

Suppressing Jitter by Isolating Bare-Metal Tasks within a General-Purpose OS on x86 NUMA Systems

**Dämpfung des Jitters durch Isolation von Bare-Metal Tasks
in Standardbetriebssystemen auf der x86 NUMA-Architektur**

Von der Fakultät für Elektrotechnik und Informationstechnik
der Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften
genehmigte Dissertation

vorgelegt von

Diplom-Ingenieur Georg Wassen
aus Leverkusen

Berichter: Prof. Dr. rer. nat. Rainer Leupers
Prof. Dr.-Ing. Stefan Kowalewski

22. Juni 2016

Diese DiSSERTATION ist auf den Internetseiten der
Hochschulbibliothek online verfügbar.

Bildungsgang

1985 – 1989 Gemeinschaftsgrundschule im Kirchfeld, Leverkusen-Lützenkirchen.

1989 – 1998 Werner-Heisenberg-Schule, städtisches Gymnasium Leverkusen.

1999 – 2007 Diplomstudium Elektrotechnik und Informationstechnik, Studienrichtung Informations- und Kommunikationstechnik. RWTH Aachen University.

2008 – 2015 Promotionsstudium Fakultät für Elektrotechnik und Informationstechnik. RWTH Aachen University.

Abstract

Today, multi-processor systems have become commonplace in the computer market from High Performance Computing down to small embedded systems. This shift of paradigm comes at the cost of adapting the software for concurrent execution. Besides correctness, real-time systems have the additional requirement of predictable timing, especially to reliably meet deadlines. This predictability is particularly hard to implement and verify on multi-processor systems. Different approaches exist that mainly restrict what the entire system executes.

This thesis presents a new concept to execute hard real-time tasks in isolation besides a General-Purpose Operating System. This is accomplished by restricting the environment of isolated tasks but still allowing the full power and performance for the remaining system. To limit the impact of arbitrary applications onto isolated tasks via shared resources, the Non-Uniform Memory-Access (NUMA) architecture is analyzed and presented as capable of eliminating the performance interference.

The isolated task concept is generally portable to arbitrary General-Purpose Operating Systems (GPOSs) and architectures but was implemented for Linux on x86 multi-processor systems. The isolation of tasks revokes control of that CPU from the operating system which is not tolerated by Linux. Therefore, several modifications are described to restore system stability. The communication and Data Acquisition is realized with shared memory and user-mode drivers which limits the impact of concurrent execution and simplifies the Control Flow Graph (CFG). Consequently, the execution time of real-time tasks is only influenced by the execution time of basic blocks. Hence, the architectural analyzes can be applied to the isolated task concept but also to all other Real-Time Operating System and research approaches. The main contribution in this area is the presentation of NUMA systems being suitable to separate memory and I/O streams to accompany the interference isolation with a complete functional partitioning of the system.

The isolated task design allows to extend existing applications with hard real-time tasks for high-frequency Programmable Logic Controller implementations. The application can still base on all available libraries and tools of current Linux distributions extended by the Real-Time Preemption Patch (RT-Patch) for better soft real-time. Hard real-time tasks previously implemented on external Micro-Controllers (μ Cs) can be merged on the same x86 multi-processor NUMA system to reduce hardware costs, improve communication throughput and latency, and simplify development and verification.

The entire concept is formally verified where the documentation allows, specifically for the interrupt-freedom and mutual impact on the CFG. The hardware effects that can not fully be derived from the documentation are analyzed based on extensive benchmarks that show the value of the NUMA partitioning in real-time systems.

Zusammenfassung

Mehrprozessorsysteme haben sich vom Hochleistungsrechnen bis zu kleinen eingebetteten Systemen durchgesetzt. Dieser Paradigmenwechsel geht auf Kosten der Anpassung der Software an die parallele Ausführung. Außer Korrektheit müssen Echtzeitsysteme auch vorhersehbares zeitliches Verhalten garantieren – das bedeutet meist das Einhalten von Zeitfristen. In Mehrprozessorsystemen ist diese Berechenbarkeit besonders schwierig zu realisieren und zu beweisen. Verschiedene Lösungen existieren, die hauptsächlich einschränken, was das gesamte System ausführen kann.

Diese Arbeit stellt ein neues Konzept vor, das harte Echtzeitaufgaben neben dem Betriebssystem isoliert. Realisiert wird das durch eine Beschränkung, was in Isolation ausgeführt werden darf. Die volle Mächtigkeit und Leistungsfähigkeit des übrigen Systems wird dabei erhalten. Der negative Einfluss des übrigen Systems auf isolierte Tasks wird weiter eingeschränkt durch die Analyse und Nutzung der NUMA- (Non-Uniform Memory-Access-) Architektur, die sich dadurch für Mehrprozessor-Echtzeitsysteme besonders anbietet.

Das Konzept isolierter Tasks kann generell auf verschiedenen Betriebssystemen und Hardwarearchitekturen realisiert werden. Im Rahmen dieser Arbeit wurde es für Linux auf x86 realisiert und untersucht. Das Isolieren von Tasks entzieht dem Betriebssystem die Kontrolle über diese CPUs, was von Linux nicht ohne weiteres toleriert wird. Daher werden einige Änderungen beschrieben, die das Gesamtsystem wieder stabilisieren. Kommunikation und Datenaustausch mit physischen Systemen werden durch gemeinsamen Speicher und direkten Hardwarezugriff aus dem Benutzermodus realisiert. Diese Maßnahmen beschränken den Einfluss gleichzeitiger Ausführung auf den Kontrollflussgraph. Daraus folgt, dass die Ausführungszeit der Basisblöcke von Echtzeittasks nur noch von Hardwareeinflüssen bestimmt wird. Daher können die Erkenntnisse dieser Analyse auf alle anderen Echtzeitbetriebssysteme und Forschungsarbeiten übertragen werden. Die wichtigste Erkenntnis ist die Eignung der NUMA-Architektur zur Trennung der Speicher- und E/A-Datenströme, um die Partitionierung der isolierten Tasks durch eine vollständige funktionelle Trennung zu unterstützen.

Der Entwurf erlaubt die Erweiterung bestehender Anwendungen um echtzeitfähige, hochfrequente Regelaufgaben. Die Anwendung kann weiterhin alle verfügbaren Bibliotheken und Werkzeuge von beliebigen Linuxdistributionen verwenden. Die Echtzeiterweiterung *Linux Preemption Patch* kann genutzt werden, um weiche Echtzeitaufgaben zu verbessern. Harte Echtzeitregelungen, die zuvor auf externen Mikrocontrollern realisiert wurden, können nun auf dem gleichen x86 Mehrprozessor-NUMA-System ausgeführt werden und damit Hardwarekosten reduzieren, Kommunikationdurchsatz und -latenz verbessern und die Entwicklung und Verifikation vereinfachen.

Das gesamte Konzept wurde formal verifiziert, so weit die Dokumentation dies zulässt. Insbesondere gilt das für die Freiheit von Unterbrechungen und den gegenseitigen Einfluss auf den Kontrollfluss. Die Effekte der Hardware, die wegen mangelnder Dokumentation nicht vollständig modelliert werden können, wurden durch umfangreiche praktische Messungen untersucht. Dies unterstreicht den Wert der NUMA-Partitionierung für Echtzeitsysteme.

Contents

Abstract	v
1. Introduction	1
1.1. Motivation	3
1.2. Contributions	8
1.3. Structure	11
2. Fundamental Principles	13
2.1. Hardware	13
2.1.1. Processor	14
2.1.2. Protection Levels	15
2.1.3. Memory	15
2.1.4. Cache	17
2.1.5. Interrupts	19
2.1.6. Input and Output	20
2.1.7. x86 Architecture	21
2.1.8. Performance Details: Micro-Benchmarks	26
2.1.9. Multi-processor Systems	32
2.1.10. System Performance: Application Benchmarks	39
2.1.11. Embedded Systems	40
2.2. Operating Systems	42
2.2.1. General Concepts	43
2.2.2. Hardware Abstraction	46
2.2.3. Programming Languages and Application Programming Inter- faces	47
2.2.4. Multi-Processor Aspects	48
2.2.5. Actual Implementation: Linux	50
2.3. Real-Time	52
2.3.1. Application Architecture	54
2.3.2. Scheduling	60
2.3.3. Real-Time Operating Systems	61
2.3.4. Multi-Processor Aspects	62
2.3.5. Performance of Real-Time Systems	64

3. Related Work	71
3.1. Fundamental Real-Time Research	71
3.1.1. Isolation or Partitioning on Single-Processor Systems	71
3.1.2. Foundations of Real-Time Computing on Multi-Processor Systems	72
3.1.3. Scheduling Theory	73
3.2. Real-Time with a Linux Kernel	74
3.2.1. The Real-Time Preemption Patch	75
3.2.2. Interrupt Abstraction.	77
3.3. Partitioning and Isolation Approaches	80
3.4. Previous publications	84
3.5. Supporting Student's Theses	84
4. Isolation of Bare-metal Tasks	87
4.1. General Concept	88
4.2. Setup and Partitioning	89
4.2.1. Hardware Configuration	90
4.2.2. Linux Settings	92
4.2.3. Partitioning	93
4.3. Process Setup	96
4.3.1. Moving to the Dedicated CPU	97
4.3.2. Synchronization	97
4.3.3. Memory Management	98
4.3.4. Implementation	99
4.4. Full Isolation	100
4.5. Analysis	101
4.6. Practical Evaluation	103
4.6.1. Risks	109
4.6.2. System Stability	109
5. Linux Kernel Modification	111
5.1. Monitoring System Stability	112
5.2. Linux Kernel Patch	114
5.2.1. Infrastructure	115
5.2.2. CPU Online Mask	116
5.2.3. Inter-Processor Interrupts	116
5.2.4. Read-Copy-Update	120
5.2.5. Notifier Chain	121
5.3. Deactivation of the Timer-Interrupt	122
5.4. Alternative Implementations	123
5.4.1. Kernel Module	123
5.4.2. Hotplugging	124

5.5. Evaluation	125
6. Application Support	127
6.1. Shared-Memory Communication	130
6.1.1. Signals	132
6.1.2. Mutual Exclusion	134
6.1.3. Complex IPC objects	136
6.1.4. Dynamic Memory Allocation	140
6.2. Management	140
6.2.1. Control	140
6.2.2. Supervision	142
6.2.3. Adaptability	143
6.2.4. Exceptions in Isolated Tasks	144
6.3. Input and Output	145
6.3.1. Direct Hardware Access	146
6.3.2. Protocol Stack	147
6.4. Interrupts	148
6.4.1. User-Mode Interrupts	148
6.4.2. Interrupt Sources	152
6.4.3. Preemptive Scheduling	152
7. Architectural Influence	155
7.1. Estimation of the Execution Time	156
7.2. Multi-Core Systems	158
7.3. Uniform Memory Access Systems	162
7.4. Non-Uniform Memory Access	164
7.5. Symmetric Multi-Threading	169
7.6. Input and Output	170
7.6.1. Basic Access	171
7.6.2. Non-uniform I/O Access	173
7.7. Interpretation and Recommendations	177
8. Application Examples	181
8.1. Hardware Selection	181
8.2. Software Architecture	183
8.2.1. Graphical User Interface	185
8.2.2. Manager and Isolated Tasks	185
8.2.3. Closed-Loop Control and I/O	186
8.2.4. Observation of Execution-Times	188
8.2.5. Error Recovery	188
8.3. Other Applications	189
8.3.1. Application-Specific Integrated Circuit	189

Contents

8.3.2. Network Packet Processing	190
8.3.3. Medical Imaging	191
8.3.4. More Examples	191
9. Conclusion	193
9.1. Summary	195
9.2. Evaluation	197
9.3. Future Perspectives	199
9.3.1. Hardware	200
9.3.2. Linux	202
A. Test Systems	205
A.1. Intel Core 2	205
A.1.1. Test system <code>poodoo</code>	206
A.1.2. Test system <code>octopus</code>	206
A.2. Intel Nehalem	206
A.2.1. Test system <code>xaxis</code>	207
A.3. Intel Westmere	207
A.3.1. Test system <code>haixagon</code>	208
A.4. Intel Sandy Bridge	209
A.4.1. Test system <code>pandora</code>	209
A.5. AMD	210
A.5.1. Test system <code>devon</code>	210
B. <code>smp.boot</code> Kernel	213
C. Examples for Real-Time systems	215
Acronyms	219
Glossary	225
List of Figures	245
List of Tables	249
List of Listings	251
Index	253
Bibliography	261

1. Introduction

“Actually, manycore has been around for many years in the desktop and supercomputing arenas. But it has lagged in the mainstream embedded world; it is now here for embedded as well.” Moyer [Moy13b, p. 1]

Multi-core processors and generally multi-processor computers have in fact become ubiquitous. Not only in High Performance Computing (HPC), but also in commodity computing and in recent time also in small embedded systems, multiple processors work in parallel. This architectural shift requires a more radical change of the applied software than any other improvement of the last decades because the concurrent execution requires a careful synchronization [Sut05]. In time-sensitive applications, the whole system needs also to be predictable and efficient.

This work targets applications that require hard real-time with very low latency and jitter in the Millisecond range, together with compute- and memory-intensive tasks, and also a high data-rate and low latency communication between them. The classical implementation would be a distributed system of multiple μ Cs, connected by a field bus to a high-performance computer. Recent development of multi-core general-purpose processors (and generally, multi-processor systems) enables a consolidation of multiple, distributed systems into a single computer. The applications become complex using multiple communicating processes or threads, but the development can be unified to a single architecture. Common tools (compiler, development environment) simplify debugging and verification, and the performance can be increased since Commercially off-the-Shelf (COTS) systems provide a superior performance per cost ratio. The architecture of x86 compatible systems with processors from Intel and AMD was selected, because it has the widest distribution in server systems.¹ Although ARM-based systems are closing up, those are usually built as fixed systems that are not extensible while x86 systems can be built flexibly to order from components and can be extended with devices via PCI Express (PCIe). This type of hardware is often referred to as COTS to emphasize its easy availability, wide distribution, and low costs and is utilized in an increasing number of real-time and embedded systems [Eid+04; Dom08; Moy13a].

The system software for multi-processor systems can be an arbitrary operating system (OS) extended for multiple processors or any mix of multiple virtualized OS instances. Instead of installing a multi-processor capable Real-Time Operating

¹In High Performance Computing, 458 systems in the Top 500 list of June 2014 use x86 processors [Meu+].

1. Introduction

System (RTOS) that limits the general flexibility of existing applications, this work evaluates the possible limits for hard real-time tasks on commodity operating systems. Linux was selected for its wide deployment, but also because it is scalable from embedded systems and smartphones² up to most of the TOP 500 high-performance computers.³ A further advantage is its open source license that is cost-effective (no license fee per deployed system) and allows to study its implementation and to modify its source code. Linux supports many peripheral devices and a wide range of modifications by other developers and researchers is available to be integrated selectively into a self-built Linux kernel.

The approach taken here is to isolate some processors for a bare-metal execution of hard real-time tasks while the remaining processors execute standard software of the Linux system. This is a partitioning approach of a Symmetric Multi-Processing (SMP) OS into Bound Multi-Processing (BMP) [Moy13d] extended with bare-metal execution on the isolated processors. The isolation of the real-time processors must be arranged with the Linux kernel that is generally not prepared for such an operation. Further, the hardware effects of concurrent access to shared resources (Last-Level Cache, system buses, memory) is analyzed to find solutions to isolate bare-metal tasks from interfering execution on other processors. The specific goals are:

- Use a standard operating system on common multi-processor hardware. The experiments are implemented on Linux for x86 (32 and 64 Bit) systems, but should be transferable to different operating systems and architectures.
- On such a multi-processor system, reserve a (or some) processor(s) for hard real-time work, leaving other processors fully usable for soft real-time and compute-intensive tasks to exploit the power of the Linux system (with all services and libraries).
- Execute hard real-time tasks with full control of every execution cycle of their Central Processing Unit (CPU) (bare-metal execution).
- Allow a flexible reconfiguration of the partitioning of CPUs for real-time, system and high-performance jobs.
- Evaluate the effect of *stealing* processors from the Linux kernel to execute isolated tasks and find a configuration that keeps the system stable preferably without or with only minor modifications to the Linux kernel.
- Analyze the special execution environment of bare-metal tasks for limitations and provide solutions for the implementation of usable applications. This includes the management of isolated tasks, Inter-Process Communication and hardware access.

²in mid 2013, nearly 1 billion Android smartphones are based on a Linux kernel.

³486 out of 500 systems in the Top 500 list of June 2014 [Meu+] run Linux.

- Analyze the timing behavior: Is this really hard real-time? Evaluate sources of jitter (software and hardware) and reduce them as much as possible to provide strong guarantees for a tight Worst-Case Execution Time (WCET) estimation.
- Solve the jitter problem on x86 multi-processor systems by analyzing the sources of jitter and providing guidance to design the partitioning to reduce hardware influence. By using NUMA systems, it will be shown how the performance interference can be constrained.

1.1. Motivation

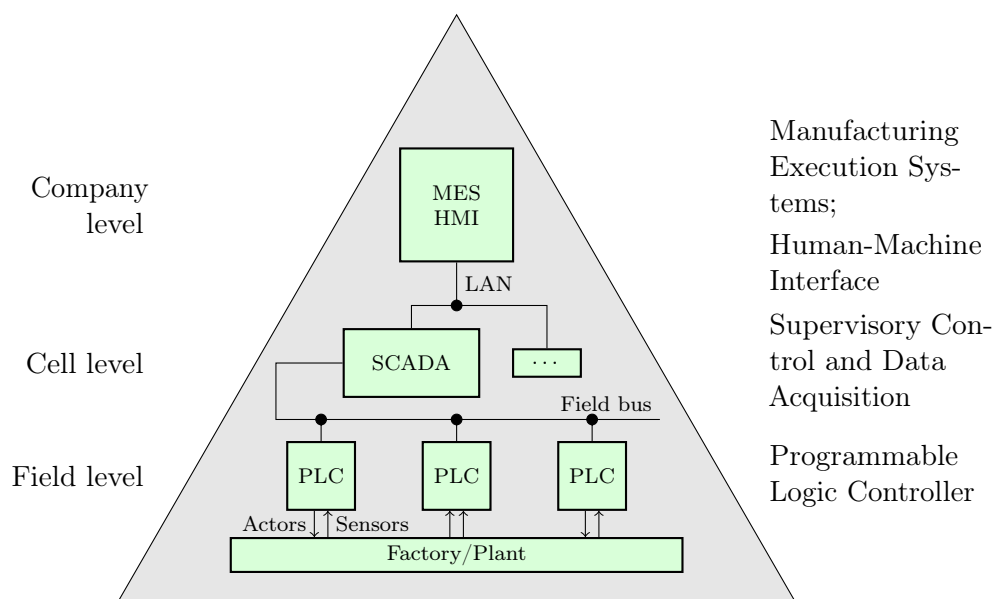


Figure 1.1.: Automation pyramid (simplified) [Sau07]

Real-time applications require not only a correct result but need the reaction also by a defined deadline (the terms will be defined precisely in Section 2.3). This is a very wide field of research. Typical application areas of real-time systems can be illustrated in the automation pyramid (Fig. 1.1) [Sau07]. The field level is the lowest as it directly interacts with physical systems of a factory (plant). The components are typically Programmable Logic Controllers (PLCs) that use sensors to detect the state of the plant and actors to adjust the plant's properties. The cell level controls multiple components of the field level and is itself managed by the above company level. From top to bottom, the timing requirements become tighter and the frequency increases. In reverse, in the higher levels, the complexity of the algorithms raises as optimization and guidance of multiple systems must be accounted for. The company level may even connect multiple buildings or remote sites over Wide Area

1. Introduction

Network (WAN). Classically, the components of the automation pyramid at cell level and below are dedicated embedded systems that communicate via field bus. The company level and sometimes also the cell level uses Personal Computer (PC) type systems with Human-Machine Interfaces (HMIs) and database access.

So far, digital controllers with hard real-time and low latency requirements are often implemented on μ Cs without an OS. Programming the hardware in such a way is referred to as bare-metal execution. This allows the full control over every functional unit and CPU cycle but waives the comfort of the OS's hardware abstraction (Section 2.2.2) and complicates the portability [Lal13, Sect. 1]. The current realization of complex systems is based on special embedded systems communicating via field bus. The concept of isolated tasks presented in this work allows to horizontally integrate multiple PLCs and to vertically integrate multiple levels. This can be realized on a single x86 multi-processor system with various tasks from digital control up to command and control. This work targets complex time-sensitive applications that include multiple levels of the automation pyramid, for instance hard real-time tasks with a time-frame below 10 μ s as well as multiple compute-intensive jobs. An exemplary application is a chemical experiment and measurement control system. This can be used as prototypical example to present the application of the following concepts.

- Control a research experiment by reading sensor data and setting actor commands, e. g. a chemical reaction experiment with sensors such as temperature, pressure, filling level and settings such as valves, electric pumps, heaters, coolers, blender, etc.
- Guarantee hard real-time for the correctness of the results. This is not because somebody could get hurt if a deadline is missed or the system fails (which would be *criticality*, Section 2.3) but the results would be biased, a scientific finding must be based on trusted measurements and the expensive or long-running experiment had to be repeated.
- Constrain latency and maximum jitter to the low Microsecond range.
- Log sensor values to a database, pre-process results to reduce the data volume.
- Provide an interactive Graphical User Interface (GUI) to display the current state, allow to change parameters, start and abort the experiment, and execute a first analysis of results to indicate success or failure of the experiment.
- Integrate algorithms flexibly: Use an external program to convert a graphical representation of a control (e. g. Matlab Simulink, Labview) into code and compile it into a library loadable by the control application.

- Include compute-intensive jobs, e. g. the analysis of a video stream (soft real-time, but best-effort), or a complex simulation of the reaction model to derive non-measurable physical quantities.
- Provide high-throughput and low-latency communication between tasks, especially between real-time and non-real-time tasks without introducing possible blocking into the real-time execution.

Liu lists many examples of algorithm complexity demonstrating that an increasing computing power is required for modern real-time data processing systems [Liu00, Chap. 1]. For example, real-time databases contain perishable data with degrading precision (e. g. in radar tracking of airplane positions). Each data entity has an age and a temporal dispersion. Additionally, the query process must be deterministic in time. In contrast, general purpose databases (e. g. a company payroll) contain data that remains correct in the absence of updates. More examples of real-time applications are listed in Appendix C on page 215.

This wide range of applications illustrates some more use cases for the presented concept. All these systems require a combination of hard real-time with very low latencies (below 10 μ s) and a high compute performance. Two trends become apparent, the use of multi-processor systems to exploit their increased compute-performance and the application of commodity hard- and software instead of specialized μ Cs and RTOSs [DM03a]. The first step towards using multiple processors are distributed systems [Kop97] (federated architectures [WR12]). These use multiple distinct computers that tightly cooperate by using interconnects [KW05; Kop08] for message passing and synchronization [KG93]. In large-scale computing, this architecture [Sta09, Chap. 16] is used to increase the performance (e. g. in HPC) or availability (e. g. internet servers, “cloud-computing”).

The trend of multiple cores per processor package will probably continue, Agarwal and Levy expect to see processors with 1000 cores around 2017 [AL07] and Vajda estimates 250 to 4000 cores for 2020 [Vaj11, Sect. 1.2]. At the time when this project was done⁴, systems with two to 16 processors were common. The PC and server market is currently dominated by the x86 architecture with processors from Intel and AMD. The strongest competing processor architecture is ARM that leads the market for mobile devices and embedded systems. Various companies announced an emphasized effort to enter the server market with new energy efficient multi-processor ARM systems.

With the ubiquitous availability of multi-processor computers on every scale from small embedded systems up to high-performance servers, these systems are already installed in real-time applications [HX96; VT10; WR12; Moy13a]. This is supported

⁴Planning from 2010, main implementation 2012, evaluation 2013

1. Introduction

by many developers who are experienced with the architecture they know from desktop computers.

A straightforward approach to transfer multiple existing single-processor systems to a multi-processor system is Asymmetric Multi-Processing (AMP) [NKN08]. This partitioning allows to execute single-processor software on dedicated CPUs. It can be realized with limited effort and the individual applications keep their timing and verification because they are functionally isolated. However, this leaves only benefits from saving multiple (possibly similar) embedded systems by replacing them with one multi-processor system. The communication done by field bus before must now be implemented in software. With more effort, more complex software architectures can be realized to exploit the scalability of multi-processor platforms [BCA08; NP12].

Independently from the OS architecture, the realization of real-time applications on multi-processor hardware offers many challenges. An RTOS is designed from the ground up for determinism. It is far more difficult to modify an existing GPOS for predictable response times because all non-preemptive code paths must be evaluated and possibly modified for a bounded execution time [MS05]. The extension of single-processor RTOSs to support multiple processors is challenging to scale well. For time-sensitive applications, it is far more complicated to avoid all possible indeterministic algorithms. A third way is a combination of multiple separated OSs for different tasks.

A complex OS needs computing time for itself to maintain internal structures. This time missing to the real-time application is called OS noise [Lam09]. Further, if applications – executed concurrently by multiple processors – share functional units, they influence the execution time of each other because of hardware effects. All these causes increase the total jitter experienced by the applications running on multi-processor systems. This is also unwanted in HPC where real-time methods are applied increasingly to fine-grained synchronization to improve the lock-step of massively concurrent applications [McK96; Tsa+05; Boc+09; Mor+11].

Unpredictable timing stems from macroscopic causes like scheduling, interrupts, operating system jitter, or indefinite loop counts. All these are in the scope of functions. Further, the execution time of an uninterrupted piece of code is subject to *hardware jitter* caused by caches and shared resources. *Timing anomalies* [LS99; Rei+06] are architecture-dependent and can hardly be solved (completely) by software. All dedicated RTOSs have to work around these problems.

Many researchers try to estimate interrupt handling time to account for it more precisely in the Worst-Case Execution Time [BLA11]. Part of that theoretical work uses assumptions not applicable to actual systems or proposing improvements for future architectures. However, if the duration of interrupt handlers can only be predicted inaccurately, their frequency of occurrence is even harder to foresee. This results usually in an overly pessimistic estimation.

Other challenges include the hardware abstraction provided by OSs that must be realized with predictable timing if the application depends on it. If no suitable RTOS is available, a new one could be implemented or an existing GPOS could be modified. For complex processors such as the x86 architecture, the overhead of initializing the processor and implementing drivers for all hardware devices would be very costly. If the device drivers provided by the GPOS are not real-time capable, the drivers required by the real-time application are sometimes implemented in user-space. Examples are General-Purpose I/O (GPIO) [Abb13, Chap. 7], user-space network drivers [Cor13d] and block Input/Output (I/O) [Cor13g].

The synchronization in multi-processor RTOSs must be predictable and avoid or prevent deadlocks. In priority-driven scheduling, the *priority inversion* problem describes a higher prioritized task waiting for a resource blocked by a lower prioritized one. Since the resource can not be revoked, a viable solution is to increase the priority of the blocking task giving it priority over the waiting one until it releases the resource (priority ceiling or inheritance). Message passing and other concurrent functions can be implemented with *lock-free* or even *wait-free* algorithms. [SS11]

Linux⁵ is an operating system with a very versatile application range. Its availability under the terms of the open source license GPL version 2⁶ allows to use and redistribute it free of charge and to modify it as required. Linux is the basis of embedded and mobile systems (e.g. the Android⁷ smartphone OS, or the Raspberry Pi⁸), desktop and workstation computers (e.g. in the form of distributions like openSUSE⁹ or Red Hat¹⁰), and is used also by performance critical servers, cloud installations and in High Performance Computing. Further, Linux is increasingly utilized in embedded control systems [Dom08; BA09a].

Various efforts are taken to improve the timing predictability of Linux as operating system for real-time applications [Sri+98], most notably the RT-Patch¹¹. This is a collection of modifications that change the behavior of many places in the Kernel (Linux with RT-Patch is referenced as Linux-rt). Since the introduction, many parts of this collection have already been integrated into the Linux mainline Kernel (and can be activated on demand). The major improvements for a better (hard) real-time support are to reduce non-preemptive sections, to include a real-time capable Read-Copy-Update (RCU) variant, and to improve the timing predictability of synchronization methods [MS05]. The long-term evaluation of Linux-rt [Emd10] demonstrates impressive numbers, but this can still not be considered *hard* real-time because that would require a formal verification.

⁵Online: <https://www.kernel.org> (visited November 12, 2015)

⁶Linux file COPYING (Linux 3.12)

⁷Online: <http://www.android.com> (visited November 12, 2015)

⁸Online: <http://www.raspberrypi.org> (visited November 12, 2015)

⁹Online: <http://www.opensuse.org> (visited November 12, 2015)

¹⁰Online: <http://www.redhat.com> (visited November 12, 2015)

¹¹Online: <http://rt.wiki.kernel.org> (visited November 12, 2015)

1. Introduction

Interrupt handlers on current Linux systems interrupt the running task for 2 to 40 μs and even more with applied RT-Patch. If the hard real-time task requires less jitter than that, even a single interrupt will cause a violation of the timing. It is a common practice to confine the handling of Interrupt Requests (IRQs) to dedicated CPUs [DW05]. Brosky and Rotolo propose to allocate some processors of a multi-processor computer solely for real-time applications [BR03] (Shielded CPUs). A fairly recent publication promotes CPUs isolated for real-time applications because the scheduling latency of Linux-rt (Linux 3.8.13-rt) was measured between 11 μs in an idle system and 44 μs during I/O bound workloads [CB13]. Further, the integration of separately developed components for a real-time system often reveals timing problems due to collisions on the bus level caused by memory and I/O transfers [PC07; Das+11; Das+13].

Several approaches further optimize the real-time behavior and predictability of a system running a Linux kernel. Between changing the Kernel itself (which will always remain soft real-time) and employing a dedicated Real-Time Operating System (that will never provide full Linux compatibility), several approaches execute the Linux kernel under control of a sub-kernel. That hypervisor or Hardware Abstraction Layer (HAL) is a Real-Time Operating System executing all hard real-time tasks. When no time-critical task or deadline is pending, the Linux system is executed as idle task. This type of system includes two different domains in which a task can be executed. Originally, both the sub-kernel with the hard real-time tasks and the GPOS with its jobs running when the real-time part is idle were executed on the same processor. With the advent of commonly available multi-processor systems, these approaches adopted partitioning to execute the hard real-time tasks on a processor reserved solely for this purpose. Although the development environment is quite advanced and tasks can be moved between the domains, the whole system is not fully compatible to modern Linux distributions and it is still uncertain, if an existing complex application can be ported to them.

1.2. Contributions

The broad availability of multi-processor systems (Sect. 2.1.9) enables the application of concepts to real-time systems, that are established in high-performance computing since long ago. Having multiple tightly cooperating concurrent processes or threads is a foundation of parallelizing programs. The optimization on NUMA systems demands careful planning of where a task will run and where its data should be stored. However, common algorithms and synchronization functions are optimized for average throughput. But for large systems, the temporal predictability becomes more relevant because with fine-grained synchronization, a deviation on one processor has a large impact on the overall throughput. In this work, best practices from

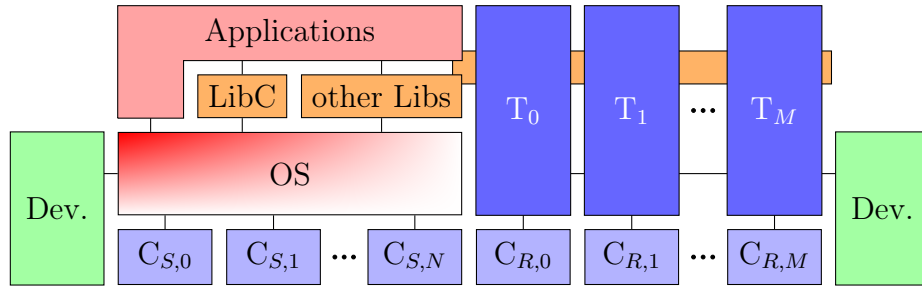


Figure 1.2.: Layered diagram of the proposed system architecture

HPC optimization like using CPU affinity to assign specific tasks to dedicated CPUs, careful partitioning of memory access (esp. in NUMA systems), fine-grained synchronization, and user-mode implementation of synchronization methods will be applied to multi-processor real-time systems.

To combine the comfort and versatility of a GPOS with the predictability of bare-metal programming, this work examines how to partition a multi-processor system into a general-purpose partition and one or more hard real-time partitions (called *bare-metal tasks*) that are isolated from the OS.

The concept of isolated CPUs executing bare-metal code addresses the macroscopic cause of jitter from OS and interrupts and proposes a method to execute tasks with a predictable timing. That is hard real-time with regard to task scheduling (no interruptions from unknown code), but the execution time of every uninterrupted piece of code or basic block on the x86 architecture can not be determined that way. Only by estimating the WCET very pessimistically or by applying advanced scientific methods, the whole solution can satisfy the academic definition of *hard real-time*. To solve or at least attenuate the problem of architectural jitter, the causes in multi-processor systems are analyzed and it will be shown how the NUMA architecture can be utilized successfully.

The layers of a typical GPOS based system are displayed in the left half of Fig. 1.2. The OS controls the CPUs and devices. The applications can directly access the OS interfaces but typically use libraries. The main contribution are several bare-metal tasks executed directly on dedicated processors so that they become isolated from the system partition and their performance is not influenced. They control their assigned devices and can communicate with other bare-metal tasks and applications in the system partition without use of the OS.

The general concept of bare-metal tasks was first published in 2012 together with benchmark results with static loads [WLB12]. The isolated task concept was brought to practical usability for a contract partner that deploys its product including multiple isolated tasks successfully. In that project, the problem of x86 multi-processor hardware jitter was solved by utilizing a NUMA system. This work now presents detailed evaluation and advice how to realize bare-metal tasks with

1. Introduction

minimum hardware jitter on x86 NUMA systems. Specific features of the presented architecture are:

- Isolated bare-metal tasks are UNIX processes (or threads) executed in user-space. Their memory protection from each other and from general-purpose processes is guaranteed by the operating system architecture which bases on hardware features.
- Isolated bare-metal tasks can be implemented as threads of a larger application where threads can be configured for different timing guarantees. Implementation is generally possible in every compiled programming language provided that its run-time is free of system calls. For example, C++ can be used if all objects are created statically on start-up and only their internal methods are invoked during run-time (e. g., `new` must not be executed during bare-metal execution).
- Initialization at start-up time: All resources (memory allocations, shared buffers, IPC, devices) should be initialized at start-up time. However, local real-time proved allocators can use preallocated memory pools to allow dynamic programming.
- Single process per CPU: Each isolated process executes solely on a dedicated CPU (time-triggered paradigm [Kop91]). With user-mode interrupt handlers presented in Section 6.4.1, event-triggered applications and preemptive scheduling can still be realized.
- The process that will be an isolated task can be initialized as usual process with all libraries. It can even switch between fully isolated mode and normal operation, but the duration of the switch to isolated mode is not predictable.
- Hard real-time execution can only be guaranteed when not issuing system calls. This prevents the use of the operating system's device drivers. For lowest latency and jitter, device access must be implemented from user-space via direct or memory-mapped I/O.
- The general-purpose part of an application can utilize all Linux services and all available libraries such as, for example, Qt for Graphical User Interfaces, ImageMagick for image processing, Open CV for video processing, data-base connections, GSL for data processing, or the inclusion a scripting language like Lua. If that part of the application has soft real-time demands, the Linux system can be implemented with the RT-Patch to increase its predictability.
- Because of the bare-metal execution, the latency of hard real-time tasks is the optimum, that can be realized on the x86 architecture.

- The predictability of the bare-metal execution is further improved by analyzing the hardware architecture of multiple processors and presenting methods to avoid mutual influence on NUMA systems.
- High Performance Computing optimization experience is applied to parallelized mixed hard real-time and compute-intensive applications.
- The eviction of private cache elements by execution of other cores on the shared Last-Level Cache was not described before. As was the use of PCIe in hard real-time systems.
- Hardware-in-the-Loop simulators realized as distributed system need to handle clock drift [Pal+11]. The isolation approach enables the implementation as a single multi-processor system with a consistent clock.

The concept of isolated tasks on a multi-processor system allows to extend existing complex applications using the well-known x86 environment. Besides the availability of libraries for large (existing) applications that should be extended with high-frequency real-time tasks, also their use of a specific compiler (e. g. through compiler-defined behavior) may be a reason not to change the execution environment or operating system version. The entire application can be built with the same tools. Further, this concepts enables a rapid implementation to gain short time to market, prototypes, and small series. Both research on such systems as well of research applications (control of experiments in other fields) can benefit from this concept.

1.3. Structure

In the following Chapter, the fundamental principles of the used hardware, operating system, and real-time research is introduced. Chapter 3 presents related work and explains where this concept differs from previous efforts. The main part starts with the presentation of the isolation concept in Chapter 4 and a description of the required modifications to the Linux kernel to stabilize the system in Chapter 5. Chapter 6 presents solutions to circumvent the limited execution environment of bare-metal tasks on isolated processors. The hardware effects of jitter induced by the x86 architecture is addressed in Chapter 7 where the NUMA architecture is presented as a viable solution for both memory and I/O transfers. Chapter 8 demonstrates how applications based on the presented concepts can be implemented or how existing complex applications can be extended. Finally, Chapter 9 concludes this work and presents future perspectives.

2. Fundamental Principles

This Chapter introduces to the basic principles required for the following Chapters. The hardware is described to derive models of the system's behavior needed to analyze the predictability of the isolated processor concept and the hardware jitter effects. Further, operating systems and real-time systems are presented to centralize definitions and to explain details built upon later.

2.1. Hardware

This work mainly concentrates on current multi-processor systems of the 80x86 (x86) architecture using processors from Intel and AMD. This group of processors founded the class of Personal Computer (PC) systems, multi-purpose computers that quickly gained a large market share and built the base of today's ubiquity of computing. Besides the originating PC systems, this architecture also powers servers, mobile devices and embedded systems.

Contenders of the x86 architecture are, among many others, ARM, MIPS, and PowerPC. All these processors are descendants of the *Von Neumann* architecture. Programs in binary code can not directly be exchanged between different architectures, but the principles are similar and high-level languages can typically be compiled to all of them. Although this work employs the x86 architecture, the concepts can be transferred to other architectures.

The *Von Neumann* computer [Neu93] describes a basic model of a computer consisting of a Control Unit (CU), an Arithmetic Logical Unit (ALU), memory and Input/Output (I/O). Today, the former two are integrated in a processor and all parts are still the main functional units of computer systems. Current processors are more complex, but from the software side, they can still be modeled as *Von Neumann* architecture.

This Section introduces only the foundations required for this work. More information can be gathered for example in the textbooks *Computer Architecture* by Hennessy and Patterson (Design decisions) [HP07], *Computer Organization and Design* by Patterson and Hennessy (Hardware functionality, emphasis on parallel processors) [PH11], and *Advanced Computer Architecture* by Hwang (High-performance computing) [Hwa93]. All details about the applied x86 processors are documented in the according manuals of Intel [Intel13e] and AMD [AMD12d].

2. Fundamental Principles

2.1.1. Processor

The Central Processing Unit (CPU) is the active part in a computer that executes the programs. It contains a Control Unit (CU), at least one Arithmetic Logical Unit (ALU), and a set of registers. It accesses the memory via a data path and the Input/Output (I/O) capabilities either via the same path (system bus) or using a separated specialized path. The CU controls the execution of code by reading instructions and managing the ALU and registers. The set of registers contains data registers for operands and results, and control registers, most notably the Instruction Pointer (IP), the Stack Pointer (SP) and a Flags register. The IP holds the address of the next instruction to execute. The SP manages the top of the stack (see below) and the Flags register helps with arithmetic and logic operations and branches (e.g. indicates whether an overflow occurred in the last operation).

An instruction is a directive to the CU or an ALU, possibly accompanied by one or more parameters. Examples are the loading of data to a register, an arithmetic operation on two registers, or the jump to a different location to continue execution there. The instructions understood by a Central Processing Unit (CPU) are defined in the *instruction set*. It is typically consistent for a selected architecture. The bit-width defines the size of basic registers and the size of pointers to addresses. Therefore, a 32 Bit system supports generally up to 4 GiB of address space ($2^{32} = 4 \text{ GiB} \approx 4\,000\,000\,000$) and the 64 Bit instruction set is an extension supporting a much larger address space. The instruction sets can be classified for the extend of supported instructions and combinations. A Reduced Instruction-Set Computer (RISC) architecture has generally are smaller set of instructions and is more streamlined easing the CU's work. On the opposite, a Complex Instruction-Set Computer (CISC) architecture offers specialized instructions that can only be realized with multiple RISC instructions. Thus, many tasks can be accomplished in various ways and optimization becomes important and powerful.

A typical instruction is accompanied by up to two operands, the *destination* and *source* argument. An example x86 instruction is `ADD rax, rbx` that instructs the processor to add the value of the register `rbx` (source) to `rax` (destination) storing the result in `rax` (in other words: $\text{rax} \leftarrow \text{rax} + \text{rbx}$). The IP is either advanced to the next instruction or moved by a larger, given distance by a control transfer instruction or diverted by an interrupt (Section 2.1.5). The former two cases immediately follow the internal state of the processor. Although external state can take influence by I/O instructions, it is controllable if those are used. In contrast, the advent of interrupts is generally unpredictable.

The processor executes instructions with a high frequency, today's processors easily reach more than 2 GHz. This equals a cycle length of less than 0.5 ns (a time in which electrical signals can travel no further than 15 cm). At such high frequencies, not every instruction can be executed completely in a single cycle. Especially loading data from memory takes considerably longer and will delay the execution.

For a long time, the performance of processors measured in (million or billion) operations per Second (MFLOPs, GFLOPs) could be increased by higher frequencies. But this resulted also in a higher energy consumption and the heat dissipation became a problem. Other possibilities to increase the performance were introduced, for example pipelines to interlace execution advancement [PH11, Sect. 4.5], Instruction-Level Parallelism (ILP) to charge multiple ALUs from a single instruction stream [PH11, Sect. 4.10] resulting in super-scalar execution (multiple instructions per cycle), out-of-order execution to temporarily overtake non-depending instructions, and more. Thereby, the complexity of processors raises and the pipeline length grows steadily. Since wrongly predicted branches require the pipeline to be filled again, the predictors became more complex and performance-critical.

Today's x86 processors are very complex and this renders them very hard to predict. They are optimized for instruction throughput, but the time required for a given program depends on many parameters, some of which are very hard to investigate [Fog13a].

2.1.2. Protection Levels

Many processor architectures support different protection levels. The Current Privilege Level (CPL) is a part of the state of the processor and the code it is currently executing. Memory regions have associated a Descriptor Privilege Level (DPL) and code is only allowed to access a certain part of memory if the CPL is lower or equal to the DPL. Figure 2.1 displays an extended ring structure [SS72] of privilege levels. Generally, code executed in a certain level has full access to the above levels (memory, function calls), but can only use well-defined services and interfaces from lower levels.

2.1.3. Memory

The Turing Machine's band extends its state space over a finite state machine. Likewise, the memory extends the state space of the Von Neumann computer beyond the internal states of the processor (i. e. registers and flags). Each location in memory can directly be accessed by its address. In modern operating systems (OSs), every process has a distinct, linear address space, that is mapped to physical page frames with the help of a Memory Management Unit (MMU). Thereby, multiple processes can execute concurrently (or interleaved) each having their own view to a distinct linear address space without interfering with other processes. In x86 processors, the MMU is a physical unit doing the address translation automatically and transparently for the running program. The details of the translation of virtual to physical addresses (Paging and Segmentation [Intel13c; AMD12b]) are of less importance for this work. A number of recent translations is cached in the Translation Look-aside Buffer (TLB). On missing that cache, the translation takes a considerable amount of time which must be regarded in time-critical systems.

2. Fundamental Principles

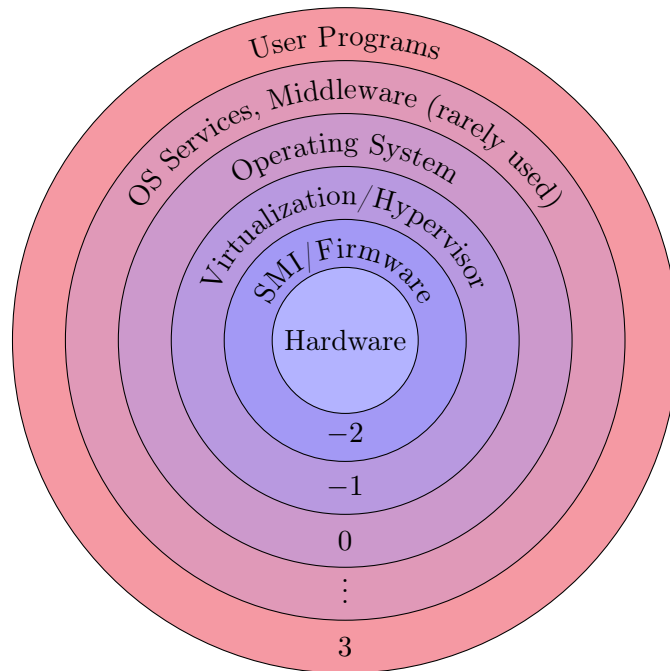


Figure 2.1.: Ring structure of privilege levels. The levels 0–3 are exemplary for the x86 architecture, the levels –1 and –2 are optional.

In a typical multiprogramming OS [Tan07, Chap. 2], every process uses its own memory sections for code, data and stack (Fig. 2.2). The stack is a First-In, First-Out (FIFO) data structure for temporal storage supported by the processor’s SP register. Further, the OS is embedded in the address space, but not directly accessible from the process to protect global data structures. It is possible to map physical memory into multiple processes to share data between them (shared memory). After its initialization by the OS, the access to shared memory is handled by hardware in the same way as the access to private memory.

The *demand paging* is a feature to make more memory available than physically present in the system. It is based on the observation that some parts of the memory are seldom used, for example only during the initialization or in the case of an error and that most processes show a temporal and spacial locality, i. e. focus on a working set. Some regions can be swapped out of the main memory and stored on the hard disk on the granularity of pages (usually 4 KiB). Optimally, the logic chooses pages, that will not be used in the time to come. When a processor tries to access such a relocated page, a *page fault* handles the reloading of the respective page (by displacing another page). This proceeding is transparent for the execution of the CPU except for the lapse of time caused by the interruption.

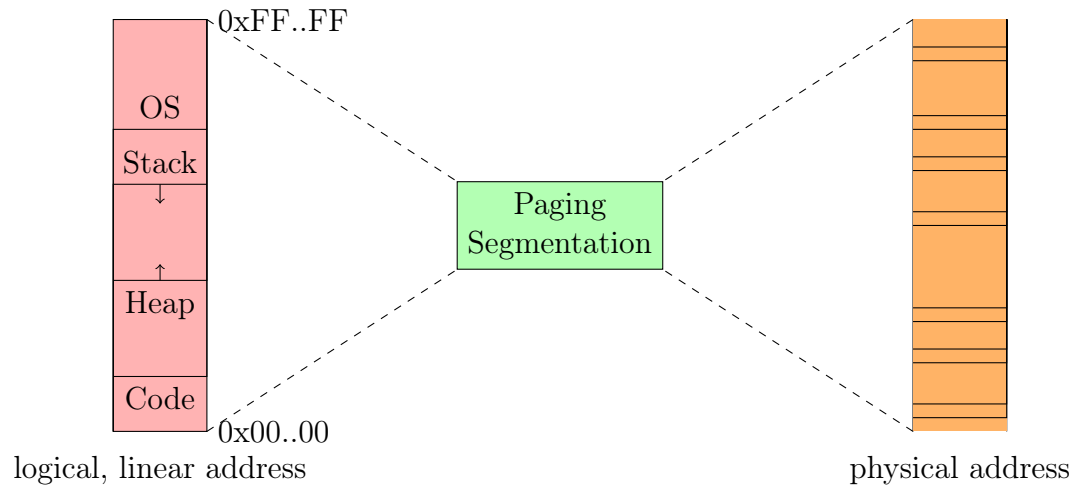


Figure 2.2.: Address space of a process and mapping to physical memory.

2.1.4. Cache

Over the years, the CPU speed increased faster than the memory transfer rate (Moore's law [Moo06]). To reduce the effect of this *memory wall* [WM95], caches were introduced. Caches benefit from the same *working set* observation, that also makes demand paging effective: Most programs have a small set of memory that they use at a time. This working set moves, but the acceleration of successful accesses more than balances the required reloading of new regions. Since faster caches are more expensive, current processors use multiple levels of caches with a larger capacity with growing distance to the CPU. Most strategies try to approximate required memory by keeping the most recently used data or the most frequently used data. The locality also results that the probability that data in near locations to recent accesses will be used soon is higher than the request for distant data. This makes caches efficient that can keep recently used data and pre-fetch adjacent elements.

Current x86 processors include up to three levels of caches named from L1 (near to the processor, very fast, in the range of 32 to 64 KiB) to L3 (far, slower, 8 to 16 MiB). Figure 2.3 displays the cache hierarchy of the Pentium processor. The First Level Cache (L1\$) is usually separated into two parts exclusively for data (L1d) and instructions (L1i). Without loss of generality, only the data cache is regarded in the following. The term Last-Level Cache (LLC) can be used independently if two or three levels exist. The access times range from 3 cycles for the private L1\$ up to some tens of cycles for the LLC which is a major improvement to over 100 cycles for main memory access [Fog13c, Chap. 11].

The smallest entity in caches is a cache line. This is a block of Bytes (in current processors usually 64) that is always loaded from or written to lower levels contiguously. If a variable with a size of 4 Bytes is written, the entire cache line must be

2. Fundamental Principles

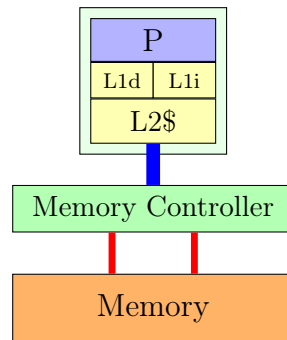


Figure 2.3.: Cache levels of Intel P4 processors.

read into the L1\$ (if not preexisting) and subsequently be transferred back to the lower levels and the main memory.

Consequential, all cache-levels must be synchronized to guarantee data consistency. The policies supported by x86 processors are Write-Through (changes are immediately propagated to lower levels and the main memory) and Write-Back (changes are written back not until the eviction of a cache line) [Hwa93, Sect. 4.3.2]. The former results in higher cache and memory contention but faster eviction to load new data while the latter reduces the bus load at the cost of eviction time.

Multiple cache levels can further be organized inclusively or exclusively [BW88; Dre07, Sect. 3.2]. Inclusive caches keep all elements of higher levels in the lower levels so that evicted unchanged or written-through cache lines can simply be dropped [Intel13c, Chap. 11]. This is at the cost of the total amount of cacheable data being the size of the largest cache. In contrast, exclusive caches allow replacing cache lines from lower levels if they are moved to higher levels increasing the total cacheable amount of data to the sum of all caches [AMD12b, Chap. 7]. The drawback of this strategy is that modified cache lines must be written back eventually evicting other cache lines from the lower levels.

A cache can be organized with physical or with virtual addresses. If using virtual addresses, the access can use the address directly without converting it to the physical address. But since virtual addresses are only valid for the current process, every switch to a different context invalidates the entire virtually indexed cache. Therefore, current processors implement the smallest cache with virtual addressing and the others with physical addressing [PH11, Chap. 5.4, p. 508].

The cache management must be able to decide quickly, if a requested element is present. To simplify the logic, caches are usually organized with a limited associativity [Hwa93, Sect. 5.2.2]. N -way associative means a cache line can be placed in one of only N associated places in the cache (with N being commonly in the range of 2 to 16). Figure 2.4 displays an example of a two-way associative cache where the elements of the memory can be placed in two different locations of the cache and vice-versa every cache element can store one of four different memory elements.

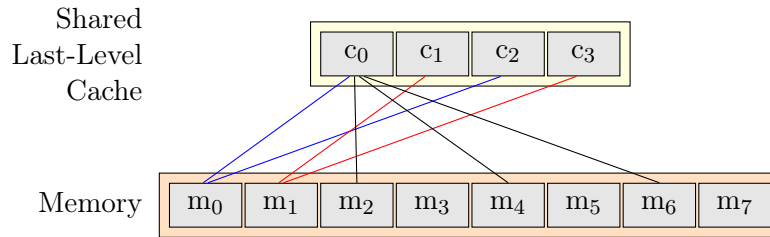


Figure 2.4.: Cache associativity: 2-way.

If all activity on a cache as well as its management strategy are known in detail, the availability of items in the cache is predictable. But since the OS and possibly other processes also use the cache, it becomes increasingly difficult to track what data can be kept in the cache. A deeper understanding of caches can be obtained from the processor documentation [Intel13c; AMD12b] and computer architecture text books [PH11, Chap. 5; HP07, Chap. 5; Hwa93, Chap. 5.2; Dre07].

2.1.5. Interrupts

On most common processors, the current execution can be temporarily suspended by interrupts [PH11, Chap. 4.9]. This mechanism saves the current state of the processor, executes an *interrupt handler* and subsequently restores the previous state to continue the suspended execution. Other than a time lapse, the interrupted program does not notice the interruption.

In General-Purpose Operating System (GPOS), the interrupts are initialized and controlled by the OS itself and can not directly be manipulated by user-mode processes. Interrupts can be used to signal events from peripheral I/O devices (Interrupt Request, IRQ) or to execute code triggered by a clock (periodically or at a given point in time). Software interrupts can be triggered by an executed program to notify the OS. This mechanism can be used to realize the system call interface because user-mode programs are not allowed to call kernel-mode functions directly.

Another source of interrupts are exceptions that are triggered by program or hardware errors, demand paging, and for system maintenance (debugging). Brandenburg et al. group interrupts into four categories: Device Interrupts (DIs), Timer Interrupts (TIs), cycle-stealing interrupts (CSIs), and Inter-Processor Interrupts (IPIs) [BLA09]. In this work, CSIs are called Platform Interrupts (PIs) because they are inherent to the system and the others do also “steal” cycles.

The most important example for exceptions is the page fault, that instructs the OS to provide memory that is currently not present. The handler can provide a new page or swap in a page that was evicted to the hard disk by demand paging. The execution repeats the triggering instruction that should execute successfully once the

2. Fundamental Principles

conditions are fixed. Other exceptions signal illegal actions of the executing program leading to the OS aborting the process.

Processors usually have an Interrupt Flag (IF) that blocks the serving of interrupts. The OS may use it to temporarily delay incoming interrupts during time-critical operations (e. g. hardware control) and to protect shared data structures. Since user-mode processes could interfere with the OS, they are usually not allowed to change this flag.

With long pipelines in the CU, interrupts are difficult to manage. Synchronous interrupts are triggered by an instruction somewhere in the pipeline with other instructions already partly executed. An asynchronous interrupt must be asserted between two distinct instructions. In both cases, preceding instructions must be completely executed (retired) and side-effects of following instructions must be rolled back. This overhead increases with the length and complexity of the execution pipelines. RISC processors have generally a lower interrupt latency. Additionally, the MIPS architecture provides register banks that are used during interrupt handling so that the registers of the running process do not need to be temporarily saved to the stack in memory. In contrast, handling an interrupt on current x86 processors introduces a large overhead.

2.1.6. Input and Output

The third important part of a computer besides processor and memory is the Input/Output (I/O) system. This includes keyboard and monitor of a PC, but also clocks, background storage (i. e. hard disks), network and raw electric signals. The access to I/O devices can generally be realized with special instructions (usually **IN** and **OUT**) to access I/O ports or with registers embedded in the memory address space (“memory-mapped I/O”). In the latter case, control registers are read and written using the usual load and store operations identical to other memory locations (e. g. **MOV**). It depends on the device, which mechanism must be used, what the addresses are and how the bits in the registers are defined. Essentially, all interaction with hardware (peripheral devices) is based on setting and interpreting bits in certain positions. The I/O ports may be connected to a dedicated I/O bus or accessed using the same system bus as memory transfers. Similarly, memory-mapped registers are routed via the memory bus out of the processor and are located somewhere in the chip set or interconnect. The software side does not depend on the implementation, but the performance may be impacted.

The first single processor systems used straight-forward implementations connecting the CPU with memory and I/O. Usually, the address was put on a parallel address bus and the data to or from that address was then transferred on the data bus. This allows to access multiple devices or memory units by addressing them in the more significant bits and thereby extending the address range. The parallel I/O bus Peripheral Component Interconnect (PCI) [[PCI02](#)] was a widely used standard

until it was replaced with its successor PCI Express (PCIe) [PCIe10] that uses serial point-to-point links. Both PCI and PCIe are used for internal wiring of the Integrated Circuits (ICs) that compose the chip set on the main board and also provide standardized slots to install expansion cards.

2.1.7. x86 Architecture

The previous descriptions about important computer system functionalities apply to most modern computer architectures. Since the major part of this work is based on the x86 architecture, the details referenced in later parts are presented in this Section.

Intel introduced the 8086 processor in 1978 [Intel13a, Chap. 2.1] followed by a series of compatible processors (80286, 80386, 80486, Pentium, Core, Xeon) that are referred to as x86 architecture. All new models are downward compatible so that old software (OSs and applications) can still be executed on the most modern x86 computers. Other companies, e. g. AMD, Via, and Cyrix, provide processors of the same architecture capable of running the same OSs and programs (i. e. binary compatible). Based on this processor architecture grew the PC ecosystem with compatible chip sets and peripheral devices (parallel and serial ports, graphics adapters, network interfaces, printers, etc.). Today, the x86 architecture is continued with Intel's Xeon, Core i and Atom processors and AMD's Opteron, Athlon and Geode processors. The universal programmability was so successful, that cost-effective systems became available from small embedded devices up to high-end servers and many programmers are familiar with these systems.

The processors from the leading suppliers Intel [Intel13e] and AMD [AMD12d] are fully documented. This allows to completely understand their working and to easily implement or adapt OS-level code.

“The choice of hardware platform has become less important than it used to be. The distinctions between RISC and CISC processors, between PC's and mainframes, and between simple processors and vector processors are becoming increasingly blurred as the standard PC processors with CISC instruction sets have got RISC cores, vector processing instructions, multiple cores, and a processing speed exceeding that of yesterday's big mainframe computers.” Fog [Fog13b, Sect. 2.1]

While hand-held devices (smartphones, tablets) are increasingly used for web browsing, consuming multimedia content and communication, the PC and notebook class remains important for demanding tasks such as constructing (CAD), graphics, video editing, games, and programming. This architecture even gains new application fields in server and cluster environments where Commercially off-the-Shelf (COTS) systems replace more and more workstations, mainframes and specialized High

2. Fundamental Principles

Performance Computing (HPC) systems. For example, Beowulf [Ste+95] was the first widely noticed cluster of COTS systems using open source software. It established a boom of such systems. Today, the architecture of most HPC systems in the Top 500 list [Meu+] consists of clusters of x86 nodes connected with a high-speed interconnect. There are standards for the software design, but the lack of standardization in hardware implies extra difficulties in implementing on a given processor architecture or porting to a different platform [Fue+12]. COTS systems and especially x86 provide such a standardized platform.

Processors

Current x86 processors still start executing in the legacy 16 Bit mode for downward compatibility, but the major modes are the 32 Bit and 64 Bit protected modes IA-32 (x86_32) and AMD64 (x86_64), resp. Compared to other architectures like ARM and MIPS, the x86 architectures provide relatively few registers (8 multi-purpose registers in 32 Bit mode, 16 in 64 Bit mode) and do not support register banks for low-overhead context switching on interrupts. These design decisions originate from the early implementations and were hard to change without breaking the compatibility. The x86 architecture uses the *Little Endian* byte ordering [Intel13a, Sect. 1.3.1; AMD12a, Sect. 2.2.1] storing the least significant byte of a multi-byte value at the lowest byte address.

The x86 processors support four privilege levels, but most GPOSs only use two of them, the kernel-mode (level 0) and the user-mode (level 3). Besides the address range, that can be configured to be reserved for privileged access, some instructions are reserved for kernel-mode execution but can be configured to be used also in user-mode programs (e. g. the I/O port instructions **IN** and **OUT** and the manipulation of the IF with **CLI** and **STI**). The extension to the inner rings -1 and -2 was introduced to illustrate the higher privilege of hypervisors and the firmware over the OS [WR09].

Instructions of the x86 architecture can be grouped into the following sections [Intel13b; AMD12a]:

- Data transfer and conversion: Move data between registers and memory or registers of different size.
- Arithmetic: Calculation of mathematical and logical operations.
- Logical: Comparing and decision based on binary values.
- Control transfer: Conditional (e. g. based on logic compares) branches and unconditional jumps and function calls.
- I/O: Access to I/O ports (e. g. peripheral devices).
- FPU and SSE: Floating point and vector instructions.
- System: Privileged instructions generally reserved for use by the OS.

System Management Mode

An integral part of Intel [Intel13c] and AMD [AMD12b] processors is the System Management Mode (SMM). This is a privileged execution mode beyond the kernel-mode that usually can neither deactivate nor take influence on what is executed in this mode. Therefore, it is sometimes referenced as privilege level -2 (Fig. 2.1 on page 16) [WR09]. This feature is intended to be used by the firmware (BIOS, UEFI) for tasks such as energy management or security features that should be protected even from the OS. They are implemented as System Management Interrupts (SMIs), the most notable form of PIs on the x86 architecture. The memory where the handler is stored is not accessible from other privilege levels, thus it is beyond control what code is installed by the firmware. This mode has attracted attention from security research [DEG01; BcD08; ESZ08; Duf+09; Wec09; WR09] and recently also in the light of real-time systems [Man09; SO13]. Wallraf has analyzed the effects of the SMIs to real-time systems and how to detect, circumvent or handle its influence [Wal10a].

Interrupts

The x86 architecture supports 256 different interrupts, called *vectors* because each of them may point to its dedicated handling routine. The first 32 vectors are reserved for processor exceptions, the remainder can be used for hardware Interrupt Requests (IRQs), software interrupts and IPIs. The handler functions for each vector are registered in the Interrupt Descriptor Table (IDT) which is located in the main memory and usually organized by the OS but can be manipulated by user-mode programs [kad02] if they manage to access the memory where the IDT is stored.

The functional units controlling interrupt handling in the x86 architecture are summarized as Advanced Programmable Interrupt Controller (APIC). Located in the global chip set, the Input/Output Advanced Programmable Interrupt Controller (I/O APIC) detects signals from peripheral devices and forwards them to the CPU where its Local Advanced Programmable Interrupt Controller (localAPIC) handles the interruption of the running code. In the case of multiple CPUs (Section 2.1.9 on page 32), the I/O APIC allows to load-balance the interrupts between them or to route interrupts to dedicated CPUs. It also allows to trigger IPIs on addressed target CPUs [Intel13c, Sect. 10.4].

The APICs are programmed with memory-mapped I/O. Their physical addresses can be mapped to the virtual address space and then accessed with usual load and store operations. The registers of the localAPIC are documented in the processor manuals [Intel13c, Chap. 10; AMD12b, Chap. 16] and the I/O APIC is documented in the chip set documentation (e. g. south bridge Intel ICH7, Chap. 5.10).

The Flags register provides the Interrupt Flag (IF) that disables interrupt handling when cleared. It does not block exception handling, nor does it influence Non-

2. Fundamental Principles

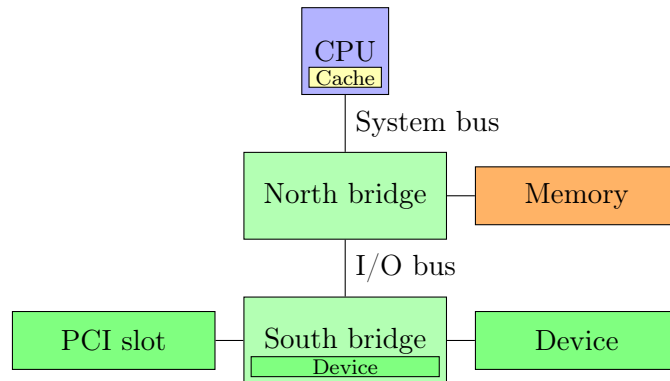


Figure 2.5.: Classical PC I/O architecture.

Maskable Interrupts (NMIs). It is intended to be cleared only for short periods of time by the OS, but the instructions `CLI` (Clear Interrupt Flag) and `STI` (Set Interrupt Flag) can be permitted to selected user-mode processes.

Memory

The x86 architecture supports paging and segmentation, although the latter is mostly deprecated in the 64 Bit mode. The page size is 4 KiB with 2 or up to 4 levels of page tables (in 32 and 64 Bit mode, respectively). The page table of the lower level can be omitted to create *Huge Pages* of 2 or 4 MiB (32/64 Bit mode, resp.) which saves TLB entries for large contiguous memory regions.

Input/Output

The x86 architecture supports both I/O ports and memory-mapped I/O. The I/O ports are accessed with the `IN` and `OUT` instructions for single, double and four Byte transfers. These instructions are usually privileged to the kernel-mode but can be permitted to user-mode processes. The access to memory-mapped I/O is restricted to processes that can access the according physical address range. Similarly, this is usually true only for the kernel-mode but can be established for user-mode processes too by mapping the physical address range into the virtual address space of that process.

The devices in PC systems are classically organized as depicted in Fig. 2.5. The CPU is connected via the system bus to the north bridge [PH11, Chap. 6.5]. That chip hosts the Memory Controller (MC) to physically control the memory chips. Further, the north bridge directs I/O transfers to the I/O bus that connects the south bridge containing some common devices such as Serial and Parallel ports, USB, network interfaces, hard disk controllers etc. The south bridge also provides

Product Name	Micro-architecture Name	Introduction	Pipeline Length
Intel Pentium	P5	1993	10
Intel Pentium III	P6	1999	10
Intel Pentium IV	Netburst	2000	20–31
Intel Core	Yonah	2006	14
Intel Core 2	Merom, Wolfdale	2006	15
Intel Core i7	Nehalem, Westmere	2008	17
Intel 2nd gen. Core	Sandy Bridge	2011	17
Intel 3rd gen. Core	Ivy Bridge	2012	17
Intel 4th gen. Core	Haswell	2013	17
AMD Athlon/Opteron	K8	2003	12
AMD Athlon/Opteron	K10	2007	12
AMD Bulldozer	Bulldozer	2011	12
AMD Piledriver	Piledriver	2012	12

Table 2.1.: List of micro-architectures from Intel [Intel13d] and AMD [AMD05], Pipeline information estimated by Fog [Fog13d]

connections for further devices fixed on the main board and provides standardized connections such as PCI slots to install expansion cards.

With the AMD Opteron and Intel Nehalem generation (ca. 2005), the MC was moved from the north bridge into the processor chip to reduce its communication costs. The north bridge persisted some time to host PCIe connections, but in recent systems (e. g. Intel Sandy Bridge generation, ca. 2011), the PCIe connections are also provided by the CPU and a system can be constructed with a single I/O controller (with the functionality of the former south bridge).

Current Implementations

Some recent examples of x86 processors from Intel and AMD are listed in Table 2.1. Since the marketing names (e. g. Pentium, Core i7, Xeon, Opteron) often obfuscate the implementation details (micro-architecture), the code names of micro-architectures are listed. These names reflect performance-critical features [Fog13d, Table 1.1]. The instruction `CPUID` can be used to obtain detailed information about the executing CPU [Intel08a; AMD10].

The length of the instruction pipeline grew from 10 stages in the Pentium generation (1993) up to more than 30 stages in the latest Pentium IV generation (2005). The simultaneous increase of the processor frequency to above 4 GHz led to an extreme increase in energy consumption which was finally a dead-end for further performance

2. Fundamental Principles

growth. Intel went back to a simpler design with the Core generation (Yonah micro-architecture) with a pipeline reduced to 14 stages and enhanced that to about 17 stages in current processors.

The Core 2 and Nehalem [Fog13d, Chap. 8] micro-architectures introduced considerably faster atomic operations. Since Nehalem, misaligned memory access costs hardly any penalty. Fog has listed many details about the current micro-architectures Sandy and Ivy Bridge [Fog13d, Chap. 9], Haswell [Fog13d, Chap. 10], and AMD Bulldozer and Piledriver [Fog13d, Chap. 14].

In the Intel Nehalem and AMD Bulldozer generations, processors gained dynamic frequency scaling capabilities. A power management unit may change the speed of single Cores/Compute Units. On AMD Bulldozer, the maximum clock speed is reached only after a long sequence of compute-intensive code [Fog13d, Sect. 14.1].

Besides architectural optimizations, multi-core and multi-processor systems became common since approximately 2006 [Dom08, Sect. 1.4]. The specifics of multi-processor systems will be treated in Section 2.1.9 on page 32 further down. Before that, the performance of single-processor systems is presented.

2.1.8. Performance Details: Micro-Benchmarks

On complex processors, the performance depends on the characteristic of an application. Therefore, to estimate the performance, some low-level details must be regarded. With more knowledge about how exactly a processor works and behaves, elementary metrics can be used to form models for a wider range of applications. This allows to decide if a program can be further improved or if it is limited already by a hardware restriction like the memory transfer rate. HPC has revealed a vast knowledge about optimizing algorithms [HW10; Fog13a] and where bottle-necks are (e.g. Memory Wall [WM95], Roofline model [WWP09]). In time-sensitive applications, it is desirable to estimate execution time and the variance for a series of executions. Further, it is beneficial to know which instructions or operations are *fast* or *slow* and to get a relative impression of what could be accomplished in a given time.

This Section presents basic techniques for evaluating small portions of code from basic blocks down to single instructions. The details and provided values are related to the x86 architecture [Pao10], but most methods apply in a similar way to other architectures [PH11, Chap. 1.4]. More details about specific CPUs are presented in the optimization manuals of Intel [Intel13d] and AMD [AMD05].

A benchmark measures the performance. For a computer, low-level or micro-benchmark analyze basic parameters [MS96a] while applications benchmarks measure the execution time for a given task. The latter ones are only relevant for comparable work-loads. Some examples for application benchmarks will be presented in Section 2.1.10 on page 39 and Section 2.3.5 on page 64.

The most important metrics are latency and throughput. The latency is defined as the response time from an event triggering an action up to the handling. Applied to executing tasks, the latency is measured up to the beginning of the handler function, while in network transfers, the latency is the time between the beginning of sending data and the completion of receiving it. The throughput can be defined for instructions executed or bytes transferred per time unit. Usually, the latency is measured for very short instances (single instructions, one byte transfer) and the throughput is measured as average for longer times. This results in the latency being subject to a larger variance. The jitter is defined as difference between minimum and maximum latency. While the minimum resulting from the hardware implementation is generally stable, the maximum latency is often subject to a large variance between test series. In a network transfer, this can be caused by concurrent network transfers, and the instruction latency can be interfered for example by cache eviction or interrupts. Therefore, the observed jitter may differ from the maximum possible real jitter if the maximum latency was not restricted by deactivating/avoiding interfering effects.

Time-Stamp Counter

To be able to measure very short execution times of single instructions, a timing function of low overhead and with only minor variance is required. This is provided by the `RDTSC` instruction that reads the cycle count from a 64 Bit timer register. This time stamp is reset at system initialization and incremented with the CPU frequency. The 64 Bit width guarantees that this counter will not flow over. The documentation [Intel13c, Chap. 17.13] states that this instruction is not serializing, therefore a super-scalar out-of-order processor may reorder instructions resulting in the measured routine may being moved (partly) outside the timed region. This should be avoided by serializing instructions such as memory fences (available with the SSE instruction set) or the `CPUID` instruction.

The cycle counter used by `RDTSC` may depend on the current frequency of the CPU. This results in powered down processors counting slower than those running at full speed. Newer processors tend to provide *invariant* time-stamp counters that always run at the maximum frequency they support. Since the measuring routine is executing CPU-bound code, the processor will run at maximum speed, anyway.

The lowest observed difference of two consecutive `RDTSC` instructions varies between 5 to 100 cycles (Table 2.2). On Intel processors, the latency appears to be influenced by the pipeline length. The serializing memory fence is generally much faster than the `CPUID` instruction and should be preferred if available. Single instructions between two measurement points are covered by the super-scalar execution. In this case, the affected instruction must be repeated multiple times and other effects such as cache misses and loop overhead should be considered [Fog13c].

2. Fundamental Principles

Micro-architecture	RDTSC	MFENCE	CPUID
Intel P III	31		158
Intel P IV	97	244	730
Intel Core 2	80	88	380
Intel Nehalem	24	44	450
Intel Westmere	24	40	640
Intel Sandy Bridge	21	44	228
Intel Ivy Bridge	18	45	225
AMD K8	5	25	65
AMD K10	60	100	119
Intel SCC	13		30
Intel Xeon Phi	6		180

Table 2.2.: Latency (in CPU cycles) of the RDTSC and serializing instructions (where available).

An alternative to the time-stamp counter are the Performance Measurement Counters (PMCs). Those special function registers allow to count specified events with high accuracy and without overhead but are less portable.

Most instructions (especially data transfer and integer arithmetic) can be executed in a single cycle. Due to the super-scalar property, multiple of such instructions can be executed concurrently if they use execution units that are available multiple times. The execution stream is limited by the decoding unit that fills the pipeline and by dependencies between instructions. The branch prediction is highly optimized and can adapt to the code within two to three iterations [Fog13c, Sect. 18.1]. Since pipelines in current processors have 12–22 stages [Fog13c, Sect. 9.6], a branch mis-prediction [Fog13d, Chap. 3] causes a pipeline-flush and costs in the order of 12–50 cycles. In the following, some characteristics of current processor architectures are presented for reference and to rank subsequent results.

Cache and Memory Access Latency

The execution time of a given routine depends largely on the efficient utilization of the caches. The latency of register, different cache level, and main memory access differs by several orders of magnitude. To reliably measure the access latency of different cache levels, a routine with predictable cache utilization must be used. One such micro-benchmark is presented by Saavedra and Smith [SS95], McVoy and Staelin [MS96a, Sect. 6.2], and Babka and Tůma [BT09]. The *Membench* routine traverses an array of size *range* with a step of *stride* which results in a predictable number of cache misses given that the cache size is known and the benchmark is

Micro-architecture Name	Main Memory			
	L1\$	LLC	Sequential	Random
Intel Core 2	4	13	54	271
Intel Nehalem	4	12	21	91
Intel Westmere	4	12	18	123
Intel Sandy Bridge	4	16	23	123
Intel Ivy Bridge	4	12	15	91
AMD K8	6	17	60	136
AMD K10	3	20	49	194

Table 2.3.: Memory access latency in CPU cycles.

the only user of the cache. Repeating this test for a series of ranges and strides allows further to determine the size of the cache line and of different cache levels, the associativity, and the TLB size [YPS05].

The results for some systems of different micro-architectures are presented in Table 2.3. The First Level Cache access lasts in the range of 3–4 cycles on current systems, the Last-Level Cache takes between 12 and 20 cycles. If a L2\$ is available, its access latency lies between those. The main memory access depends on the memory interface and quality of the installed modules and on the access pattern. The sequential access benefits from pre-fetching while a random access pattern yields slower results. Further, the access to main memory is independent from the CPU frequency, therefore, the access latency measured in CPU cycles depends on this frequency. The values in Table 2.3 are taken on CPUs running at 2.3 to 2.6 GHz. This test series shows that the access to variables can vary between 4 and 200 cycles depending on the availability in the caches or main memory.

On 64 Bit systems, the MMU uses four levels of page tables. In case the address resolution of a virtual to the physical address can not be served by the TLB, the table walk requires four memory accesses. If the required tables are not present in the cache, this can result in an additional slowdown of up to 800 cycles.

Notable Instructions

Both Intel and AMD implement their x86 processors with RISC cores and a decoder unit that translates x86 CISC instructions into micro-operations. Table 2.4 lists some exemplary values for recent processors [Fog13e]. Most data transfer and simple arithmetic functions are executed in a few cycles. The division is more complex taking 10 to 40 cycles. The latency is the time it takes to execute a single instruction. The *reciprocal throughput* defined by Fog [Fog13e, p. 3] is the time it takes in average for a chain of similar instructions. For example the reciprocal throughput of 0.33

2. Fundamental Principles

denotes three functional units that allow to complete three such instructions in a single cycle provided they are not functionally dependent (i. e. depending on the result of another). Hence, most simple instructions are executed after few cycles and the super-scalar out-of-order pipeline is able to execute in average two to three instructions concurrently (if well utilized) [HW10, Sect. 1.2.4].

Function and system calls and interrupts are more complex since they need to store some context to be able to return to the instruction following the call. A function call pushes the current IP on the stack and the return from that function uses the topmost stack element to restore the IP. System calls and interrupts usually switch to the kernel-mode and save some more registers. A function call is voluntary, therefore caller and callee can agree on the registers belonging to each of them (this is usually defined in a calling convention). In contrast, interrupts can occur between arbitrary instructions and must therefore save and restore the complete state of the CPU. Table 2.5 lists the measured latencies of calls to an empty function, a software-triggered interrupt and the execution of the `syscall()` function that is usually implemented with the efficient `SYSCALL` or `SYSENTER` instructions. All times include the returning and restoring of the interrupted execution. Most notable is the acceleration of all latencies with newer micro-architectures. In the P III and P IV generation, the system call takes nearly as long as a software interrupt while in newer variants, it is considerably faster.

Input and Output

In terms of I/O capabilities, the throughput is measured most often. For many applications, the effective throughput of a certain device is important. For instance hard disk or network transfer rates are lower than the internal system and I/O buses, therefore they do not saturate them. A range of specialized benchmarks was designed to measure classes of peripheral devices such as main memory (Stream¹, Calibrator²), hard disks (IOzone³, Bonnie++⁴), or network connections (iperf⁵).

The raw access parameters of the internal buses can be measured by reading and writing the configuration registers. The access latency of I/O ports with the `IN` and `OUT` instructions is in the range of 1000 to 5000 cycles. It largely depends where the device is connected to the internal buses and on the device itself. The latency of memory-mapped I/O is in the same order of magnitude but supports the transfer of larger buffers with a better throughput. Compared to simple arithmetic instructions, the I/O on the x86 architecture is very slow. This will be investigated further in

¹Online: <http://www.cs.virginia.edu/stream/> (visited May 6, 2014)

²Online: <http://homepages.cwi.nl/~manegold/Calibrator/> (visited November 12, 2015)

³Online: <http://www.iozone.org> (visited November 12, 2015)

⁴Online: <http://www.coker.com.au/bonnie++/> (visited November 12, 2015)

⁵Online: <https://github.com/esnet/iperf> (visited November 12, 2015)

Micro-architecture Name	Instr.	Nbr. of Micro-Ops	Latency	Recipr. Throughput
Intel Core 2	MOV	1-2	1-3	0.33-1
	ADD	1-2	1-6	0.33-1
	MUL	1-3	3-7	1-4
	DIV	4-7	9-23	?
	JMP	1	0	1-2
Intel Nehalem	MOV	1-2	1-3	0.33-1
	ADD	1-2	1-6	0.33-1
	MUL	1-3	3-5	1-2
	DIV	4-8	19-28	7-17
	JMP	1	0	2
Intel Sandy Bridge	MOV	1-2	1-3	0.5-1
	ADD	1-2	1-6	0.25-1
	MUL	1-4	3-4	1-2
	DIV	9-11	20-28	11-18
	JMP	1	0	2
AMD K8	MOV	1	1	0.33
	ADD	1	1	0.33-0.5
	MUL	1-3	3-5	1-2
	DIV	31-88	15-41	15-41
	JMP	1	0	2
AMD K10	MOV	1	1-8	0.33..0.5
	ADD	1	1-4	1.33-1
	MUL	1-3	3-4	1-2
	DIV	?	15-87	15-87
	JMP	1	0	2

Table 2.4.: Latency of selected x86 instructions (in CPU cycles) [Fog13e].

2. Fundamental Principles

Micro-architecture	Function Call	INT 80	syscall()
Intel P III	38	322	350
Intel P IV	97	1680	1720
Intel Core 2	88	970	450
Intel Nehalem	24	575	200
Intel Westmere	24	630	160
Intel Sandy Bridge	22	460	153
Intel Ivy Bridge	21	700	200
AMD K8	12	630	400
AMD K10	63	500	370

Table 2.5.: Latency of function and system calls (in CPU cycles).

Section 7.6 on page 170. Application benchmarks using common tasks to rate a whole system will be presented in Section 2.1.10 on page 39.

2.1.9. Multi-processor Systems

High Performance Computing (HPC) started to use multiple cooperating processors long ago [Hwa93] to obtain results faster and to increase the manageable problem size or resolution. Instead of engineering or buying a faster CPU, computer systems with multiple CPUs were constructed. With scaling algorithms, this allowed a nearly linear performance increase. Around 2006, Intel and AMD started to promote multi-core processors for commodity PCs [Dom08, Sect. 1.4]. The optimizations of processors led to overly complex designs and the power consumption outgrew the benefits. To gain such a performance increase with architectural improvements, a doubling of the frequency (costing four times the power consumption, quadratic increase) or extremely more complex instruction-level improvements (that only yield a few percent more performance) would be necessary [MS13, Sect. 2.1, p. 336]. By integrating two simpler processors, the power consumption was (only) doubled but also the performance could reach a factor of two. (linear scaling). Some new technologies can be exploited by optimizing compilers, for example automatic vectorization to use the SSE instructions. However, the new architecture introduces concurrency with new challenges, or, as Sutter expressed: “The free lunch is over” [Sut05]. This refers to single-threaded programs that do no longer benefit automatically from faster processors but that must be parallelized to exploit the new multi-processor systems [Gee07]. And parallelizing programs efficiently is not trivial since errors in the synchronization yields the risk of wrong results and bad performance [Moy13c; McK14b; Lue12].

Today, mobile systems (notebooks, smartphones and tablet computers) use two to four processors, desktop systems provide four to eight processors, and typical nodes of high-performance computers integrate 16 to 20 processors [HW10]. The *New Moore's Law* predicts a doubling of cores per chip every two years [Vaj11, p. 3]. This leads to *Many-Core* processors, loosely defined as chips with more than tens of cores or, more timelessly defined, substantially more than commonly or currently available). This development poses new challenges in avoiding bottlenecks where too many cores share the memory interface.

Flynn classified computer system architectures depending on the number of execution contexts and data partitioning [Fly72]: Single Instruction Single Data (SISD) stands for single-processor systems. Single Instruction Multiple Data (SIMD) denotes systems where a single program instruction processes multiple data items, e. g. elements of a vector. This technique is also called *vectorization*. The classification Multiple Instructions Multiple Data (MIMD) stands for multi-processor systems executing multiple programs concurrently (including shared memory multi-processor and distributed systems, also called clusters) [PH11, Sect. 7.6].

Today's HPC systems can not easily be classified with these terms although these concepts invented in the 1960s are still widely applied, but combined. Beginning with the Pentium MMX (1993), SIMD instructions for vector processing were integrated in the x86 architecture. Today, the successors SSE and AVX process up to eight single-precision or four double-precision floating-point operations with a single instruction. With the Pentium IV, Intel introduced *Hyper-Threading*, a technique for Simultaneous Multi-Threading (SMT) that presents two processors to the software [Vaj11, Sect. 2.2.1.7]. But those two CPUs are not fully featured but the logical processors share some resources of the physical one gaining between 140 and 160 % of the performance of a single processor. AMD processors may integrate two "Cores" into a "Compute Unit" [AMD13, Sect. 2.1] that share some resources in a similar way. To software, the logical processors can hardly be differentiated from two physical ones. Subsequently, chips with multiple independent processors integrated in the same package (Multi-core processors) became available.

A more empirical classification oriented to the software architecture is the division into *data parallelism* and *functional parallelism* [Moy13c]. Data parallelism denotes multiple similar tasks working with different data, also called Single Program Multiple Data (SPMD) in the style of Flynn's classification. In functional parallelism, also called *Master-Worker* or Multiple Programs Multiple Data (MPMD), the tasks do different things and often forward results to subsequent processing (data streaming). This can be realized on every architecture, but has no advantage if only a single instance of data is processed. It benefits from a pipelining effect when the first stage of the next instance can overlap the processing of subsequent stages.

2. Fundamental Principles

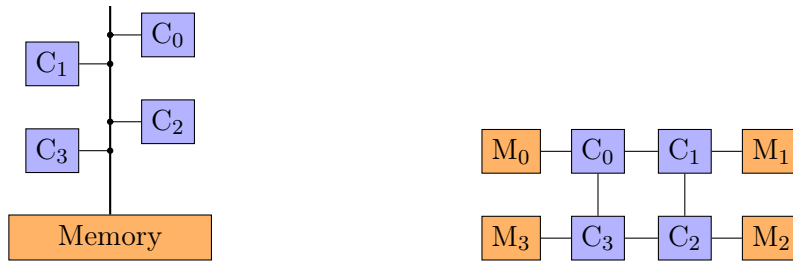
To the software, it does not matter if multiple processors are integrated on a single chip (usually called “Multi-Core”), on different chips (in the same package or installed in different sockets), or if they are logical processors (Simultaneous Multi-Threading, SMT). The OS lists them all as equal processors. Although for optimization reasons, the layout and interdependencies are important to detect bottlenecks. For code that is bound by memory transfer, it is sensible to use fewer CPUs than available to avoid contending the system bus. Compute-bound code can benefit from using only one logical processor per core. Other optimization methods regard the physical dependencies between processors and physical units, such as the distance (latency, throughput) to memory, and place threads according.

In this work, *CPU* and *processor* synonymously denote a Von Neuman processor visible to software independently if it is a logical processor (SMT, Hyperthread) or a physical one. If a distinction is required, *logical* and *physical* denote this characteristic. A *chip* is a single piece of silicon carrying one or more physical processors (that may provide multiple logical processors each). A *package* contains a single or multiple chips and is installed in a *socket* on the main board. A *multi-core* integrates multiple processors in a single package, while for a *multi-processor* system, it does not matter if the processors are integrated on the same chip, are logical processors, or are installed in multiple sockets.

Memory Architecture

Although from a programmer’s perspective, the memory architecture is transparent, the optimization for speed or timing predictability requires knowledge of the layout and consideration of the behavior. The first real multi-core processors shared the system bus and had an equal access to the main memory called Uniform Memory-Access (UMA) (Fig. 2.6a). But this architecture is subject to contention on the shared system bus. Systems with more processors avoid this bottleneck by placing multiple memory regions near to the individual processors resulting in faster access to near memory. These nodes consist either of a single processor or a multi-core processor with integrated Memory Controller (MC) each with local memory. Non-Uniform Memory-Access (NUMA) systems (Fig. 2.6b) can still run the same programs because all the memory is addressed in a linear manner. No Remote Memory-Access (NoRMA) systems are distributed systems where, multiple systems cooperate via network and can not directly access other node’s main memory. This type of systems requires special programming to exchange messages (Message-Passing, MP).

The current generation of x86 processors by Intel and AMD supports the NUMA architecture. Depending on the target group, this can be a single NUMA node per package, thus only show effect on systems with multiple sockets, or carry multiple Memory Controllers (MCs) in a package. Figure 2.7 shows an example system with one Interconnect Controller (ICC), one MC and two CPUs per node. The processors can access the local MC directly. To access the other memory regions, a request



(a) Uniform Memory-Access (UMA) (b) Non-Uniform Memory-Access (NUMA)

Figure 2.6.: Comparison of simplified UMA and NUMA architectures.

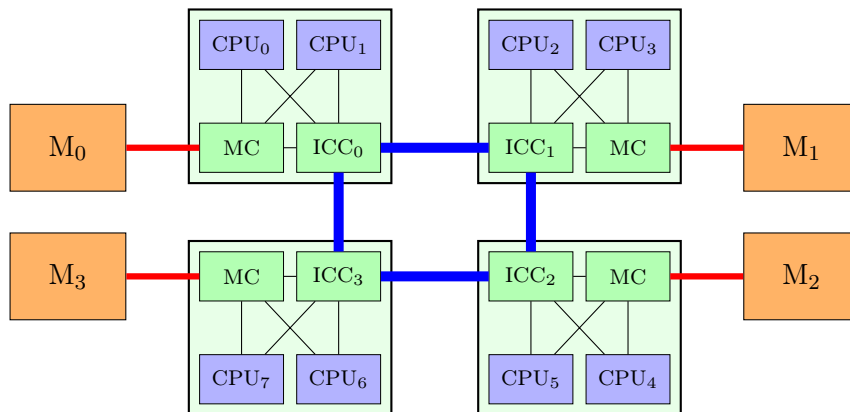


Figure 2.7.: Multi-processor architecture with NUMA characteristic.

2. Fundamental Principles

must be transmitted over the interconnect to a different MC. In Fig. 2.7, node 0 can access the memory of nodes 1 and 3 over one network link while the access to node 2 requires two links. The access to local memory has always the lowest latency. The remote accesses are slower and may be all similar or depending on the network distance. For a remote access, the probability of contention (lower throughput, larger outliers in latency) increases with the number of links (distance). Even more complex NUMA characteristics are possible, e.g. multiple systems as displayed in Fig. 2.7 could be connected by a different (external) interconnect.

In single-processor systems, the software developer can apply a simple model of sequence and dependencies to understand what data is written when and where and what a read access will see. The processor optimizations of ILP and out-of-order execution may reorder instructions, but they are still bound to functional dependencies (read-after-write, etc.). Additionally, the compiler may optimize a high-level language and emit machine instructions in a different order since the compiler can only regard the current code and does not know about concurrent access. In multi-processor systems, other execution threads may concurrently access the same variable and modifications may become visible to other processors in a different sequence than suggested by the instructions or programming language. Various consistency models exist [AG96; Des13a, Sect. 3.2.3, p. 138] to describe the hardware behavior of different architectures and to help programmers verify their code. If the sequential visibility of memory transfers is crucial and some of those transfers use memory shared with concurrent execution, the consistency must be instructed to both the compiler and the processor. Compilers usually offer synchronizing instructions that are regarded by its optimization and emit memory fence instructions to the processor. A load or store fence guarantees, that all previous reads or writes (resp.) are committed and globally visible, before any subsequent instruction is executed. A memory fence is the combination of load and store fence. The x86 architecture provides the serializing assembly instructions **SFENCE** (store), **LFENCE** (load), and **MFENCE** (memory). The Gnu Compiler Collection (GCC) provides the intrinsic function `__sync_synchronize()`⁶ and the C11 standard [C11, Sect. 7.17.4.1] introduces `atomic_thread_fence()` for the same purpose.

Cache Hierarchies

The cache hierarchy used in single-processor systems (Section 2.1.4) introduced additional challenges when extended to multi-processor systems [Hwa93, Chap. 7.2]. Combining multiple processors to a shared memory system offers two options, *private* and *shared* caches. If every processor has its own stack of private caches, a lot

⁶Online:

https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/_005f_005fsync-Builtins.html
(visited November 12, 2015)

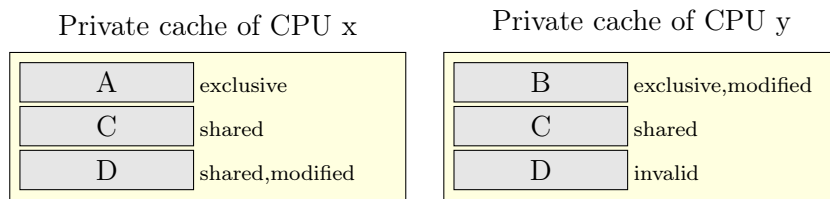


Figure 2.8.: Examples for coherent cache lines.

of data will probably be present in several of them. This effect is reduced by implementing shared caches at a lower level. The existence of multiple caches introduces the *coherence* problem because the private copies in all caches must be held in a synchronized state. Whenever a processor changes a value in its private cache that is present in other private caches, these copies must be either updated or invalidated. The x86 architecture provides cache coherence in hardware, so that the programmer does not have to care for this function-wise.⁷ This hardware-management of cache coherence uses states attached to each cache line. The most common protocol is MESI using the states *modified*, *exclusive*, *shared*, and *invalid* [HP07, Chap. 4.2]. Current x86 processors use that [Intel13c, Chap. 11.4] or a variant (e.g. MOESI [AMD12b, Chap. 7.3], MESIF [Intel09, p. 15]).

In Figure 2.8, some examples are given for the management of cache coherence. Cache line A is exclusive to CPU x and could be dropped without further action. CPU y has line B also exclusively, but it is marked as modified, so it must be stored back before evicting it. Cache lines C and D are shared between both. But while C is the same for both, D was modified by CPU x, rendering it invalid for CPU y which must reload the data before the next access.

Generally, the synchronization of caches can be organized by *bus snooping* (i.e. listening on the global system bus, what other CPUs are requesting and storing) or message passing protocols. While bus snooping is easily possible in UMA systems (Fig. 2.9), NUMA systems must provide a broadcast of memory operations to support those protocols. If snooping the transfers of all processors is too inefficient, directory-based protocols can be used [HP07, p. 208].

If multiple processors alternately modify the same variable, the cache line containing this memory location oscillates between their private caches. The same is true for two variables that are placed nearby sharing the same cache line. In this case, it may happen that two processors each write to their own, distinct variable but in effect to the same cache line with drastically reduced performance which is called *false sharing*. To avoid this, data structures used exclusively by concurrent tasks should be placed at least a cache line away from each other.

⁷However, the performance usually benefits from regarding the cache layout.

2. Fundamental Principles

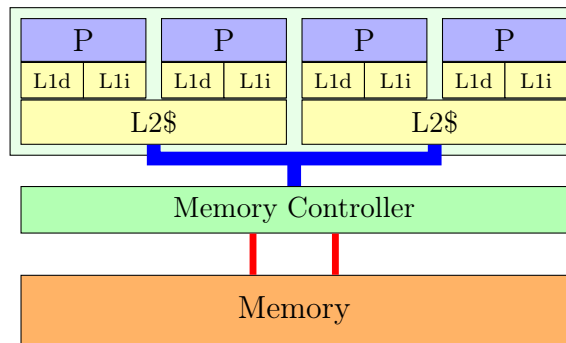


Figure 2.9.: Processor

Input and Output

The data transfers to peripheral devices are implemented similarly to single-processor systems. The I/O bus is either a dedicated one or uses the same interconnects as memory transfers. Depending on that layout, its performance may interfere with the I/O and memory transfers of other processors. Since peripheral devices are not aware of multiple processors, software should ensure a reservation of devices to avoid races.

Multi-Processor Synchronization

Multiple processes on a single-processor system are executed alternating in time-slices (pseudo-parallel). In this case, blocking all interrupts avoids interfering with other processes because the context switch is initiated by a timer interrupt. In multi-processor systems, processes on different CPUs can in fact be executed concurrently and other means of synchronization are required. Therefore, processors support special atomic operations that guarantee inseparable execution. Reading and writing Integer sized variables is usually atomic, i. e. all bytes are read together so that a modification of one of them during loading is not possible.

For more complex synchronization mechanisms, atomic operations [Sta09, Table 5.1] are supported that are guaranteed to be completed uninterrupted (by other processor's execution and by interrupts). Recent x86 multi-processor CPUs provide the **LOCK** prefix to execute a following instruction atomically. It can be used with a wide range of instructions, such as **ADD** (addition), **BTS** (bit Test-and-Set, TAS; Conditionally sets a bit), **XCHG** (Exchange, XCHG; Exchanges the content of two memory variables), and **CMPXCHG** (Compare-and-Swap, CAS; Exchanges conditionally) [Intel13b; AMD12c].

Two other important atomic operations used in many algorithms and provided by other architectures are FAD and LL/SC. Fetch-and-Add (FAD) loads a variable into a register while adding another. The Load-Linked/Store-Conditional (LL/SC) operation is the only atomic operation on ARM. It loads a memory variable in

a linked state and later stores it only if the source was not touched by other processors since loading. The conditional store can fail and the algorithm must handle this by aborting or retrying. The newest generation of Intel processors (micro-architecture Haswell, 2013) introduced *transactional memory* that can be used for multi-processor synchronization [Intel13d, Chap. 12]. It is a generalization of LL/SC [Vaj11, Sect. 5.4.5]. A transaction is a group of load and store operations, that are either executed completely (and uninterrupted) or rolled back. In the latter case, the implementation can decide to retry or to abort and use other means of synchronization. On large machines with many processors, the synchronization can alternatively use special instructions using I/O devices or dedicated interconnects.

All these instructions can be used in programs to realize synchronized access to resources. Instead of implementing these mechanisms repeatedly, they are usually provided by the OSs (Section 2.2.1 on page 44).

A different method for multi-processor synchronization is the use of Inter-Processor Interrupts (IPIs). Similar to a software interrupt, they allow to trigger an interrupt, but on a different CPU. On the x86 architecture, they are realized in the localAPIC that has configuration registers to issue such interrupts. With this mechanism, an interrupt can originate from hardware (peripheral devices, IRQ), software (voluntarily called locally with `INT`), an exception (involuntarily triggered by the local code), or a different processor.

2.1.10. System Performance: Application Benchmarks

For the detailed understanding of a system's behavior, elementary metrics (Section 2.1.8) are very helpful. Further, they help to rate the performance and grade of optimization of a given application. But for the evaluation of applications, an exact model of their working is required (e. g. the *Roofline* Model [Wil08; HW10; PH11]). For many applications, this effort is undesirable. Instead, the performance of a given application can be measured and compared to modifications or to the execution on other systems. This includes *profiling* to detect where an application spends most of its execution time [Fog13b, Chap. 16].

Several collections of application benchmarks for the single-processor performance are available that measure typical algorithms. Classic benchmarks for floating-point (Whetstone [CW76]) and integer arithmetic throughput (Dhrystone [Wei84]) measure the respective operations per Second. The Standard Performance Evaluation Corporation (SPEC)⁸ provides various benchmarks, e. g. CPU2006 to measure the processor performance [Dom08, Sect. 3.5.1]. The Embedded Microprocessor Benchmark Consortium (EEMBC)⁹ Suites target embedded systems [Dom08, Sect. 3.5.1].

⁸Online: <http://www.spec.org> (visited November 12, 2015)

⁹Online: <http://www.eembc.org> (visited November 12, 2015)

2. Fundamental Principles

The BDTI Benchmark Suites¹⁰ analyze digital signal processing performance [Dom08, Sect. 3.5.1].

The macroscopic analysis can use different clocks for its timing: OS functions offer a varying resolution from Seconds down to Nanoseconds or the CPU's time-stamp counter (Sect. 2.1.8). It is reasonable to take the time for a function repeatedly: The first iteration will be the slowest, because the caches are cold [Pao10]. Subsequent executions will be faster depending on how much of its data can be held in the caches. To reduce the probability of a task switch distorting the results, the process priority can be increased.

On multi-processor systems, the execution time may depend on the assignment of processes to CPUs, the placement of memory buffers (esp. on NUMA systems), the cache utilization, and the load balancing (how often processes move to other CPUs) [Kle04]. All this should be considered when setting up experiments to evaluate the performance of applications. Notable benchmarks for overall system performance on HPC systems are Linpack [Don88] and the NAS Parallel Benchmarks [Bai+91] that cover a range of typical algorithms found in many real applications. Further, there exist specialized benchmarks to evaluate the power usage or graphics processing performance.

2.1.11. Embedded Systems

Embedded computer systems are defined as computers that are part of a larger system [Sta09, Chap. 13] and that have a dedicated function that is started automatically on system initialization [Moy13a]. In contrast to general-purpose computers (e. g. PCs, Laptops), they usually have limited interactive components but interact closely with their (physical) system. Examples are the control computers of washing machines, electronic music keyboards, control units of internal combustion machines, digital controllers for factory production processes, and medical devices. According to some opinions, a web server, data base or a render farm for an animated movie can also be regarded as embedded systems since they serve a specific purpose and are carefully designed and optimized for their task [Moy13a]. Generally, the lines are blurry: While an electronic book (eBook reader) is a device dedicated solely to reading texts and thus is clearly an embedded system, a smartphone or tablet computer has a similar form (large screen, few keys, touch sensitive screen) but allows installing applications (called "Apps" in this context) and may also be classified as general-purpose computer.

The function of an *embedded system* may often also be provided by electronics, but in the context of this work, it refers to the combination of computer hardware and software. An embedded systems usually start its predefined program after

¹⁰Online: <http://www.bdti.com/Services/Benchmarks> (visited November 12, 2015)

being switched on. In contrast, general purpose computers usually start applications interactively on user request.

An embedded system has often special input and output means. Instead of keyboard and monitor, they usually employ sensors, actors, limited displays (e. g. LEDs) and buttons or touch-screens. They often interact with real-world processes and thus, they are often also real-time systems. But both terms are not congruent. Special elements of embedded systems usually not present in PCs:

- Micro-Controllers (μ Cs) are ICs that combine a CPU with internal memory and peripheral devices for I/O capabilities. A μ C enables the construction of cost-effective computer systems with very few additional components while general-purpose CPUs must be installed on a complex main board providing special voltage supply, memory, chip set, and many more.
- Co-processors (e. g. Digital Signal Processor, DSP; Field-Programmable Gate Array, FPGA; Application-Specific Integrated Circuit, ASIC) can be added to calculate special functions more efficiently than a small processor. They are less flexible but more powerful and energy-efficient.
- Input/Output: General-Purpose I/O (GPIO) provides digital interfaces, Analog to Digital Converters (ADCs) and Digital to Analog Converters (DACs) allow the interaction with variable voltage. These units are summarized as Data Acquisition (DAQ).
- Read-Only Memory (ROM) for a fixed software (firmware) provided by the producer instead of allowing a user to install programs on a hard-disk.

Embedded systems range from very limited capability in processing power and memory (e. g. coffee maker, airbag control), up to very sophisticated machine controls (e. g. airplanes, power plants, video systems). They are specifically designed and evaluated (tested). Updates are similarly tested and usually only provided to fix existing errors. On most embedded systems, no other software can be installed by a user. Very small systems can waive memory protection as only verified tasks are executed. However, there exist also embedded systems being very sensitive to security threats: Smartphones allow the installation of “Apps” while expected to protect user data. Network equipment is subjected to hostile traffic engineered to attack vulnerabilities. Many embedded systems must endure extreme environmental conditions:

- Dust and water in industry applications,
- Extreme temperatures in automotive systems,
- Limited energy in portable devices,
- Severely optimized for costs in consumer products,
- Superb reliability in life-critical systems (e. g. airplanes, medical devices),

2. Fundamental Principles

- Very long lifetime (cars are expected to operate for decades compared to PCs and smartphones which are replaced after a few years),
- Difficult accessibility for service (e. g. satellites).

Therefore, embedded systems can be classified according to different aspects [ISO06, Chap. 4.1]. Some examples are: *scale* (small, medium, large), *timing* (real-time, non-real-time, how predictable the execution must be to correctly interact with real-world processes), or *criticality* (consequence of failure, e. g. toy car vs. airplane).

2.2. Operating Systems

An operating system (OS) is a software that acts as an *intermediary between user and hardware* [SG94] also called *user/computer interface* or *resource manager*. The main objectives are *convenience*, *efficiency*, and *ability to evolve* [Sta09, Chap. 2.1]. It acts as government, resource allocator, and control program. It governs the users and processes supervising their privileges and guarding their integrity, it allocates resources avoiding conflicts of concurrent usage and exhaustion, and it controls complying behavior allowing to prevent or detect malpractice.

Operating systems are available in a wide range of types. General-Purpose Operating Systems (GPOSs) are known from PCs: They are installed on a persistent storage (e. g. a hard disk), started first, present an interface to the user and control the applications. In contrast, the smallest scale embedded OSs are just a set of helper functions serving the same purpose (control, resource manager) but they are integrated with the user functions to a monolithic firmware. Embedded systems (Section 2.1.11) may either use a GPOS that is usually configured in a way that only the required tasks and services run, or a special embedded OS.

A GPOS is usually installed on the bare hardware, and consists of the kernel, libraries and tools. The *kernel* is the supervisor running with highest privileges. *Libraries* are collections of functions that can be used by programs. *Tools* are programs accompanying the OS to provide essential services, for instance compile code to object files (Assembler, Compiler), handle object and executable files (Linker), load a program to execute (Loader), manage users and their privileges, and a variety of background services such as printer spooler, network servers, and other *daemons*.

This Section introduces concepts implemented in most common GPOSs. There exist experimental OSs that demonstrate radically new ways, but as this work targets common systems, only conventional methods are regarded. Unless otherwise stated, the information in this Section is based on the Operating System textbooks from Tanenbaum [Tan07], Stallings [Sta09], and Silberschatz and Galvin [SG94].

2.2.1. General Concepts

Multi-tasking OSs support the execution of multiple execution contexts that are separated from each other. These tasks all have a notion of exclusive execution, but are executed alternatively in time slices. Depending on the hardware capabilities, the OS supports processes that have their own, unique, protected address space and/or threads that share the address space. Typical GPOSS support both and multiple threads of a single process are separated from other processes (and their threads). Here, the term *task* is used for an execution context (process or thread) if the address space does not matter.

The architecture of an OS is divided into the classes *monolithic* and *micro-kernel*. While the Monolithic OS is a single program with a unique address space (the kernel-space), in a micro-kernel, only the most basic services are implemented in the actual OS while all other functions are provided by dedicated server processes with reduced privileges and protected from each other [Tan07, Chap. 1.7]. The micro-kernel architecture is considered cleaner and more secure but their performance is more challenging due to Inter-Process Communication (IPC) being required where the monolithic kernel can use function and system calls [Sta09, Chap. 4.3]. In an embedded OS, tasks can be static, that means they are created at system initialization time and live as long as the system is executed (although they can be suspended and reactivated). More complex OSs and GPOSS support the dynamic creation of tasks. Usually, they inherit most properties from their parents that initialized them.

Each task owns some resources (independently if they are protected from access by other tasks). The *code section* holds the instructions to execute. The code is loaded together with initialized data in the *data section* when a new task is created. The *stack* is a First-In, First-Out (FIFO) data structure for temporal storage of data used for function calls and local variables. The *heap* is a region of dynamically allocated memory for the storage of data with beforehand unknown size. On hardware with memory protection (e.g. a MMU), each process is provided with its own address space. These *virtual addresses* are translated to *physical addresses* in hardware (see also Fig. 2.2). This allows to execute programs that do not need to know in advance, where their allocated memory lies but can rely on always using the same virtual addresses.

Scheduling

Multi-tasking OSs commonly manage states of their tasks. Those states often include *ready*, *running*, *blocked*, and *finished*. New tasks are created with the state *ready*. When they are executed, their state changes to *running* while the interrupted task is set back to *ready*. All tasks in the state *ready* are waiting to be executed again. They can be managed in a *ready queue* or by more complex data structures. If a running task has to wait for the availability of a resource or decides to sleep, its state

2. Fundamental Principles

is changed to *blocked*. After the time-out or when the resource becomes available, the blocked task changes to *ready* and becomes electable for execution, again. When a task ends, its state becomes *finished*. This allows to keep the data structures for evaluation before deleting all traces of that task.

The temporary suspension of a task and the activation of another one is called task switch. A *preemptive* OS assigns time slices to tasks and is able to interrupt them whenever a time slice ends or a more important action is required. To the tasks, the preemption is transparent. They are temporarily suspended by an interrupt and their state is fully preserved until they are continued. The opposing technique is *cooperative* scheduling where tasks are only replaced by another task when they call system functions or voluntarily return the control of their CPU.

The scheduling algorithm selects a new task from the ready queue containing the tasks in the state *ready*. A very simple concept is *round-robin* where just the next task in the list of all tasks is selected. Different optimizations aim for various purposes, *fair* scheduling improves the utilization under differing workloads (e. g. short running latency-sensitive tasks vs. compute-heavy number-crunching), and *priority* scheduling allows to specify which tasks are more important than others. Common goals of scheduling algorithms include system throughput, interactivity, and power awareness.

A task waiting for something (time, resource becoming available, notification by another task) can do so by polling the state repeatedly (spinning, busy waiting). This allows a prompt reaction to a state change but wastes execution time. The alternative is setting the own state to *blocked* so that other tasks can execute. In this case, the OS can wake up the blocked task or another task can do so. For example the *sleep* function call temporarily blocks the current task with a time-out after which it is reactivated. A hybrid approach is to poll with short periods of sleeping in between. This releases the CPU to other tasks while still controlling the frequency of checking the state and it can be realized for actions that are not supported directly by the OS.

Synchronization

In a multi-tasking environment, tasks can be executed concurrently. Independently if they are executed alternatively in fine-grained time-slides (pseudo-parallel) or on a multi-processor system, sharing resources may cause races. Since all arithmetic operations are done in the CPU registers, a common proceeding is to load the content of a variable into a processor register, change its value (e. g. by adding a constant) and store the result back into main memory. If this action is interrupted by another task that changes (e. g. increments) the same variable, the storing task will overwrite the modification of the other. Such a region of code modifying a shared resource is a *critical section*.

Simple modifications of shared variables can be solved with atomic instructions (Section 2.1.9). If the critical section contains more complex operations (e.g. to change multiple fields of a structure that must be visible to others only completely because single changes are inconsistent), it must be protected from concurrent access. On single-processor systems, the temporary deactivation of interrupts inhibits task switches. On multi-processor systems, mutual exclusion [Dij65] is required, that allows only one task at a time to enter the critical section. This can be realized by checking a shared object (e.g. a Mutex) before entering and, if needed, waiting until the section becomes free. After leaving the section, the Mutex is notified that the section is free so that a waiting task can get the Mutex and proceed into the critical section.

A more complex synchronization object is the Semaphore [Dij68], a shared variable that counts the number of free slots. The test to enter is successful, if the value is above zero. This check must be done atomically with reducing the value on success. A Semaphore with only one slot (binary Semaphore) is a Mutex. Many classical synchronization problems like *Dining Philosophers* and the *Barbershop Problem* [Dow08] can be solved with Semaphores. These synchronization objects can be constructed from atomic instructions like Compare-and-Swap (CAS) [Sta09, Sect. 5.2], Exchange (XCHG) [SG94, Sect. 6.3], Test-and-Set (TAS) [Tan07, Sect. 2.3.3], Fetch-and-Add (FAD), or Load-Linked/Store-Conditional (LL/SC) [Her91] depending on their availability on the applied processor.

These implementations wait actively (*spin-locks*). This is a disadvantage because longer critical sections waste execution time of the waiting tasks. It would be more efficient to suspend a waiting task (state *blocked*) and switch to a different task that is able to progress. Further, if a task holding a Mutex is interrupted, the waiting time of others increases or even may result in deadlocks. The alternative are blocking synchronization objects. Since they influence the state of tasks, they can only be provided by the OS. Some implementation-dependent properties of Mutexes and Semaphore are:

- Acquire and release functions with timeout,
- Assignment to waiters: queue, priority, random,
- Waiting: active (spin-lock) or passive (blocking and wake-up by OS), and
- Ownership: release only by requirer or arbitrary.

If multiple participants access shared Mutexes, the risk of a deadlock becomes evident [Tan07, Chap. 6]. Four conditions are necessary for a deadlock, mutual exclusion, hold and wait, no preemption, and circular wait [Sta09, Chap. 6]. Operating systems can decide to handle this situation differently. *Deadlock prevention* is a system design that prevents the situation by breaking at least one of the four conditions [Tan07, Sect. 6.6], *avoidance* checks every request for its risk [Tan07, Sect. 6.5], and *detection* inspects the running system to resolve occurred deadlocks in hindsight. However, most current GPOSs just ignore deadlocks.

2. Fundamental Principles

The presented methods are based on the agreement, that every access to a shared resource will be protected by the programmer. It is still possible to ignore the convention. A different approach is to use higher level methods like *monitors*. This is an object with integrated shared storage, but the access is only possible using proposed functions. These functions inherently protect the concurrent access. Alternatively, the class of Message-Passing (MP) functions can be used. This avoids shared variables and only communicates with standardized send and receive functions that implicitly synchronize. Inter-Process Communication (IPC) are methods to pass messages between processes available in most OSs. The communication is classified as:

Asynchronous (no-wait) Send and immediately proceed.

Synchronous Proceed only after the message has been received.

Remote invocation Proceed only after a reply has been received.

The *asynchronous* communication can be used to construct the others and is therefore the most flexible type [BW01, Sect. 9.1]. On single-processor systems, it depends on the scheduling policy, which task (sender or receiver) executes first. On multi-processor systems with real concurrency, asynchronous communication becomes feasible and overlapping pipelines stages are fast and efficient.

Time

Most OSs offer their tasks the service of time keeping. This includes functions to ask for the current time (Wall-clock time), measure time differences, install timers to wake up or get signaled at a specified time, and to frequently repeat actions.

2.2.2. Hardware Abstraction

Hardware abstraction makes programs more portable to transfer them to different systems. Without, software must detect the peripheral devices and know exactly how to control each of them. To avoid reimplementing these details and to make a wider range of hardware available, OSs offer unified interfaces that abstract the implementation details. Further, the hardware abstraction allows to protect data from users. For instance a file system abstracts from the storage details on the hard disk and introduces file ownership and access rights so that one user can not delete other user's data or modify the OS without permission.

A basic abstraction is the task scheduling (Section 2.2.1) that creates the notion of a dedicated CPU for every task. On modern processors with multiple privilege levels (Section 2.1.2), processes are protected from each other. The most commonly used privilege levels are the kernel-mode with a CPL of 0 and the user-mode with a larger CPL (e.g. 3). Consequently, kernel-mode code is allowed to access all regions of memory while user-mode code may access only its own user-space memory. The memory protected from user-mode processes is the kernel-space. Classically, an

operating system executed in ring 0 controls the hardware and offers system calls to applications running in user-mode. The kernel can copy data to and from the application's memory, but those applications can only use the offered interfaces. A hypervisor [Tan07, Sect. 1.7.5] can be classified as executing in ring -1 (Fig. 2.1) because it controls its guest OSs. This can be transparent to the guest OS that sees no difference to be executed on real hardware.

The *memory management* is an important mission of OSs. To processes and threads, a linear virtual address space is presented. The physical page frames are managed by the OS with the processor features *paging* and *segmentation*. The OS initializes and controls the Memory Management Unit (MMU). The available main memory can be extended with *demand paging*, the storing of temporarily not used memory regions to background storage (e.g. a hard disk). Since the *swapping* is time-consuming, it is crucial to apply suitable algorithms to select which regions shall be displaced.

The most important part of hardware abstraction are *device drivers* that can be modularly be added to the OS to extend its functionality. If a new peripheral device is released, an according device driver allows to instruct existing OSs how to control this device without the need to install a new OS. In this way, existing software can also make use of new devices by adhering to the same abstracting interfaces. However, the details how a device is controlled are often badly documented and only a driver provided by the vendor exploits the full functionality or performance.

Many devices are accessed as part of a layered architecture where devices are grouped into classes and abstracted by the OS. For example hard disks are not accessed on the bit level, but a *file system* offers the abstraction of files with names and directories independently if the data is stored on hard disk, USB stick, or a network drive. In the latter case, a network adapter is used transparently without the user-mode software needing to adapt. Another layered software architecture is the network stack that abstracts from the physical transport layer.

2.2.3. Programming Languages and Application Programming Interfaces

Software is usually written in a programming language and *compiled* into machine code. High-level programming languages abstract from the machine code and compilers allow to optimize the generated instructions. Optimization targets are time to execute, code size [Fog13b, Chap. 4], data size, or time and/or space of edit/compile/link process [ISO06]. If utmost optimization is required, different compilers should be tried and the recommendations given in optimization manuals [Intel13d; AMD05] and textbooks [Fog13a; HW10] should be followed. For very performance-sensible tasks, parts of programs can also be implemented in Assembly language that can be converted instruction for instruction to machine code. The

2. Fundamental Principles

machine code is closely tied to an architecture. Depending on the task, a suitable language should be selected [Fog13b, Sect. 2.4]. Common programming languages are [Jai13]:

Assembly: Low-level machine language, complex to use but offers direct access to all features. Tied to an architecture.

C: Mid-level language for hardware interaction, operating systems implementation, and performance critical applications.

C++: Object-oriented high-level general-purpose language with good performance.

Java: Very portable general-purpose language with sufficient performance.

Python: Scripting language for quick modifications and easy maintenance.

Ada: Real-time language for safety-critical embedded systems.

Besides the OS interfaces, Application Programming Interfaces (APIs) offer the extension of functionality and ease development [Dom13]. Different OSs that support the same API allow to use the same programs on them (portability). Notable interfaces are the C library that is part of the programming language's standard, or the UNIX standards POSIX [POSIX08] and System V [SysV95a; SysV95b; SysV95c]. The MCAPI [MCAPI11; Wal10b] is a communication standard for shared memory multi-processor systems. The Multicore Programming Practices guide is published by the inventors of MCAPI to provide "best practices" [Lue12] in writing code that easily can be parallelized [Mul13]. Other standards for multi-processor systems are currently in development by the Multicore Association.

2.2.4. Multi-Processor Aspects

Current general-purpose processors offer parallelization based on Single Instruction Multiple Data (SIMD, vectorization) and Multiple Instructions Multiple Data (MIMD, specifically shared memory multi-processor). The SSE and AVX extensions support embedding vector operations in otherwise serial code. This does not require fundamental modifications of OSs. However, to support multiple processors, an OS must be constructed accordingly [Tan07, Sect. 8.1.2]. Different strategies are Asymmetric Multi-Processing (AMP) and Symmetric Multi-Processing (SMP) [Sta09, Sect. 4.2]. An AMP system can be implemented on heterogeneous processors where the OS is executed on a general-purpose CPU. It delegates work (tasks) to restricted or optimized processors. But, more relevant to this work, this approach can also be implemented on homogeneous systems (like x86 multi-processor systems) by executing multiple independent OS instances on dedicated processors [Moy13d, Sect. 6.3]. Likewise, single-processor software (OS and applications) can be transferred to a multi-processor system with only minor changes. However, either the OS must be instructed to use only the dedicated resources or a hypervisor enforces the separation by generating the impression of private systems (*virtual machines*). In a SMP [Vaj11, Sect. 3.3.1] system (Fig. 2.10), a single OS instance is executed on all processors. Tasks can be executed on all processors and the load is balanced to efficiently utilize

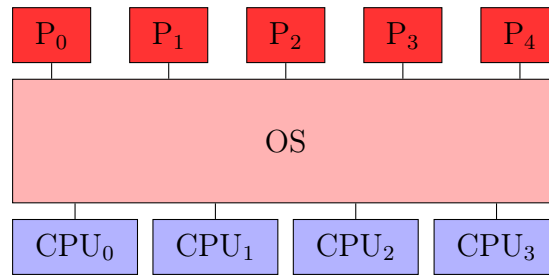


Figure 2.10.: SMP Operating system on multi-processor system.

all processors. The Scheduling becomes more complex [KCS04; RLA07] by the load-balancing between the processors, fair sharing of common caches, and energy saving attempts [SSP08]. A special case of this type is Bound Multi-Processing (BMP) [Vaj11, Sect. 3.3.3] where the (or some) tasks are not shifted between processors but bound to fixed CPUs to enhance the performance or predictability.

An existing single-processor OS can be extended with SMP support by adding the process of initializing the other processors, enabling multi-processor interrupt handling, extending the scheduling and load-balancing algorithms to include multiple processors, mutually excluding the concurrent access to shared resources, and providing those synchronization mechanisms also to user-mode applications [Dom08, Sect. 4.2].

Current GPOSs are SMP systems. But the lines are diffuse, the common OSs Linux, Windows and Mac OS all support binding tasks to dedicated processors (Bound Multi-Processings, BMP) and can act as hypervisor for virtual machines (that function is usually provided by additional software).

On multi-processor systems, scheduling and load-balancing are an important factor. Global scheduling algorithms are possible, but current GPOSs implement a run queue per CPU with single-processor scheduling algorithm and balance the load between them. The goals of load-balancing vary depending on the application and include evening the load to maximize system throughput, concentrating the load on few CPUs to switch some off (power-awareness, [Cor13b]), or placing tasks near to their memory (e. g. on NUMA systems [Bre93]).

The memory management must be adapted to multiple processors concurrently accessing its data structures. For NUMA systems, a further optimization potential is the placement of allocated memory. The strategy *Affinity on First Touch* uses the minor page fault to assign a physical page frame local to the processor that first accesses (reads or writes) a memory page independently which processor did the allocation call. Further features discussed for HPC systems are the migration of memory if the access pattern moves to a different memory node or executing tasks as near as possible to their memory working set.

2. Fundamental Principles

The synchronization objects provided for user-mode applications were presented in Section 2.2.1. These are suitable for multi-processor systems [HS08], although systems with many processors (also called *many-core* systems) demand additional optimization methods [Reb09]. Instead of spin-locks on the same shared variable, fairness and performance may require multi-level algorithms. The internal synchronization of the OS is done with the same algorithms and with Inter-Processor Interrupts.

2.2.5. Actual Implementation: Linux

This work concentrates on Linux because of several reasons. Linux is *open source*, that means all its components are available as source code and can be analyzed and modified. Further, it is very well documented [BC05; Lov10] and the base of many research projects. Above all, Linux is very versatile and widely accepted, it can be – and is in fact often – used from small embedded systems, mobile and desktop computers, internet servers and many HPC systems. Linux is a UNIX descendant, but it is not fully POSIX [Lew91] compatible, although it mostly adheres to that standard. The Linux API [Ker10] is stable, therefore, programs can be used on a wide range of Linux distributions and future versions are guaranteed to execute old programs without modification.

A *Linux distribution* consists of the Kernel being the core OS and a wide range of supporting software that provide the libraries (e. g. GNU LibC), tools (e. g. GNU binutils), compiler (e. g. GCC or clang) and other software (e. g. Graphical User Interface (GUI) like KDE or FVWM). The Linux kernel published¹¹ by Linus Torvalds is called mainline Kernel or Vanilla kernel. Many projects modify their distribution Kernel to support more features.

Linux is a very modern OS that supports many architectures and includes many optimizations. It is actively developed with a new Kernel version every two to three months. Since it is applied in many large computers, its scalability is highly optimized [GSC07]. Some features utilized later in this work are presented below.

Kernel Threads

For the internal maintenance, Linux uses *kernel threads* which are tasks executing in kernel-mode [Vir12]. Some kernel threads exist for special tasks (e. g. to process the items waiting in the work queue) and others are *pinned* to their CPU to execute only there. The pinned kernel threads usually respond only to requests on their processor.

Hotplugging

Linux supports the deactivation and removal of components during system runtime with the *Hotplugging* system [Cor13f]. This allows to switch off certain processors.

¹¹Online: <https://www.kernel.org> (visited November 12, 2015)

The OS handles remaining processes by moving them to other CPUs and shuts down the processor. After restarting a CPU, it is added to the set of available (online) processors and the load-balancer is free to execute tasks there.

Timer and Wall Clock

Linux uses a timer interrupt with a frequency of 100 to 1000 Hz (depending on the configuration). Recent versions support a flexible setting that adapts itself to the system load. The timer interrupt is used for keeping the wall clock time and for internal maintenance.

Linux supports various sources for this timer interrupt. The classic unit of the x86 architecture is the RTC (real-time clock), current systems use the internal localAPIC of each CPU. A higher precision is supported by the High-Precision Event Timer (HPET). Linux supports the POSIX timer API for user-mode programs with resolutions down to Nanoseconds.

Multi-processor Support

Linux supports Symmetric Multi-Processing (SMP) and Bound Multi-Processing (BMP). Tasks can be bound to single processors or groups of them with the CPU-Affinity function that sets process properties or the CPU-Set¹² method that supports the management of containers. For the optimization of HPC applications, the topology of CPUs and memory nodes can be queried (as documented in [LINUX:cputopology.txt]¹³) or managed with tools like Likwid¹⁴. The memory management [Gor07] is optimized for NUMA systems [Kle04; Bli+04].

Read Copy Update

Read-Copy-Update (RCU) is a synchronization mechanism to optimize shared resources that are mostly read and rarely changed [McK+01]. With conventional locking (e. g. a Mutex), every access requires two additional operations to lock and unlock the critical section. On large multi-processor systems, this operation causes cache contention which costs a considerable amount of time, especially related to short read accesses [McK09]. RCU takes an optimistic approach to allow reading without further protection. Generally, data can always be read. Removing no longer needed data structures is postponed until no readers are active on the respective element. Instead of modifying elements, new elements are created and the old ones removed when no reader is affected anymore. The writer can either wait for the next grace period and proceed with deleting or modifying an element or it can register

¹²Online: https://rt.wiki.kernel.org/index.php/Cpuset_management_utility/tutorial (visited November 12, 2015)

¹³Documentation/cputopology.txt (Linux 3.12)

¹⁴Online: <https://github.com/RRZE-HPC/likwid> (visited November 12, 2015)

2. Fundamental Principles

a callback function with RCU system that will be called later to catch up with required actions. The waiting approach would block the execution for unpredictable times, particularly with the pessimistic estimation of grace periods. Therefore, most parts of the Linux kernel use the callback method. The drawback is that every modification is more costly. Since the readers do not register themselves, a heuristic, overly pessimistic approach is taken. The detection, when no more readers are using an element is done pessimistically when all CPUs have changed their execution context. With the exact knowledge about the scheduling inter-processor relations, the algorithm can define a grace period after that every reader has left. An overview of the Linux implementation and further literature is given in [LINUX:RTFP.txt]¹⁵. One important use case of RCU is the Slab system [Bon94] that provides reusable memory buffers and that automatically deallocates no longer used buffers after an RCU grace period to ensure that no processor still uses them.

2.3. Real-Time

A real-time system has time-critical jobs that not only need to calculate a *correct* answer but also need to provide this answer at a *specified* time. The job can be a function call or the reaction to a trigger (e. g. an interrupt handler). The time specification can be a maximum time-out but also a minimum time is possible. Recommendable text books about real-time systems are available by Kopetz [Kop97], Liu [Liu00], and Burns and Wellings [BW01].

Being often embedded systems, real-time systems are commonly part of larger systems interacting either with other computers or with physical processes (plants). The system designer specifies the timing constraints, but they are usually derived from external real-world processes or the laws of control stability.

An often conceived misconception is the confusion with fast systems [Sta88; McK08]. Although a faster system may be more successful in meeting the deadline than a slower system, solid real-time systems are designed differently for being able to provide guarantees. They are optimized for the *predictability* of timing at the expense of raw performance.

When speaking of real-time systems, an important distinction is between hard and soft real-time. Burns and Wellings define real-time as any task with time sensibility in the sense of a delayed result being as harmful as a wrong result [BW01, Chap. 1.1]. Their separation of *hard* and *soft* real-time is however guided by criticality (defined below). In contrast, Liu presents a very sophisticated definition of real-time [Liu00, Chap. 2]. After listing common definitions (based on criticality, quantization of timing failures, and determinism) and arguing their advantages, real-time systems are defined simply depending on whether the system designer requires an elaborate formal verification (i. e. hard real-time) or if statistical demonstration (i. e. soft

¹⁵Documentation/RCU/RTFP.txt (Linux 3.12)

real-time) suffices. This is sometimes phrased as *guaranteed* and *best-effort* service. Soft real-time is sometimes associated with a Time/utility function [JLT85] that specifies how the utility of an answer degrades over time after the deadline. A steep decrease to zero is then defined as hard real-time. The definition agreed upon in most publications and used in this work is that a hard real-time system must provably comply *always* to the specification while a soft real-time system is allowed a certain degree of failure (e. g. 95 % of all events must be on time and for the remaining, a deviation of 10 % is allowed).

Following that definition, hard real-time systems must be *formally verified* while soft real-time can be validated more heuristically. The formal verification includes the analysis of the Worst-Case Execution Time (WCET) and the specification of frequency and probability of all events. Since this proceeding comes at a high effort [Liu00, Chap. 2.4.1], the system designer must balance the real-time requirements with development costs. Many applications include tasks with a wide range of specified demands. However, if a task depends on other tasks, its verification is simplified if it can be based on hard deadlines by the other tasks [Liu00, p. 31].

Besides the *hardness*, the magnitude of the specified latency is important. A process depending on the temperature may have a deadline in the Second to Minute range while other systems must respond in the Micro- or even Nanosecond range. Obviously, a larger latency requirement, even with hard limits, may be easier and cheaper to realize than a soft but extremely short deadline.

These terms should not be confused with *criticality* which is often done with the examples nuclear power plant and video codec for hard and soft real-time, resp. Anderson et al. provide examples for criticality from an aviation standard with the levels Catastrophic, Hazardous, Major, Minor, No Effect [ABB09, Table 1]. A similar term is *availability* being mentioned as advantage for distributed systems [Sta09, p. 708]. The often cited reason for hard real-time being *people's safety/injuries* may derive from the fact, that especially for systems that are safety critical, the “designer has the burden of proof that bad things will never happen” [Liu00, p. 31]. This is similar to the definition of hard real-time and in fact, many critical systems are also hard real-time systems (but not the opposite).

The assessment of a system's criticality must include the failure rate of the computer hardware. If the hardware has a certain probability to fail, the system must include a strategy to handle timing violations and outages. In critical systems, this could be a second fail-over system (hot stand-by) while in less critical systems, a detection, verbose warning, and transition to a safe state (e. g. stopping a robot arm) may suffice [KSB10, Sect. III-C]. But the real-time property is restricted to the software, and hardware failures are beyond the scope of this work. On the x86 architecture, the hardware failure expectation is higher than on special embedded systems. Therefore, a full evaluation must include software and hardware and the equation becomes more simple if a strong assurance for as most parts as possible can be given.

2. Fundamental Principles

The terms process and thread are defined in OSs as is task to be either of them (Section 2.2.1). The smallest entity of work (e. g. query a sensor, set an actor) is a job [Liu00], of which multiple can be in the responsibility of a task. An application is a collection of tasks that jointly serve a purpose (e. g. control a plant).

The measurements latency and jitter were already defined (Section 2.1.8) as time between a stimulus and the triggered event and difference between minimum and maximum thereof, respectively. Examples for stimuli are an external event (electric impulse), a message from another CPU (IPI on the hardware level), a message from another process (IPC on the OS level), or a previously blocked resource becoming available. The latency is generally not constant. The jitter is a measure for the absolute variability, hence the difference of minimum and maximum latency. For hard real-time, the maximum latency and the Worst-Case Execution Time (WCET) are important [Wil+08]. The WCET is the longest run-time for the according task. In a formal verification, this must include all possible code-paths of basic blocks in the Control Flow Graph (CFG) using the worst expectations for retries, loops and instruction latencies [Fer+01]. In the presence of caches, it must also include cache misses where they may occur. Both values can be measured or calculated. To measure the worst-case, strong efforts must be done to provoke the most disturbances possible. This is usually done by loading the system with more work than realistically present during real execution and measuring for a long time to increase the probability of catching incidences as near to the worst-case as possible. Since hard real-time requests a *formal verification*, it can not be proved by experiments but *must* be calculated abstractly.

On real hardware, the minimum latency (or execution time) is dependent on the hardware effects (e. g. instruction latency, memory transfer). If the average or median value is near to the minimum, this is an indication that outliers are seldom. For the analysis of jitter sources, more sophisticated statistics or histograms are helpful. But if the investigated effect is very short, the statistic must be recorded with minimum impact to avoid distorting the results.

2.3.1. Application Architecture

The automation pyramid (Fig. 1.1 on page 3) illustrates how real-time systems are applied in most factory control systems [Sau07]. To narrow down the topic and to precise general application areas, Liu classifies real-time systems as follows [Liu00, Chap. 1]:

Digital control Field level (lowest): Form (feedback) control loops by directly reading sensors, computing an output value according to a control law, and modifying actors.

Guidance and control Cell level: Management of multiple digital controllers, e. g. in an aircraft: Flight level control (to control the physical plant), flight man-

agement (to control the virtual plant provided by flight level control, manage the flight path), air-traffic control (to control coarse grained navigation on the basis of flight management).

Optimal control Cell or company level: Implement optimization problems (of path, cost, etc.).

Command and control Company level (highest): Control multiple systems on the level of other controllers. This usually queries and displays their state and processes user input and calculates plans.

Generally, the timing periods grow from *digital control* (Micro- to Millisecond) to *command and control* (Seconds to Minutes). Similar, the “hardness” of the real-time becomes softer in the same direction. For example, if an airplane does not get an update from *optimal control* at the expected time, it can still follow its previously set course (safe state).

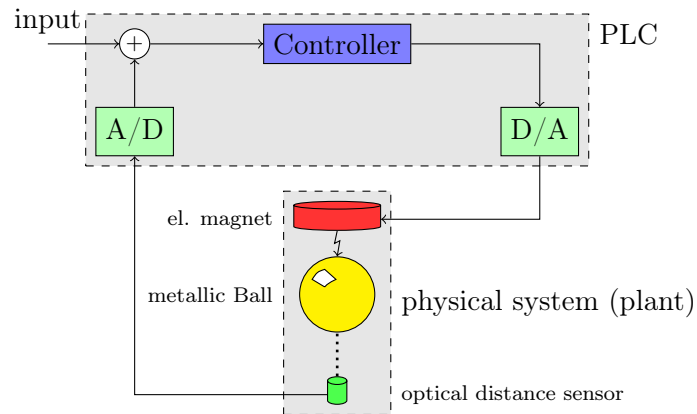
The components with the lowest and highest timing accuracy requirements are the Programmable Logic Controllers (PLCs). They often form closed-loop feedback controls as illustrated in Fig. 2.11a. Digital controllers commonly need a conversion of analogue sensor readings to digital values (Analog to Digital Converter, ADC) and a generation of analogue current for actor settings (Digital to Analog Converter, DAC). The sampling period of the ADC and DAC is *a key design choice*. Their selection may be based on the *perceived responsiveness* of the system or on the *dynamic behavior* required by the laws of control theory and stability theory to keep the oscillation and thus, the process, under control.

A similarly demanding application is Hardware-in-the-Loop (HiL) Simulation (Fig. 2.11b). It is used to verify the implementation of PLC embedded systems if the physical system is not (yet) available. The simulator is driven with the same signals and currents as real actors and provides the same readings as the sensors. A physical model is implemented to simulate the behavior of the real system. The timing is very demanding to correctly verify if the PLC implementation would work against the real plant.

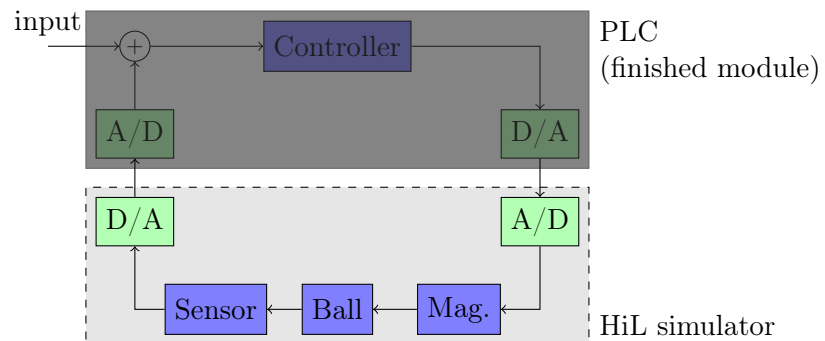
In *multi-rate systems*, multiple sensors collect multiple states and multiple actors influence multiple dimensions of behavior. A digital control could implement different rates: In a combustion engine control unit, the rotation can change faster than the temperature, thus the distinct control computation tasks can use different rates or the frequency of task can depend on the rotational speed. This is where tasks with different priorities or frequent deadlines origin from.

In *signal processing*, applications stream data with real-time constraints, e.g. for digital filtering, video and voice (de)compression, and radar signal processing. Although many of these systems are subject to *soft* real-time, there may be systems, where hard real-time is required (e.g. radar, in addition to high computing demands).

2. Fundamental Principles



- (a) Closed-loop control: The Programmable Logic Controller (PLC) controls a physical system (plant).



- (b) An implemented PLC is tested against a simulator that behaves equal to the plant.

Figure 2.11.: Hardware-in-the-Loop Simulation.

In special μ Cs and appliances, Digital Signal Processors (DSPs) are added for stream signal processing.

Signal processing may be of constant complexity (in the radar example: basic data processing) or time variant (in the radar example: dependent on the current number of tracked objects). A similar example is the Positron Emission Tomography (PET) data processing, where medical images are created based on associated events that must be discovered in a high-volume data stream.

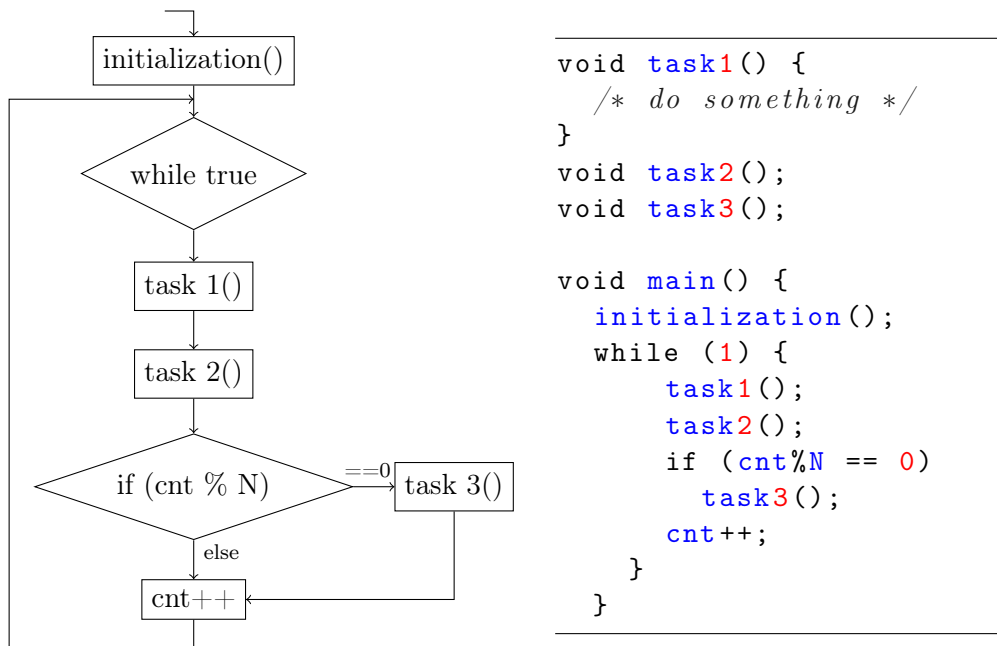


Figure 2.12.: Control flow diagram and example code of time-triggered paradigm.

The possible implementations range from single-processor μ C running only the application without OS (i. e. bare-metal) or with a statically linked embedded OS up to multi-processor high-end servers and distributed systems. However, some paradigms are common to a range of systems. In time-triggered systems [Kop91], all tasks are executed periodically in a large endless loop (Fig. 2.12). If a task is required less often (with a different frequency) than the others, it can be activated every N^{th} loop iteration. This paradigm is very basic and can be implemented on the most simple processors. It is well suited for closed control loops where the first task reads sensor values, the second calculates the control function and the third one sets actor values (Input-Process-Output (IPO) model). An optional task could be used for communication with a production control system and send the current state and receive the set point for its control function. The frequency of the entire loop can be set to a given value by waiting before the next iteration until the start time has come. As long as all tasks keep their expected execution time, the timing is correct. Thus, the formal verification of the time-triggered paradigm is simple.

2. Fundamental Principles

The Parallax Propeller¹⁶ is an eight-core μC that does not support interrupts. It is intended to be programmed in a time-triggered manner to poll its inputs. Since it has enough processors (when it was introduced in 2006, it was one of the first multi-processor μC available to the wide public) reserving a core for the sole purpose of polling inputs is considered acceptable.

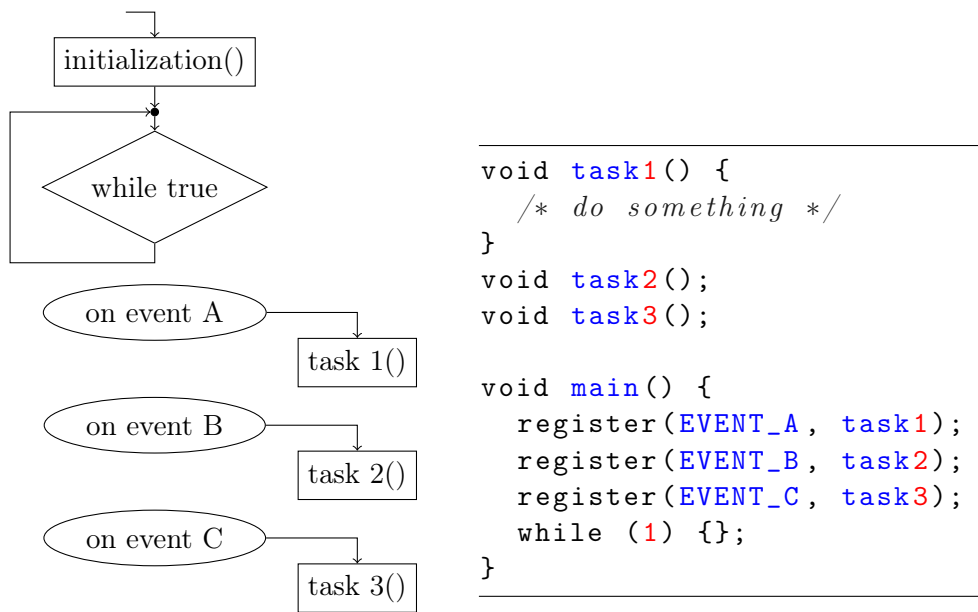


Figure 2.13.: Control flow diagram and example code of event-triggered paradigm.

The opposing paradigm is event-triggered [Kop91] where all tasks are only executed when triggered by events. After the initialization, the processor halts or waits in an empty loop and interrupts call the tasks (Fig. 2.13). This method is suitable for control applications where a small reaction latency to many possible events is required. The events can be assigned with priorities to interrupt less important tasks. While the latencies can be better than in a time-triggered system, the formal verification must now include the probability of occurrence and the maximum frequency of events which is more complex.

More realistic than the pure paradigms is a merger, the hybrid paradigm that includes tasks triggered by events and also an endless loop with some time-triggered tasks (Fig. 2.14). If one event is triggered by time (e.g. a frequent timer interrupt), the limits blur. All tasks called in a loop are defined as time-triggered, all interrupt handlers are called event-triggered. If an OS implements cooperative scheduling, this can be regarded as time-triggered because the tasks decide when to return control to the OS and that activates the next task.

¹⁶Online: <http://www.parallax.com/microcontrollers/propeller> (visited November 12, 2015)

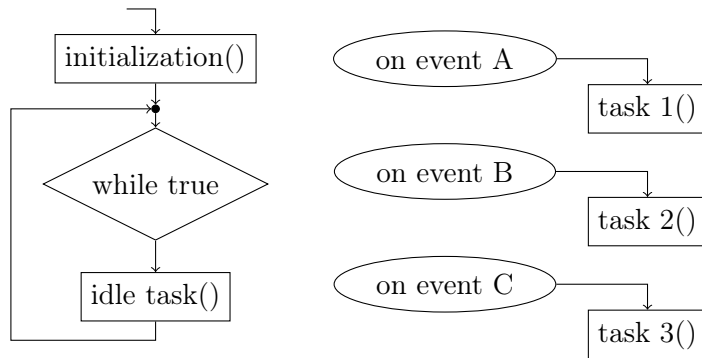
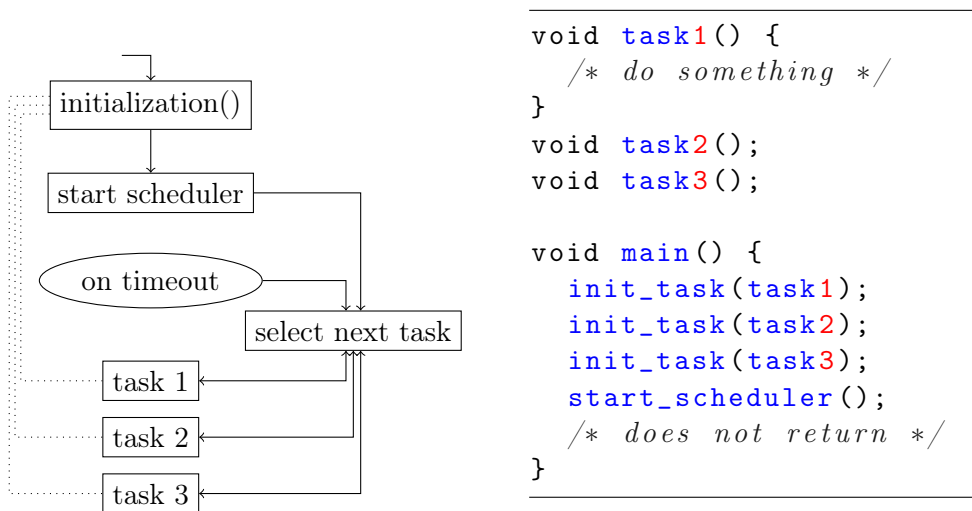


Figure 2.14.: Control flow diagram of hybrid (time- and event-triggered) paradigm.



```

void task1() {
    /* do something */
}
void task2();
void task3();

void main() {
    init_task(task1);
    init_task(task2);
    init_task(task3);
    start_scheduler();
    /* does not return */
}
  
```

Figure 2.15.: Control flow diagram of scheduling.

2. Fundamental Principles

A preemptive multi-processing OS assigns time slices to its tasks and is able to revoke control at events (interrupts or the end of their time slice). The selection of tasks can be regarded as time-triggered, but the system surely has also event-triggered capabilities for the timer and external events (Fig. 2.15). The selection of a paradigm must comprise the requirements of latency, predictability, performance and also the targeted hardware [Kop93].

Tasks can be classified as periodic (i.e. at regular time intervals, generally implemented time-triggered), aperiodic (i.e. irregular but frequently, both event- and time-triggered) and sporadic (infrequent and random, usually event-triggered) [Liu00, Sect. 3.3].

2.3.2. Scheduling

The sequence of tasks can be *fixed* or *dynamic*. A fixed schedule is calculated in advance (at system build time) and stored with the firmware. This allows to evaluate the possibility to schedule all tasks according to the real-time requirements (schedulability). Dynamic scheduling implements the algorithm in the OS and calculates at each invocation, which task to activate. Since the optimal solution is very compute-intensive, fixed schedules can be more elaborate while dynamic scheduling uses simplified methods. The schedulability analysis of a task-set with dynamic scheduling includes security margins and generally achieves a lower utilization. If tasks are created during run-time, dynamic scheduling must be implemented and the OS can also include a verification if the new task will not corrupt the system's timing to decide if it can be accepted.

In real-time systems, specialized scheduling algorithms are used that are different than those implemented in GPOS (Section 2.2.1). The most common strategy is priority-based [LL73]. In this scheduling algorithm, every task is assigned an integer valued priority and the *ready* task with the highest priority is executed. If multiple tasks meet this criterion, two strategies are common, *round-robin* executes them in time slices alternately and *first-in, first-out* completes the longest waiting one before activating the next. A task should block itself or lower its priority after a while to let lower-prioritized tasks execute. If a task misbehaves (on an error or intentionally), the whole system can be blocked. This scheduling algorithm is offered by most GPOSs, Real-Time Operating Systems (RTOSs), and embedded OSs.

A different method is *Deadline scheduling* where every task defines a deadline when it must be completed. Further, it must be specified, how long the task intends to work. These information allows to calculate the latest possible activation time for each task to complete the work in time. Various algorithms are known (earliest deadline first, latest deadline first, etc.). Other algorithms, in recent times also for multi-processor real-time scheduling, are discussed in the literature, e.g. Pfair [Bar+96; BS97].

2.3.3. Real-Time Operating Systems

Common operating systems (OSs) are optimized for average throughput (Personal Computer, PC; server; and High Performance Computing, HPC) or energy efficiency (mobile systems). The required predictability for real-time systems is supported by specialized Real-Time Operating Systems (RTOSs). Since the range of hardware and applicability is very wide, many very different RTOSs are offered. They range from very small embedded OS that are compiled and linked with the application code like a library up to OSs that are installed on hard disk like a GPOS. Also available are distributed real-time systems that include message-passing on a real-time capable interconnect [KG93; Pop+04]. Because of their differences, it is difficult to transfer an application to another RTOS. This is addressed by API standards like the real-time extensions of the UNIX standard, POSIX.4 [Gal95; Obe00]. Some RTOSs mimic the behavior and API of competing systems. Some notable products are [Lue13]:

FreeRTOS An open source embedded RTOS¹⁷. It is very portable and offered for a wide range of processors and μ Cs. The code is linked statically with the application to a binary firmware.

MicroC/OS-II Embedded RTOS for commercial applications and learning [Lab97].

QNX A commercial RTOS¹⁸, that is based on a micro-kernel and used in many products.

VxWorks A commercial RTOS¹⁹, also used in many products.

Linux

Being a General-Purpose Operating System, Linux is optimized for average throughput and energy efficiency (Section 2.2.5). Nevertheless, it provides the real-time scheduling classes *Round-robin* and *FIFO* that implement priority-based scheduling. Only if no task in these classes is *ready* to execute, the other processes with fair scheduling are executed. This allows to implement soft real-time applications but since interrupts and Kernel tasks can interject the real-time tasks at any time, the predictability is low. Although, if the latency requirements are in the Second to Millisecond range and the criticality is low, a plain Linux system can be utilized. Further, these mechanisms can be used to improve best-effort applications on Personal Computers (PCs), e. g. audio, video, and games. Linux also provides real-time capable synchronization objects and high-precision timing functions to better support real-time applications.

In fact, Linux is widely applied in a wide range of real-time systems from embedded [Abb13; SGD09] to high-end servers. It has the advantage of a very flexible configurability and the possibility to modify its source code due to the open source

¹⁷Online: <http://www.freertos.org> (visited November 12, 2015)

¹⁸Online: <http://www.qnx.com/products/rtos/> (visited November 12, 2015)

¹⁹Online: <http://www.windriver.com/products/vxworks/> (visited November 12, 2015)

2. Fundamental Principles

license. Further, it is well documented and many developers have experience with implementing and optimizing it.

After early attempts to use Linux for real-time systems [Sri+98], many different approaches have been made to improve its predictability and suitability for real-time applications [Hag05; SL06]. The main types are improvements to Linux itself converting it more into a RTOS and the sub-kernel approach to execute a different RTOS below the Linux kernel and virtualize the Linux system [Bar+07]. These will be presented in detail in Section 3.2.

2.3.4. Multi-Processor Aspects

Since multi-processor systems offer high compute-performance and are energy efficient, they are utilized since long ago for real-time systems that need more compute performance [HH89] and are of growing importance in recent times [SMZ11].

The common Symmetric Multi-Processing (SMP) concept can be applied to RTOSs, but their complexity increases. The scheduling problem already being hard on single-processor systems is fundamentally more complex on multiple processors [Liu00, Chap. 9; ACD06]. The scheduler and load-balancer have to decide additionally, on which processor the tasks should be executed. If the system has caches and a NUMA characteristic, those should be included in the decision process [Gua+09]. Similar to the other designs, most SMP RTOSs support *partitioning* to include Bound Multi-Processing (BMP) properties. Therefore, a wide variety of OS concepts is applied to real-time systems [VT10].

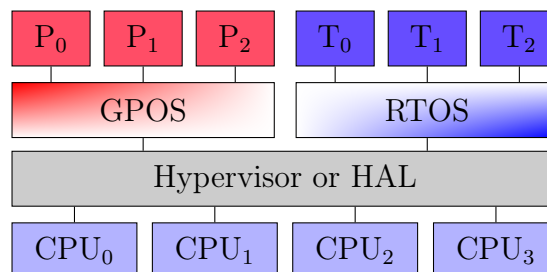


Figure 2.16.: Real-time Asymmetric Multi-Processing (AMP) system based on a hypervisor.

A straight-forward approach to integrate the functionality of multiple embedded systems into a new multi-processor system is to use an hypervisor to integrate multiple OS instances (Fig. 2.16 and Section 2.2.2). This layout is called Asymmetric Multi-Processing (AMP). The instances can be single-processor OSs tied each to their own CPU like the tasks in a BMP system. Further, a mixture of different systems or real-time with non-real-time OSs can be realized. The guest-OSs need either to cooperate with the *partitioning* approach (e. g. by using the Hardware Abstraction Layer (HAL)

interfaces instead of direct hardware access, i. e. para-virtualization) or must be executed in a virtual machine that imitates a private system. The communication between the instances can be realized by a virtual interconnect provided by the Hypervisor. The advantage of this principle is the possible maintenance of evaluated systems on consolidated hardware. But this principle does not exploit the gains of multi-processor systems. In addition, the Hypervisor or HAL layer introduces additional overhead and it influences the real-time capabilities of its guests, that can not be better than their foundation.

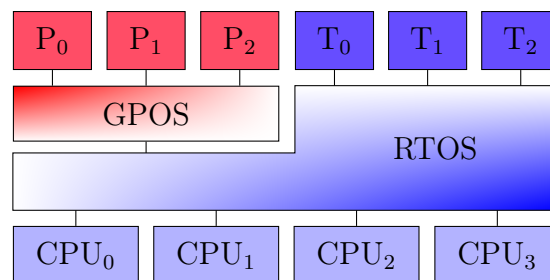


Figure 2.17.: Operating system based on a sub-kernel.

The sub-kernel approach (Fig. 2.17) is very similar, except that the hypervisor is simultaneously a complete RTOS that directly executes the hard real-time tasks and executes a virtualized GPOS as its lowest prioritized (idle) task. It can easily be extended for multiple CPUs by combining it with *partitioning* to reserve processors for certain tasks and others for the GPOS.

The above methods are common on homogeneous systems with equal processors. In embedded systems and as extension for commodity systems, heterogeneous processors are also employed [Sch13]. If the processors have different performance, capabilities or even different architectures (i. e. instruction sets), an Asymmetric Multi-Processing (AMP) structure is more suitable. These systems execute the OS on a subset of the available CPUs, generally the more capable ones [Moy13d]. The other CPUs are assigned tasks in a master/worker concept. Usually, the less capable or more specialized processors execute special code without OS system support (bare-metal) [Lal13]. Distributed systems or clusters are multiple systems that do not have access to the main memory of the others. Instead, they communicate with message-passing over an interconnect. This type of system is also used for real-time applications [BW01; Kop97] with their own challenges.

An important factor for the extension of existing real-time systems to multiple processors is the *scalability* of the implemented algorithms [BCA08; Vad+09]. The extension to few processors is well known from GPOSs, but the more CPUs are involved, the more complex algorithms must be implemented to avoid unequal chances and bad scaling. McKenney presents challenges that a growing number of processors pose to OSs with a special focus to Linux-rt [McK07].

2. Fundamental Principles

Synchronization

The synchronization in real-time systems may use similar means as in other OSs, but special challenges have to be considered apart from fairness and general throughput. If a task holds a Mutex and is then interrupted by a higher prioritized task, that task may wait on that same Mutex forever. This effect is called *priority inversion* because a task is waiting for another with a lower priority [SRL90]. Common solutions are *priority ceiling* and *priority inheritance*. These strategies increase temporarily the effective priority of a task holding a lock to the maximum priority or that of the waiting task, respectively. It is depending on the features of a RTOS, which synchronization objects (e.g. Mutex, Semaphore) support these strategies.

If a lower latency is required than the synchronization objects provided by the OS, user-mode solutions like spinlocks can be used. They wait busily on a shared variable by frequently polling that object using atomic operations. This has the advantage of a very fast reaction time and low overhead but since they do not block the waiting task, it should only be used for short periods of time. Further, there is no solution for the priority inversion problem and the wasted processor cycles could be used more productively by other tasks [SG94, Chap. 6.4.2].

Instead of locking, special algorithms can be used that do not need locking. These lock-free algorithms are efficient and can be used for message-passing protocols or shared resources. Since some of these algorithms include retry loops in case of a collision, the property wait-free was defined to always terminate in a constant time [Her93].

If tasks do not share resources, critical sections are avoided. Communication via *Message-Passing* is inherently synchronizing because a message is sent after the data is available and a task waiting to receive will only proceed after the data has arrived. This is also more secure, because distinct processes without shared memory can not corrupt others. However, a message not sent will also violate the timing and received messages must be checked for integrity [Vaj11, Sect. 5.4.8].

Synchronization is very important in concurrent real-time systems. It has a direct effect on the performance and correctness of the system. Errors (both in the OS implementation as well as in the usage by the application) are hard to detect since races may occur only seldom. The timing of synchronizing tasks is very difficult and may even introduce *timing anomalies*. This behavior can be observed in out-of-order architectures due to dynamic reservation of functional units within the processor. Besides cache misses, these timing anomalies can result in execution time jitter [LS99; Rei+06].

2.3.5. Performance of Real-Time Systems

A real-time system can comply to the specific requirements or not. Therefore, real-time research generally prefers the formal verification over benchmarks emitting a

performance measure. But some concepts and OS implementations are too complex and to prove soft real-time systems or to show improvements to OSs, real-time benchmarks are used. Further, the behavior of some hardware architectures is very difficult to predict, partly because of its complexity, but also because of missing documentation about implementation details. Real-time benchmarks are used to quickly compare OS modifications, configurations settings, or different hardware. Further, for soft real-time and very complex systems, it is more cost-effective to measure the performance.

Verification

Formal verification [Els+72] uses mathematical and algorithmic methods to prove if a program adheres to a specified model. Besides the functional correctness, real-time systems require the analysis of their timing. This must include the levels basic blocks, tasks and schedulability [CP01]. *Basic blocks* are linear parts of code that have single entry and exit points and do not contain any loops or branches. They always execute the same instructions in the same sequence. Their WCET depends on the hardware and, on multi-processor systems, the contention of shared resources. *Tasks* are build from basic blocks that are connected by loops and branches to a Control Flow Graph (CFG). The timing depends on the number of loop iterations and the probability of branches. Finally, the *Schedulability* analyzes if all tasks can be executed with the selected scheduling algorithms (e. g. preemptive priority scheduling) under inclusion of blocked shared resources.

Execution Time of Tasks

For soft real-time applications, the average and variance of the execution time may suffice, but for hard real-time systems, the Worst-Case Execution Time (WCET) must be regarded [PB00]. In some settings, the Best-Case Execution Time (BCET) is also used. Generally, the WCET can be estimated by formal methods or measured. In hard real-time systems where a formal verification is required, the WCET must also be derived by formal methods.

Processors for embedded hard real-time systems (Programmable Logic Controller, PLC) often use a RISC architecture that is optimized for constant execution time instead of average throughput. For example, the Atmel²⁰ AVR series of Micro-Controllers [Küh98] executes most instructions in one cycle and the documentation gives an exact cycle count for every instruction [AVR10]. This allows to determine exactly the execution time of a linear (non-branched) code by counting instructions and identifying the number of loop iterations. Varying code paths can be evaluated with established methods.

²⁰Online: <http://www.atmel.com> (visited November 12, 2015)

2. Fundamental Principles

In contrast, modern processors for commodity systems are optimized for the average throughput posing challenges for a predictable execution time [PC07]. The execution time of a task or function varies the more, the more complex an architecture is [Pet02]. For architectures applied in this type of systems (e. g. x86 and ARM), the estimation becomes thusly very diffuse. The span between quickly increasing processor speed and memory throughput widens [WM95]. To mitigate memory transfers slowing down the instruction throughput, optimizations like multiple levels of caches, long execution pipelines, ILP (super-scalar out-of-order execution; branch-prediction), vector instructions, multi-core processors, and more were introduced [HP07, Chap. 2]. Therefore, the history of the pipeline filling and branch predictors, the use of private and shared caches and the contention of shared resources and I/O [SBF05] all contribute to the jitter of a basic block. The dependencies are hardly documented and difficult to measure [And+97; Fog13a]. Therefore, the derivation is very complex as it must consider the most pessimistic blocking, cache misses and contentions [Pao+13]. But this would yield an overly pessimistic value that would allow only few tasks per processor resulting in a massive under-utilization. Therefore, the aim is a *tight* estimation of the WCET. A realistic result can be obtained by an estimation on the base of a cycle-accurate processor simulator [YL06] (e. g. PTLSim [You07], MPTLsim [Zen+09]) or by measuring the execution time under load on the real system. Since so many factors are unknown in this setting, they must be accounted for by a security margin.

Benchmarks

Benchmarking real-time applications, OSs, and hardware is difficult, and easily results in misdirections [DM04]. To obtain the WCET experimentally, the worst-case must be observed. The probability to provoke events causing the worst-case (or a delay close to it) increases if the test system is loaded with more work than realistically present in the final system [Fog13b, Sect. 16.2]. It is advisable to warm up the tested routine by executing some iterations before recording a statistic. This avoids including cache effects that will not be present in the running application. To create a more than realistic load, the following measures can be taken [Fog13b, Sect. 16.2]:

- Executing the whole application with all dynamic link libraries.
- If using the network or a server, loading them too.
- For files and databases, using large data-sets.
- If the application will be installed on various systems, testing on slow hardware with insufficient RAM and with many background processes.
- Using different brands of CPUs, different graphics cards and other relevant peripheral devices.
- Running multiple instances and, if testing on a multi-processor system (or SMT), binding them to the same physical core or socket.

Log mode	Gaps (CPU cycles)		
	Minimum	Average	Maximum
Only statistics	72	93.50	2303304
Histogram	96	178.08	236488

Table 2.6.: Hourglass overhead.

- If using shared memory for communication and synchronization, testing scenarios with all on the same CPU, all on the same NUMA node, and all on different NUMA nodes.
- Provoking swapping by allocating more memory than physically available.
- Further types of load are: CPU bound, memory transfer, I/O, OS services, and errors or exceptions.

A universal load often used is to build the Linux kernel during the benchmark. This is not comparable to other systems, but can be used to rate single changes to a system.

In the following, some benchmarks are presented that are used in the literature [Hal+00; Dan12]. The *Realfeel* benchmark is a comprehensive suite of functions that tries to rate the quality of a real-time system [Web02]. Abeni et al. evaluate the real-time performance of the now outdated Linux 2.4 kernel [Abe+02]. But the paper contains guidance on *loading the system* during real-time benchmarks. Gonçalves and Melo present rules and best practices for testing real-time applications with Linux-rt [GM08], that also apply to general real-time programming. Rostedt and Hart present some simple benchmarks [RH07] used for the improvement of the Real-Time Preemption Patch (RT-Patch) . Williams presents how to analyze the scheduler [Wil02]. This includes stressing loads and some test programs for specific details.

Hourglass

An important benchmark used repeatedly in this work is the *Hourglass* benchmark [Reg02]. It was introduced to analyze the time slices assigned by the operating-system to multiple processes. In this work, the concept was used to analyze gaps in the execution time caused mainly by interrupts.

The example code in Listing 2.1 presents the basic principle. The minimum loop is short enough to detect a variance in the range of a few CPU cycles of the mean execution time. To measure more realistic scenarios, it can be extended with a synthetic workload of variable range, e.g. the writing of a memory buffer or an I/O access. The size of such a buffer can be selected to fit into the different cache levels so that working in the L1\$ or even cache misses to the main memory can be ensured similar to the Membench (Sect. 2.1.8).

2. Fundamental Principles

```
int hourglass(uint64_t duration, uint64_t threshold)
{
    uint64_t t1, t2, t_end, diff;           /* timestamps */

    rdtsc(&t1);                             /* start-time */
    t_end = t1 + duration;                  /* calculate end-time */

    while (t1 < t_end) {                   /* loop until end-time */
        t2 = t1;
        rdtsc(&t1);
        diff = t1 - t2;
        if (diff > threshold) {
            store_timestamp(t1, diff);
        }
        /* Note: Additional task workload may be added here */
    }
}
```

Listing 2.1: Hourglass benchmark routine (simplified).

The function `store_timestamp()` must be implemented as efficiently as possible to avoid a measurable impact that would corrupt the results. The most simple implementation only records the minimum and maximum values (Listing 2.2). More meaningful is the collection of a histogram to understand the distribution of gaps. The impact of logging histogram data can be seen in Table 2.6. The statistic recording results in the average being very close to the minimum, indicating that the outliers up to the maximum are very rare. The absolute value of the maximum in this test series depends on the OS scheduling. The histogram results nearly in a doubling of the average loop execution time. On recent x86 systems, the loop is very short (20 to 100 cycles) when executed in the L1\$ and depends on the CPU micro-architecture. Figure 2.18 displays the distribution of results on the desktop system `poodoo` (System specification in Appendix A.1.1) during normal work. The number of events is in logarithmic scale. The highest bar is more than two orders of magnitude bigger than any other.

With a threshold well above the mean execution time, only interruptions of the normal execution are detected and recorded limiting the number of events so that they can be recorded in a list to plot their length over timestamps. This allows a more precise evaluation of the distribution and clustering of events and to find the source of disturbing events. Figures 2.19 and 2.20 illustrate this: With a threshold of 1000 cycles, the first 1000 gaps are nearly all in the range of 4000 cycles length. A burst between 0.4 and 0.5 Seconds includes many gaps up to 200 000 cycles. The

```

uint64_t min=U64_MAX, max=0, sum=0, cnt=0;      /* globals */
uint64_t t_min, t_max;                        /* timestamps of min/max events */

int store_timestamp(uint64_t offset, uint64_t gap)
{
    if (gap < min) {
        min = gap;
        t_min = offset;
    }
    if (gap > max) {
        max = gap;
        t_max = offset;
    }
    sum += gap;
    cnt++;                                     /* avg = sum/cnt */
}

```

Listing 2.2: Minimal implementation of timestamp recording function.

threshold of 10 000 hides the large number of short gaps and records more of the longer ones.

The task switch depends on the OS but its magnitude is also influenced by the hardware overhead of interrupt handling. For this test series, the original Hourglass benchmark was used. Two processes are executed on the same CPU and if they are the only *ready* tasks, they will run alternately. Both Hourglass routines detect the gaps in their execution time when an interrupt or the other process executes. The difference between one task interrupting and the other continuing is the task switch latency. The minimum task switch time on the test system *poodoo* with Linux 3.11 was measured at 5600 cycles (2.11 μ s). The minimum time for an empty

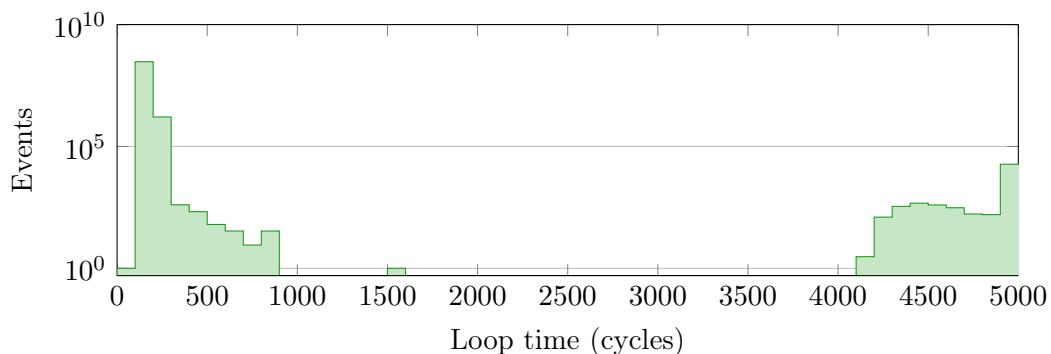


Figure 2.18.: Hourglass histogram.

2. Fundamental Principles

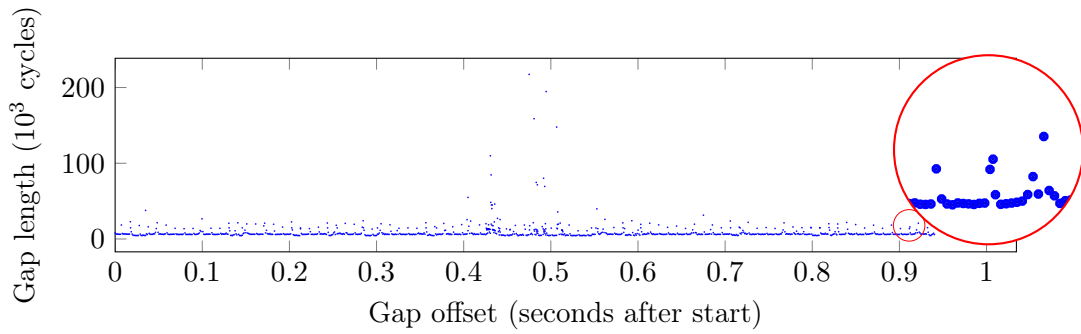


Figure 2.19.: Scatter plot (threshold 1000).

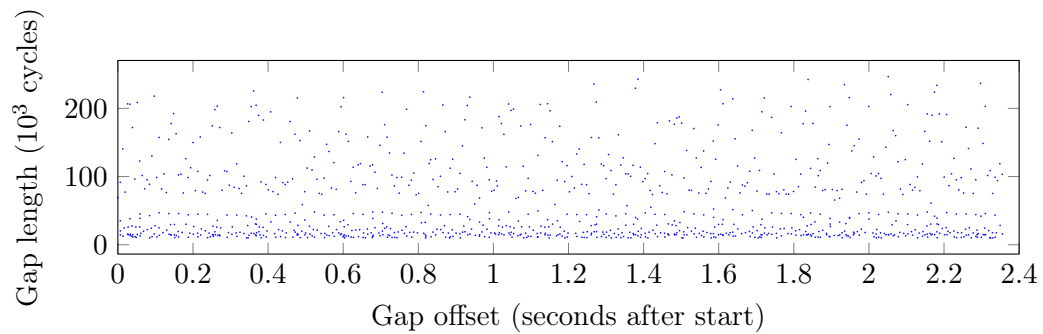


Figure 2.20.: Scatter plot (threshold 10000).

interrupt handler on the Core2 is 970 cycles (Table 2.5) and the majority of detected interrupts on a real system was detected at approx. 4000 cycles. From this follows, that the task switch causes slightly more overhead than the typical timer interrupt. The median of approx. 1000 recorded task switches in 10 Seconds was 7800 cycles and 95% were below 13000 cycles ($5 \mu\text{s}$). The maximum was 200000 cycles, but all values above the 95% quantile can be biased by long interrupts and other tasks executing in between.

3. Related Work

This Chapter presents related work with the same motivation, similar goals, or likewise approaches and highlights where these publications helped to advance this project and how it distinguishes itself from others. The field of research includes High Performance Computing (HPC) and real-time research that are combined to improve multi-processor real-time systems. Further, supervised student's theses are presented that had an influence on this work.

3.1. Fundamental Real-Time Research

Before approximately the year 2000, multi-processor systems were mostly reserved to HPC applications in compute centers of science and research facilities. Apart from some special applications requiring real-time as well as high compute performance that were implemented on special multi-processor systems, specifically designed Systems-on-a-Chip (SoCs) and elaborated software, the concept of multiple operating system (OS) instances and real-time-capable scheduling was invented for single-processor systems.

3.1.1. Isolation or Partitioning on Single-Processor Systems

Lipari et al. use the term *isolation* in the context of a single-processor scheduling algorithm to emphasize the protection of an application from other, misbehaving applications [LCB00]. However, they do not present an implementation nor do they consider the hardware effects caused by other applications.

Bollella and Jeffay present a way to execute multiple OS instances on a single processor system with real-time constraints [BJ95] (Asymmetric Multi-Processing, AMP). They use virtualization to use existing General-Purpose Operating Systems (GPOSs) and Real-Time Operating Systems (RTOSs) without the need to modify them. The RTOS is granted time slice reservations, i. e. temporal partitioning or Time-Division Multiple Access (TDMA).

Kuo et al. present an *open environment* for parallel and distributed real-time systems [KLW00]. This appears to be a special RTOS, that uses time slices for applications and each application schedules its own tasks. Time-based resource reservations solve the *real-time composability problem*, but waste computing resources in the same way as the spacial partitioning presented here. Another approach of

3. Related Work

temporal resource partitioning on single-processor systems by Mok et al. aims to simplify the global verification by creating partitions that can be evaluated separately [MFC01]. A similar project is also presented by Deng et al. with a *virtual processor* isolating the hard real-time application from other applications [Den+99].

The text book of Buttazzo promotes a more scientific approach to the implementation of real-time systems [But05]. The author presents categorized scheduling algorithms for single-processor systems and introduces to the practical problems of synchronizing access to shared resources and available OSs. Regehr and Stankovic observe the problem of predictable execution on loaded single-processor systems and look out to upcoming multi-processor systems where a separation becomes feasible [RS01].

3.1.2. Foundations of Real-Time Computing on Multi-Processor Systems

First systems with multiple processors were connected single-processor systems without a common memory space. These distributed systems were either special embedded systems (e.g. sensor networks) [Kop97] or HPC clusters used for soft real-time (e.g. streaming).

With the advent of multi-processors for Personal Computer (PC) type systems (Simultaneous Multi-Threading (SMT), e.g. Intel HyperThreading [Cha02] in 2002; multi-core processors such as the Intel Pentium D¹ in 2005), parallel computing became main-stream (since the Intel Core i7 in 2008, all x86 processors – except the smallest scale Atom series – have at least two physical cores) and Micro-Controller vendors began to promote multi-core processors for embedded systems (e.g. ARM-based by FreeScale², Texas Instruments³, etc. around 2007). The following publications present a first overview of real-time approaches on multi-processor systems. These works use neither Linux nor the x86 architecture.

The *Spring* kernel is claimed to be a new paradigm for RTOSs [SR89]. This and the Hawk kernel [HH89] are two early examples of RTOSs for embedded multi-processor systems. They are intended for *Guidance and Control* systems and *Command and Control* applications with hard real-time requirements. These applications have a lower frequency than digital controllers (Programmable Logic Controllers) but higher compute demand. The special processors and OSs are of lower relevance today, but their concepts are still applied.

¹Online: <http://www.intel.de/content/www/de/de/processors/pentium/pentium-d-processor-brief.html> (visited November 12, 2015)

²e.g. <http://media.freescale.com/investor-relations/press-release-archive/2007/16-10-2007-d.aspx> (visited November 12, 2015)

³e.g. <http://newscenter.ti.com/index.php?s=32851&item=123723> (visited November 12, 2015)

3.1. Fundamental Real-Time Research

Sindhvani et al. present RTOS acceleration techniques for embedded systems with multi-core processors [Sin+04]. They include some interesting ideas, e. g. a co-processor for interrupt-handling (which is related to this work) and a Hardware-RTOS. The paper was presented at the Real-Time Linux Workshop 2004 in Singapore, but is not covering Linux in particular. They target special Micro-Controllers (μ Cs) and soft-core processors implemented in an Field-Programmable Gate Array (FPGA) missing the opportunity of today's systems with eight and more x86 processors. Scheler et al. present their OS extension for real-time interrupt handling using an heterogenous co-processor [Sch+09]. The architecture is a Tri-Core μ C, but the partitioning approach is similar. Dubrulle and Ohayon present a special micro-kernel for embedded real-time systems that can provide different demands such as hard real-time and streaming [DO13]. Although this project supports both hard real-time and compute-intensive applications, it is not as versatile as a Linux system.

The advantage of non-x86 architectures like many μ Cs and Scalable Processor Architecture (SPARC) [Lic+08] is their execution with a predictable timing. This is further examined by the MERASA [Ung+10] and parMERASA [Ung+13] projects that emphasize the need for powerful multi-processor systems providing hard real-time capabilities. A new architecture of small and simple cores, a cache-less memory hierarchy and a static switched network on chip is promoted [MMU11] to improve the Worst-Case Execution Time (WCET) estimation in multi-processor systems with caches [MU12].

There are some multi-processor capable RTOSs, for example RTEMS⁴, the *Real-Time Executive for Multiprocessor Systems*. This open source licensed project provides some POSIX interfaces and BSD sockets but can not be used for existing Linux applications to extend them with hard real-time tasks.

3.1.3. Scheduling Theory

Goossens and Richard present an overview of real-time scheduling problems and algorithms [GR05], both for single and multi-processor systems. Anderson et al. provide basics on real-time scheduling theory on multi-processor platforms [ACD06]. Carpenter et al. present an extended overview of scheduling algorithms for hard real-time tasks on multi-processor systems. The scheduling problem for multiple processors is still hard to solve efficiently. They describe two approaches as tradition: *Partitioning* and *global scheduling*. In *global scheduling*, all tasks are organized in a single queue and a global scheduler distributes them to the Central Processing Units (CPUs). With *partitioning*, the tasks are assigned to a fixed CPU. The authors further present a hybrid approach where jobs are assigned fixed, but tasks are allowed to migrate [Car+04].

⁴Online: <http://www.rtems.org> (visited November 12, 2015)

3. Related Work

The formal verification and some scheduling algorithms require an estimation of the execution time. The challenge of a tight estimation in the presence of concurrent resource users is thoroughly analyzed considering interrupts [Abe01; BLA11] and device drivers [Lew+07]. They all conclude, that the prediction is very complex and that it can hardly be handled without deep modifications to the OSs.

Abeni and Buttazzo debate how to integrate soft real-time and compute-intensive workloads into hard real-time scheduling algorithms like Earliest Deadline First (EDF) [AB98]. This indicates a need for the combination of those applications on multi-processor systems. Starke and Oliveira analyze a heterogeneous scheduling approach that is part preemptive and part non-preemptive [SO12]. They use a partitioned system to avoid migration overhead. This encourages a spacial partitioning given that enough processors are available.

3.2. Real-Time with a Linux Kernel

This Section gives an overview of the different approaches to build real-time systems with Linux. The amount of publications concerning soft and hard real-time improvements for Linux indicates its wide application in those systems and a strong interest to be used for even more applications [Fin01; WL04; Duv09]. Scordino and Lipari group those approaches into *Interrupt Abstraction* and *Kernel Preemption and Real-Time Improvements* [SL06]. Here, modifications to the Linux kernel are presented first. They intend to improve the preemptibility and reduce the worst-case reaction time (latency). Some separate efforts led to the Real-Time Preemption Patch (RT-Patch), a joint project that collects and improves Linux kernel modifications. Further, a variety of approaches to execute Linux under the supervision of or concurrently to a RTOS micro-kernel are presented. The underlying sub-kernel or abstraction layer has a prioritized access to the interrupts and executes the Linux system only when the timing allows it. On single-processor systems, both OS instances are executed with different priority while on multi-processor systems, a spacial partitioning can be applied.

Predecessors of the RT-Patch.

The Linux kernel was first published in 1991 and quickly gained developers and users. In the late 90s, Linux was mature for a wide range of applications and could be installed on many different compute architectures. Naturally, it also attracted developers of real-time systems. However, the standard Kernel was not well suited for predictable execution, but since the source code is available and the open source license allows modifications, several approaches were started. Among them is “KURT” (Kansas University Real-Time), a collection of modifications to provide

higher resolution timing and a priority based scheduler [Sri+98]. The authors describe their system as *firm* real-time, a slightly weaker definition than hard real-time.

Goel et al. have presented Time-Sensitive Linux [Goe+02], a modification of Linux for soft real-time applications and multimedia [Kra+09]. Other independent Linux modifications are a Linux kernel patch to improve predictability and reduce latency [Yan+05], a scheduler extension supporting reservations and deadline scheduling [KRI10], and another scheduler replacement supporting partitioning and reservations [Sou+11].

3.2.1. The Real-Time Preemption Patch

A more systematic approach to increase the predictability of the Linux kernel were the *Low Latency Patch* of Andrew Morton and *MontaVista's Preemptible Kernel Patch* by Robert Love [SL06]. They analyzed the code for long regions of uninterruptible execution due to deactivated interrupts or holding locks and modified those sections either with voluntary interruption points or by using smaller grained locks and preemptible spin-locks. Today, many of the fundamental modifications are integrated in the mainline Kernel [McK05b]. Since real-time support generally comes at the cost of a lower general throughput, the grade of preemptibility can be configured. The remaining modifications not yet included in the mainline Kernel were collected by the PREEMPT_RT project⁵ where many well-reputed kernel developers collaborate to improve the real-time capability provided by the Real-Time Preemption Patch (RT-Patch) and to include the modifications into the mainline Kernel [DW05] [Cor10].

The major modifications target the predictability by lowering the worst-case latency and added infrastructure required by real-time applications. To protect global data structures from concurrent access, the earlier Kernel deactivated interrupts on single-processor systems and used a global lock (“Big Kernel Lock”) to synchronize multiple processors. When a higher prioritized task was due while a lower one executed such a critical region, the other had to wait until the blocking one left that region and removed the global lock. The duration of such blocking situations is obviously not predictable and scales badly with the number of processors. Therefore, blocked sections were made interruptible to decrease the maximum time other tasks are blocked [McK05a]. Since an interrupt handler blocks the handling of new interrupts, the interrupt system was split into minimal interrupt handlers and deferred work queues that process the pending requests when appropriate [GDA08]. Further, synchronization objects such as spinlock, Mutex and Semaphore were extended with priority-awareness to better support real-time applications and to avoid priority inversion [RH07]. The infrastructure is further supported by high-resolution timers [GN06] to exploit the hardware capabilities of modern systems providing a resolution of Microseconds and below. An important part is the enabling of Read-Copy-Update

⁵Online: <https://rt.wiki.kernel.org> (visited November 12, 2015)

3. Related Work

(RCU) (Section 2.2.5 on page 51) for the fully preemptible kernel. It was discussed a long time, if and how RCU's determination of grace periods can be made aware of preemption [MS05], but it was finally solved [McK+06] and further optimized [McK09] also for real-time systems.

Scheduling

After several attempts to add new scheduling algorithms to the Linux kernel, LITMUS^{RT} is a test bed to empirically evaluate new multi-processor scheduling algorithms in a Linux system [Cal+06]. It is used in several publications to prove the models of their scheduling theory. In a practical evaluation, LITMUS^{RT} was compared to the RT-Patch and its overhead was rated acceptably low, but it inherits the sensibility of Linux for Input/Output (I/O) load [CB13].

Kato and Yamasaki present a Linux kernel module to exchange the scheduler with their own implementation [KY08]. The motivation for a kernel module is that this requires the least modifications to the kernel source code and thus, an easy porting to various versions.

Recently, a deadline scheduling class was integrated into the mainline Linux 3.14 kernel [Cor14a]. However, all scheduling work is still bound by the task switching overhead, especially in multi-processor systems where load-balancing additionally causes cache misses and thus, a larger jitter. The use-cases where a jitter in the Microsecond range is required as targeted by this work, can not switch tasks, at least on the CPU where such a high-frequency task is executed.

Tickless Kernel

In time-sensitive as well as in HPC applications, the frequent timer interrupt was identified repeatedly as unwanted overhead because of its compute time missing to the application and due to cache-effects further reducing the cycles spent productively. The Linux configuration allows to select the frequency of the timer interrupt between 100 and 1000 Hz to adapt the OS to specific work-loads. This work was extended by the *Dynticks* modification initially developed as part of the RT-Patch [Cor07]. It reprograms the timer interrupt depending on the situation and can rigorously reduce the number of such interrupts in the best case.

The kernel configuration option `CONFIG_NO_HZ` turns off the timer interrupt whenever a CPU is idle. This allows a deeper sleeping mode without frequent wake-ups, but increases the latency to reactivate idle/sleeping CPUs. Thus, it is applicable only to desktop systems and not suited for scheduling real-time systems. This approach was continued in the `NO_HZ` development of Linux 3.9 [Cor13a] and extended in Linux 3.10 [Cor13e]. Currently, a new *full tickless* mode is implemented, that further reduces the timer interrupts if only a single task is active on a given CPU. In this case, no scheduling is required, but one interrupt per Second is still needed for

time-keeping. There are RCU modifications [Cor12] required to support the full tickless mode. The constraint of the remaining timer interrupt should be removed in later kernel versions [Cor13c], [McK13a].

It is expected by the kernel developers, that HPC and real-time applications will benefit from these improvements since it is common for those applications, to dedicate CPUs to single tasks [Cor13e]. Thus, the Linux kernel supports the reduction of the timer interrupts up to their complete eviction. This is very similar to the complete isolation of a CPU, but since other interrupts, especially the Inter-Processor Interrupts (IPIs) are not blocked, this does only improve the average performance but does not grant hard real-time. However, this approach could be further investigated as alternative to the kernel modification presented in Chapter 5.

Evaluation

The RT-Patch is still actively developed and its inclusion into the mainline Kernel is promoted [Edg13b]. The real-time capability of Linux with RT-Patch (Linux-rt) was evaluated many times [AEM07; BCB09; GL11] but always experimentally. Since the Linux kernel is very complex, a formal verification is probably not feasible. The most thorough practical evaluation is presented by Emde consisting of a large range of systems tested under heavy load over long times [Emd10]. This coverage does not replace a formal verification, but demonstrates that Linux-rt is applicable to a range of demanding real-time applications. However, the Linux kernel can only be considered for soft real-time systems. In the scope of this project, the RT-Patch can be applied to the Linux kernel to improve the timing of applications running in the system partition.

3.2.2. Interrupt Abstraction.

A different approach to increase the real-time suitability of a system running Linux is to abstract interrupts. In other words, Linux is executed under supervision of a hypervisor (Section 2.2.2) or another OS (sub-kernel) creating effectively an Asymmetric Multi-Processing (AMP) system. The RTOS is executed directly on the hardware and executes Linux as task with the lowest priority. This results in all hard real-time tasks being preferred to all Linux tasks. The interrupts are caught by the RTOS and passed through to the GPOS when no time-critical work is pending. Since the modification of Linux itself remains a complex system with more possible code-paths than coverable, these approaches allow to use Linux applications together with hard real-time tasks on the same system.

The first extension of Linux for hard real-time applications was RTLinux⁶ [Yod99]. It was designed to require as little modifications to the Linux kernel as possible.

⁶not to confuse with the RT-Patch, that extends the name of the kernel to Linux-rt, but some authors use “RT-Linux” when speaking about Linux with RT-Patch [GL11].

3. Related Work

Essentially, it replaces the interrupt handlers and the code to disable interrupts [BY97]. A thin layer below the Linux kernel controls the real-time tasks and executes Linux as a para-virtualized guest OS when no time-critical work is pending. Devices are either controlled by the real-time layer if they are used by time-critical tasks or assigned to Linux drivers [Bar97]. This allows to use a wide range of available hardware that is supported by Linux and requires only to implement drivers for time-critical I/O devices. The product was offered by FSMLabs and later acquired by Wind River⁷ and is discontinued today. The tight control by a patent-holding company was probably the trigger of several imitations with similar goals but sometimes slightly different details.

One of these similar approaches is RTAI⁸ that was created in the style of RTLinux. But instead of a dedicated Hardware Abstraction Layer (HAL) that is integrated in the Linux kernel, RTAI is implemented as a kernel module that requires only a small patch [MDP00] to facilitate its porting to new kernel versions. Real-time tasks are also implemented as kernel modules that can be loaded or changed at run-time. RTAI promotes the use of low-cost high-performance hardware (Commercially off-the-Shelf, COTS) [DM03a] and allows to implement distributed systems [DM03b]. It was also used to improve hard real-time applications on multi-processor systems with an isolation approach executing a GPOS and RTOS on dedicated CPUs [SBF04]. The latest version of RTAI was 4.0 published⁹ on December 6, 2013 supporting recent Linux 3.x kernels.

RTAI was initially based on RTLinux for its HAL but moved to ADEOS as a free alternative. The *Adaptive Domain Environment for Operating Systems* [Yag01] is a so-called nano-kernel (i. e. smaller than a micro-kernel) intended as real-time capable hypervisor for the construction of AMP systems. In the case of Linux, the kernel is moved from privilege level 0 to 1. This causes instructions with direct hardware access to trigger CPU exceptions that are handled by ADEOS. Its virtualization implementation is very specific to the x86 architecture and less depending on Linux. The motivation is to ease OS development by enabling debugging under control of the hypervisor and to execute multiple OS instances. With RTAI, all interrupts are handled with priority in the ADEOS layer and forwarded to the guest OS after hard real-time tasks were executed.

The project Xenomai¹⁰ was started separately from RTAI, merged in 2003 and separated again in 2005 [Ger04]. Xenomai has its own HAL RTOS kernel called *nucleus*. The motivation is to allow an easy migration from RTOS applications developed for other, commercial RTOSs and to integrate them with Linux [Ger01]. This is realized with various *skins* that translate the Application Programming

⁷Online: <http://www.windriver.com> (visited November 12, 2015)

⁸Online: <https://www.rtai.org> (visited November 12, 2015)

⁹Online: https://www.rtai.org/?Archive_announcements&id=32 (visited November 12, 2015)

¹⁰Online: <http://www.xenomai.org> (visited November 12, 2015)

Interface (API) of other systems to Xenomai's own API. Xenomai is actively developed with the latest release 2.6.3 on October 5, 2013 supporting up to Linux 3.10.

RTAI aims for the lowest, technically possible, latency while Xenomai also aims for clean, extensible interfaces. RTAI appears to be more of a research project while Xenomai is developed and embraced by the industry.

Another project, L⁴Linux [HHW98] uses the L4 micro-kernel as hypervisor for a guest Linux system to execute hard real-time applications. The hypervisor uses para-virtualization. Therefore, Linux is modified to use L4 and L4Env interfaces instead of direct hardware access. Mehnert et al. compare the approaches of two separate OS instances with their own address space and a single, real-time-enabled OS instance [MHH02]. In both cases, the integrated Linux system allows to re-use existing software while not taking influence on the timing of the tasks executed in the sub-kernel.

The real-time capabilities of standard Linux, the RT-Patch and various sub-kernel solutions are compared practically by many authors. Brown and Martin conclude that Xenomai provides a better guarantee of timeliness but requires more effort to implement [BM10]. The sub-kernel approaches are in fact able to provide hard real-time promises because the timing depends only on the hypervisor or hardware abstraction layer which is much smaller than Linux and thus, makes a formal verification feasible. Although, an available verification is not advertised by any of these projects. Sub-kernels provide a solution for single-processor systems or where real-time and general-purpose tasks need to share CPUs (e. g. for load-balancing). The isolated task approach is simpler by requiring less changes to the Linux kernel and can be used for systems with enough processors or if real-time tasks utilize a full processor on their own.

Suites Based on Interrupt Abstraction Projects

Several research projects are based on the above approaches. The Hyades project aims for distributed systems that offer both hard real-time for data acquisition and intensive computing capabilities with soft real-time [Cha+04]. After evaluating shielding approaches, RTLinux and RTAI, this project followed the approach to implement a new RTOS *DIC* on top of Adeos. The same group developed ARTiS, a partitioning approach with a new Linux scheduler providing only soft real-time [MM01]. That Linux extension creates two partitions for real-time and general-purpose tasks. In the real-time partition, only preemptible code is allowed to confine the maximum latency [Mar+04]. Tasks requiring system calls are automatically migrated to the general-purpose parting [Pie+04]. The evaluation of ARTiS is done with benchmarks under load [Pie+06] which does not qualify for hard real-time. The last published version¹¹ is from 2005 based on Linux 2.6.12.

¹¹Online: <http://www.lifl.fr/west/artis/> (visited November 12, 2015)

3. Related Work

Ragot et al. present a Linux extension for high performance and real-time computing [Rag+04]. It is related to ARTiS covering the same field of applications and listing some examples.

Betti et al. present the Linux kernel modification *ASMP-Linux* [Bet+08]. This is an approach of shielded CPUs [BR03] to create domains of real-time CPUs and system CPUs. But this work goes further than the shielded CPUs of Brosky and Rotolo and also than ARTiS of Marquet et al. [Mar+04]. A simple interface `/proc/asmp` allows the configuration. Programming is done with POSIX.4 real-time extensions [Gal95]. A special feature is the complete suppression of interrupts (including the Local Advanced Programmable Interrupt Controller (localAPIC) timer) to guarantee hard real-time. This voids preemptive scheduling (similar to the approach presented in this work) but keeps cooperative scheduling. The paper includes extensive benchmarks and evaluation. The current state¹² is a patch for Linux 2.6.19.1 from 2008.

Guiggiani et al. presented the *Realtime Suite*, a collection of Linux tools to create real-time capable I/O applications based on RTAI [Gui+11]. The suite of software, tools and documentation is intended to compete with commercial real-time products. The AMP hypervisor SPUMONE shares the motivation and concepts with RTAI and Xenomai but is implemented for the SH4a architecture (related to MIPS) that is used in many embedded systems developed in Japan [Mit+11]. Twin-Linux is another AMP approach for Linux. It does not aim for real-time, but for the concurrent execution of multiple Linux instances [Jos+10], thus it belongs to the partitioning concepts. Instead of a hypervisor, the guest systems are started with different parameters by the boot manager GRUB which requires no modifications to the Linux kernel but is limited by assigning all Peripheral Component Interconnect (PCI) devices to one system and all PCI Express (PCIe) devices to the other system.

All these projects recognize the advantage of combining a full-featured Linux system with hard real-time tasks. Some of them include partitioning, but none uses a bare-metal execution on isolated processors to reach the lowest possible latency and jitter.

3.3. Partitioning and Isolation Approaches

The above listed approaches realize real-time computing on multi-processor systems with a common OS but provide either only soft real-time (in the case of kernel modifications) or require multiple OS instances. The latter is realized either with deep changes to the GPOS or with an overly complex system architecture. In the following is shown, how these systems and new approaches are adapted to the availability of an increasing number of processors per system. An early mentioning of *isolation* originates from High Performance Computing, where partitioning is used

¹²Online: <http://www.sprg.uniroma2.it/asmplinux/> (visited November 12, 2015)

3.3. Partitioning and Isolation Approaches

to improve the performance of CPU, memory, and I/O and to provide a sort of quality-of-service in heavily loaded systems [VGR98].

The first description of isolated tasks on dedicated processors is the *shielded processor* concept by Brosky and Rotolo. They base on a multi-processor system and reserve some of the CPUs for real-time applications. Depending on the requirements, multiple tasks are scheduled together on a CPUs or, in the most demanding case, a single task owns its processor. The Linux kernel is modified for the affinity of tasks, new scheduling algorithms and some other improvements similar to those provided today by the RT-Patch. They describe the implementation of real-time applications with memory locking, priority-based scheduling, and interrupt and system call avoidance but provide less details about their kernel modifications [BR03]. In comparison to RTLinux and RTAI, the shielded task concept has the advantage of a simpler overall architecture and allows to re-use Linux drivers and protocols and to use high-level languages for the implementation of applications [Bro04]. The concept is still available as part of the commercial RedHawk Linux distribution¹³. These publications are not detailed enough to re-implement them.

In a publication discussing how RCU can be modified for a fully preemptible kernel required for the RT-Patch, McKenney and Sarma also list some ideas for partitioning and isolation [MS05]. They consider isolation as a workaround as long as the RT-Patch is not able to provide hard real-time. In their concept, a migration approach is presented that groups CPUs into real-time and general-purpose partitions [MS05, Sect. 3]. If a real-time task executes a system-call, it should be moved to a non-real-time processor. This implies executing only (hard) real-time tasks on dedicated CPUs and avoiding system-calls on these processors. Other conditions are described that are similar to the isolation approach presented here, but were not realized because it is only drawn as a concept as long as the RT-Patch does not provide sufficient predictability. The proposed implementation seems to be a kernel modification but since the original ambition is a modification of RCU being a part of the mainline Kernel, it is expected that the isolation approach would be pursued in a portable way that allows its inclusion in the mainline Kernel.

Checconi et al. describe an extension of the Linux scheduler with reservation properties. This would not change the user-interface and allows various partitioning schemes [Che+09] but remains a Linux kernel modification still subject to unpredictable latencies.

In a conference talk summary, *CPU isolation* is cited as future goal of the RT-Patch [Edg11]. It appears not to be seen in relation to real-time applications but being a topic that the developers of the RT-Patch should be able to realize with their experience. The state of this project is further explained in a presentation of

¹³Online: <http://real-time.ccur.com/home/products/redhawk-linux> (visited November 12, 2015)

3. Related Work

McKenney given at various Linux conferences in 2013 and 2014¹⁴. The motivation to remove the timer interrupts is derived from energy conservation and High Performance Computing and extended to real-time applications that would also benefit from reduced execution interruption [McK14a]. The presentation introduces the vast number of modifications that led to the tickless kernel presented above.

A project to execute multiple OS instances on top of a hypervisor is frequently promoted by Xenomai developer Jan Kiszka. Citing actual use-cases where more functionality than a RTOS and a harder real-time than a Kernel modification like the RT-Patch can provide, Virtualization with the existing projects KVM and QEMU is evaluated to build an AMP system [Kis09]. The hard real-time guest system uses only a reduced set of resources while the remaining resources remain under control of the GPOS for soft real-time tasks [Kis13b]. These considerations led to the development of *Jailhouse*, a hypervisor explicitly advertized as base for AMP systems executing GPOS and RTOS instances on dedicated CPUs [Kis13a]. The project was publicly announced in November 2013 [Cor13i] and made available as open source software¹⁵. The system is started and initialized by a normal Linux kernel. In the running system, the Jailhouse hypervisor is placed below the Kernel by loading a module. It integrates advantages from Xenomai, RTAI, and earlier discontinued approaches like RTLinux into a modern, lightweight hypervisor based on newest Linux kernel technology. Compared to this work, Jailhouse shares the motivation and goals but uses the different approach of multiple OS instances (AMP).

Partitioning on other operating systems

A precondition of this work is to use Linux as base system. However, similar concepts of partitioning were realized for other OSs, mainly Microsoft Windows. Since the source code is not freely available, modifications are not possible. Still, dual-Kernel approaches (AMP) with hypervisor or HAL solutions [MČ08] were developed based on published driver interfaces and CPU properties or with special development contracts. Most of the following products are or were commercially available. Therefore, their inventors did not publish every detail about their implementation. Similar to other real-time systems, the approaches were initially intended for single-processor systems [TVU98], but were generally extended for multi-processor systems to allow spacial partitioning. In this context, the various suppliers often use the classification as *hard real-time* in a weaker definition of *better than soft* or just *lower measured latency* [Obe+99].

The product *INtime for Windows*¹⁶ from tenAsys is a hypervisor approach that runs the INtime RTOS besides the Windows Kernel. It supports single and multi-processor systems, the latter through partitioning. Shared Memory and shared

¹⁴Online: <http://www2.rdrop.com/~paulmck/scalability/> (visited November 12, 2015)

¹⁵Online: <https://github.com/siemens/jailhouse> (visited November 12, 2015)

¹⁶Online: <http://www.tenasys.com/index.php/overview-ifw> (visited November 12, 2015)

3.3. Partitioning and Isolation Approaches

virtual I/Os are available to create distributed applications on the base of the INtime SDK.

The product *Kithara RealTime Suite*¹⁷ extends Windows with a driver for real-time tasks supporting single and multi-processor systems. The Kitara RTS kernel runs inside Windows and real-time tasks are executed in kernel mode. The API supports interaction with standard Windows programs.

The product *RTX Real-Time Platform for Windows*¹⁸ [RTX12] from InvervalZero is also a partitioned hypervisor approach where a real-time kernel runs besides Windows on dedicated CPUs. The RTX architecture [RTX09] is based on a Windows Driver executing in kernel context. It can share a CPU with Windows or run on dedicated CPUs. The RTSS Environment shares a single RTOS instance with preemptive multi-tasking. Applications using RTX are described with a relatively long cycle time of 2 to 50 ms [XX11; LQ12]. In a comparison of RTX and RT-Linux, RTX is rated better [MČ08] but the applied benchmark and the discussion reveal a weak understanding of real-time principles by the authors.

The product *RTPRO-PC*¹⁹ from ETAS realizes a hypervisor approach to run Windows in parallel to an RTOS on a multi-processor system. The hardware abstraction is closely tied to selected processors and main boards and only available for few certified systems. Its use is demonstrated in a rapid prototyping application [NW12]. *Hyperkernel* from Imagination Systems²⁰ was a similar product for single processor systems, but it is no longer available.

The modifications of the Linux kernel are not suited for the strong definition of hard real-time requiring a formal verification of all possible code paths because of the complexity of the Linux kernel. All presented products for Windows and the Linux extension Jailhouse are AMP systems with a spacial partitioning. They realize a fully isolated CPU that executes only hard real-time tasks while the other CPU(s) execute a GPOS with the full comfort. The realization can either be a hypervisor that starts multiple guest OS instances or a HAL that is implanted into a running GPOS taking over some resources for its real-time control. To the best of my knowledge, the concept to isolate a usual UNIX process from all interruptions to the level of bare-metal execution was not publicly presented by other authors. The alternate concept of hot-unplugging a CPU and rebooting it without notification of the OS – that is outlined in Section 5.4.2 on page 124 but was not implemented – was also not presented before.

¹⁷Online: <http://www.kithara.de/de/produkte/realtime-suite> (visited November 12, 2015)

¹⁸Online: <http://www.directinsight.co.uk/products/venturcom/soft-control-architecture.html> (visited November 12, 2015)

¹⁹Online: http://www.etas.com/de/products/rtpro_pc.php (visited November 12, 2015)

²⁰Online: <http://www.nematron.com/products/legacy/hyperkernel.html> (visited November 12, 2015)

3. Related Work

3.4. Previous publications

The concept of isolated bare-metal tasks was first presented in a state of an early prototype with a preliminary practical evaluation [WLB12]. This publication included a prediction of OS conflicts that led to the analysis presented in Chapter 5. This concept was also presented as poster [Was+12] in conjunction with a HPC cooperation. In 2013, the concept was brought to production quality in a confidential industry cooperation. The result is successfully deployed in automotive Hardware-in-the-Loop (HiL) simulators.

The experience made with low-level benchmarking and real-time interferences supported the research on Inter-Process Communication (IPC) for many-core systems [RW14].

3.5. Supporting Student's Theses

The following students helped advancing the project and wrote their Bachelor's or Master's theses under my supervision (listed in chronological order).

In a more distant project on many-core systems, Pablo Reble researched synchronization mechanisms that scale well to hundreds of processors [Reb09]. He was mentored by Jan Kiszka who develops Xenomai and Jailhouse. The experience from his work together with frequent discussions about his current research founded a basic experience for the implementation of the application support library for real-time capable shared memory communication presented in Section 6.1.

Maximilian Marx ported the Real-Time Operating System FreeRTOS to a Linux process using UNIX signals as timer interrupt [Mar10]. This Bachelor's thesis paved the way for the porting of a RTOS to isolated tasks using user-mode interrupts as shown in Section 6.4.3.

For her Diploma thesis, Lena Oden evaluated the application of the real-time methods of isolated tasks for High Performance Computing [Ode10]. However, in the scope of our institute's possibilities at that time with clusters up to 16 nodes and Symmetric Multi-Processing (SMP) systems up to 8 processors, it was difficult to prove positive effects. However, a promising trend for larger systems could be predicted.

Based on an early prototype of isolated tasks, Raphael Bruns implemented a first real-world application example [Bru10]. His Diploma thesis describes how to realize a software-defined protocol using digital I/O.

Gerhard Wallraf analyzed the x86 System Management Mode in his Diploma thesis [Wal10a]. Although the System Management Interrupts are hardly detectable by system software, he managed to successively cut out all other interruptions until the influence on hard real-time applications could be evaluated.

3.5. Supporting Student's Theses

Sebastian Meßingfeld designed a real-time sensor based on the shared memory Inter-Process Communication for his Bachelor's thesis [Meß11].

Josef Raschen thoroughly analyzed the possibilities of hard real-time inter-task communication as described in Section 6.1 and implemented a portable version [Ras11]²¹. His Diploma thesis describes portable wait- and lock-free algorithms for a wide variety of communication objects for x86 and Advanced RISC Machines (ARM) systems.

Robert Uhl analyzed the x86 interrupt handling and how Linux uses it [Uhl11]. His Bachelor's thesis supported the modification of the Linux source code presented in Chapter 6 and the analyzing of remaining interrupt effects described in Chapter 7.

Jens Dankert researched the implementation of real-time applications [Dan12]. This Master's thesis assisted the application support layer (Chapter 6) and provided experience in related work (Chapter 8).

Stephan Alt implemented a complex real-world application example based on isolated tasks [Alt13]. This Bachelor's thesis is a further demonstration of the applicability and versatility of the concept. It is shortly described in Chapter 8.

During the project, the student assistants Christian Spoo and Sonja Kolen helped in implementing, debugging, system administration, benchmarking and discussing current topics.

²¹This thesis was awarded in the Graduation Challenge 2012 of the German *Fachausschuss Echtzeitsysteme* (Technical Committee for Real-Time Systems)
<http://www.real-time.de/preise/grad2012.html> (visited November 12, 2015)

4. Isolation of Bare-metal Tasks

Even using a Real-Time Operating System (RTOS), scheduling hard real-time tasks is subject to difficult to predict task switch and kernel latencies. Instead of installing a specialized RTOS where the inner structure favors predictability over performance, a different approach is presented to realize hard real-time applications using only the General-Purpose Operating System (GPOS) Linux. Having multiple processors in current systems, a spatial partitioning is selected over temporal partitioning. The Bound Multi-Processing (BMP) (Section 2.3.4) approach is taken one step further: A Central Processing Unit (CPU) is reserved only for a real-time task while other processes and the OS's internal maintenance are confined to the other CPUs. The isolated task is executed bare-metal without operating system (OS) support. This concept can freely be adapted to the application's requirements of timeliness and compute power. The extreme case would be a single CPU for the OS and all others for a single real-time task each.

After servicing an interrupt handler, the Linux kernel checks internal work queues and kernel threads. This can add an unpredictable overhead to the execution time of each interrupt handler. Besides the difficult to account frequency of interrupts, their execution time increases the inaccuracy of the Worst-Case Execution Time (WCET) which must be estimated with a sufficient safety margin. The same holds for the task switch and system call because their handling in kernel-mode is managed accordingly.

This Chapter presents the implementation of bare-metal tasks for x86 systems running Linux. The concept allows to use an arbitrary Linux distribution (possibly with Real-Time Preemption Patch) with all updates and all available libraries. This eases the porting of complex applications with multiple threads that may have high computing demands as well as hard real-time requirements with latencies below 10 μ s. Starting with an off-the-shelf Linux system, standard tools are used to modify a user-mode process to run uninterrupted on a dedicated CPU. This Chapter demonstrates, that the goal of a user-mode process fully controlling its CPU can be reached without modifications to the Linux kernel. However, the system stability is weakened. This will be addressed in Chapter 5.

4. Isolation of Bare-metal Tasks

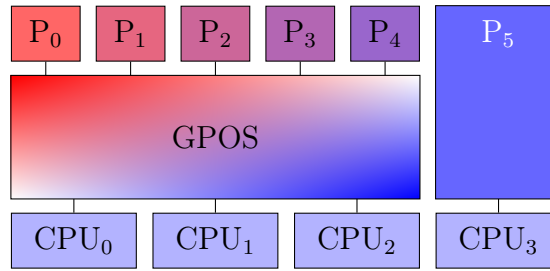


Figure 4.1.: Concept of a strictly isolated task P₅ on a dedicated processor.

4.1. General Concept

Having a multi-processor system running a single OS instance, a standard process will be *isolated* on a dedicated CPU to run its code uninterrupted like a bare-metal single-processor system (Fig. 4.1). To retain full control over that processor, the isolated task will not be allowed to use services of the operating system because they may invoke further Kernel functions and their execution time is not predictable. This allows a formal verification of the hard real-time tasks while the other CPUs can fully utilize the power and convenience of a GPOS. Generally, the partitioning of multiple processors supports an arbitrary number of isolated tasks provided that at least one CPU remains available for the operating system and its services.

The operating system is restricted to a smaller number of CPUs than before, similar to the CPU Hot-plugging feature where some CPUs can be switched off. Except from the reduced number of available CPUs, everything is executed as usual. The normal processes are load-balanced between the existing processors and may use the operating system's support for soft real-time scheduling.

The isolated process runs bare-metal on its processor, which means it is not under the control of the operating-system (Fig. 4.1). Its timing can be verified like on a single-processor system. However, the interdiction of using operating system services like Inter-Process Communication (IPC) and device drivers locks in the isolated task which reduces its applicability for useful workloads. This restriction can be bypassed by using shared memory communication and direct hardware access as covered extensively in Chapter 6. Here, the uninterrupted operation is analyzed in practice using the Hourglass benchmark (Section 2.3.5) with small synthetic workloads. This serves as first validation and to identify remaining interruptions. The final verification is formally based on the processor documentation.

Brandenburg et al. examine the influence of interrupts onto the execution time. They identify three choices to limit interrupt-related delays: Splitting the handlers into short interrupt handlers and deferring the bottom halves into work queues, using only polling to avoid Device Interrupts (DIs) but not the others, and masking all interrupts during real-time task execution. Their conclusion is that interrupts can not be avoided completely and therefore must be included in the schedulability analysis

[BLA09]. This isolation concept promotes a different solution by using time-triggered bare-metal tasks on dedicated CPUs and handling interrupts on the other CPUs or a multi-processor system.

Synchronous control transfers can be voluntary (e.g. jumps) or involuntary (e.g. exceptions) because they are detected while executing an instruction. Other interrupts are *asynchronously* triggered by devices (Device Interrupts: Interrupt Requests, IRQs and Timer Interrupts), other CPUs (Inter-Processor Interrupts, IPI), or the firmware (Platform Interrupts). Synchronous events are in the responsibility of the programmer and can be confined. The duration of a branch or function call is static except for the variation caused by caches and shared resources. The time for executing a called function can be analyzed for known functions. Unknown functions must be replaced with analyzable code. System calls and software interrupts must be avoided during time-sensitive tasks because those are functions of the OS that can not be analyzed with all side-effects. The goal is to deactivate asynchronous interrupts whose length and frequency is out of the user's control. The use of *full isolation* employs a time-triggered, uninterrupted execution model which is easier to manage and analyze.

In the remainder of this Chapter, the successive steps are described to set up a completely isolated process for the x86 architecture running a standard Linux OS. Initially, the configuration of the hardware (Firmware setup) and operating system is optimized for real-time applications. These methods are reasonable for every real-time system. Further follows the partitioning of the CPUs into a system share and dedicated CPUs for isolation. All Device Interrupts must be routed to the system partition. The tasks that will be isolated are moved to their CPU and must be configured and initialized to avoid page faults and other synchronous interrupts and exceptions. Finally, the full isolation is reached by blocking all remaining interrupts (mainly Timer and Inter-Processor Interrupts) and adhering to the coding standard of not using system calls.

4.2. Setup and Partitioning

After being powered up, a single CPU (the Bootstrap Processor, BSP) starts executing the firmware initialization code from a read-only memory. This program is called BIOS (Basic Input/Output System) or UEFI (Unified Extended Firmware Interface). The hardware initialization starts a boot manager that selects and loads an OS from mass storage and hands over control to the boot code of the OS. In multi-processor systems, this is all executed with only a single CPU active. It is the OS that starts and initializes the other processors (called Application Processors, APs).

Many features of the CPU, the memory, and the connected devices can be configured in the firmware setup program. These settings may change how the OS sees the hardware. This way, behavior can be affected that can not be changed by other

4. Isolation of Bare-metal Tasks

means. Other adjustments can be given to the OS by the boot manager or later be modified in the running system. The partitioning of processes and hardware interrupts intervenes with the OS's data structures. Linux provides standardized functions for these essential methods, which are also available on most other OSs. The settings and methods presented in this Section are also advisable for all other real-time systems.

4.2.1. Hardware Configuration

The firmware can be configured by entering a setup program shortly after powering on a system (often by pressing the keys DEL or F1 at the appropriate time). All following methods improve the general predictability of the execution time of code. This setup is the same as for RTOSs on the x86 architecture [Zha+05] (e.g. RTX [RTX09], Xenomai¹).

Simultaneous Multi-Threading (SMT)

This feature splits a physical CPU into multiple logical processors (Sect. 2.1.9) that share some resources like First Level Cache (L1\$) and Instruction Decoder. This feature is called Hyper-Threading for Intel processors and Cores of a Compute Unit on AMD processors. Hardware threads sharing resources impact each other if one of the shared units is a bottleneck for both. It is expected, that the impact on hard real-time tasks spoils the predictability up to the point where hard real-time can not be guaranteed.

If the system supports Simultaneous Multi-Threading (SMT) it should be deactivated in the firmware configuration. Alternatively, the Linux kernel can be instructed to boot only the number of physical CPUs via the boot parameter `maxcpus=N`. It will select only distinct physical CPUs in this case.

Virtualization Support

Since 2006, both Intel (VT-d) and AMD (AMD-V) processors provide hardware-based support for virtualization, the Virtual Machine Extensions (VMX) [Intel13c; AMD12b]. Further, memory and Input/Output (I/O) virtualization may be supported by the chip set (e.g. IOMMU, I/O Memory Management Unit). The technology is based on an additional layer of indirection for memory and I/O addresses. Current Linux versions activate these features even for non-virtualized systems. The support should be deactivated in the firmware configuration to reduce the I/O latency.

¹Online: <http://xenomai.org/2014/06/configuring-for-x86-based-dual-kernels/>
(visited November 12, 2015)

Energy Saving and Sleep Modes

Recent processors can change their frequency in a wide range to reduce power and heat (Intel Speedstep [Intel13c, Chap. 14], AMD Performance Control [AMD12b, Chap. 17]). The decision about when to reduce the frequency or how to balance performance and power consumption can either be delegated to the firmware (e. g. BIOS) or be controlled by the OS. For real-time systems, it is crucial to keep the control in the OS because the transition of operating modes induces extra latency and the instructions per time depend on the CPU speed.

Some settings in this regard can be configured in the firmware while others are accessible in the running system. The firmware configuration generally supports a setting to maximum performance that should be selected for real-time systems. If the Linux system is configured with power management, the governor² should be set to “performance” to force the OS to keep all CPUs at the maximum frequency [Emd11]. Additionally, driving processors to their maximum performance has the advantage to keep them at their maximum frequency avoiding the latency to power them up when needed.

System-Management Interrupt

Being a Platform Interrupt (PI), the firmware initializes this interrupt and provides the handler. The x86 architecture includes the System Management Mode (SMM) mechanism to hide the code from users by making the memory region inaccessible. When a System Management Interrupt (SMI) is triggered, *all* CPUs suspend their work while one of them handles the interrupt.

The SMI may be used for various tasks, e. g. power-management, thermal care, OS interaction, and emulation of legacy hardware. It depends on the firmware and its settings, which functions are implemented with SMIs. Some systems, especially laptops, have time-triggered SMIs that occur with a period between 2 Hz and every 15 Seconds [Wal10a]. Other triggers for SMIs are the system temperature (in case the OS misses to activate a fan), the failure of a fan, user interaction such as display brightness on laptops or case intrusion detection on servers. The x86 architecture promises to be backwards compatible, thus still running 1980’s DOS. To support such OSs where only USB keyboards are present, the firmware uses SMIs to provide a USB driver and to translate input events to a PS/2 compatible interface.

The time of SMI handling is very long. This is due to the complex mode switch including the synchronization of all CPUs and the access to the hidden SMM memory region that is switched in the chip set. Analyzes have revealed SMIs with times usually above 100 000 CPU cycles (50 μ s) and up to 250 ms [Wal10a]. This would spoil every hard real-time application.

²/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor and respectably for other CPUs.

4. Isolation of Bare-metal Tasks

Only few settings with regard to SMIs can be changed in the firmware configuration. Often, the legacy USB keyboard support can be deactivated. This may result in the boot manager no longer being able to use the keyboard. Once the system is booted, the OS's USB drivers take care. There is a configuration register in Intel chip sets to deactivate the SMI, but this register can be locked by the firmware after booting to irrevocably disable its modification. The detection of SMIs can only be done by excluding all other interruptions, e. g. by fully isolating a CPU and watching for remaining gaps in the execution of a short hourglass loop (Section 2.3.5) [Wal10a]. If a given system shows frequent time-triggered SMIs, it is not suited for real-time applications [Zha+05].

4.2.2. Linux Settings

The Linux kernel can be configured in various ways, most notably by build-time configuration, boot parameters, and run-time settings. The *build-time configuration* allows to omit features that will not impose any overhead in the running system. But changing those setting requires recompiling the Kernel. *Boot parameters* allow configuring deeply invasive settings by rebooting. A large number of *run-time settings* finally can be accessed via the `/sys` and `/proc` virtual file-system hierarchies. Those are usually limited to the administrator's access and allow the fine-tuning of the system's behavior. Some of the settings presented below can be modified by various methods.

NMI Watchdog

A watchdog timer [Wil06, Sect. 15.17] frequently checks the correct operation and may warn the user or restart the system. Linux provides an Softlockup Detector³ that watches over all CPUs for blockings. It issues a Non-Maskable Inter-Processor Interrupt (NM-IPI) to unblock or at least notify the user.

As the projected isolation of a process on a CPU will preclude the OS and its interrupts from executing on that CPU, a watchdog timer would interfere and should be deactivated. This can either be configured at build-time or run-time. The latter is done by writing the value 0 into the file `/proc/sys/kernel/nmi_watchdog`. If this file does not exist, the watchdog was deactivated at compile-time. Since Linux 3.12, the time-out of detected blocked tasks can be configured in the file `/proc/sys/kernel/hung_task_timeout_secs`. Writing 0 to this file avoids warnings in the system log and Non-Maskable Interrupts (NMIs) on isolated processors.

³Linux file `Documentation/lockup-watchdogs.txt` (Linux 3.10)

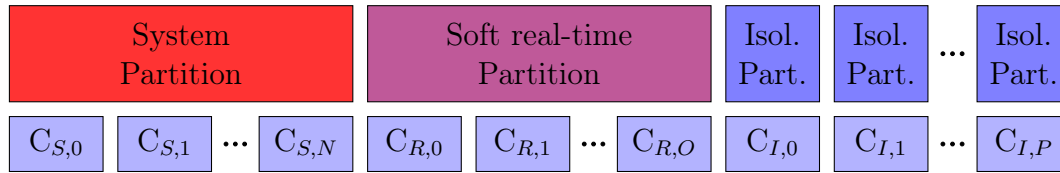


Figure 4.2.: Example setup for multiple partitions.

Time Keeping

A Linux system uses Timer Interrupts on each CPU to keep their meaning of wall clock time. Different modes using either the Local Advanced Programmable Interrupt Controller (localAPIC) timers or High-Precision Event Timer (HPET) are available. There is a routine checking the cadence of all CPUs that may interfere if it detects that the OS is not running on an isolated CPU. This will be addressed in Section 5.2 on page 114. The boot parameters `tsc=stable` and `clocksource=tsc` instruct the Kernel to use the localAPIC method and not to wrongly value it as unreliable. This avoids warnings in the system log and possibly NMIs to the isolated processors.

4.2.3. Partitioning

Like in a Bound Multi-Processing (BMP) system, assigning tasks and interrupts to dedicated CPUs increases their predictability. The extreme variant of executing only a single process or thread on a CPU prevents all task switching overhead. The disadvantage is a weak utilization of the compute resources of the whole system. Having more than 10 processors in recent x86 systems allows to partition the system into some CPUs dedicated for the least latency while the remaining CPUs can build a partition that utilizes the included CPUs as usual at the cost of only the generally available soft real-time latencies. The *System Partition* (Fig. 4.2) comprehends one or multiple processors and executes the basic system services (called *remaining system* in the context of this work). An optional soft real-time partition can include additional CPUs for prioritized work. The isolated partitions include always a single CPU and will execute a single process or thread. The partitioning can be adapted to the needs but must remain constant for the run-time of the application.

The remaining system lacks the CPUs dedicated to the hard real-time tasks like if they were switched off by the Hotplugging system. The current hotplugging implementation in Linux does not support removing CPU #0 (the Bootstrap Processor, BSP), because this CPU executes some internal services that can not be moved to other processors so far. Likewise, the CPU #0 should not be selected for isolation. Since this CPU remains always functional, it can be selected to handle all interrupts. If a certain interrupt is crucial for the application (e.g. the network interrupt for low-latency transfers), it can be bound to another dedicated CPU.

4. Isolation of Bare-metal Tasks

Changing the interrupt and process affinity requires administrator privileges. All the following methods can either be executed from the shell (e. g. by an initialization script) or from the application itself by writing to the appropriate pseudo files. Instead of running with administrator privilege, the application can also be configured to use Linux Capabilities [HM08]. This would reduce the security risk of running an application with maximum rights.

Interrupt Affinity

On x86 multi-processor systems, hardware interrupts are distributed by the Input/Output Advanced Programmable Interrupt Controller (I/O APIC) (Section 2.1.9 on page 32) to CPUs where a localAPIC accepts the notifications and manages the interruption of the current instruction flow. The I/O APIC has a routing table that allows to assign where each IRQ should be handled. By default, the interrupts are sent to an arbitrary target negotiated in hardware. In Linux, the system service `irqbalancer` reprograms the routing table frequently to distribute the load caused by interrupt handling. Before changing the IRQ Affinity to match the selected partitioning, the service `irqbalancer` must be stopped to avoid it disturbing the intended setting.

Linux provides the interface `/proc/irq` to set the affinity of each Interrupt Request (IRQ) by writing a bit mask into the file `smp_affinity`. The `smp_default_affinity` can be configured for devices that are initialized later to get this bit mask as default. If a different OS does not offer an interface to the I/O APIC routing tables, the interrupt affinity can also be configured directly in the hardware. The configuration registers are memory-mapped and can be accessed by a privileged process.

Having removed the handling of Device Interrupts from the dedicated hard real-time CPUs, there can still be delivered Inter-Processor Interrupts (IPIs). There is no hardware means of routing them because they can be addressed directly to a particular CPU. This will be handled in Section 4.4.

System Partitioning

Processes and threads (or generalized “tasks”) are a concept of the operating system and managed in its data structures. Thus, configuring on which CPUs a task is allowed to run requires using an interface of the OS. Restricting the running processes to the CPUs of the system partition and moving processes or threads dedicated for isolation to their CPUs can be accomplished in Linux systems with the means of CPU Affinity or CPU-Sets. The CPU Affinity is not a POSIX standard but should be portable to other operating systems. CPU-Sets are a Linux feature that supports assigning processes and threads to partitions. These partitions can be spread over CPUs and Non-Uniform Memory-Access (NUMA)-nodes. Both mechanisms are

applicable to realize isolated tasks. However, CPU-Sets were used in subsequent implementations of the concept because of their convenience.

CPU-Sets are part of the control-group feature [LINUX:cpuset.txt]⁴. They are created as a hierarchy of groups. The interface is a virtual filesystem mounted at `/cpuset/` or `/sys/fs/cgroup/cpuset/`⁵. New sets can be generated by creating a directory. The configuration is handled by writing to the virtual files within the set directories. Each CPU-Set can be assigned CPUs by writing to `cpuset.cpus`. Then, processes and threads can be moved to a CPU-Set by writing their TIDs (Task-IDs) to the file `cpuset.procs`. For a process with multiple threads, the threads can be assigned to different CPU-Sets. Child processes and threads created by tasks in a CPU-Sets remain in their parent's set. The list of tasks in a set can be checked by reading `cpuset.procs`.

For the isolation concept, multiple CPU-Sets are created, one for each partition. The sets for isolated tasks each contain only a single CPU to run on. At least one system partition containing CPU #0 must be created. All running processes must be moved to that CPU-Set.

An alternative implementation of process partitioning can be realized with the interface CPU Affinity. A process can change its own affinity to a bit mask defining the CPUs it is allowed to run on. Threads of a process can be configured to different affinity masks using `pthread_setaffinity_np()`. With appropriate privilege, the affinity can also be changed for other tasks. According to its documentation⁶, a new process or thread inherits the affinity of its parent. Similar to using CPU-Sets, all existing tasks must be configured to run on the CPUs allocated for their partition. The CPUs where isolated tasks will be executed must be freed from other tasks. This interface is portable to other operating systems, either supporting POSIX or similar functions (Windows uses `SetProcessAffinityMask()` for this concept).

Both interfaces can be accessed from the shell to set up the system in a script or from within the application. The shell interface for CPU-Sets is the previously described pseudo directory hierarchy described above. This can also be used from arbitrary programming languages by creating directories and writing to files. The CPU Affinity can be changed from the command line with help of the program `taskset`.

In Linux systems, many kernel threads (Section 2.2.5 on page 50) exist besides the user-mode processes. They are scheduled by the Kernel like other tasks, but execute in kernel-mode. They are used for the internal maintenance of data structures and for asynchronous I/O. Most of them are blocked for longer times and only activated when needed. These kernel threads can be divided into two types: The so-called “pinned” kernel threads are tied to a CPU and generally exist multiple times, once

⁴Documentation/cgroups/cpusets.txt (Linux 3.12)

⁵The actual mount-point can be found in `/proc/mounts` searching for “cpuset”

⁶man-page `sched_setaffinity()`

4. Isolation of Bare-metal Tasks

for each CPU. The other kernel threads can be executed on an arbitrary CPU like normal tasks.

The distinction between user-mode process and kernel thread can be made with help of `/proc/<pid>/cmdline`. If this file is empty, the PID belongs to a kernel thread. Otherwise, it contains the command line that was used to load that process. The command line tools `ps` and `top` indicate kernel threads by displaying their name in square brackets (e.g. `[kthreadd]`). Using `sched_getaffinity()` or by reading `/proc/<pid>/status`, the affinity bit mask can be identified. Kernel threads with only a single bit set are “pinned” (e.g. `[kworker/0]`). All non-pinned kernel threads should be moved together with the user-mode tasks.

Pinned kernel threads should not be changed in their affinity and can not be moved to other CPU-Sets. But they are blocked on their dedicated CPU until they are activated by other code on that particular CPU. Since isolated tasks are not allowed to use operating system interfaces, they will not activate pinned kernel threads on their CPU. It is assumed that when the isolated task takes full control of its CPU, pinned kernel threads are not harmed by excluding them from execution.

Processes can change their CPU Affinity as well as their assignment to a CPU-Set. In High Performance Computing (HPC), it is a common technique to use the CPU affinity to distribute the work over available CPUs and to keep the code close to its memory (especially in NUMA systems). But generally, programs will not change their affinity or CPU-Set. It is a good convention to let the administrator decide if a program should be balanced by the OS or fixed to dedicated CPUs. For the isolation concept, it is crucial that no program interferes with the partitioning. In all experiments, it was never observed that a program escaped from its partition.

4.3. Process Setup

The isolated task can be a process or a thread of the main application. Processes have the advantage of memory protection, thus they have a reduced risk of faults injected by other parts of the application via shared memory. Using processes, they can still interact using shared memory segments but other parts of the memory can not be accessed by a different execution context. Programming with threads is easier but requires additional care for synchronization and protection from concurrent access to shared data structures.

An isolated task can be a stand-alone application or part of a larger one. The latter case is reasonable to assemble different tasks that interact with each other. In a larger application, a manager task can be implemented to start and handle isolated tasks (Fig. 4.3). A collection of isolated hard real-time tasks, an optional manager task and other (soft or non real-time) processes and threads are regarded as a *real-time application*. By carefully assigning CPUs to applications, this concept can be extended to multiple applications running concurrently on the same system.

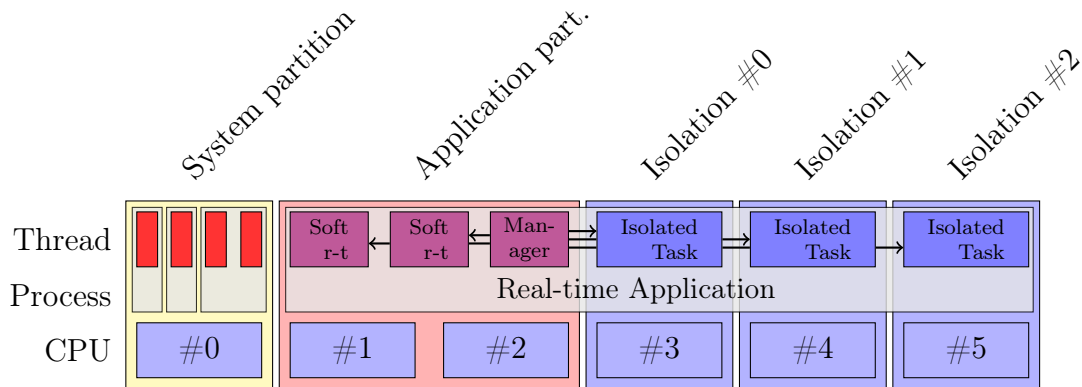


Figure 4.3.: Example application with manager task controlling multiple threads in its partition and multiple isolated task in dedicated partitions.

In the following and without loss of generality, only a single real-time application is executed on a system.

Since the isolated task are not allowed to use OS interfaces (system calls), all allocations and initializations must be done before the application enters its time-critical part. The isolated tasks are separated into a setup phase and real-time execution. The methods presented here are mostly generally applicable to all real-time applications⁷ [BYT12; Fin01; Duv09].

4.3.1. Moving to the Dedicated CPU

The first action after creating a new thread or process dedicated for isolation should be moving it to its CPU. The assignment of a task to its CPU is done by the same means as the partitioning explained above (Section 4.2.3). This can be configured by the task itself or from the outside by the manager task that started the isolated task (Fig. 4.3).

Experiments using the CPU-Sets feature observed problems on current Linux version when moving threads between CPU-Sets when another task is completely isolated (i.e. in its *real-time execution* phase). Therefore, all isolated task must be started and set up before one of them enters the isolated state.

4.3.2. Synchronization

Inter-Process Communication (IPC) will be a topic in Section 6.1 further below. However, for the setup of isolated tasks, a means of synchronization is required. This method must work without relying on the OS after its initialization. Shared memory

⁷Online: https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application (visited November 12, 2015)

4. Isolation of Bare-metal Tasks

is suited as it is a hardware feature which works by referencing the same physical memory page from the virtual address space of different processes. Threads of the same process can share global variables naturally. For different processes, the UNIX standards System V and POSIX provide their variant of shared memory which are both supported by Linux [Ker10, Chap. 48 and 54].

Shared memory provides the lowest latency between two processors. One task can wait actively (busy waiting, polling) on a shared flag until the other one changes (e. g. increments) that variable. On multi-processor systems, the time between changing and detecting that modification is caused by the store-back from the First Level Cache to the level shared with the other processor. This can be the Last-Level Cache or main memory (Section 2.1.9). Although functionally irrelevant, the placing of communicating tasks on the same multi-core processor or distinct ones has impact on the synchronization latency.

Generally, the access of multiple tasks to a shared resource must be coordinated. If only a single writer changes a flag, multiple readers can wait on that flag without further synchronization. On the x86 architecture, single read and write accesses to integer variables up to the bit-size of the architecture (32 or 64 Bits) are always atomic. More complex synchronization primitives (Mutex, Semaphore) can be realized in the user-mode using atomic operations (Section 2.1.9).

4.3.3. Memory Management

Newly allocated memory regions are usually only initialized in the page-tables but not backed with physical memory at the time when the allocation function returns. Linux implements a first-touch approach: The pages for virtual addresses are prepared but marked non-present because they are not yet assigned to physical page frames. On the first access, the CPU exception *page-fault* is triggered. This *minor* page fault will choose a physical page frame (and free it if occupied) and connect it to the virtual address.

To avoid the page fault handler to be executed during the hard real-time execution, all dynamic memory regions must be allocated and first-touched at initialization time. The first assignment to all virtual addresses is sometimes called “pre-faulting”. It suffices to assign a default value to one memory address of each virtual page of 4 KiB. The stack must be handled in a similar way because it grows by function calls and can extend to new pages not used before. For the stack, it suffices to call a function containing a large local array. But care must be taken that the compiler optimization does not waive this function or its data initialization.

If the total amount of allocated memory exceeds the physically available memory, the OS can swap pages to background storage (demand paging, Section 2.1.3). In that case, the page table entries remain in place but are marked non-present. On the next access, a *major* page fault will replace a page-frame with the displaced data from background storage and mark the page as present. The accessing process can

continue without noticing the swapping mechanism. But for real-time applications, the latency for swapping back pages causes an unpredictably long blocking of the execution. The demand paging can be avoided for certain regions of memory by marking them as *locked*. The POSIX standard offers the function `mlock()` to lock a region and `mlockall()` to lock all of a process's pages into memory [Ker10, Sect. 50.2]. For isolated tasks, all dynamically allocated memory must be initialized and all accessed memory regions (dynamically allocated, stack, heap, code) must be locked into memory.

If the memory requirement is larger, the use of huge-pages (Section 2.1.7) could be considered. This would reduce the Memory Management Unit (MMU) overhead and Translation Look-aside Buffer (TLB) usage [ALL12]. But since the hard real-time task should restrict itself to the cache (as shown later), it is not advisable to use large amounts of memory.

4.3.4. Implementation

As already mentioned in the previous Sections, tasks in isolation are not allowed to use system calls. To enable a formal verification of all running code, all calls to libraries should also be confined as much as possible. Particularly, all utilized functions must be checked for not including system calls or unpredictable execution times.

The basic structure of an isolated task is an initialization phase before the activation of the full isolation and an endless loop repeating its work until notified to exit. In the initialization, the task is moved to its CPU and initializes its memory (locking and pre-faulting). The transition to the hard real-time execution can be managed by a task of the application using shared flags.

During real-time execution, the memory footprint should be kept as small as possible to keep most of the data in high cache levels near to the processor. Shared variables should be used only when required because they must be transferred via the Last-Level Cache or main memory between CPUs. Read and written shared flags should be placed in different cache lines to avoid the bouncing of cache line between different First Level Caches. Even shared variables used by different CPUs must be placed in dedicated cache lines to avoid *false sharing*.

The real-time jobs are called repeatedly in an endless loop (time-triggered paradigm). Thus, it is possible to slow down the next iteration using the time-stamp counter and to call certain functions only every N^{th} iteration (Listing 4.1). Isolated tasks can also be held actively waiting for a notification to execute some work. The timing resolution depends on the WCET of the called functions.

When notified to terminate the real-time execution phase, the loop is left and all resources should be gracefully returned. Memory allocated and locked will be automatically deallocated (and implicitly unlocked) when the process or thread ends. Generally, it is considered good programming style to undo all allocations

4. Isolation of Bare-metal Tasks

```
extern unsigned volatile stop; /* global flag: end of appl. */

void task1(void)
{
    unsigned iter = 0; /* iterator variable */
    uint64_t t, tnext; /* curr. time and end of time slice */

    setup(); /* setup thread and start isolation */

    rdtsc(&tnext);
    while (stop == 0) { /* execute until notified to quit */

        func1(); /* call every LOOP_TIME cycles */

        iter++;
        if (iter > 10) {
            func2(); /* call every 10*LOOP_TIME cycles */
            iter=0;
        }

        tnext += LOOP_TIME;
        do {
            rdtsc(&t);
        } while (tnext < t) /* wait for end of timeslice */
    }
    setdown(); /* terminate isolation */
}
```

Listing 4.1: Task Example using the time-triggered paradigm.

and lockings. When the thread or process exits, the isolated CPU remains with no assigned task. It is possible to create a new task and restart from the beginning. As described so far, the concept implements BMP with a single task on a CPU and regarding all real-time programming advice. In the next Section, the full isolation will be activated.

4.4. Full Isolation

As explained in Section 2.1.5, only interrupts (and exceptions being a subclass of them) can asynchronously interrupt program execution. Those are Device Interrupts (DIs) by the hardware (i. e. IRQs), the Timer Interrupt (TI), and IPIs from other CPUs. Synchronous interrupts are caused by the code itself (i. e. system calls and

exceptions) and must be excluded by implementing the isolated task accordingly. The Platform Interrupts (PIs) (such as the SMI) are asynchronous but were already regarded and excluded on the utilized systems.

A brute-force approach to block all maskable interrupts is to clear the Interrupt Flag (IF) with the `CLI` instruction. This is usually forbidden for user-mode processes but can be allowed on Linux systems with a call to `iopl(3)` if a process is run by `root` or has the appropriate capability.

The IF masks the asynchronous interrupts from the hardware (IRQs) and other CPUs (IPIs). Synchronous interrupts triggered by the running code (software interrupts: `INT N` and system calls: `SYSCALL`, `SYSENTER`) and CPU exceptions are not maskable. As explained in Section 4.3.4, the task in isolation is not allowed to issue system calls and must be programmed carefully to avoid CPU exceptions and page faults. Further, the Non-Maskable Interrupt (NMI) is not blocked by the IF. It can be triggered by hardware (e.g. a watchdog timer) or software (Non-Maskable IPI). The watchdog must be deactivated (Section 4.2.2). Non-maskable IPIs are used after severe errors in the Linux kernel. Their safe exclusion is handled in Section 5.2.3.

If a standard process masks its asynchronous interrupts without further preparation, it is very likely that the system will block completely because other tasks and interrupts on that CPU will no longer be serviced. For short periods of time, this can be tolerable. But for the execution longer than a Minute, the methods described above will address these risks. The most important precondition is the partitioning of the system with removing all IRQs and processes from the isolated CPUs to ensure their further ability to be executed.

An even stronger means to inhibit all interrupts is deactivating the localAPIC [Wal10a] to a *soft-off* state. This blocks even NMIs from being serviced. The only interruptions that can still occur are PIs (i.e. the SMI). However, this was not required in all experiments.

Since Device Interrupts are routed to other processors, the IF or deactivation of the localAPIC are only required to avoid interrupts sent from other CPUs and from the timer of the localAPIC. Therefore, an alternative implementation without using the IF would be to deactivate both IPIs and the local timer. The local timer can easily be reprogrammed by writing to the memory-mapped configuration registers of the localAPIC which can be accomplished by a sufficiently privileged user-mode process. Constraining the Linux kernel from sending IPIs to the isolated processor requires a modification of the Linux kernel. This will be addressed in Section 5.2.3.

4.5. Analysis

According to the documentation of Intel [Intel13a, Sect. 3.5] and AMD [AMD12a, Sect. 3.7] processors, the control flow is sequential, i.e. the instruction pointer can not be changed directly. Instructions implicitly affecting the control flow are voluntary

4. Isolation of Bare-metal Tasks

jumps (`JMP`), branches (`Jxx`, e.g. `JLE`, jump if less or equal), function calls (`CALL`), and the return from functions and interrupt handlers (`RET` and `IRET`). The only other events causing a divergence from the sequential instruction execution are interrupts and exceptions. Interrupts can be asynchronously triggered by the hardware (via I/O APIC or directly from the localAPIC) or synchronously with the instructions `INT`, `SYSCALL` and `SYSENTER`. Exceptions are always synchronous, i.e. triggered as side-effect of an instruction.

All deviations from a sequential code path caused by instructions and synchronous interrupts and exceptions must be predictable to make the entire execution time predictable. If state-dependent branches use only conditions resulting from previous execution, they become predictable. If they depend on global variables influenced by other processors or hardware registers (I/O), care must be taken in validating all possible code paths. All called functions must be available for analysis, hence available in source code. This is given for all functions of the application itself. Library functions must be included in the verification, their use should be reduced to a minimum. Synchronous interrupts and exceptions must be avoided by carefully programming because they execute kernel code⁸. That is available for analysis, but very hard to evaluate for its execution time because of its complexity. The asynchronous interrupts are routed to other processors (IRQs), masked by the Interrupt Flag (IPIs), or deactivated (localAPIC timer). Only the Non-Maskable Interrupt (NMI) and the System Management Interrupt (SMI) can arrive, but they were prevented by system configuration.

From this follows that the OS can no longer interrupt the isolated task (cooperative scheduling) and the isolated task does not use system calls. Thus, the inner privilege rings (Section 2.2.2) have no control over the user-mode task. The result can be modeled as a non-preemptive execution on a single-processor system. The application is executed bare-metal without supervision by and support of an OS. Hence, existing scheduling research for this type of time-triggered applications often used in embedded systems is applicable.

Besides the remaining risk of being interrupted by NMIs and SMIs, hostile applications could interfere with the hard real-time tasks on isolated processors. It is still required to examine all applications for improper actions like escaping from their CPU affinity or CPU-Set or changing the OS configuration to unwanted settings (the extreme would be shutting down the system). But real-time systems must control to a certain extend what runs concurrently. This isolation concept removes the influence of external, unknown code paths from hard real-time execution, but does not protect from hostile programs and attacks. This could be probably accomplished by executing virtual machines in non-isolated partitions. The potential problem can be illustrated with memory management: Physical memory can go offline (as also CPUs can) [Edg13a]. This would lead to the system moving memory used by the

⁸In Section 6.4.1 will be shown how interrupts can be used without Kernel code being executed.

real-time task to different physical addresses although it is locked causing page faults with unpredictable latency in the hard real-time tasks.

So far, it is proved that it is possible to formally verify all code paths in the hard real-time tasks. But for transposing a given code path to execution time, the instruction latencies must be known. Unlike RISC processors for embedded systems, the execution time of an instruction on recent x86 processors is not constant due to a long pipeline, multiple execution units (super-scalar out-of-order execution), and different cache levels. The prediction is further complicated by shared functional units like Last-Level Caches, main memory and devices where the code executed concurrently on other processors must be regarded, too.

Hence, for the Control Flow Graph (CFG), unbounded execution delays in waiting loops can be ruled out because all running code can be formally verified. What remains is unpredictability in the execution time of basic blocks due to hardware effects from the caches and shared resources. A first impression will be obtained in the following practical evaluation. The detailed analysis and opportunities to get a tighter estimation of the worst-case execution time depending on the multi-processor architecture will be presented in Chapter 7.

4.6. Practical Evaluation

According to the analysis, the code executed in full isolation is not interrupted by the OS. To demonstrate the gain of a reduced worst-case latency, a practical evaluation of different degrees of partitioning and isolation was done with a benchmark similar to Hourglass (Section 2.3.5 and Listing 2.1 on page 68). In this test routine, a small loop repeatedly samples the time-stamp counter that increases with every CPU cycle. During all following measurements, the CPU executes with maximum speed. Therefore, the cycle counts can easily be converted to Seconds by dividing them by the frequency.

The following results were obtained with an early prototype of the concept implemented in C. The test system is a desktop computer (System `xaxis`, Appendix A.2.1 on page 207) with an Intel Core i7 920 2.667 GHz with 3 GiB RAM. The OS is Linux 2.6.27⁹. The minimum and average loop times are below 30 cycles on an idle system, therefore the following benchmark uses a threshold of 50 cycles. All loop iterations longer than that are counted as gaps. The *idle* system was running a graphical user interface, but left without further interaction. The *load* scenario consists of four CPU intensive processes, a memory intensive process using a 256 MiB array, two processes doing file I/O (with read/write and memory-mapping), and a heavily forking process to engage the Kernel. The benchmark was executed for 60 Minutes each with *fair scheduling* without further real-time programming measures, *partitioning* to execute solely on a CPU, and *full isolation* by additionally

⁹Fedora 10 (64 Bit) with Linux kernel version 2.6.27.24-170.2.68.fc10.x86_64

4. Isolation of Bare-metal Tasks

System	Benchmark	Avg. loop	Max. gap	Total gap	Plot
idle	fair scheduling	25.16	2 894 613	0.71 %	Fig. 4.4
idle	partitioning	25.15	36 897 576	0.72 %	
idle	full isolation	24.76	166	0.000 <i>ppm</i>	
loaded	fair scheduling	74.84	144 043 058	65.3 %	Fig. 4.5
loaded	partitioning	26.14	123 790	0.6 %	Fig. 4.6
loaded	full isolation	26.13	1 135	0.01 %	Fig. 4.7

Table 4.1.: First evaluation on Intel Core i7 920 (in CPU cycles).

clearing the Interrupt Flag (Table 4.1). The benchmark program collects offset (time-stamp) and length of detected gaps but the size of the buffer for this list must be kept small enough to fit in the First Level Cache. The plots below show only the first 2000 detected gaps with the x axis converted to Seconds to better illustrate the time span. To avoid impacting the results, further gaps are not recorded but only statistically evaluated.

In the *idle* system, the benchmark detects large gaps above 30 million cycles (more than 10 ms) unless it is fully isolated from all interrupts. In absence of other runnable processes, the benchmark is interrupted for 0.7 % of its execution time by the timer interrupt (all other hardware interrupts are routed to the other CPU). In the case of a cleared Interrupt Flag, the maximum gap detected in one Hour execution is 166 cycles (62 ns) and a total of 5 gaps above 50 cycles. As shown in Section 2.1.8, the minimum disruption time for an empty interrupt on the Nehalem micro-architecture is 575 cycles (Table 2.5). Typical Linux interrupt handlers are in the range of several 1000 cycles (Section 2.3.5). Therefore, the observed gaps in the fully isolated system are not resulting from interrupts triggered by hardware or software on the remaining system, but must be due to hardware effects caused by shared functional units (e. g. cache misses, bus collisions).

On a heavily loaded system, the improvement of the real-time behavior can directly be seen. The benchmark executed with *fair scheduling* must compete for a CPU and is executed for only 35 % of the time. With *partitioning*, the gaps are reduced to 0.6 %, approximately the same as on an idle system. With *full isolation*, the percentage of disruptions is further reduced, but the maximum gap was detected with a length of 1135 cycles (425 ns). This is the dimension of an interrupt and must be analyzed for its real cause.

Figure 4.4 shows the duration (in cycles) over the time (Seconds after start) of the first 2000 gaps in the *idle* system with *fair scheduling* (first row in Table 4.1). The maximum gap in one Hour was 2.9 million cycles, but the majority of gaps is in the range of 20 000 cycles. A lower level of clustered gaps is in the range of 50–70 cycles. In short bursts (e. g. around 1.2 Seconds), many interruptions of varying length can

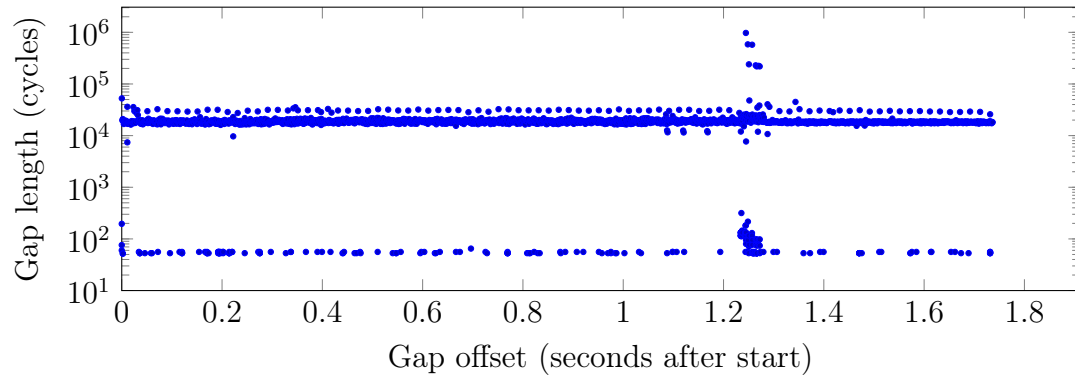


Figure 4.4.: First 2000 gaps (above 50 cycles) on **idle** system, **fair scheduling** for benchmark.

occur. In this test series, no other processes compete for the CPU, the most probable source of interruptions is the local timer interrupt used for time management and scheduling.

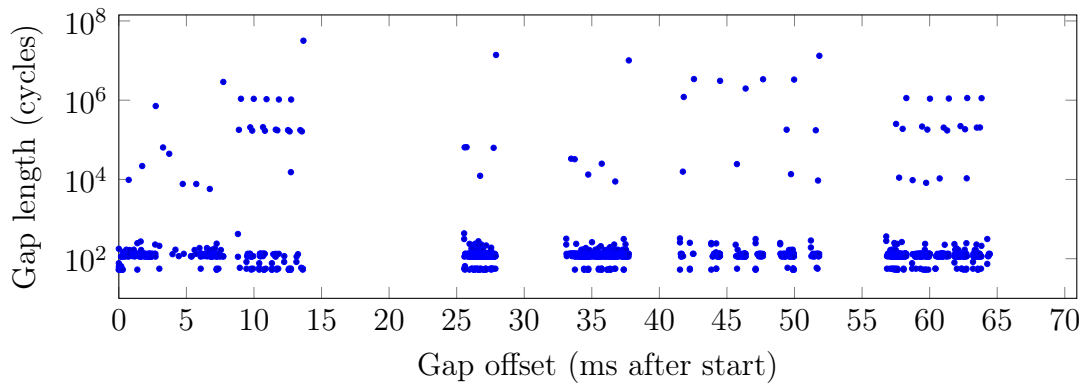


Figure 4.5.: First 2000 gaps (above 50 cycles) on **loaded** system, **fair scheduling** for benchmark.

The same benchmark with *fair scheduling* executed on the *loaded* system collects 2000 gaps in only 0.07 s (Fig. 4.5). In this resolution, the gaps in the execution time can also be identified on the x axis. As in the previous plot, certain levels of gaps around 70 (here extended to slightly above 100) and around 20 000 cycles can be identified.

If the benchmark is executed in its own *partition* on the *loaded* system (Fig. 4.6), most of the larger gaps no longer occur. The most gaps are in the range of 70–100 cycles with fewer, but very periodic ones at 10 000 cycles. This is an indication that the gaps below 1000 cycles are caused by hardware effects such as cache misses while the disruptions at 10 000 cycles are the periodic timer interrupts. When the

4. Isolation of Bare-metal Tasks

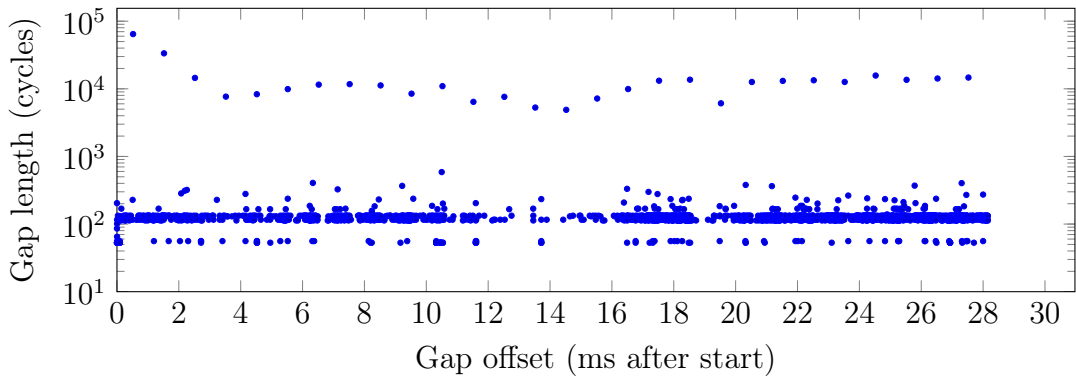


Figure 4.6.: First 2000 gaps (above 50 cycles) on **loaded** system, benchmark executed in its own **partition**.

benchmark is fully isolated (Fig. 4.7), the gaps above 1000 cycles no longer appear and the higher resolution on the y axis makes visible distinctive levels at 56, 116, 133 and spread between 150 and 500 cycles.

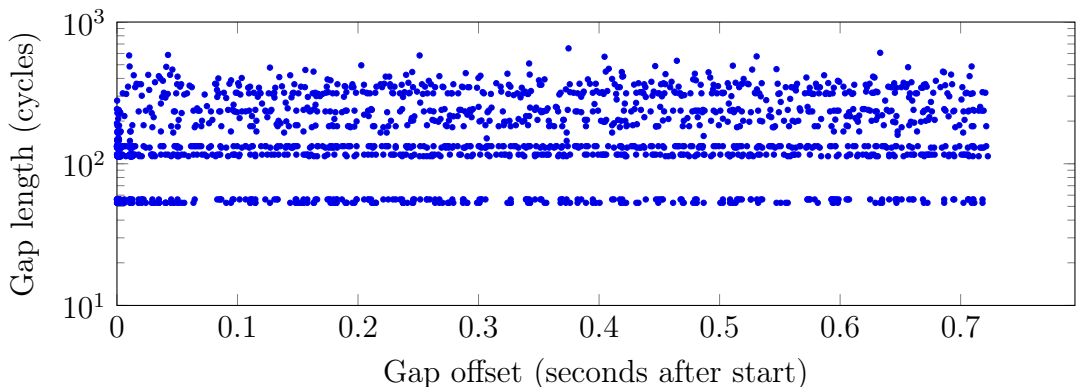


Figure 4.7.: First 2000 gaps (above 50 cycles) on **loaded** system, benchmark executed **fully isolated**.

Using a more complex benchmark on haixagon (Intel Xeon Hex-Core E5645 2.4 GHz, Appendix A.3.1 on page 208) with the advanced implementation presented later, a similar result can be observed. The histograms in Fig. 4.8 display the distribution of detected gaps during 10 Minute test series using different real-time methods comparable to the results presented above. With *fair scheduling* and *partitioning*, a considerable number of iterations are interrupted for 10 000 to 25 000 cycles. However, the majority lies in the lowest histogram range, below 1 500 cycles occurring more than 5 orders of magnitude more frequently. The partitioning results in less interrupts in the range above 30 000 cycles ($12.5 \mu\text{s}$). Only the fully isolated execution is able to prevent all delays above 2 500 cycles.

4.6. Practical Evaluation

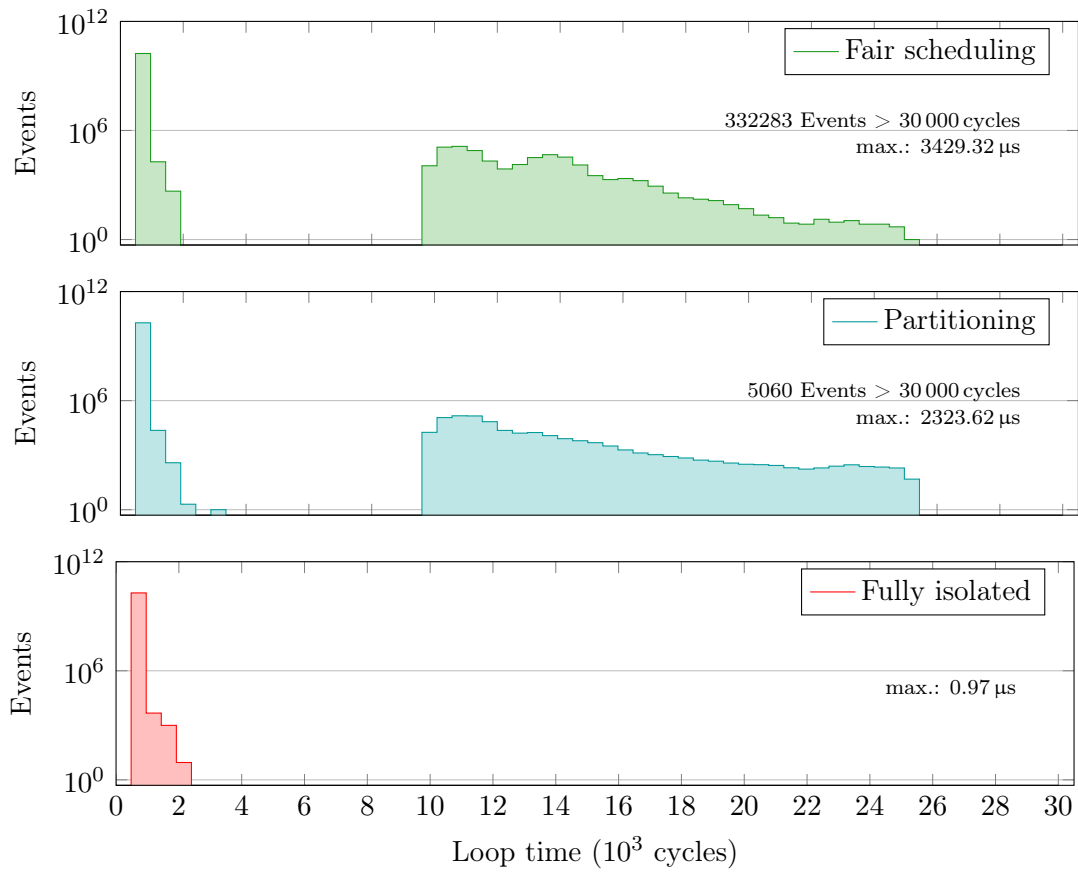


Figure 4.8.: Histograms on **loaded** system.

4. Isolation of Bare-metal Tasks

To rule out interrupts being the cause of the gaps between 100 and 1000 cycles, a similar benchmark routine was executed with a minimal OS kernel (*smp.boot* Kernel, Appendix B on page 213). This bare-metal micro-kernel initializes all CPUs of a multi-processor system to a known and well controllable state. Particularly, the interrupt system (I/O APIC, localAPIC, sending IPIs) is deactivated and the observed jitter can only result from CPU and cache effects.

Load range	Hourglass max. gap	
	Write-Back	Write-Through
16 KiB	64	64
128 KiB	64	64
256 KiB	52	52
512 KiB	52	52
1 MiB	52	52
2 MiB	52	48
4 MiB	1080	52
5 MiB	1260	52
6 MiB	1420	48
8 MiB	1552	52
16 MiB	1480	52

Table 4.2.: Maximum gap on the *smp.boot* Kernel with different cache configurations for varying load ranges (in CPU cycles).

An hourglass benchmark routine was executed on one CPU while a load with a varying memory access range is executed on another CPU. The remaining CPUs are idle (halted). The results (Table 4.2) of *Write-Back* caching clearly show, that a load using less than 2 MiB of the shared Last-Level Cache has no impact on the benchmark. However, if the range of the load increases above half of the Last-Level Cache size, the benchmark experiences occasional gaps above 1000 cycles similar to the previous test series with an isolated task on a Linux system. Since interrupts can be excluded in the *smp.boot* Kernel, this proves that the shared Last-Level Cache in fact can slow down the execution of a processor that only works in its First Level Cache. In all tests, the minimum and average loop time are 44 and 47 cycles (resp.) which indicates that the deviations are very seldom because they have no impact on the average. With a *Write-Through* caching strategy, the impact of larger loads on the hourglass benchmark is avoided. But if the benchmark uses a real payload instead of the nearly empty hourglass loop, Write-Through has the drawback of always impacting the measurement task and its average decreases to 288 cycles. The estimation of the worst-case execution time must pessimistically include these effects if other processes use the Last-Level Cache shared with a hard real-time task. In Chapter 7, methods are presented to analyze and handle this jitter.

4.6.1. Risks

Certain actions in the system partition may harm the real-time execution of isolated tasks. In all tests, these did not occur, but all applied software should be evaluated for compliance as derived in Section 4.5.

The partitioning relies either on CPU-Sets or on CPU Affinity and both can be modified by other processes. If an application uses them for its own purpose, it will most probably interfere with the isolation. Only processes with *root* privilege can modify the base Control-Group hierarchy. However, sub-trees of CPU-Sets can be consigned to users to create further subdivision without influence of neighboring containers. Other unwanted behavior includes hotplugging of CPUs, memory and other devices and the shutdown of a system.

Using system calls from completely isolated tasks is prohibited. Obviously, their timing is not predictable and may include the execution of pending work queue jobs. If code to be executed in an isolated task is provided by a user, that must be analyzed for compliance. Another reason to prohibit system calls was observed on some Linux versions. The Kernel may restore the Interrupt Flag implicitly after returning to the user-mode. This would involuntarily terminate the isolated state of the task.

The isolated task concept removes concurrent execution on other processors from the formal verification and the estimation of the WCET. But the whole system must still be certified and controlled for other (possibly hostile) applications.

4.6.2. System Stability

The tests presented here were done with a very simple C program and setup shell scripts. The load applications were started before the benchmark initialized the isolated task and stopped afterwards. With tests on interactively used systems, the system under test remained responsive.

However, some more complex tests and heavier use of the system during active isolations revealed problems. Some processes were blocked until the ending of all isolations. Other problems were lost network connections, no login possible, and wrong wall clock time. All these problems are addressed in the following Chapter.

5. Linux Kernel Modification

In the previous Chapter, the concept of isolated bare-metal tasks on dedicated Central Processing Units (CPUs) is presented and evaluated. The evaluation concentrates on the execution of code in isolation and shows its uninterrupted execution even with the concurrent execution of heavy system load.

However, certain loads cause the system to stop reacting to user input. As predicted before [WLB12]¹, the masking of Inter-Processor Interrupts (IPIs) is expected to block the execution on other, non-isolated CPUs. This may happen due to the synchronous `call_function()` mechanism that waits for the completion of a remotely triggered function. Among others, this is used in the Linux function `tlb_shootdown()` that instructs other CPUs to flush their Translation Look-aside Buffer (TLB) after changes were made that affect other processors. An IPI is sent to the target CPU that reads the function pointer to execute from a global variable, calls the function and acknowledges the completion by setting a flag. The sender can not notice the masking of its IPI but will wait for the acknowledge by spinning on the flag until the masking of interrupts is undone. This will not happen until the bare-metal task terminates its isolated state. It is highly probable, that after some time, every CPU gets stuck in such a synchronous waiting.

Indeed, various more elaborate tests have shown, that it depends on the executed programs, if and when a complete system blocking will occur. There were successful executions of an isolation lasting several Minutes while the system was used for office work. But the start of new processes as well as heavy memory and network load tend to block one CPU after another within few Minutes after the begin of an isolation. If the isolation ends, the execution of the blocked CPUs proceeds within instants. This is plausible because masked IPIs have their assertion bit set and will be executed as soon as the Interrupt Flag is set again (i. e. interrupts are unmasked).

The most notable problems observed during all tests are the blocking in kernel-mode of some or all CPUs in the system partition (i. e. with the effect that processes are no longer executed), no logins possible, instable wall clock time (and related problems such as sleeping processes not waking up because the clock does not advance over a certain point), increasing memory consumption (because Kernel buffers are not freed), and reported faults (often false-positives) in the Kernel log (output of the command `dmesg`) sometimes causing Kernel-Bugs, Non-Maskable Inter-Processor Interrupts, Stack-Dumps, or even a reboot. Some problems followed up to Kernel

¹First prediction by Jan Kiszka in personal communication, 2009

5. Linux Kernel Modification

modifications, for instance that pinned Kernel-Threads appear in created CPU-Sets that can not be moved back leading to the CPU-Set not being removable, too.

In this Chapter, the methods are presented to analyze the system state during the execution of isolated tasks along with approaches to address the observed problems. Although it was initially intended to use the isolation concept on unmodified systems, some changes to the Linux kernel are required to keep the system responsive for longer executions.

5.1. Monitoring System Stability

To detect problems with all processes executed on the system, a full logging of a wide range of system states was implemented. Since the system is always tested under load, this integrated recording does not affect the measurements in a negative way but instead increases the load. Besides observing the system from user-mode and searching the source code for the triggers of problems, kernel debuggers and tracers like *SystemTap* and *ftrace* [Gre14] can be helpful.

As mentioned, prolonged periods of isolated tasks can disturb the time keeping system. This leads to unreliable answers of the functions reporting the wall clock time and to problems with the `sleep()` function. Thereby, a script recording some information and then sleeping for five Minutes can not be trusted to create a track of information obtained at a constant interval every five Minutes. Additionally, recording the system's notion of the current time can also be misleading. Hence, it is advisable to use the processor's time-stamp counter (`RDTSC`) as additional point of reference because it is independent from the operating system (OS) and monotonically increasing. This allows to detect, if the wall clock time reported by system functions is correct. It is also reasonable to compare and synchronize the output of different logging systems such as own log files and the Kernel log (`dmesg`).

Further, the start and end of the real-time application (including the isolated tasks) should be monitored with time-stamps to match recorded events with the start, ongoing execution, or shutdown of isolated tasks. A viable solution to record the state of the real-time system is to monitor it from a second system via network. A shell script was used to connect to the system under test with Secure Shell (SSH) and record all required information together with its own wall clock time. It is advisable to process this massive amount of data automatically and create a report including graphics to quickly detect anomalies in the execution. Further, the recording should start some Minutes before and end sufficiently after the real-time application with its isolated tasks to understand how the values evolve during normal operation.

Some examples of data sources for recording the system state are listed in Table 5.1. The active processes can be monitored with the command `ps`. This command can be configured to include more information in its output, especially the CPU a process is actually executed on, and its scheduling priority (if applicable). Many files in

Data Source	Type	Comment
<code>date</code>	Shell command	Print wall clock time
<code>uptime</code>	Shell command	Time since boot and system load
<code>ps</code>	Shell command	running processes and state
<code>dmesg</code>	Shell command	print Kernel log buffer
<code>RDTSC</code>	Assembly instr.	Time-stamp counter
<code>/proc/meminfo</code>	virtual file	contains free and occupied memory
<code>/proc/interrupts</code>	virtual file	displays number of interrupts per CPU
<code>/proc/slabinfo</code>	virtual file	number of Kernel memory objects
<code>/cpuset/*/tasks</code>	virtual file	existing CPU-Sets and tasks therein

Table 5.1.: Data sources for logging the system state during isolation

the virtual file systems `/proc` and `/sys` help to understand the Kernel's internal state and behavior. The Kernel log is a ring buffer holding the last 16–64 KiB of log messages. The output of `dmesg` is the current content of the buffer. This command can be instructed to clear the buffer reducing the output of the next invocation to only new entries. Alternatively, following recordings must be revised from duplicate entries. To avoid missing entries when the buffer flows over, the messages should be stored to a file sufficiently frequently.

During the practical evaluation, some effects were observed that only occurred very scarcely (e. g. only in the second isolation after a reboot or only in the second isolation after every fifth to twentieth reboot). The utmost security can not be proved with experiments (especially for hard real-time execution), but to get reproducible results, the testing procedure should be automated. This was realized by controlling the system under test from a second computer that repeatedly reboots the first system and iterates many test series. The test sequence should include many short isolations as well as few very long ones.

Some negative effects are triggered by the modifications to the Linux kernel presented in the next Section (and addressed by other modifications or settings). Generally, more data must be included in the logging if undesired behavior is experienced to be able to understand the cause and find remedial measures.

Figure 5.1 is a plot of the *free memory* as reported by `/proc/meminfo`. The x axis displays Minutes after the start of the system. The test series includes a 15 Minute isolation and then six executions of two Hours each. During each execution, the free memory decreases linearly. It is not the test application, that consumes this space, but the Kernel itself. Comparing the *free memory* trend with the *Slab size*, it is obvious, that the decreasing of free memory is caused by an increasing total Slab size that is somehow aggregated during an active isolation instead of being returned to the free memory pool. The Slab is a Kernel data structure to manage small memory allocations. All memory blocked during the isolations was immediately

5. Linux Kernel Modification

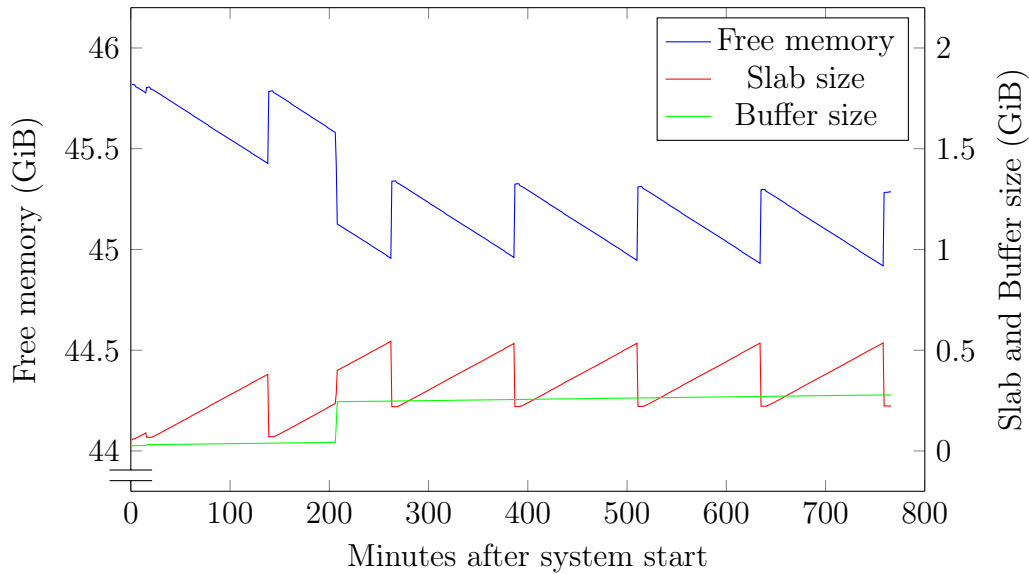


Figure 5.1.: Example trend of various `/proc/meminfo` values during six isolations of two Hours each (System: `haixagon`, Appendix A.3.1 on page 208).

freed at the ending of the isolated tasks. The falling rate depends on the frequency of SSH logging, hence it is the network connection and the creation of processes that leaves those traces. It can be interpolated after how long the free memory would be exhausted. In this example, the free memory decreases by approximately 1 GiB in 6 h. A system with 4 GiB of main memory would be exhausted after an isolation of 24 h duration.

In the example of Fig. 5.1, a singular event 200 Minutes after system start consumed memory that was not freed after the isolation ended. A part of this consumption can be seen in the Slab size, another part caused a step increase of the *buffer size* that otherwise remained nearly constant. This event's time of day could be matched with the execution of the Cron daemon's daily maintenance jobs that were deactivated subsequently.

5.2. Linux Kernel Patch

The predicted internal blocking of synchronous function calls happened while actively waiting for the completion of a called remote procedure. And this could only be addressed by a modification of the Linux kernel. Therefore, the initial ambition to realize the concept on an unchanged Linux system could not be met. Other problems observed by the methods described in the previous Section need to be addressed with other modifications. To ease porting the changes to other Kernel versions, the changes are confined as much as possible. Finally, a total of six files,

four in the architecture-independent `kernel/` directory and two x86-specific files below `arch/x86/kernel` were changed. The described modifications were initially developed for Linux 2.6.31-rt and later ported with only little effort to Linux 3.2, Linux 3.2-rt, Linux 3.9, Linux 3.11, Linux 3.12, and Linux 3.12-rt. The following descriptions are based on Linux 3.12, the most recent official version used during the project. The modifications are similarly valid for the same Kernel version with the Real-Time Preemption Patch (RT-Patch) Linux 3.12-rt. The changes are collected as patch `isol.patch`. The term “the Patch” refers to all modifications described in this Chapter.

5.2.1. Infrastructure

Similar to Hotplugging CPUs (Section 2.2.5), the isolation can be interpreted as a state of certain processors. Therefore, the changes to the Linux kernel need to be activated for the isolated processors if and only if a task is executed in the isolated state. A convenient method to offer an interface accessible by user-mode applications is to create a virtual file in `/proc` or `/sys` [CRK05]. The latter is the more modern approach that was selected. During initialization, the Patch creates the virtual directory `/sys/isol` with files to configure the behavior and to activate isolations for individual or multiple processors. The `sysfs` virtual file system registers a function that is called when a user reads or writes a file in that directory. The reading function can create a character string out of internal data structures to pass information to the user while the writing function receives the written data to store them internally and to optionally execute required actions.

The actions to isolate a task now include the additional step of notifying the Kernel about the activation of the isolation. This should be the second-to-final step just before clearing the Interrupt Flag (IF). After the real-time operation, the Interrupt Flag is restored and the Kernel must be notified about the ending of the isolation before any other take-down steps are executed.

So far, using system calls from within a completely isolated task was prohibited. This collides with notifying the Patch about activating or ending the isolation. Strictly, a task completes the initialization phase with clearing the Interrupt Flag. Hence the assembly instruction `CLI` is the point when the full isolation begins. Likewise, restoring the Interrupt Flag with `STI` terminates the isolation before the other measures are reverted. The important reason for not using system calls is their unknown duration and the potential implicit restoration of the IF. If the writing to the `sysfs` file happens before masking interrupts, this is acceptable. Similarly, after the real-time application has ended (the control loop was left), the deinitialization and restoration of normal system operation is not time-sensible and can be done securely by using a system call. Generally, it is recommended to use the low-level UNIX interface (e.g. `open()` and `write()`) for accessing `sysfs` files instead of the C library (e.g. `fopen()` and `fprintf()`) or C++ (e.g. `class ostream`) because those

5. Linux Kernel Modification

may buffer the output. The handle of the open file `/sys/isol/cpu_mask` should be left open to be used again for terminating the isolation.

The current Patch supports two methods of adding a processor to the set of isolated CPUs. The first method is writing a bit mask of isolated CPUs to the file `/sys/isol/cpu_mask`. The internal infrastructure will compare this new bit mask with the previous setting and activate the isolation for all additionally set bits and revert the changes for all cleared bits. To avoid race conditions by reading the previous state, setting the bit corresponding to the own CPU and writing the new bit mask, the alternative method supports adding and removing individual CPUs by writing `+N` or `-N` to the same virtual file to add/remove only CPU `#N`. The management functions use a Kernel spinlock to avoid problems with concurrent access.

Other files in the directory `/sys/isol` serve to configure the behavior of the modifications and to ease debugging. The file `flags` sets a bit mask to selectively activate individual modifications to test different combinations of them without the need to rebuild the whole Linux kernel. Similarly, the individual entries of the notifier chain (Section 5.2.5) can be activated with the bit mask `notifier`. The amount of logging output can be configured with `verbose`. During evaluation of the Patch and the behavior of the system, it is very beneficial to be able to deactivate selected methods and to understand exactly when different parts of the modification become active.

5.2.2. CPU Online Mask

The first approach was to change the bit mask that is used by the Kernel to track available CPUs. Usually, it is used if restricting the number of started CPUs (boot parameter `isolcpus`) or for Hotplugging. It was expected that all Inter-Processor Interrupt (IPI) sending functions check this bit mask and refuse to send to offline CPUs. But this did not solve the problem and even caused CPUs to not work properly after an isolation. Thus, clearing such bits may lead to shutting down systems in the Kernel that are not restarted automatically after restoring the bit.

5.2.3. Inter-Processor Interrupts

Inter-Processor Interrupts (IPIs) are used in multi-processor systems to invoke interrupt handlers on other CPUs. This is required to wake up idle CPUs to schedule work on them, to invoke arbitrary functions to maintain the OS's internal structures, handle hardware errors and lockups, for performance monitoring, and so forth. IPIs can be sent by a CPU either to a dedicated target or be broadcasted to multiple or all CPUs.

The delivery of IPIs to isolated CPUs that have masked all interrupts has the potential of blocking the remaining system if the sender waits synchronously for the

completion of the requested function. As experiments have shown, it depends on the load on the system partition after how long complete system lockups become more probable. To allow all types of programs in the non-isolated parts for arbitrary times, the dispatching of IPIs to isolated CPUs must be prevented.

Linux offers variants of the function `smp_call_function()` to execute a given function on either a given CPU or on all CPUs. To manipulate this system, the call tree was analyzed to find places where changes should be applied. The analysis is made intricate by the coding style of Linux involving many macros and data structures of function pointers to allow the porting of Linux to a wide variety of architectures.

Generally, two approaches exist to understand the interactions of many functions in different parts of the Kernel, Bottom-up and Top-down. The *Bottom-up* approach either starts at the low-level functions that directly access the hardware of the Local Advanced Programmable Interrupt Controller (localAPIC) or at a level where all functions involved with IPIs can clearly be identified. If this is not the layer where modifications are suitable, all places must be identified, where these functions are called. This recursive process must be followed until a layer is found, where modifications are reasonable. The `apic` structure contains function pointers that are initialized depending on the type of localAPIC. They are invoked mostly by macro abstraction, therefore the macros must be identified that use the function pointers, then the places must be found, where those macros are used. In this direction, the tree becomes very widely branched. The *Top-Down* approach starts at an similarly identifiable point and follows down function calls to the functions that actually access the configuration registers of the localAPIC directly. The challenge is to detect the value of the function pointers in the `apic` structure that are initialized at boot time. For this task, the Patch includes a function to read the current values of the five `apic` elements concerned with IPIs and resolves their addresses to symbols. Both methods can be combined to make sure that the whole IPI system is understood and covered.

The implementation of the x86 architecture branch of Linux 3.12 is displayed in Fig. 5.2. The `apic` structure in `apic.h`² contains pointers to low-level sending functions. On the test system `haixagon` (Appendix A.3.1), the functions of the `physflat` module are applied (except `apic_send_IPI_self()` which is similar for both modules as indicated by commentaries in the source code).

The Top-down approach covers all branches of callers but fails on waiting inter-processor function calls: The architecture-independent sending function instructs low-level functions to transmit an IPI and waits for the target to acknowledge the execution. If the IPI is just not sent, the acknowledge will never arrive. As the sending functions synchronously block until notification, the waiting task will block until the isolation ends and IPIs are reactivated. Finally, at least the synchronously waiting inter-processor function calls must be modified in the Bottom-up path, the low-level functions in the Top-down direction can then forestall all further IPIs with

²`arch/x86/include/asm/apic.h` (Linux 3.12)

5. Linux Kernel Modification

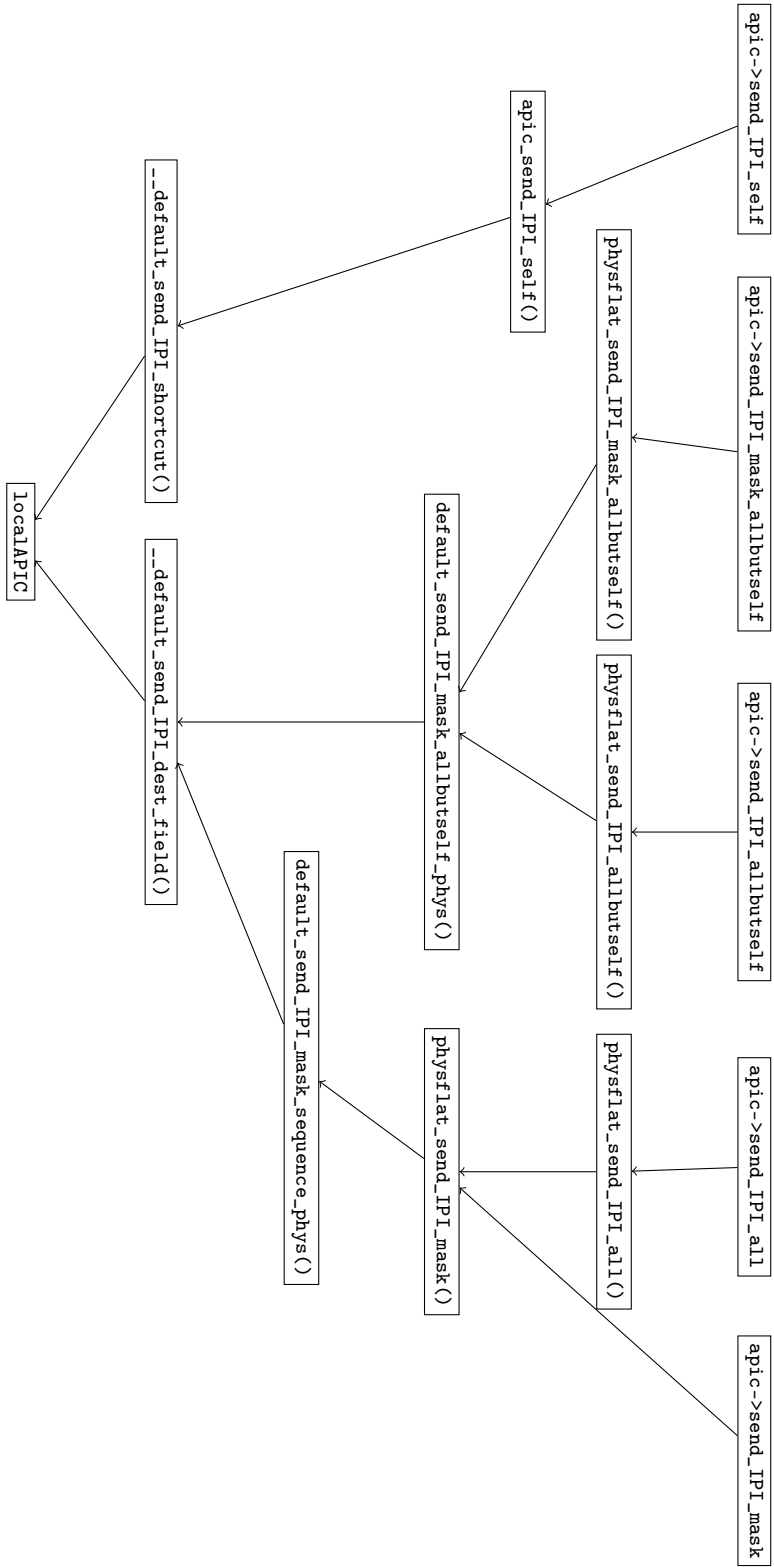


Figure 5.2.: Inter-Processor Interrupt triggering functions call graph of Linux 3.12.

Function	File	Type
<code>default_send_IPI_mask_sequence_phys()</code>	<code>ipi.c</code> ^a	single, low-level
<code>default_send_IPI_mask_allbutself_phys()</code>	<code>ipi.c</code>	single, low-level
<code>apic_send_IPI_self()</code>	<code>probe_64.c</code> ^b	single, low-level
<code>resched_task()</code>	<code>core.c</code> ^c	single, high-level
<code>wake_up_idle_cpu()</code>	<code>core.c</code>	single, high-level
<code>kick_process()</code>	<code>core.c</code>	single, high-level
<code>ttwu_queue_remote()</code>	<code>core.c</code>	single, high-level
<code>generic_exec_single()</code>	<code>smp.c</code> ^d	single, waiting
<code>smp_call_function_single()</code>	<code>smp.c</code>	single, waiting
<code>__smp_call_function_single()</code>	<code>smp.c</code>	single, waiting
<code>smp_call_fuction_many()</code>	<code>smp.c</code>	mask, waiting

^a`arch/x86/kernel/apic/ipi.c` (Linux 3.12)

^b`arch/x86/kernel/apic/probe_64.c` (Linux 3.12)

^c`kernel/sched/core.c` (Linux 3.12)

^d`kernel/smp.c` (Linux 3.12)

Table 5.2.: Functions modified to block sending IPIs in Linux 3.12.

least effort. The lowest-level functions in Fig. 5.2 use physical localAPIC IDs that can not easily be converted back to the CPU IDs used by the isolation bit mask. Therefore, the Patch modifies the three functions of the level above (marked “low-level” in Table 5.2). These three functions block sending IPIs to the isolated CPUs on the lowest level. Even the Non-Maskable Inter-Processor Interrupt (NM-IPI) used for debugging Kernel problems is prevented from being asserted to an isolated CPU.

The higher-level functions use low-level ones to send an IPI. Further, the *waiting* functions actively wait for the completion signaled by an acknowledge flag in a shared variable. If the IPI is never handled, the synchronous waiting is blocked. The IPIs blocked by this modification not even result in the corresponding pending bit to be set, therefore they are not caught up after all isolations end. The type marked *single* is used for a single CPU by providing its ID and the *mask* function is given a bit mask of multiple CPUs to execute and wait for multiple CPUs. The three functions marked *low-level* suffice to handle also the *high-level* functions. An exception are the four *waiting* functions that must be modified in addition to the low-level functions because they synchronously wait for the completion of the triggered function.

With this modification, no IPIs are sent to isolated processors which is a basic requirement to avoid blockings caused by IPIs. If the hardware interrupts are routed correctly to other CPUs and all IPIs are blocked by the Patch, only the timer interrupt is still triggered and blocked by the Interrupt Flag. This allows to fully isolate a task without clearing the IF which will be described in Section 5.3.

5. Linux Kernel Modification

If IPIs are sent but masked on the target CPU, their *pending* bit is set and the interrupt handler is called when the IF is reset. The described changes inhibit sending IPI, therefore they are not caught up after the isolation ends. The `tlb_shootdown()` is executed on all processors after global changes on the Kernel page tables. If a CPU is not notified, it would access memory with out-dated TLB entries. This is avoided in the Patch by generally executing this function on all CPUs that revert their isolation.

5.2.4. Read-Copy-Update

Read-Copy-Update (RCU) is widely used in Linux (Section 2.2.5) for mostly read data structures [BC05, p. 207]. By default, the variant `rcutree` is used. In preemptive real-time Kernels (especially with the RT-Patch), the detection of a grace period is more complicated. This is implemented in the special version `rcupreempt`. For the isolation context, it is important to understand, that the RCU system tracks the state of all CPUs and either waits for them incrementing a counter or periodically executes a function (via IPI) on all of them. If a CPU is isolated, it neither increments the counter nor can the remote function be executed. This blocks the RCU system never ending a grace period. This can either result in an indefinite blocking (if synchronously waiting) or in not freeing data structures (if callbacks are never invoked).

The Slab system uses RCU to manage allocations of memory buffers. If a buffer is returned, it is not immediately freed, but a callback is scheduled, to free the buffer when it can be guaranteed that not more readers are holding a reference to that buffer. Hence, if the RCU system is blocked and its grace period never ends, returned Slab elements are never freed and can not be reused. This is the cause for a growing Kernel memory consumption during an active isolation. As observed, the growing rate depends on the type and amount of load in the system partition.

The solution can be adapted from the Hotplugging system. Since the RCU management keeps track of all CPUs, it must be notified of unplugged and isolated CPUs. The straight-forward solution is to notify the RCU subsystem about the unavailable CPU like the Hotplugging does. The Patch includes functions `rcu_offline_cpu()` and `rcu_online_cpu()` in `rcutree.c`³ that call the same internal functions that are invoked by Hotplugging events. The process of registering a CPU offline is notified by the sequence of `CPU_DOWN_PREPARE`, `CPU_DYING` and `CPU_DEAD`. The inverse operation includes the events `CPU_UP_PREPARE` and `CPU_ONLINE` (the meaning of these events will be described in the next Section). Essentially, they clean up pending RCU callbacks on the current CPU and (de)register the processor in the list of active CPUs. Since an isolated task is not allowed to use system calls, it can not interfere with data structures protected by RCU. In the next Section will be described, how

³`kernel/rcutree.c` (Linux 3.12)

Event	Description
CPU_DOWN_PREPARE	ask for approval
CPU_DOWN_FAILED	only for rollback on disapproval
CPU_DYING	shortly before CPU is switched off
CPU_DEAD	after CPU is powered down
CPU_DEAD	after CPU is powered down
CPU_UP_PREPARE	ask for approval
CPU_UP_CANCELED	only for rollback on disapproval
CPU_STARTING	CPU will be started
CPU_ONLINE	CPU is running

Table 5.3.: CPU notification events

the Hotplugging system can be imitated more closely which will include notifying the RCU system.

5.2.5. Notifier Chain

The Notifier Chain is a mechanism of the Linux kernel to allow other subsystems to register to be notified on certain events. Since the activation of isolated CPUs is similar to Hotplugging, the CPU notifier chain is especially interesting. Every kernel module (e.g. device drivers) or subsystem (e.g. time keeping, RCU) can register its callback function with the CPU notifier chain. During shutdown or startup of each CPU, multiple events (Table 5.3) are issued to all registered callbacks. The *_PREPARE events are inquiries that can be answered with disapproval. In that case, all previously notified callbacks are rolled back with an abort event. With this mechanism, a callback can place a veto to keep a certain CPU running.

Many parts of the Kernel need to be notified about unavailable CPUs. Examples are the time system, watchdogs, and RCU. Therefore, it is sensible to imitate this mechanism also when isolating a CPU. A copy of the original functions was extended with a bit mask to disable the notification of selected entries. Some notifiers should be omitted if the CPU is not in fact switched off because the isolated task needs to be further executed. Therefore, the *migration* notifier is excluded. The modifications are implemented in `cpu.c`⁴. The functions `cpu_isol()` and `cpu_unisol()` imitate the function `notifier_call_chain()` originally from `notifier.c`⁵.

Since the inclusion of the notifier chain, another problem arose with kernel threads. The imitation of Hotplugging terminates many pinned kernel threads on the respective processor. When reactivating it, the kernel threads are restarted by the `kthreadd`

⁴`kernel/cpu.c` (Linux 3.12)

⁵`kernel/notifier.c` (Linux 3.12)

5. Linux Kernel Modification

(Kernel Thread Daemon). If this non-pinned kernel thread was moved to a CPU-Set containing only CPU #0, the spawned threads inherit this property and remain pinned to CPU #0. Therefore, in the system initialization when moving all processes and non-pinned kernel threads to a system CPU-Set, the Kernel Thread Daemon must be pinned to CPU #0 and remain in the root CPU-Set (that contains all CPUs). During restart, the pinning of the new threads is changed to their CPUs and they remain in the root CPU-Set as before they were terminated.

5.3. Deactivation of the Timer-Interrupt

With preventing the sending of IPIs to isolated CPUs described in Section 5.2.3, the only interrupt still blocked by the Interrupt Flag (IF) is the local timer triggered by the localAPIC. That interrupt is used by the Linux kernel for periodical maintenance (e.g. advance the wall clock and usage statistics) and preemptive scheduling. Since the isolated CPUs host only a single process and this interrupt is masked anyway, the local timer can be stopped during active isolations.

The local timer is a functional unit of the localAPIC (Section 2.1.7 on page 21) and can be configured per CPU by the memory-mapped localAPIC configuration registers [Intel13c, Sect. 10.5.4]. Those are available at constant physical addresses which makes them accessible both for the Kernel and for user-mode applications. The latter can map the respective region of the virtual file `/dev/mem`⁶ and modify the localAPIC behavior directly.

The timer uses two registers, `Initial Count` and `Current Count`. It is stopped by writing zero to `Initial Count`. To restart the timer, either the old value of `Initial Count` or the last value of `Current Count` before stopping the timer must be written to `Initial Count`. Each CPU can only configure its own localAPIC. To be able to remotely activate an isolation, a remote function call (Section 5.2.3) via IPI is used. This must be done before the Patch starts blocking IPIs to that CPU.

If the partitioning redirects all hardware interrupts, IPIs are blocked by the Patch and the local timer is deactivated, the IF must not be cleared to achieve the state of complete isolation. Moreover, the IF can only be changed by the CPU itself. Therefore, the complete isolation supported by a cleared IF can only be reverted by the CPU itself. This could be triggered by an IPI, but the handler executes in kernel-mode and can only influence the user-mode flags by manipulating the stored flags in the user-mode stack. In contrast, activating the local timer of another CPU by IPI is easier.

With this modification, the isolation can be controlled from another CPU. This makes managing multiple isolated tasks (Section 6.2) more convenient and enables handling errors in isolated tasks. For example, if an isolated task gets stuck in an

⁶Newer Linux distributions tend to deactivate this interface for security reasons but it can be reactivated.

endless loop, a cleared Interrupt Flag would only allow an Non-Maskable Interrupt (NMI) to pass. If the blocked task does not need to clear the IF, the isolated state can easily be terminated by another CPU and the isolated task becomes a normal thread again that can be terminated with OS methods.

A similar approach of deactivating or reducing the timer interrupt and modifying the Linux kernel was done in a project to increase the performance of High Performance Computing (HPC) applications. They changed the timer frequency and removed the major source of IPIs by changing RCU instead of systematically verifying that no further interruption is possible which is crucial for hard real-time execution. They did not reach a fully stable solution but proved their concept for HPC systems [ALL12].

5.4. Alternative Implementations

During the development and evaluation of the described changes to the Linux kernel (the Patch), alternative methods were evaluated but not implemented. The most notable alternative way of handling the problems that the complete isolation causes in the Kernel is the Hotplugging approach. At the time of implementing, the current method of changing some places in the Kernel was rated to lead faster to a successful result. Since the modification of the Kernel and porting the changes to a different version is a major effort, the alternative concepts are described. Though, it was not analyzed if these methods are realizable.

5.4.1. Kernel Module

One motivation of the isolation concept was initially to use an unmodified Linux system. This had offered the benefit of independence from the underlying OS and to easily transfer the real-time application to other systems or to upgrade the OS [KY08]. However, Kernel problems enforced their addressing by modifications to the Linux source code by a patch. This Patch changes only six files and was already ported to several new Kernel versions. This did not demand deep analyzes, instead the only problems were functions moved too far resulting in the Patch tool not being able to identify the places to change. Otherwise, kernel modules are considered easier to port to new Linux versions. They use a more fixed Application Programming Interface (API) but are restricted to the released interfaces.

Another disadvantage of the Patch is the requirement of rebuilding the Kernel after every change. In contrast, a kernel module can be unloaded, recompiled and just loaded again. If the package management of a distribution installs a new Kernel (e.g. with security fixes), it would lack the Isolation Patch but could probably just load automatically the same kernel module as before.

5. Linux Kernel Modification

It was tried to realize the changes of the Patch as kernel module. The infrastructure with the `sysfs` interfaces and the management of isolated CPUs can easily be implemented in a module, after all this is how device drivers work. But the modification of IPI sending functions is a modification of several high- and low-level functions (in the architecture-independent and the x86 parts of the code, respectively). Other functions like the notifier chain and the notification of RCU are not part of the module API. They could be exported by a Patch again, but this would foil the kernel module advantage.

5.4.2. Hotplugging

Like introduced in the motivation for this Kernel Patch (Section 5.2.1), during an active isolation, the respective CPU is unavailable for the other CPUs like a switched-off processor is. The procedure of activating the isolation is in large parts imitating the Hotplugging system. Consequently, an alternative method to unregister isolated CPUs from the Kernel would be in fact switching it off with the Hotplugging system. To execute the isolated real-time task, the halted processor could be waked up again without using the functions of the Linux kernel. This would result in a CPU that executes some code but without the Kernel registering this CPU as available.

The waking-up of a shut-off CPU is described in the processor manuals [[Intel13c](#); [AMD12b](#)]. It is the same procedure used by a Symmetric Multi-Processing (SMP) OS when booting the other processors (Application Processor, AP) from the Bootstrap Processor (BSP). The boot process of a multi-processor OS starts with a single processor that initializes some early required devices and then wakes up the other processors with a Startup-IPI. This is a special IPI that can be generated similar to usual IPIs by the localAPIC. Instead of an interrupt vector number, the Startup-IPI carries the address of a physical memory page as payload. This page must contain the boot code starting from 16 Bit Mode to initialize the CPU to 32 Bit Mode (and possibly to 64 Bit Mode) and to set up some processor-specific functional units like the localAPIC and the page-tables used by the Memory Management Unit (MMU).

To exploit this hardware feature circumventing the Linux kernel, a memory page in the physical address range below 640 KiB must be allocated and initialized with the compiled boot code. Then, a Startup-IPI can be sent to the switched-off processor with the physical address of the page holding the boot code. The boot code is standard for x86 processors for switching from 16 Bit Mode to the required 32 or 64 Bit mode that the other CPUs use.

Bringing up the CPU is straight-forward and certainly possible. However it must be analyzed how the localAPIC must be configured to work correctly. It will not receive any IPIs, because the Kernel expects this CPU to be switched off. But it must be evaluated, if the reactivated localAPIC reacts to broadcast IPIs.

The final action of the boot procedure is generally handing over to a scheduler that manages multiple tasks. In this setting, only a single task (the isolated one)

should be executed. The code is stored in a process running in the remaining Linux system. It is probably a viable way to use the same page tables for the waked-up CPU than a manager process and jump to the address of the code to be executed in isolation.

To revert the state of a CPU executing without being registered by the Linux kernel, the isolated task must shut down the CPU to a halted state. Then, Linux can re-initialize the CPU with the Hotplugging system. It is not clear, how the CPU would react to faults and errors and how a stuck isolated task could be recovered. It would probably be required to implement interrupt and exception handlers for those occasions.

Generally, this method contains a lower risk than the Patch for causing errors in the OS that were not yet discovered because the Hotplugging system handles all subsystems correctly and is widely in use and tested while the Patch is only evaluated with a selection of loads. Further, this proceeding can probably be implemented as easily portable kernel module because it is based on processor features and requires only minor interaction with Kernel functions. It remains as future work to test the viability of this approach.

5.5. Evaluation

The implemented modifications were successfully tested with a wide variety of loads and benchmarks with isolations up to 5 Days. This included heavy system load and remote logins and recorded the load processes to evaluate if they work properly. Further, there were no critical messages in the Kernel log (`dmesg`).

Using system calls now includes another risk (additionally to those explained in Section 4.6.1). The Patch deactivates some subsystems for the isolated CPU. Using system calls that rely on these functionalities can either block indefinitely or reactivate sleeping services or even block the whole system. Although, single isolations of up to 120 Hours and tests with many reboots and many short executions (up to 50 isolations of 5 Minutes each) went all fine.

The hard real-time property of non-interrupted execution on the bare-metal is not changed in this Chapter. But the fact, that a system can not crash is not easily formally verifiable running a Linux kernel. Therefore, the criticality does not hold as strong as the temporal predictability.

6. Application Support

In the previous practical evaluations, the isolated task has always executed either the empty loop of the hourglass benchmark or small synthetic benchmark routines doing substitute work on memory buffers and multiple isolated tasks did not cooperate. Since the isolation concept presented in Chapter 4 does not allow system calls, the isolated tasks can neither communicate nor use device drivers so far. This constrains their applicability for real work. The solution to restore the usability already began to show in using shared variables for notifying the isolated tasks. This Chapter presents methods for shared memory communication without interference of the operating system (OS) and the direct hardware access for interacting with the physical world. A similar trend to avoid context switches into the OS can be observed in High Performance Computing (HPC) where communication methods implemented in the user-mode are increasingly popular. Supported by Remote Direct-Memory Access (RDMA) interconnects, the latency is decreased and the OS jitter reduced [Gio+04].

The objective of the isolation concept is to integrate multiple components and even levels of the automation pyramid (Section 1.1) into a single system with multiple processors. While applications of the company level like Manufacturing Execution Systems (MES) and Human-Machine Interface (HMI) are already implemented customarily on PC systems, the real-time requirements of the cell level and even more the hard real-time and very low-latency demands of Programmable Logic Controllers (PLCs) in the field level are still in the domain of embedded systems. For development systems, prototypes and unique research specimens, the rapid prototyping approach using the isolation concept enables the integration of hard real-time tasks on commodity x86 systems. A possible starting point is an already existing complex application that should be extended with low-latency closed-loop control or a distributed system of multiple external micro-controllers that should be consolidated into fewer systems for cost-efficiency or simplified management.

Generally, common guidelines to implement real-time applications are valid [Lab97; SRH99]. The exemplary application introduced in Section 1.1 is used as a motivation to present how all required means of complex real-time applications can be realized with the isolated task concept. The main application contains a Graphical User Interface (GUI) (e. g. a Qt¹ application) to display the state of the experiment and to allow to configure and start test series. It uses a network connection to store results in an external database. Multiple threads are started to execute longer running analyzes

¹Online: <http://qt-project.org> (visited November 12, 2015)

6. Application Support

and data preprocessing. The GUI should always display a rather up-to-date snapshot of some graphs representing the current state of the experiment. The user interface must always be responsive to adapt parameters or even to abort the experiment in case of a dangerous condition. Decisions of the operator must be transmitted to the experiment control process. Several actors of the physical system (plant) must be regulated by closed-loop controllers with a frequency of 100 000 Hz (10 μ s) and a time-out of 2 μ s with hard real-time constraints. The controllers should update their parameters with soft real-time behavior (quickly after they are changed by the operator) and they must make available their current state to the main application for post-processing.

For debugging purposes, a recording of every 10th state (100 μ s) of the controllers can be activated. Alternatively, the debugging can be supported by providing a special soft real-time mode where a bare-metal task on an isolated processor is not completely isolated. This would allow to execute the isolated task under the control of a common debugger. Software tracing [Sto13, Sect. 3, p.569] can be implemented via a shared memory buffer where the bare-metal task writes trace entries in a ring buffer structure and a system process reads and evaluates the traces. The overhead depends on the granularity and volume of trace events and has the worst impact to the cache resulting in a larger jitter than with deactivated tracing. On x86, the Performance Measurement Counters (PMCs) [Intel13c, Chap. 19; AMD12b, Chap. 13.2] can be used to acquire processor details such as cache misses or branch mis-predictions. The infrastructure supports interrupts, but can also be used on a polling base only counting events that are reported via shared memory to a processing task in the system partition. Debugging mode and tracing should be deactivated during time-sensitive execution.

Between test series (i. e. not during real-time operation), the code implementing the closed-loop control can be replaced. The chemical and physical models are developed using a graphical tool (e. g. Simulink), converted to C code, compiled to shared libraries and dynamically linked to the application. The interaction of the tasks and their required timing are illustrated in Fig. 6.1.

The term “hard real-time” is interpreted in this setting that not a single violation of the defined deadlines is allowed. This should be proved by formal verification relating to the code. But since the execution time of instructions on x86 hardware is difficult to predict, the practical execution must be monitored. This enables registering timing errors, because even a single timing violation invalidates a test series. It is not a safety-critical application (where lives depend on), but failures can be expensive if long-running test series must be repeated or hardware is damaged.

The GUI, the Manager and the Calculation tasks can be implemented as threads in a process. As system, an x86 server with two sockets is projected that can be configured with 8 to 20 processors. The high-performance calculation can therefore

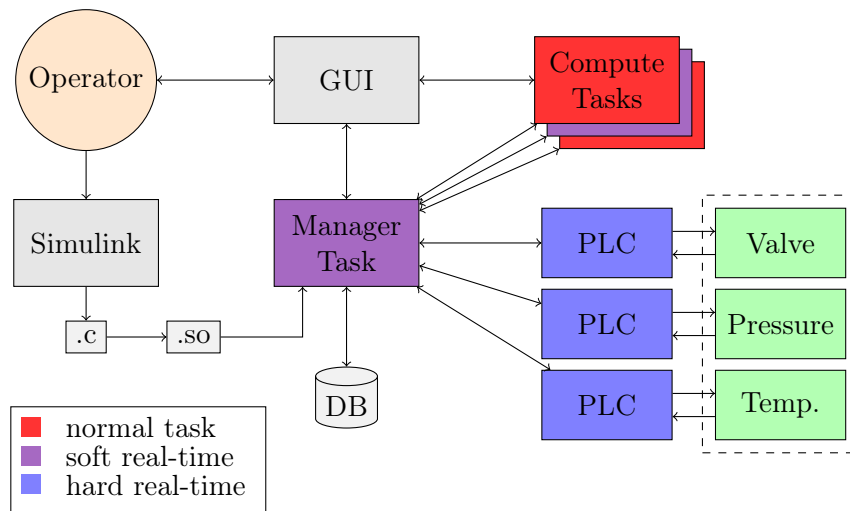


Figure 6.1.: Example application: Tasks and interaction

be implemented with OpenMP² and use several Central Processing Units (CPUs) as well as the soft real-time scheduling methods provided by Linux. It would be possible to install an accelerator card (e. g. Intel Xeon Phi³ or a graphics card for general-purpose processing⁴) and delegate some work. The Manager Task interacts with the GUI using the functions of the framework (e. g. Qt). It starts and interacts with the calculation threads and writes records to the database using a network connection.

Instead of implementing the PLCs on external embedded systems with Micro-Controllers (μ Cs) connected by a field bus, they are realized with the isolated task concept on three dedicated CPUs. Each PLC executes a time-triggered loop every $10 \mu\text{s}$ and calls its functions periodically. These functions include the receiving of updated control parameters, the reading of sensor values, the calculation of the control function, the setting of actor values and sending the current state to the Management Task. The Input/Output (I/O) of physical signals is provided by peripheral Data Acquisition (DAQ) devices installed as PCI Express (PCIe) expansion cards with digital General-Purpose I/O (GPIO), Analog to Digital Converter (ADC) and Digital to Analog Converter (DAC) interfaces. The control function is provided by the Management Task during the initialization and not changed during real-time operation.

In the following Sections is presented how the required means of communication, management and device interaction can be realized under the constraints of the com-

²Online: <http://openmp.org> (visited November 12, 2015)

³Online: <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-overview.html> (visited November 12, 2015)

⁴Online: <http://ggpu.org> (visited November 12, 2015)

6. Application Support

pletely isolated task concept. Additionally, a mechanism to allow certain interrupts without losing control to the OS is presented allowing to use the event-triggered paradigm and even to embed a Real-Time Operating System (RTOS) providing preemptive scheduling into an isolated task.

6.1. Shared-Memory Communication

Isolated tasks can be threads (e. g. Pthreads [NBP96]) or forked [Roc08] processes. Threads have the advantage of a shared address space enabling the common use of global variables. But this also implies the risk of mis-implementing synchronization and generating data races. With distinct processes, the separated address spaces protect the tasks from accidentally corrupting other's data. Processes can still establish shared variables using shared memory segments (e. g. System V shared memory, Section 2.2.3 on page 47). On most architectures, shared memory is a hardware feature implemented in the paging system. At initialization, allocation or mapping, the OS must be consulted, but afterwards, the hardware manages the access to shared memory and the coherence of the memory view. On the x86 architecture, multiple processes can point their page table entries to the same physical page frames [Intel13c, Chap. 4; AMD12b, Chap. 5]. According to the Consistency model (Section 2.1.9) data stores of one process or CPU to an address will be immediately visible to the others that share this variable.

The access to shared resources must be protected by mutual exclusion or special serializing algorithms. This can be enforced by using only common functions to access those shared variables (e. g. Message-Passing or the Monitor concept). Generally, mutual exclusion can be realized with OS methods (e. g. Mutex, Semaphore) that usually block a task until the resource becomes free or using user-mode implementations waiting actively (e. g. spinlock). Since system calls must not be used by isolated tasks, the standard library mechanisms like PThread-Mutex (on Linux realized with Futex [Dre11; FRK02] which may use a system call⁵), pipes and message queues [Ker10] must be replaced with other functions. Spinlocks have the advantage of a very low overhead and latency but they inherently carry a risk of starvation and deadlocks [Tan07, Chap. 6].

Completely isolated tasks are executed bare-metal and the rules of OS development apply. For all shared resources, the hardware-induced latency for the access from different processors must be regarded. If a variable is written by a processor, other processors having the same line in their cache must invalidate it and reload its content from the level below when they access this data the next time. On the x86 architecture, the cache coherence is managed by the hardware, this is transparent to

⁵Züpke presented an alternative implementation which is claimed to be capable of hard real-time [Züp13], but his definition of *hard real-time* is different (weaker). This approach still uses a system call in the contented case.

software. The common level can be a shared Last-Level Cache or the main memory. The latency between storing on one processor and another CPU detecting the change includes the write-back to the shared level and the loading into the other cache. This effect will be analyzed in detail in Chapter 7. A similar effect in this context is false sharing (Section 2.1.9). Therefore, by implementing all following algorithms, variables written by distinct processors should be placed in different cache lines. This can be realized by including padding (unused variables) or allocating aligned buffers e. g. with `posix_memalign()`.

Further important details to watch for while implementing synchronization and generally access to shared variables follows from the compiler optimization. In the programming language C, the `volatile` modifier should be used for variables that are read repeatedly in a loop while another processor may change them. Without this modifier, the compiler may optimize the memory access and only use a register which will never mirror the subsequently changed state of the originating memory location. A similar effect can be realized with memory and compiler barriers or intrinsic atomic functions (Section 2.1.9, also explained in `volatile-considered-harmful.txt`⁶).

A lot of research was published for multi-processor real-time systems and RTOSs proposing a wide range of algorithms for real-time capable synchronization. Generally, all methods can be applied to isolated tasks if they use only hardware-supported instructions like Compare-and-Swap (CAS) or Fetch-and-Add (FAD) (Section 2.2.1) and if they consider multi-processor systems. Of special interest for hard real-time systems are non-blocking [MS96b], lock-free [FLO02; LS04] and wait-free [Her91] algorithms. Because of its emphasis on a low overhead and predictable latency, some research of the area of HPC is also applicable [MS91; Dak09] if it is evaluated for the special (hard) real-time demands. However, the verification of the correctness of complex lock- and wait-free algorithms is more complex than using mutual exclusion [Des13b]. Therefore, the communication functions should be encapsulated in a library that can be verified independently and that can be reused.

Complex nonblocking data structures can be constructed to the need, e.g. for different numbers of readers and writers and with different progress guarantees (e. g. single producer wait-free enqueue and multi-consumer lock-free dequeue FIFO queue) [AIB13, p. 56]. Brandenburg et al. compare blocking to non-blocking algorithms. Generally, the non-blocking kind can be implemented in user-mode based on atomic operations. And the non-blocking variants are also rated better for hard real-time applications [Bra+08]. This is demonstrated by Pohlack et al. in a simple, low-overhead implementation that is advertised as being easily to be integrated into existing projects, possibly based on a wide variety of environments [PAH04]. Their concept was later implemented in a framework to monitor real-time behavior without unpredictably disturbing the timing [PDL06]. This monitoring system is “solely based on shared memory regions, not using any system calls” [PDL06, Conclusion]

⁶`Documentation/volatile-considered-harmful.txt` (Linux 3.12)

6. Application Support

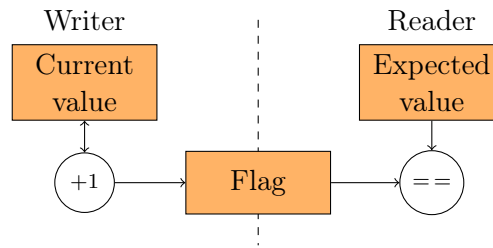


Figure 6.2.: Simple flag operation.

which is similar to the shared memory Inter-Process Communication proposed here. Richter et al. present a framework for real-time communication on Linux-rt [RWK11]. Libraries for user-space synchronization based on atomic operations support non-blocking data structures. Recent examples are *Userspace Read-Copy-Update*⁷ and *Concurrency Kit*⁸ that are both for HPC but might be applicable – at least in parts – to real-time programs.

However, in this Section is presented how such synchronization and Inter-Process Communication (IPC) mechanisms can be implemented to the need of the application. In the following, the synchronization of conditions with signals [BW01, Sect 8.1], the protection of shared resources with mutual exclusion [Liu00, Chap. 8] and some message-based IPC objects [BW01, Chap. 9] are regarded.

6.1.1. Signals

The signaling between isolated tasks and with tasks in the system partition can easily be realized with flags [Sta09, Table 5.3]. This concept is also called *condition variable* [Tan07, Sect. 2.3; Dic13, Sect. 5.2] providing `wait()` and `signal()` methods. Here, a flag is an aligned shared memory variable of 4 or 8 Bytes with a defined meaning of either its bits or integer value. The *writer* sets a bit or stores a new value and the *reader* can test if the writing has already happened (Fig. 6.2). This procedure can be applied in a *one-way* direction with a single writer and one or multiple readers that wait for certain signals from the writer. Alternatively, a *two-way* usage allows a handshake protocol for two partners if they alternately read and write the same shared variable. Essentially, it must always be clearly defined who is allowed to write the flag while the other waits to change roles. With store operations on the x86 architecture being always atomic, no further synchronization is required.

In both the one-way and the handshake case, all members of the operation need a local copy of the shared variable. The writer increments its local copy and writes the new value to the shared variable to start the next *epoch*. The readers compare their local copy with the shared variable to detect the change to the expected value.

⁷Online: <http://ltnng.org/urcu/> (visited November 12, 2015)

⁸Online: <http://concurrencykit.org> (visited November 12, 2015)

6.1. Shared-Memory Communication

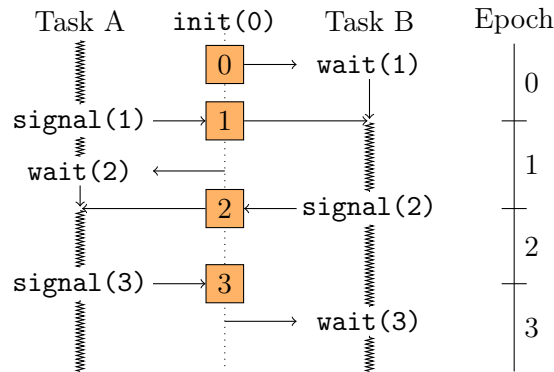


Figure 6.3.: Flag used for a handshake operation.

The readers must increment their local copy between two waiting operations to wait for the next epoch because the writer can already have written the value or they can even have skipped epochs.

In a Handshake algorithm (Fig. 6.3), two members alternatively signal the start of an epoch and wait for the next epoch. In the example, Task A signals the odd epoch numbers and waits for the even numbered ones. When Task B waits for epoch 3, it has already begun and it can immediately proceed with its work. This demonstrates, that a task can still execute in the notion of a previous epoch even if the other task has already begun the next. With two participants signaling alternately, they can omit the local copy of the expected value and wait for an even or odd flag value. This procedure can be extended to multiple members passing a virtual token in a circle. If all members wait for all others to complete the current epoch before jointly proceeding, a *Barrier* can be realized.

For hard real-time tasks, the blocking wait operation can be dangerous to the timing because the partner has the power to jam the other task. If one task has a lower priority or is implemented with only soft real-time constraints, the duration of blocking is unpredictable (i. e. priority inversion). In such combinations, a test or try function can be useful that does not include active waiting (like a spinlock), but only returns if the requested epoch was already signaled. With this sort of functions, the programmer can decide how to proceed if the waiting condition is not met, yet.

Allowing only busy waiting on spinlocks instead of suspending a task wastes CPU cycles and reduces the productive work that can be done on that CPU. Suspending is not possible in the bare-metal execution of the isolated task concept. Although, the advantage of busy waiting is the extremely low latency to notice a change of the flag. On current x86 processors, an empty loop that only compares a flag with a constant value can be executed in 20 to 40 cycles. That time must be extended to the Last-Level Cache or memory latency, where the memory is shared between the respective processors. With a common Last-Level Cache two processors can exchange

6. Application Support

signals with a latency of 42 to 125 cycles. In the absence of a shared Last-Level Cache the latency increases to 208 to 250 cycles⁹.

In the example application presented above, the flag can be used to notify isolated tasks that they need to read new control parameters. The Handshake mechanism is useful to manage the isolated tasks by instructing them to initialize, start and stop their work.

6.1.2. Mutual Exclusion

Critical sections are parts of the code that access shared resources and that must be protected from concurrent access (Section 2.2.1). This can be a conditional update of a variable that must not be interrupted by another task changing this value or it can be the update of multiple entries of a record that other tasks must not read with only some values changed (i. e. in an inconsistent state). On multi-processor systems, those sections can not be protected by blocking interrupts on the current CPU because tasks on other CPUs execute concurrently. A straight-forward solution is to use mutual exclusion and enter a critical section only if no other task is currently executing in this section. If blocked, the task must wait until the section becomes free. After leaving the section, the Mutex is notified so that a waiting task can proceed into the critical section.

Instead of using the methods provided by the OS, a task can implement actively waiting variants of Mutexes, Semaphores and more complex synchronization means on top of flags or by directly using atomic instructions (Section 2.1.9) on shared variables. In any case, the applicability of synchronization primitives based on shared variables must be evaluated for hard real-time demands [BW01, Chap. 8] because they imply a risk of priority inversion (Section 2.3.4) and deadlock (Section 2.2.1). In the following Section, complex synchronization objects like message queues will be presented, that are easier to use and do not entail these risks.

A Semaphore as described by Dijkstra [Dij68] provides two operations, P and V. They use an unsigned integer value initialized to the number of free resources or the number of participants that are allowed to enter the critical section at the same time. The P operation (*prolaag*, Dutch portmanteau of *probeer te verlaag*: try to reduce [Dij63]) probes if the calling task is allowed to advance into the critical section and returns only when the condition is met. Implementations often call similar functions `down()` or `wait()`. On leaving the critical section, the V operation (Dutch *verhoog*: increase [Dij63]) increases the Semaphore to indicate that a resource became free. This operation is also called `up()` or `signal()`. The binary semaphore with the initial value of 1 is often called Mutex and used with `lock()` and `unlock()` operations. A Mutex implemented with busy waiting is called spinlock.

⁹Results taken on test system `haixagon` with isolated tasks on CPUs #6 and #7 with a shared Third Level Cache (L3\$) and on CPUs #5 and #6 using the main memory.

```

unsigned int sem = 1;           /* initialize as 1 (free) */

void P(unsigned int *sem)      /* "Prolaag" ~ try to reduce */
{
    while (1) {
        ATOMIC {
            if (*sem > 0) { *sem--; break; }
        }
    }
}

void V(unsigned int *sem)      /* "Verhoog" = increase */
{
    *sem++;
}

```

Listing 6.1: Semaphore implementation using pseudo atomic functions.

The typical implementation of a Semaphore is shown in Listing 6.1. The section marked `ATOMIC` can not be implemented in a high-level language like C¹⁰ but must use an atomic assembly instruction like CAS or FAD (Section 2.2.1 on page 44). The example universal Semaphore implementation can not easily be realized with the atomic instructions provided by the x86 architecture. For the binary Semaphore that is initialized with 1 and can otherwise only be 0, the implementation with an atomic Exchange (XCHG) instruction is shown in Listing 6.2. The example code uses GCC intrinsic functions instead of embedding assembly code.

All these implementations are actively waiting on spinlocks. Since suspending is not possible, the blocking `lock()` operation can be replaced with a `trylock()` operation that immediately returns with a boolean value indicating if the Semaphore was obtained or if it was not free. This kind of function is beneficial for the timing analysis because it returns after a constant execution time. The programmer can decide to execute other work before retrying to obtain the Semaphore. In case of a positive result to the `trylock()` operation, the Semaphore must be released with the `unlock()` operation after leaving the critical section.

When handling multiple locks, the risk of a deadlock is present (Section 2.2.1). Deadlock detection and subsequently recovering is in most cases not productive in real-time systems. The occurrence of deadlocks can either be avoided by only granting access to a Semaphore, if the situation is secure in every possible following path [Tan07, Sect. 6.5] or the system can be designed to prevent deadlocks by breaking at least one of the four required conditions [Tan07, Sect. 6.6]. Blocking implementations

¹⁰Although most compilers support intrinsic functions that are implemented with atomic instructions, e.g. `__atomic_compare_exchange()` in GCC.

6. Application Support

```
unsigned int mutex = 1;                /* initialize as free */

void lock(unsigned int *mutex)
{
    while (1) {
        unsigned int local = 0;
        local = __sync_lock_test_and_set(*mutex, local);
        // LOCK XCHG *mutex, local      /* write 0 (locked) */
        if (local == 1) { break; }     /* if it was 1 (free) */
    }
}

void unlock(unsigned int *mutex)
{
    __sync_add_and_fetch(*mutex, 1);
    // LOCK INC *mutex                 /* atomic increment */
}

```

Listing 6.2: Binary Semaphore (Mutex) implementation with GCC intrinsic functions.

can handle the priority inversion problem with temporarily boosting the priority of a task holding a Mutex, but spinlocks can not take influence of OS priorities. For hard real-time tasks, it is generally preferable to use Mutexes and Semaphores only if all accessing tasks are implemented for hard real-time.

6.1.3. Complex IPC objects

The synchronization objects presented in the previous Section are required to protect critical sections. However, these parts of the code must be identified and handled accordingly. The Monitor concept [BW01, Sect. 8.6; Tan07, Sect. 2.3.7] abstracts shared resources and protects the access implicitly. This can be realized in object-oriented programming with private data members. The access is handled by public methods that protect critical sections where needed. Consequently, the user (i. e. the programmer of a real-time application) does not need to analyze side-effects and the timing evaluation becomes easier. Monitors are a common concept in real-time applications and can be implemented for isolated tasks based on spinlocks.

The accessing functions of Monitors are not generally lock-free if they implicitly use spinlocks and can thus be subject to priority inversion. An alternate concept implicitly synchronizing is message-based communication [BW01, Chap. 9]. The basic example is a message queue that supports sending values between tasks. The `send()` and `receive()` functions can be implemented wait-free, i. e. they always return within a limited time.

6.1. Shared-Memory Communication

A message queue can be implemented using a ring buffer of constant size. This avoids the allocation of memory for new elements usually requiring an OS function (system call) not allowed in the isolated task concept. The disadvantage of a ring buffer is the limited number of elements so that the buffer can run full and a strategy of either discarding further elements or overwriting the oldest must be designed. Depending on the application, different properties and guarantees can be realized.

```
unsigned int next_read, next_write = 0;           /* empty */
int buffer[SLOTS];                               /* only for int elements */

void send(int value)
{
    if (((next_write+1) % SLOTS) == next_read) {
        return;                                  /* ERROR: ring buffer full! */
    }
    buffer[next_write] = value;
    __sync_synchronize();                       /* Memory barrier */
    next_write = (next_write + 1) % SLOTS;
}

int receive()
{
    int result;
    if (next_read == next_write) {
        return -1;                               /* ERROR: ring buffer empty! */
    }
    result = buffer[next_read];
    __sync_synchronize();                       /* Memory barrier */
    next_read = (next_read + 1) % SLOTS;
    return result;
}
```

Listing 6.3: Simplified wait-free message queue implementation [Lam77].

A slightly simplified implementation of a wait-free message queue for a single sender and a single receiver both being hard real-time tasks is shown in Listing 6.3. In a dynamic initialization, one of the tasks must allocate the shared resources and the other one needs to connect to them. In both the `send()` and the `receive()` function, their respective pointer `next_write` or `next_read` is first checked for a full or empty buffer (the error notification is skipped in the example). Then, the element of the ring buffer is accessed and finally, the pointer is advanced to the next element. Since each pointer is only modified by one of the two functions and the next to-be-read and -written elements are assigned to either the reader or the writer, this is not a critical

6. Application Support

section and therefore, it does not need locks. Further, the functions do not contain any loops for waiting or retrying, therefore they always return within a limited time.

As mentioned before, the cases of a full buffer while sending or an empty buffer while reading must be regarded in a protocol depending on the application. In the example of a producer/consumer scheme, the consumer could just wait if the buffer does not contain any elements to read and immediately proceed when a new message arrives. Otherwise, the producer must decide if it can wait on a full buffer or if this would violate its real-time conditions and it better drops or overwrites elements. An example for a consumer-centered setting is a DVD burner where the producer must not let the buffer run empty but can wait if it is full. In contrast, if the producer controls a plant and sends the current state to a queue, it can not wait on a full buffer.

If the transmitted elements are large messages, *zero-copy* protocols can be implemented [HP11, App. F.2; YAW05]. This can be accomplished by dividing the allocation of a ring buffer element and the increment of the pointer that signals the reader a new element. With this procedure, the producer announces a new element to the message queue and gets a pointer in return where it can store the element directly in the ring buffer while creating it. If the element is complete, the send operation is committed to make the new element accessible to the consumer. Likewise, the consumer asks for a new element and gets a pointer to the ring buffer location. It can read the element to process and when finished, free it to make the element reusable by the producer.

The example code in Listing 6.3 contains a call to the GCC intrinsic function `__sync_synchronize()`, a memory barrier to avoid consistency problems (Section 2.1.9). Current processors of the x86 architecture use a slightly weaker memory consistency [Mos93; HP07, Sect. 4.6] than *sequential consistency* [OSS09]. On a single processor, a following instruction sees all modifications done by previously executed instructions (*sequential consistency*) [Intel13c, Sect. 8.2]. For multi-processor systems, this is slightly trickier and documented by the processor manufacturers. The C programming language defines in its latest standard (C11) a *weak consistency* model for threads [C11, Sect. 5.1.2.4] and the library [C11, p. 7.17.3]. This allows the optimizing compiler to reorder operations that are not inter-dependent in the same thread [Bat+11]. In the example, the increment of the `next` pointer and the access to the referenced buffer element appear to be independent to the compiler missing the semantic dependency. Thus, the writing of the pointer to the memory could be done before the buffer element is accessed (which could be done if the pointer was held in a register). In that case, the other task would compare against an updated pointer while the value is not stored in the ring buffer, yet. To prohibit the compiler from moving memory stores across this line, the Memory Barrier is placed between those two lines [VN13]. Further, the out-of-order x86 architecture must be instructed not to reorder these instructions which is done with a memory barrier (`mfence`). It depends on the compiler on how to realize this function, e. g.

GCC provides the intrinsic function `__sync_synchronize()`, and the C11 standard introduced the `_Atomic` qualifier [C11, Sect. 6.2.5].

The MCAPI (Section 2.2.3) is targeted at embedded systems but does not consider real-time systems. The literature describes many algorithms for special settings like multiple non-real-time senders and a single hard real-time receiver. Brandenburg et al. provide an overview and evaluation of various lock and wait-free Message-Passing (MP) algorithms [Bra+08]. They consider the lock-free queue of Michael and Scott [MS96b], lock-free buffers of Tsigas and Zhang [TZ99], wait-free buffers of Anderson and Holman [AH00], and wait-free universal constructions of Anderson and Moir [AM95]. Brandenburg and Anderson further analyze the schedulability of reader-writer synchronization for real-time systems [BA09b]. Engel and Völz evaluate the application of lock-free algorithms from HPC (using the MCS lock [Kri+93]) in real-time systems [EV11]. Other related work combines shared memory synchronization with multi-processor real-time scheduling [CRJ07], applies non-blocking synchronization to a robotic system [SSL09], or advances the formal verification of hard real-time systems [Ger+12]. The application of IPC for isolated tasks was extensively analyzed by Raschen [Ras11].

The x86 architecture provides a large number of atomic instructions (Section 2.1.9 on page 38) that allow to apply a wide range of algorithms presented in the literature. Some architectures like ARM only provide the Load-Linked/Store-Conditional (LL/SC) instructions to build atomic operations. This type of implementation does always require a retry-loop in case the conditional store fails. Therefore, on architectures only providing LL/SC, most wait-free algorithms can not be implemented [Ras11].

Alt implemented a demonstration application using the isolated task concept [Alt13]. To transfer video frames from a soft real-time image processing task (producer) to a hard real-time transmitter task (consumer), he implemented a triple buffer. The switching of buffers is implemented wait-free with atomic instructions. The producer always finds a free, unused buffer. In the next iteration, the consumer switches on availability of a new element to that buffer. Raschen describes a sensor-buffer object with wait-free writing of new sensor values while the reader has no timing limits (thus can be normal process) [Ras11].

A different approach to elide locks is to wait until no more users are holding a reference to an object [McK13b], e. g. by using reference counters or Read-Copy-Update (RCU) (Section 2.2.5 on page 51). The latter uses non-blocking entry and exit functions for critical sections to register threads using data structures and subsequently pessimistically estimate when that data structure is no longer used. This is suitable and optimized for mostly read data structures if writing can wait or use callbacks to complete pending actions. RCU is implemented with various optimizations in the Linux kernel and a user-mode implementation is available [MDL13]. The variants that work without system calls are generally suited to be

6. Application Support

used in completely isolated tasks. The grace period could be determined based on the lowest control loop frequency of all isolated tasks.

6.1.4. Dynamic Memory Allocation

The requirement to allocate all needed memory before the application enters the real-time operation follows the rule not to use system calls in isolated tasks. The allocation functions provided by the standard library (e.g. `malloc()`, `new`) are not real-time capable. But it is possible to implement a user-mode allocator that provides hard real-time guarantees as long as it does not run out of space. This allocator must reserve its memory pool at the initialization time. An example is the Two-Level Segregated Fit (TLSF) memory allocator [Mas+04] that allows memory allocation with a guaranteed Worst-Case Execution Time (WCET) [Mas+08].

With dynamic memory allocation, linked lists protected by locks or RCU and Hazard Pointers [Mic04] can be applied. This enables the implementation of advanced synchronization and communication objects that are secure to be used with the isolated task concept and to synchronize between hard real-time tasks and processes in the system partition.

6.2. Management

The setup of the system (Section 4.2) and the start and initialization of the isolated tasks (Section 4.3) should be managed by a dedicated task of the real-time application. Due to unresolved problems with moving tasks to other CPU-Sets while an isolation is active in the system, all isolated tasks must be started, initialized, and isolated in lockstep. Similarly, they must be shut down in a synchronized fashion. Further assignments of a manager task can be the monitoring of isolated tasks and the handling of errors. In the following is presented, how the test and benchmark applications use flags and message queues for synchronization and communication with the isolated tasks.

6.2.1. Control

Generally, isolated tasks can be implemented as threads of a main application or as processes with their own address space. In the latter case, the communication can be based on shared memory segments allocated and connected to during the initialization phase. In the following, an approach to managing threads will be described without loss of generality.

The management task is a thread that continuously interacts with the isolated tasks. If the application requires a high throughput of data with multiple isolated tasks, the concept could even be stretched to dedicated partner tasks with soft

State	Signaled by	Description
INIT	(Manager)	Initialization value
THR_IS_RUNNING	Isol. task	Was first executed by the OS
MA_INIT	Manager	Thread should start initialization
THR_INITED	Isol. task	Thread finished initialization
MA_START_ISOL	Manager	Thread should activate isolation
THR_ISOLATED	Isol. task	Thread activated isolation
MA_RUN	Manager	Thread should enter <i>real-time</i> loop
MA_SHUTDOWN	Manager	Thread should exit <i>real-time</i> loop
THR_LEFT	Isol. task	Thread has left the loop
MA_STOP_ISOL	Manager	Thread should deactivate isolation
THR_UNISOLATED	Isol. task	Thread deactivated isolation
MA_EMERGENCY	Manager	Emergency exit (skip prev. steps)
THR_END	Isol. task	Thread has shut down and will exit

Table 6.1.: Management states as implemented in the benchmark application.

real-time behavior. Alternatively, the management could be provided by start and shutdown functions that are called from appropriate methods of the application.

The example application defines multiple states of operation. In the *initialization* state, the system is set up and the isolated tasks are started. If all threads are prepared, the *isolation* state successively activates the complete isolation for each of them. The threads wait in the *stand-by* state for a flag signal to start the *real-time* work. When the experiment is finished or interrupted by a user, the *shutdown* state handles the reverting of the isolations. Depending on the needs, the threads can be kept running and move back to the *initialized* state to start a new experiment with different parameters. If a different number of isolated tasks is required for each experiment, the threads should terminate and the manager task will start new ones.

The switch between the states is managed by a dedicated flag for each isolated task. This way, the manager can follow a handshake protocol with each task. The following protocol with the states listed in Table 6.1 is applied by the benchmark application used for the evaluation in Chapter 7. The state names indicate who signals each state while the other waits.

The manager task maintains a field of structures holding the data for each task such as the Pthread handle, the task ID (required for CPU-Set handling), the state flag, communication objects, and monitoring variables. This data structure should be carefully designed to separate elements written by the manager and the isolated task to avoid false sharing.

To start the isolated tasks, the data structures are initialized in a loop. Then, each thread is created and signals when it is running. After all threads are started, the manager waits in another loop for each of them signaling that they are running.

6. Application Support

Each thread is then moved into its dedicated CPU-Set. After all threads are moved, the manager starts to signal each of them to start the initialization. This step is required because the initialization must be executed by the task when running on the target CPU to allocate buffers on the correct Non-Uniform Memory-Access (NUMA) node as will be shown in Section 7.4. Since the initialization of the isolated tasks is not depending on each other, it can be executed concurrently with the other tasks. Later, the manager waits for all threads to signal that they have finished their initialization. In the next step, the manager signals one task after the other to start its isolation and waits for the acknowledge before proceeding to the next task. This careful synchronization is due to the Linux kernel modification requiring that only one isolated task at the same time executes the activation by writing to `/sys/isol/cpu_mask`.

After activating the isolation, each task is in a *stand-by* state waiting for the signal to start the actual real-time work. This time can be used to run warm-up routines to load the working set of variables into the caches and to train the branch predictor. This is best done by executing the same function that will be run in the *real-time* state. If that functions has side-effects, it should be modified to not interact with others. The manager then notifies all isolated tasks in a short timespan to enter their *real-time* state and start working. The tasks execute the real-time loop until they are signaled to exit.

On a *graceful shutdown*, the manager changes the state flags of all isolated tasks in quick succession to the *shutdown* state. Before the isolated tasks may deactivate their isolation, the manager waits until all of them have signaled that they have left the loop. Then, one task after the other is instructed to deactivate their isolation and the manager waits for each before proceeding to the next. After this step, the threads free their resources and terminate after signaling that they do so.

An *emergency shutdown* protocol is in place if a task misses a deadline in the handshake protocol (e. g. does not leave the *real-time* state loop) or if an exception signal (e. g. Segmentation Fault or Divide-by-Zero) was received by the application. In this protocol, the manager signals all threads this condition and the threads move through the shutdown process without further synchronizing. With the external controlling of the isolated CPUs described in Section 5.3, the manager also writes a zero bit mask into the patch control CPU mask to immediately deactivate all changes in the Kernel. If the threads have not cleared the Interrupt Flag (IF), their isolation can be deactivated from the outside and in most cases, a recovery is possible.

6.2.2. Supervision

To monitor the correct functioning of the isolated tasks, each task measures the execution time of its main loop and increments an error counter when missing a deadline. The manager periodically checks these counters of all isolated tasks and handles according if the hard real-time condition is no longer met. Depending on

the application, different reactions are advisable. In the physical experiment control application described above, the results could be refused or the test series could be aborted. In more critical application, a warning, a fail-over to other systems or a transition to a safe state should be implemented.

If the isolated tasks frequently exchange data with their manager (e. g. to record the control values), the manager can detect if the tasks are running. Otherwise, the tasks can increment a *heart beat* counter that the manager can observe. Depending on the frequency and volume of monitoring, the manager must be implemented as soft real-time task or even a dedicated manager for each isolated task is required.

For a retrospective evaluation, it is advisable to collect at least the minimum and maximum execution time of the main loops of each isolated task. After deactivating the isolation, the manager reads those values from the tasks and displays a statistic. This enables the developer to evaluate the timing, and to control changes in the code for regressions. An advanced statistic can include a histogram like in the benchmark evaluation (Fig. 2.18). The drawback of every statistic is that they require extra memory which consumes First Level Cache (L1\$) that is more valuable for the real-time work. A histogram should be tailored to have only a minor impact on the jitter and it should be deactivated during production.

6.2.3. Adaptability

Depending on the application, the assignment of PLCs implemented as isolated tasks must be flexible. During the execution of a real-time operation, the control parameters can be updated by using shared variables or message queues. If the algorithms must be modified in a more essential way, this can generally be done during the run-time or after interrupting the real-time operation. Since control algorithms are often designed with graphical tools such as Simulink or Labview, their C/C++ export can be used to generate compiled object files that can be linked to the main application.

The most fundamental change would be the modification of the application's source code and a rebuilding of the program which requires a termination and restart. A more flexible method would be to implement the functions executed by isolated tasks in a shared library that can be loaded dynamically during the execution of the application. This could be done during real-time execution by providing function pointers to the isolated tasks but it contains the risk of a high latency when executing the code for the first time on the isolated CPU because neither the code nor the variables are in the cache. This can only be faced with a sufficiently pessimistic estimation of the WCET. For hard real-time tasks, it would be better to change the fundamental algorithms only between real-time operations with a warm-up phase before the next real-time phase is started.

An alternative to statically or dynamically linked libraries are scripting languages. Klotzbücher et al. evaluated the use of Lua for hard real-time systems [KSB10]. In this

6. Application Support

work, the unpredictable memory consumption of user-supplied routines is identified as major challenge for the estimation of the execution time [KSB10, Chap. V]. This holds similarly true for every function that is provided by third parties either as abstract model that is converted to C code or as library or as script code.

A viable help would be a guiding tool that loads user-provided functions and analyzes them by executing them in a sand box prior to starting the real-time operation. The average execution time is easy to measure, but for hard real-time systems, the WCET is required but can not be measured in a strict interpretation. What the application can do automatically is only a best-effort test to estimate a *bad-case* execution time by executing the function under heavy system load with an environment (i. e. control parameters etc.) as close to the real operation as possible. However, a safety margin should be added with sufficient experience.

Since the isolated tasks are not allowed to execute system calls, this can either be prevented by linking the code with a reduced C library or by statically or dynamically analyzing the code. A static analysis can be executed on object files with library- and symbol checks such as `ldd` and `nm`. This covers the whole file independently from unused functions or code paths. A dynamic check executes the function under control of a tracer such as `strace` that displays every system call.

In any case, the running real-time application should monitor the isolated tasks for latency violations and provide the operator with detailed information about the latency distribution of all isolated tasks to help identifying errors.

6.2.4. Exceptions in Isolated Tasks

Above, the *emergency exit* protocol was presented. This protocol is also utilized if the monitoring explained in the previous Section detects a hung task or if the application receives an exception signal.

Blocked tasks can occur if their implementation uses loops that become endless loops. Generally, the code of the isolated tasks being hard real-time code should be evaluated for its WCET. But not only during development, unforeseen conditions can occur that lead to an isolation not responding to its manager. Since the manager can only interact with the isolated tasks via shared memory which requires the receiver to read its messages, a stable solution for terminating blocked tasks should be implemented.

If the isolated tasks use the Interrupt Flag method to activate the complete isolation, this can hardly be reverted from the outside. A possible approach would be to send a Non-Maskable Inter-Processor Interrupt (NM-IPI) but this had to be implemented in the Kernel. Using the Kernel Patch with deactivated Local Advanced Programmable Interrupt Controller (localAPIC) timer, the isolated tasks do not need to clear the IF to be completely isolated from external interrupts. In this mode, the activation of the isolation is done by the patch and can be controlled from other

CPUs. Thus, the manager is able to deactivate the isolation of non-responding tasks as part of the *emergency exit* protocol.

If an isolation triggers a CPU exception, for example by accessing an invalid address or by dividing by zero, the CPU exception is not blocked by the IF but the Linux kernel interrupt handler is executed. While the *page fault* can sometimes be handled by the Kernel by swapping a page back into the main memory, the fault can also be due to an overflowed array access. The latter case is called *Segmentation Fault* and forwarded to the causing application as UNIX signal. Other examples are the Bus Floating Point Exception that can be triggered by invalid arithmetic, or the Illegal Instruction Error. Especially when loading functions provided by a user (using the concept of the previous Section 6.2.3), such errors are more common than in carefully verified real-time applications.

Unhandled error signals cause the OS to terminate the application. Since the isolation concept executes tasks that can not simply be terminated by the OS, these signals must be handled to initiate an emergency shutdown and keep the system responsive.

6.3. Input and Output

A real-time task must react in a defined time to stimuli. In the previous Section, methods are presented to transmit messages to isolated tasks that can trigger such a reaction. But physical signals are still missing. In fact, most hard real-time requirements origin from the need to keep up with physical systems (e.g. Programmable Logic Controller). In such cases, an external signal from the physical world enters the computer, must be processed in real-time and a physical output is required within a maximum time frame. These signals are usually detected by sensors that convert physical quantities to electrical current or voltage. These analogue values can be converted to digital numbers by Analog to Digital Converters (ADCs). The output can be converted to voltage by Digital to Analog Converters (DACs) to drive actors. Binary values (on/off) or bit masks can be read and written by General-Purpose I/O (GPIO) interfaces. They provide multiple *pins* that can be configured to sensing input values or driving an output. These devices are subsumed as Data Acquisition (DAQ). Other functions provided by peripheral devices are high precision clocks [Bül+04] or co-processors like Field-Programmable Gate Arrays (FPGAs) or Digital Signal Processors (DSPs).

Real-time capable hardware drivers for Input/Output (I/O) devices are a challenge. General-Purpose Operating Systems support many peripheral devices but the drivers are not optimized for predictable timing. In contrast, Real-Time Operating Systems only support a limited set of devices. If real-time drivers or libraries are available by the device vendors, their Application Programming Interfaces (APIs) are not unified [Kis05]. Since the isolated tasks are not allowed to use system calls, they

6. Application Support

Interface	Application
I ² C	Bus, mostly internal functional units
RS-232, Parport	Point-to-Point legacy ports, easy to program
PCI and PCIe	Expansion card connectors, plug and play
USB	Tree of devices, complex protocol requires driver stack
Ethernet	Network, protocol stack requires driver

Table 6.2.: Common hardware extension interfaces of the x86 architecture.

can not use the usual drivers provided by Linux to access I/O hardware. The Comedi project¹¹ provides user-mode drivers for a number of DAQ devices. However, it must be evaluated, if the applied methods do not use system calls during real-time operation. The Linux User I/O driver framework [AM12] supports writing drivers that are executed in user-mode. This is not primarily targeted for real-time, but to improve the throughput and latency of high-performance applications such as networks [Cor13d] and block device I/O [Cor13g] or to improve safety-critical applications [WDL11]. In this Section, alternative methods to circumvent these constraints by using direct hardware access, user-mode drivers and even interrupts are presented. This restores the applicability of isolated tasks to the level of tasks in an embedded RTOS.

6.3.1. Direct Hardware Access

Input/Output devices can be build into the system as a fixed component of the main board or they can be plugged into an interface connector. A wide range of interfaces enables the x86 architecture to be very flexibly extended (Table 6.2). The x86 architecture generally supports two mechanisms to interact with devices, I/O ports and memory-mapped I/O. The I/O ports are accessed with `IN` and `OUT` instructions. Similar to the `CLI` instruction, they can be used in user-mode with the appropriate process privilege configured with `iopl(3)`. Memory-mapped I/O uses configuration registers mapped to the physical address space. A process with `root` privileges can map this address region into its virtual address space with the `mmap()` function. After this assignment has been installed, an application can access the device with common load and store instructions (e. g. `MOV`) or pointer assignment in the C programming language. Both methods do not require the OS after initialization.

The legacy interfaces *serial port* (RS-232) and *parallel port* (Centronics printer interface) are rarely available on current commodity systems. Some embedded systems still provide them because they are very easy to access on the hardware level. They use few I/O ports that are well documented. In contrast, the Universal Serial

¹¹Online: <http://www.comedi.org> (visited November 12, 2015)

Bus (USB) port which is common for the connection of peripheral devices such as keyboard, mouse, and printer, is very difficult to implement because of its complex protocol.

The (commonly) internal interfaces Peripheral Component Interconnect (PCI) and PCI Express (PCIe) are best known for their connectors to plug in expansion cards like graphics accelerators or I/O extension adapters. The expansion cards are initialized by the firmware (BIOS/UEFI) or the OS and assigned I/O ports, memory-mapped I/O and interrupts. After the initialization, those devices can be accessed like build in functional units. The newer PCIe interface supports very high throughput which makes it suitable to be used with graphics accelerators, Network Interface Cards (NICs), special HPC interconnects (e.g. Infiniband), etc.

The complexity of programming such a device depends on its features. Expansion cards with serial and parallel ports extend the system with these legacy interfaces that are no longer available on new systems. They can be programmed like the build-in ones. For interfacing with physical experiments and machines, DAQ expansion cards provide GPIOs, ADCs and DACs, possibly accompanied with an FPGA. Those devices are often programmed by writing output values to memory-mapped I/O registers and reading input values from there. More complex PCIe devices such as NICs require registered memory buffers to write received network packets to. Thus, it depends on the device how complex the realization is, but it is generally possible to implement a direct hardware access in the user-mode application.

It is advisable, to leave the initialization up to the driver of the OS and only implement the required interaction during real-time execution. Some vendors of I/O devices for real-time systems already provide user-mode libraries to access the device without system calls independently from the OS.

6.3.2. Protocol Stack

Since the implementation of user-mode drivers for common I/O devices connected to PCI or PCIe is much simpler than the USB or Ethernet protocol stack, these adapters should be preferred. However, if a hard real-time capable network connection (e.g. Time-Triggered Ethernet [Kop08]) or a USB device is required, their protocol stack must be implemented also in the user-mode to be usable in isolated tasks. Examples are a user-mode driver for Real-Time Ethernet [WB11], the IP network stack *lwip*¹², and the Sandstorm web server using a user-mode network stack for performance reasons¹³. Practical experience is further presented in Section 7.6 and Chapter 8.

¹²Online: <http://savannah.nongnu.org/projects/lwip/> (visited November 12, 2015)

¹³Online: <http://heise.de/-2063812> (visited November 12, 2015)

6.4. Interrupts

So far, the implementation of isolated tasks was only possible using the time-triggered paradigm of real-time programming. This is due to interrupt handlers always executing in kernel-mode which includes OS code. Additionally, by excluding every interruption, the formal verification of the hard real-time task becomes very easy.

However, in embedded systems using an RTOS, methods are known to schedule and validate event-triggered and preemptively scheduled tasks [BW01, Chap. 13]. In this setting with an unchanged Linux system, essentially every execution of Linux kernel code must be prevented because it is too complex to verify. Even short functions can have side-effects in the whole system. An important example is the low-level interrupt handler that checks the internal work queues for pending Kernel tasks before returning to the interrupted user-mode process. In other words, every timer or keyboard interrupt does not only handle its event, but possibly activates arbitrarily long running maintenance functions.

The basic concept for isolated tasks cleared the Interrupt Flag (IF) to securely block all hardware and Inter-Processor Interrupts (IPIs) (Section 4.4). With the Kernel modification deactivating the IPIs and the localAPIC timer, clearing the IF is no longer required because all external triggers of interrupts are disabled (Section 5.3). Using this mode, a selective reactivation of certain interrupts becomes possible. In the following, a method to handle these interrupts with a user-mode function without the execution of Linux kernel code is presented. This enables using the event-triggered paradigm and even to use preemptive scheduling in an isolated task. Ultimately, the execution of an embedded RTOS on an isolated bare-metal CPU becomes feasible.

6.4.1. User-Mode Interrupts

In the Linux kernel, the usual control flow of an interrupt is the discontinuation of the user-mode process by the processor's logic with the conservation of the current context (i. e. Instruction Pointer (IP), Stack Pointer (SP) and Flags register) to the stack and the context switch to kernel-mode where the `interrupt` function¹⁴ starts the handling. It saves the remaining registers and calls the actual handling function according to the interrupt vector. After the return from that function, the saved registers are restored and the handler returns to user-mode. The context switch and the restoring of IP, SP and Flags register is managed by the hardware automatically. As explained before, the Kernel may decide to start pending work before returning to the interrupted user-mode process.

¹⁴Linux file `arch/x86/kernel/entry_{32,64}.S` (Linux 3.12)

The goal is to execute only known code to handle an interrupt on the isolated CPU and exclude all Linux code from execution to be sure that not unexpected functions and delays are inserted. The x86 architecture enforces the handling of interrupts in the kernel-mode [Intel13c, Sect. 6.12.1.1; AMD12b, Sect. 8.7.4]. Therefore, the handler must be inserted into the Kernel's address space. Consequently, a method was developed to execute the interrupt handling function in the user-mode context of the isolated task. Only a springboard function is inserted into the Kernel that immediately returns to user-mode after manipulating the stack to execute the user-mode handler. This method was implemented as kernel module that can easily be loaded into the running Linux kernel and is portable to new versions of Linux. The module is configured by virtual `sysfs` files.

The detailed proceeding is as follows: During setup, an isolated task bound to a CPU can install the user-mode interrupt springboard handler for a dedicated interrupt vector by writing the address of the user-mode handler function to the file `/sys/irq/0x7A/usi`¹⁵. This installs the springboard into the Interrupt Descriptor Table (IDT) and stores the user-mode address. The address of Linux's handler function is overwritten, but stored to undo the change later. The location of the IDT data structure can be obtained without support of Linux with the `SIDT` instruction. Since the user-mode address is only valid in the context of the initiating process, the Task ID (TID) is also stored. When the springboard handler is executed, it checks if the correct task is active and aborts otherwise. This information can be found on the Kernel stack because Linux stores the TID there for easy referencing. On its kernel-mode stack, it finds the stored user-mode SP and the address of the next instruction of the interrupted task (IP). This IP is replaced with the registered address of the user-mode handler function. The overwritten return address must not be lost to be able to continue the interrupted task. Therefore, it is pushed to the user-mode stack. Then, the springboard immediately returns to user-mode.

Since the IP in the stored context was changed to the handler function, the hardware logic of the interrupt return starts executing that function instead of directly continuing the interrupted task. When the inserted handler returns, it reads a return address from its stack and that is the address where the task was interrupted which was moved from the kernel stack to the user stack. This mechanism inserts a function into the instruction stream. Of course, the handler must save the context and all registers. This is done implicitly by the interrupt handling logic (for the context) and the Kernel interrupt handler (for the registers). Therefore, the user-mode handler needs to push the Flags register and all used registers to the stack and restore them before returning. This is done in the experimental implementation by a `Manager` function that in turn calls the actual handler provided by the programmer. By doing so, the handler can be a standard C function like illustrated in Listing 6.4.

¹⁵Example for vector `0x7A = 7Ahex = 122`

6. Application Support

```
unsigned counter = 0;                                /* global variable */

void my_handler(void)                               /* user-mode handler function */
{
    unsigned char value;
    counter++;                                       /* increment global counter */
    value = io_read(CHANNEL)                        /* read value from ADC */
    msg_send(value);                               /* send to message queue */
}

int main()
{
    unsigned char value;
    /* ... */
    umi_install(0x7A, my_handler); /* install on vector 0x7A */
    isol_activate();               /* activate isolation */
    while (1) {                   /* time-triggered loop */
        if (msg_recv(&value)) {   /* if message in queue... */
            process(value);       /* ...process value */
        }
    }
    /* ... */
    isol_deactivate();            /* deactivate isolation */
    umi_uninstall(0x7A);         /* restore original int. handler */
    printf("handled %u user-mode interrupts\n", counter);
}

```

Listing 6.4: Implementation and use example of a user-mode interrupt handler.

The install function internally stores the address of the handler function and installs its own manager into the Kernel springboard handler.

If the system executes Linux in 64 Bit mode, the according calling convention includes a *red zone* [SysV10, Sect. 3.2.2] that is a region of 128 Bytes below the SP. With the stack growing down on the x86 architecture, the red zone can be used for local data by leaf functions (i.e. functions not calling other functions) without needing to adjust the SP (to reserve this space on the stack). In normal operation the red zone is never overwritten, but if an interrupt handler using the stack is unexpectedly inserted, the stack must be expanded by 128 Bytes to protect the red zone. Therefore, in 64 Bit mode, the springboard does not just push the return address to the user-mode stack but first decrements the SP by 128. The manager function in the user-mode uses the correct return address to continue with the interrupted task but needs to additionally remove the 128 Bytes of red zone

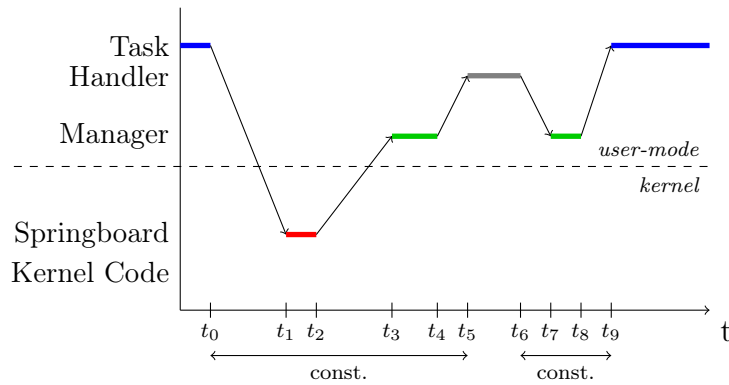


Figure 6.4.: Control flow for user-mode interrupts.

protection from the stack. This can conveniently be done with the `RET N` instruction¹⁶ that removes N Bytes from the stack after reading the return address.

On x86 systems, the context switch to the kernel-mode lasts rather long, an empty interrupt handler suspends the running task for 500 to 1000 cycles¹⁷. But the variance is low and in the absence of unknown Kernel code, the execution time of the springboard handler up to the return to user-mode is limited allowing a tight estimation of the WCET. The control flow is shown in Fig. 6.4. At t_0 , a signal triggers the interrupt and the springboard handler is executed after the switch to the kernel-mode at t_1 . The execution time of the handler is very short and at t_3 , the manager function in user-mode is executed. It stores the registers and calls the handler. The interval from t_0 to t_5 has a constant maximum. When the handler returns to the manager, that restores the context and registers of the interrupted task and returns the control to it. The interval t_6 to t_9 is also constantly bounded. In the timing verification, the WCET of the handler must be determined and added to the constant overhead of the user-mode interrupt mechanism. The implementation of this mechanism was tested on an Intel Core 2 system. The results are listed in Table 6.3.

Bracy et al. present a new paradigm light-weight inter-core communication called Disintermediated Active Communication (DAC) [BDJ06]. This is based on a hypothetical extension of the instruction set to register a memory location (on the granularity of a cache line) and an instruction to register a call-back (user-mode interrupt handler) function that is called when the memory location is invalidated by a write access of another processor. This user-mode interrupt includes minimal context saving to gain a very low latency. Such a mechanism is applied in the concept processor Pangaea [Won+08] demonstrating low-overhead fine grained thread syn-

¹⁶This instruction is commonly used by calling conventions where the called function must remove its parameters from the stack.

¹⁷On Intel Core 2: 589 cycles minimum, 643 cycles average

6. Application Support

Offset		Cycles	Diff.		Action
Begin of Springboard	t_1	313	313	$t_1 - t_0$	Switch to kernel-mode
Begin of Manager	t_3	779	466	$t_3 - t_1$	Switch to user-mode
In Handler	t_5	817	38	$t_5 - t_3$	Save context and function call
End of Manager	t_8	893	76	$t_8 - t_5$	Return and restore context
After Interrupt	t_9	931	38	$t_9 - t_8$	Return to task

Table 6.3.: Timings of user-mode interrupts on Intel Core 2 with nearly empty handler function.

chronization. This example recognizes the importance of a low-overhead, low-latency and low-jitter asynchronous notification mechanism similar to IPIs but without the overhead of entering the kernel-mode. It is shown how such a hardware support could be implemented with acceptable effort but it was not implemented on actually available general-purpose processors.

6.4.2. Interrupt Sources

The trigger of user-mode interrupts must be carefully selected. The Interrupt Requests (IRQs) of devices can be routed to a selected processor executing an isolated task if the device is not used by other parts of the system. Similar to the direct I/O, the device can be initialized by the OS and then reserved for use by isolated tasks. User-mode interrupts can also be triggered by the units of the localAPIC that mainly provides a flexible timer. Since each CPU has its private localAPIC, there is no risk of sharing. For a higher timing precision, special hardware exists (e.g. high-precision timers, HPET [ST93]). But to use them in isolated tasks, the OS must first be configured to not using them. Finally, IPIs can be used to trigger user-mode interrupts. They can even be generated from the user-mode by writing to the memory-mapped configuration registers of the localAPIC. This allows to send asynchronous signals between isolated tasks without interaction of the OS.

6.4.3. Preemptive Scheduling

With selected interrupts enabled in isolated tasks, a preemptive switching of tasks can be implemented. This is similar to user-mode threads (e.g. GNU Pth [Eng00], Application-Level Scheduling [Li+04]) but works without the interaction of the OS. The essential resources required for an embedded OS are a processor, a region of memory, and a timer interrupt for preemptive scheduling. The first two are generally available in a process and the timer interrupt can be realized with UNIX signals that behave like interrupts in the view of the process [Roc08, Chap. 9]. Signals asynchronously interrupt the process and its state is automatically preserved until

the end of the signal handling. They can be triggered by events as well as one-shot and periodic timers. Marx ported FreeRTOS¹⁸ to an isolated Linux process using UNIX Signals as timer interrupt [Mar10]. This concept can be transferred to user-mode interrupts. With the applied embedded RTOS being portable to an isolated task, existing hard real-time applications can be integrated in x86 multi-processor applications.

A proof-of-concept implementation manages registered tasks in an array of control blocks each holding a dedicated stack and a field to store the SP while other tasks execute. The task switch and scheduling function is registered as user-mode interrupt handler with the service described in the previous Section and triggered by a periodic timer of the localAPIC. For a task switch, it pushes the Base Pointer (BP) to the stack and stores the SP in the task control block. Since all other registers were preserved before by the user-mode interrupt manager, the whole context of a task can be restored with only its SP. Then, the next task is selected, its preserved top of stack is copied to the actual SP register and the previously pushed BP is restored. When the scheduler returns, the stack was exchanged with that of another task. The return address from the new stack results in executing a different task after this context switch.

During initialization, the stacks of the user-mode threads must be prepared in a state corresponding to the point when the scheduling function switches to a different task. Only the task that will be activated first starts with an empty stack. The initialization jumps to that task function instead of a function call because the call would store a return address that will never be used. On an Intel Core 2 system¹⁹, the task switch time under complete isolation was measured between 1368 to 1480 cycles which includes the user-mode interrupt handling and the context switch. Linux switches tasks with a minimum of 5000 cycles, a median of approx. 8000 cycles and a maximum an order of magnitude above that (Section 2.3.5).

¹⁸Online: <http://www.freertos.org> (visited November 12, 2015)

¹⁹Test system poodoo, Appendix A.1.1

7. Architectural Influence

Hard real-time by definition requires a formal verification of all running code. Scheduling algorithms for multiple real-time tasks are researched on for decades [LL73; BGP97; Car+04] and they can be regarded as mostly solved [But05]. However, in multi-processor systems, additional challenges must be covered [GR05; ACD06; Bak06; Cal+07; BCA08; SE11; SZ13]. Many publications make the problem manageable by simplifying the assignment by either using a synthetic (idealized) hardware or by reducing the tasks in number or dependencies. For complex real applications, the effort grows extremely. As stated before, the concept of isolated tasks on dedicated processors eases the formal verification by reducing the code that must be regarded to the real-time application itself (Chapter 4). In particular, the large code base of the General-Purpose Operating System (GPOS) (Section 4.5) as well as inter-processor dependencies (caused by Inter-Processor Interrupts (IPIs), Section 6.1) are suppressed while still keeping their services fully available in the system partition. The timing of the resulting single-processor bare-metal system can either be modeled or measured (Section 2.3.5 on page 64). Even for an embedded preemptive Real-Time Operating System (RTOS) running in an isolated task, established methods for single-processor systems can be used to predict and verify their timing. This allows to formally verify the actual application with reasonable effort without the need to simplify the problem which would distort the results. All methods of verification require the knowledge of the Worst-Case Execution Time (WCET) of the comprehended tasks [PS08]. When designing hard real-time systems, it is crucial to understand every effect and every setting of the system.

Future systems integrate multiple processors where the optimization of applications requires specific knowledge about the processors and compilers [Fog13a; HW10]. The experience from High Performance Computing (HPC) can be valuable similarly for real-time systems. Independently from real-time research, the term OS noise [Tsa+05; Mor+11] describes variations in the execution time that originates from the operating system's interrupts and scheduling. Further, the so-called timing anomalies [LS99] may even cause a faster execution when adding instructions. Following the experience, that massively parallel applications are sensible to execution-time deviations in single processes, HPC systems start utilizing real-time methods to achieve a better predictability and hence, a faster execution [ALL12].

This Chapter covers information relevant for the estimation of the execution times – especially the WCET – on x86 multi-processor systems. In Section 7.4, new research is presented that promotes the application of Non-Uniform Memory-Access (NUMA)

7. Architectural Influence

systems for isolated hard real-time tasks by combining HPC research on those architectures with hard real-time demands. Finally in Section 7.6, the experiences from Non-Uniform Memory-Access are transferred to non-uniform Input/Output (I/O) access to reduce the jitter of accessing peripheral devices. Essentially, this hardware-related Chapter is independent of the previously presented concept of isolated bare-metal tasks. Furthermore, this research applies to all real-time applications that struggle to confine the jitter on x86 multi-processor systems. Some experience for implementing hard real-time on x86 Uniform Memory-Access (UMA) systems was published before [Zha+05], but to the best of my knowledge, there are no publications available so far, that analyze memory and cache jitter and PCI Express (PCIe) I/O on NUMA architectures for hard real-time applications.

7.1. Estimation of the Execution Time

The isolated task concept targets the verification of the schedule that waives all influences by the GPOS and other processors reducing the code to cover to the hard real-time task itself. The execution time of basic blocks and tasks must be estimated for most real-time verification and scheduling methods. Further, a *tight* estimation is preferable to optimize the system utilization. Very important is the Worst-Case Execution Time (WCET) [PB00], but for some methods, the Best-Case Execution Time (BCET) is also required. Generally, the approaches of formal analysis and measurement can be distinguished. The formal verification can be separated into the verification of the task schedule and the determination of the WCET of each task (Section 2.3.5). If the hardware is too complex or not sufficiently documented to derive the WCET by formal methods, only a pessimistic approach of measuring under heavy system load is possible. To tighten the resulting WCET, the system load can also be modeled according to the real application. In the context of this work, this exception is made for the execution time of basic blocks while their combination to the Control Flow Graphs (CFGs) of tasks and the schedulability analysis remain formally verifiable.

Calandrino et al. present their expectations of future architectures: Those will have many cores with deep cache hierarchies [CAB07]. Although six years later, those architectures are not on the horizon, yet. But the general idea of deep hierarchies of shared caches is still valid (and investigated for HPC systems, e. g. with Intel's Single-Chip Cloud Computer (SCC) [SCC10]) and it is sensible to evaluate their handling for real-time systems. In Section 2.1.8, some basic characteristics were presented. For most applications, the average values (e. g. of a memory access) are relevant because a long execution time levels out deviations of single instructions. In concurrent applications, the jitter becomes disruptive if fine-grained synchronization points spread the delays to cooperating tasks. In real-time systems, the potential apposition of several instructions that are delayed to their maximum execution time

results in the WCET. Therefore, the maximum latencies of simple instructions are relevant in this context. In contrast, the minimum duration can be interpreted as the architectural effort if an elimination of the instruction (by optimization) can be foreclosed. The difference between minimum and maximum time is defined as jitter. Since a small jitter facilitates a tighter estimation of the WCET, methods to reduce the maximum deviation (and hence, the jitter) are beneficial for hard real-time systems [Bon+12].

Regarding influences to the CFG, an isolated processor can be regarded like a single-processor system. But the execution time of basic blocks depends on architectural effects in shared functional units like the Last-Level Cache (LLC). A lot of research about the reduction of cache-induced jitter on special architectures is published like specially designed Systems-on-a-Chip (SoCs) [Ros+07; And+08], locking caches [Tam+04; Cam+05], a software emulation of locking caches on x86 [BSF04], scratchpad memories for hard real-time [PP07], or a suggestion of other hardware modifications [SM08; LLX09] that are all not practicable on current x86 hard real-time systems. But the estimation can be based on existing research for x86 single-processor systems. For example, Petters and Färber present an approach to derive the WCET by automatically instrumenting the code based on the compiler’s optimization information [PF00]. An approach to control the influence of the operating system (OS) on the cache is presented by Liedtke et al. They regard single-processor real-time systems where cache effects make the estimation of WCET hard. On modern processors, the hit/miss ratio is very high, therefore disabling caches would avoid its influence but at the same time yield an extremely reduced performance [LHH97]. Altmeyer et al. present an efficient method to calculate a tight bound of the cache-related preemption delay which is needed to analyze the WCET of preemptable tasks [AMR10]. Similar analyzes were published by Reineke et al. [Rei+07], Wilhelm et al. [Wil+09], and Ishikawa et al. [Ish+13]. According to Wilhelm and Reineke, these are important steps towards better WCET estimation [WR12].

To practically evaluate the execution time jitter of a small task on an idle system, the following benchmark was executed on an isolated Central Processing Unit (CPU) of the test system `haixagon` (Appendix A.3.1). All other CPUs on the same socket were left idle and the system on the other socket was also mostly idle to reduce the influence of concurrent execution as much as possible. The test routine repeatedly accessed variables in a buffer of 4 KiB size. Both code and data are small enough to fit in the First Level Cache (L1\$). Before the measurement, some warm-up iterations were executed to load all required cache lines and train the branch-predictors [CP00]. The result for up to a million memory accesses shows a very low jitter of only 4 cycles (Table 7.1). This demonstrates, that the x86 architecture is in fact capable for hard real-time execution with low jitter if the influences of other processors on the caches can be ruled out.

7. Architectural Influence

Memory accesses	Min.	Avg.	Max.	Jitter (%)
1	24	25	28	4 (16.7)
4	24	29	32	8 (33.3)
16	48	57	64	16 (33.3)
64	184	188	204	20 (10.9)
256	792	794	796	4 (0.5)
1 024	3 096	3 098	3 100	4 (0.1)
4 096	12 312	12 314	12 316	4 (0.0)
16 384	49 176	49 178	49 180	4 (0.0)
65 536	196 632	196 633	196 636	4 (0.0)
262 144	786 456	786 458	786 460	4 (0.0)
1 048 576	3 145 752	3 145 754	3 145 756	4 (0.0)

Table 7.1.: Execution time jitter (in CPU cycles) of a small task using 4 KiB buffer (code and data in L1\$) on an idle system after warm-up.

In the following Sections, the execution time is measured under various load scenarios to illustrate the added effects in multi-processor systems over single CPUs. But depending on the requirements, more elaborate (and expensive) methods can be applied.

7.2. Multi-Core Systems

Although generally not distinguishable from a programmer’s point of view, this Section explicitly regards multi-core chips instead of general multi-processor systems. The most important common detail is that multiple processors in the same package share their Last-Level Cache and other common infrastructure (e.g. the system bus). This architecture was developed as an easy extension from single- to multi-processor systems by placing multiple dies in a single package and later by integrating multiple processor cores on a single die. Multi-processor systems with multiple sockets are regarded in the following Sections.

Since multi-core processors were the base of the first widely available multi-processor systems in the Commercially off-the-Shelf (COTS) market (Section 2.1.9), many publications regard this special type. With multiple independent execution units sharing infrastructure such as the Last-Level Cache, the system bus, and the Memory Controller (MC), new dependencies are introduced that take an influence on the execution time. Wilhelm and Reineke present an overview of currently unsolved problems coming from the unpredictability of multiple processors and architectural jitter (Cache; Pipeline; Translation Look-aside Buffer, TLB) [WR12]. Dasari analyzes the new architecture for real-time applications [DNA11] including the contention of

the memory bus [Das+11] (assuming a UMA system with a single RTOS instance and the knowledge of all real-time tasks) and other sources of unpredictability such as shared caches, interconnects, power management and some more [Das+13]. The researchers Pellizzoni, Betti, Bak and Schranzhofer cooperate for a solution for UMA systems with an emphasis on memory transfers [PC07]. They present co-scheduling of tasks [Pel+08], worst-case delay analysis [Pel+10], and resource access models [Sch+10] to finally propose an execution model to control the operating point of shared resources to remain below the saturation limit [Pel+11]. Thus, the confinement of jitter induced by cache and memory access is very complex, especially if unrestricted tasks are executed concurrently on other cores of the same package.

It follows that it is very expensive to formally derive the WCET of basic blocks in a multi-processor system covering all theoretically possible interactions and collisions. In the reality, a system behaves more friendly. Although the academic definition of *hard real-time* demands a formal analysis which must add all possible delays to the WCET, a real system shows much less jitter. It happens very rarely, that multiple *delaying events* cumulate to the calculated WCET. The benchmarks cannot prove *hard real-time*, but the distribution of the times in a histogram allows a first rating that can be extended by a security margin to a practically usable tight estimation.

Load	Min.	Avg.	Max.
16 KiB	52	52	64
32 KiB	52	52	60
64 KiB	52	52	68
128 KiB	52	52	68
256 KiB	52	52	60
512 KiB	52	52	60
1 MiB	52	52	60
2 MiB	52	52	60
4 MiB	52	52	1268
8 MiB	52	52	1660
16 MiB	52	52	1636

Table 7.2.: Influence of load using the shared cache on the inclusive private cache of another core (latencies in CPU cycles).

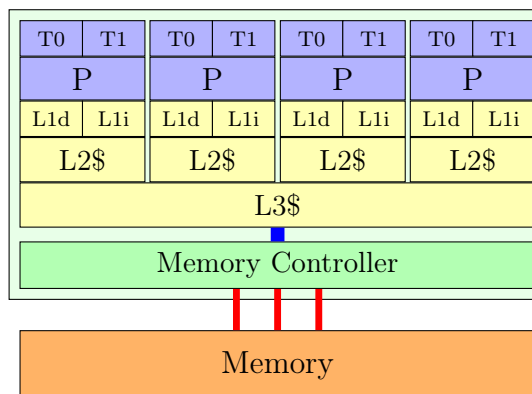


Figure 7.1.: Multi-core architecture (Intel Nehalem) with four cores and two logical threads each.

The effects of a shared cache where tasks on other processors may cause the eviction of other task's cache lines are thoroughly analyzed [YZ08]. The best results are obtained if the hard real-time tasks use only their private caches. Since the

7. Architectural Influence

isolated task concept targets at very high frequency applications, this is advisable anyway. To evaluate the influence of other tasks on such an isolated task that uses only its private caches, an Hourglass benchmark was executed on a micro-kernel (*smp.boot*, Appendix B) while a load process on a different processor using a varying memory range was executed. Only two processors execute load and benchmark, all other processors as well as interrupts are not activated to analyze the hardware effect strictly without any unknown interference. The load covers memory ranges using only the private cache, the shared Last-Level Cache and main memory access. This experiment was executed on the test system *xaxis* (Appendix A.2.1) with an Intel Nehalem QuadCore processor with a single shared Third Level Cache (L3\$). The results (Table 7.2) indicate in fact an influence of a load that uses a fair amount of the shared Last-Level Cache onto the private cache of a different processor. Minimum and average do not depend on the load, and are very close together which indicates that the maximum values are detected only rarely. The occasional outliers cause a steep increase of the maximum execution time if the load uses a large portion of the 8 MiB shared Last-Level Cache.

Since the benchmark routine uses only its private caches, its execution should not be influenced. But the inclusive caching algorithm of Intel processors is not aware of the temporal locality of the private caches and may evict cache lines from other processor's private caches when they are displaced from the shared Last-Level Cache. They are called "cache victims" by Jaleel et al. who analyze this effect inherent to inclusive caches to improve the average throughput [Jal+10]. In Fig. 7.2, a simplified multi-core processor is illustrated. This example uses a two-way associative cache (Section 2.1.4), hence every element can be stored in one of two possible cache elements of the above level. In the example, the left processor has successively used the elements m_0 to m_3 resulting in loading their contents to the shared Last-Level Cache. The right processor uses only element m_5 that was loaded into cache element c_1 (evicting m_1) and q_1 . The left processor reuses all its data beginning with $m_0=A$ which was still in the shared Last-Level Cache. This is the state displayed in Fig. 7.2. Now, element m_1 is required by the load process executed on the left processor but it is not present in the shared Last-Level Cache. It can be loaded into c_1 or c_3 (because of the two-way associativity). If the cache management selects c_1 , its content X must be evicted from the shared Last-Level Cache and also from any higher level to meet the inclusive condition that all elements must be present in lower levels. Therefore, the execution of the left processor may cause the eviction of element m_5 with value X from the private First Level Cache q_1 of the right processor. It depends on the access history which element of the shared Last-Level Cache is selected by the cache management, but the measurement indicates, that collisions in fact happen.

In an exclusive cache, the element could remain in the First Level Cache although the other CPU uses the entire shared Last-Level Cache. This effect can be observed on AMD processors that use an exclusive caching algorithm [HMN09]. The test series displayed in Fig. 7.3 were executed on the test systems *devon* (AMD Opteron,

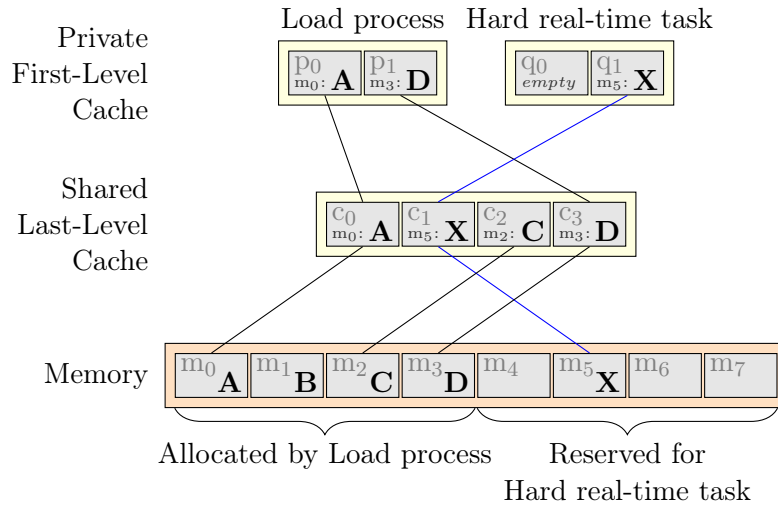


Figure 7.2.: Example: cache content of a multi-core processor (simplified).

Appendix A.5.1) and `haixagon` (Intel Xeon, Appendix A.3.1). Both systems were configured to execute only two isolated tasks on processors sharing their Last-Level Cache. One processor executed a load process reading and writing a buffer varying from 1 KiB up to 256 MiB while the measurement task accessed only a single element of a 1 KiB buffer. The benchmark was executed for 2 Minutes and collected only minimum, average and maximum values to avoid the statistic taking influence on the caches. The minimum and average are constant (216/270 cycles on the inclusive Intel system, 338/395 cycles on the exclusive AMD system) for different load working sets while the maximum depends on the load's working set size (Fig. 7.3 on the following page). On the Intel processor with an inclusive cache \bullet , an increasing of the maximum execution time at 12 MiB load working set size can be observed while the AMD processor with exclusive cache \blacksquare is much less influenced by the load working set.

On processors with inclusive caching, this effect can only be reduced by limiting the usage of the shared Last-Level Cache by all processors. There are proposals to use cache-coloring [LHH97] or hardware extensions [Seb01], but they are very intrusive or costly (large parts of the memory can not be used) or not applicable with available processors.

The cache of x86 processors can be configured by several parameters, for example the *write-through* or *write-back* strategy for modified data. The default setting is write-back which transfers only modified entries to lower levels when they are evicted or requested by other cores. The setting write-through transfers every change immediately to the lower levels so that all levels are always consistent and an entry can simply be dropped or shared with another core. But this comes at the cost of increased traffic for every modification.

7. Architectural Influence

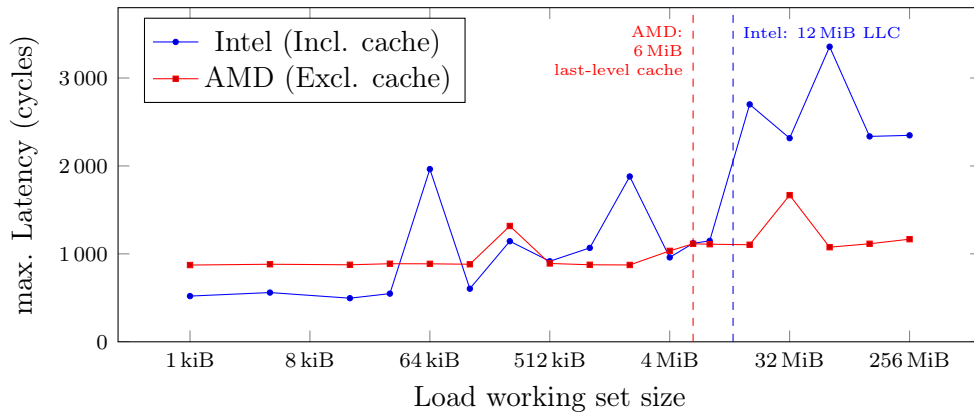


Figure 7.3.: Comparison of inclusive and exclusive caches. Impact of the load working set size to a small isolated task (max. latency).

With a longer execution time, it becomes more probable, that a task is hit by multiple delaying events slowing it further down in the worst case. The eviction effect of inclusive caches can obviously be avoided by using other processors with exclusive caching. In the following, the use of separate inclusive Last-Level Cache is analyzed. Today, systems with multiple sockets enable the construction of systems with more than 16 processors out of COTS components. A hard real-time task executes with the lowest latency and jitter if it uses as few memory accesses as possible, hence if it may keep most of its working data-set in its private caches. Since sharing the Last-Level Cache with other processes will increase its jitter especially when using inclusive caches, the other processors should be employed carefully. The requirement of having compute-intensive tasks leads to the use of multi-socket systems with multiple Last-Level Cache that are investigated in the following Section. Further, the required memory and I/O device accesses may collide on the shared system bus. The separation of data streams will be analyzed in Section 7.4 and Section 7.6.

Controlling the execution on the other cores of a multi-core processor violates the requirement of executing arbitrary load in the system partition. But multi-core processors are not the only option to use multiple CPUs. In the following Sections, the combination of single-core and multi-core chips to multi-socket systems with UMA and NUMA characteristic are analyzed.

7.3. Uniform Memory Access Systems

The classification of UMA systems also includes multi-core packages, but this Section concentrates on multi-socket systems. The differentiating detail is the existence of multiple shared caches, which results in not all processors sharing their Last-Level Cache. All sockets are connected to the same system bus and share the Memory

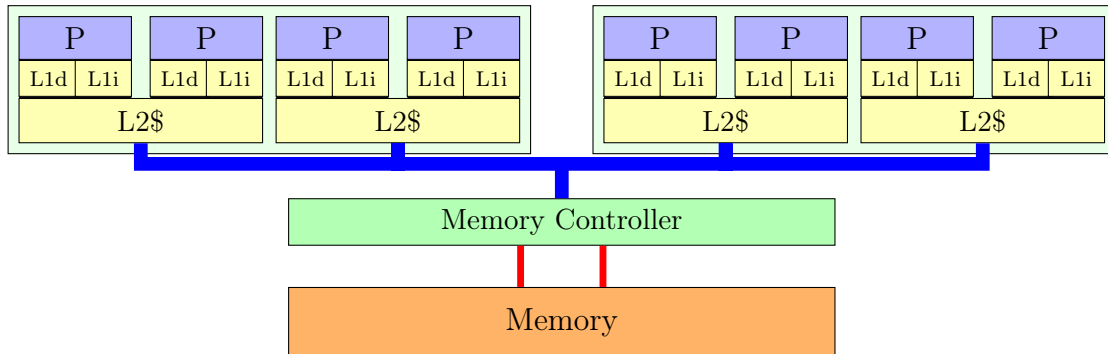


Figure 7.4.: UMA system with two sockets and four cores each (Test system *octopus*, Appendix A.1.2).

Controller (MC) located in the chip set (Fig. 7.4). Therefore, in average, each processor has the same temporal distance when accessing the memory on a cache miss. In HPC, this can become a congestion point if too many processors require large data-sets. Worse, in real-time systems, the system bus and the MC are shared functional units that can be blocked by other processors for unpredictable periods. Dasari et al. analyze very thoroughly the causes of varying execution times (jitter) in COTS-based multi-core (UMA) systems including a long list of related work [Das+11; Das+13].

To investigate the effects of different memory ranges accessed from different settings, a small benchmark mimics two isolated tasks by placing them on dedicated CPUs and using the Interrupt Flag (IF) to avoid all other influences on the run-time. A *load* task reads and writes an array of configurable size while the *isolated* benchmark task takes the timing statistic for accessing a single element of its own working set. The minimum, average, and maximum latency is recorded. The test system *octopus* (Appendix A.1.2) uses an UMA architecture with two double-core chips per socket and two sockets, hence 8 CPUs (Fig. 7.4). The settings are both tasks on adjacent CPUs sharing the Last-Level Cache, both processes on the same socket, but using their own Last-Level Cache, and both tasks on different sockets only sharing the system bus and MC.

The load was selected up to a working set size large enough to overwrite the Last-Level Cache. But the load grows not further above that to reduce the probability of triggering foreign cache eviction as demonstrated in the previous Section. The various load sizes are displayed using colors grading from blue —●— for small to red —●— for large load working sets. Here, the isolation with a small working set is not influenced in the Figures 7.5a and 7.5b. Only in Fig. 7.5c, loads with memory sizes larger than the shared Last-Level Cache cause a large jitter in isolations using only their private private caches. If two tasks are executed on different sockets using only their private caches, they should not influence each other, but apparently the cache

7. Architectural Influence

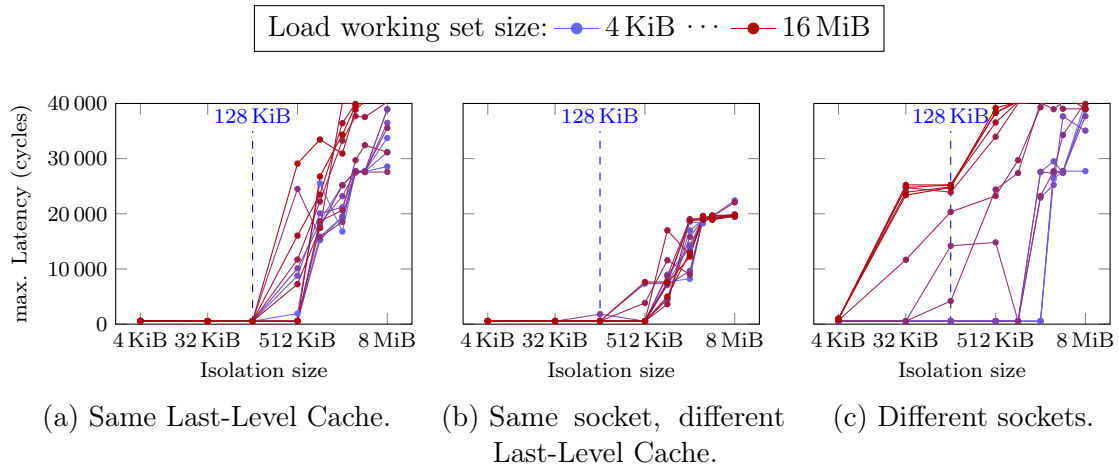


Figure 7.5.: Maximum latency depending on isolation buffer size with varying load working set sizes (4 KiB to 16 MiB).

synchronization (i. e. the coherence protocol) does so. This could be due to the MESI protocol (Section 2.1.9) that relies on bus snooping. In the case of multiple sockets holding multiple Last-Level Cache, broadcasts consume more time.

Concluding, this system architecture is not suited for hard real-time applications with latency requirements in the 10 μ s range. Generally, each system should be analyzed with similar methods before being considered for the use in a real-time system. UMA systems use other cache coherence algorithms than NUMA systems, therefore, in the next Section, systems with multiple sockets and memory connected to each socket individually will be investigated.

7.4. Non-Uniform Memory Access

Non-Uniform Memory-Access (NUMA) systems are multi-processor systems constructed of multiple nodes each with local memory and one or multiple processors. Every CPU can access the entire memory range, but the access to the local memory is more efficient (lower latency and higher throughput) than the access to remote memory connected to other nodes. In HPC systems, this enhancement of UMA systems allows to avoid the bottle neck of memory access [Bla+10] while keeping the same programming paradigm of shared memory systems (Single Program Multiple Data, SPMD). Remote accesses are less efficient, but still fully functional for Inter-Process Communication (IPC). For optimization reasons, it is advisable to partition the data in a way that code running on a node uses mostly data located on the same node's local memory [Boy+10; MG11].

Some previously cited publications mention NUMA systems in their plans for future work [Das+13, Sect. IV.C] or just explain theoretical foundations [Sto06] but they do not analyze the opportunities and weaknesses of such systems. To the best of my knowledge, no publications with experience about hard real-time applications on actual NUMA systems do exist so far.

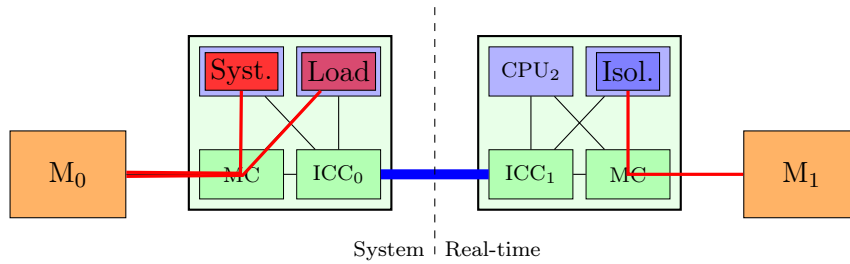


Figure 7.6.: Separation of system and real-time partitions on dedicated NUMA nodes.

In the experiments presented above, the shared functional units of the system bus and the MC were identified as contention points where exceptionally large latencies are caused occasionally by so-called *delaying events* that may spoil hard real-time tasks. With the experience from HPC optimization, it can be expected that NUMA architectures with separated memory nodes allow the creation of physical partitions for system and real-time tasks so that data paths (Fig. 7.6) are nearly completely separated. All compute-intensive processes with full system utilization are placed on their own node. The OS will by default allocate local memory from that node. The hard real-time tasks placed on the other node use their local memory so that the access paths are separated and the system interconnect (between IC_0 and IC_1) is rarely used.

The impact of separated Last-Level Caches (Fig. 7.5c) was blamed on the MESI protocol using broadcasts on the shared system bus to synchronize caches. Processors designed for NUMA systems use the modified variants MOESI (Section 2.1.9 [AMD12b, Chap. 7.3]) or MESIF ([Intel09, p. 15]) to improve the critical memory-access performance [Tho11]. The documentation does not reveal every last detail, so a measurement-based approach remains to understand the behavior [Fog13d]. It is expected, that such optimized cache coherence protocols interfere less with real-time applications.

The assumption that a NUMA system is better suited for a partitioning approach was analyzed on the test system *haixagon* (Appendix A.3.1). It has two NUMA nodes each with six physical processor cores and 24 GiB main memory. The isolation was set up with the Linux kernel patch as described in Chapters 4 and 5. The partitioning creates a *system* partition on CPU #0 for the internal maintenance, services and interrupts, a *user* partition on the remaining CPUs #1 to #5 on the same node for soft real-time tasks and six partitions for isolated hard real-time on the

7. Architectural Influence

CPUs #6 to #11 on the other node. Like in the previous test series, the benchmark uses two processes, a *load* and an *isolation*. The load process is executed in the user partition. It reads or writes the elements of a memory buffer of varying size located in its local memory. The isolated task in the other partition does so with a buffer on its local memory node. The timing includes one memory access with a stride of one cache line which equals a new cache line access for every measurement. All other CPUs, especially the ones in the real-time partition on the memory node where the isolated task is executed, are idle. Both tasks iterate over a wide range of buffer sizes to analyze the impact of the load onto the isolation.

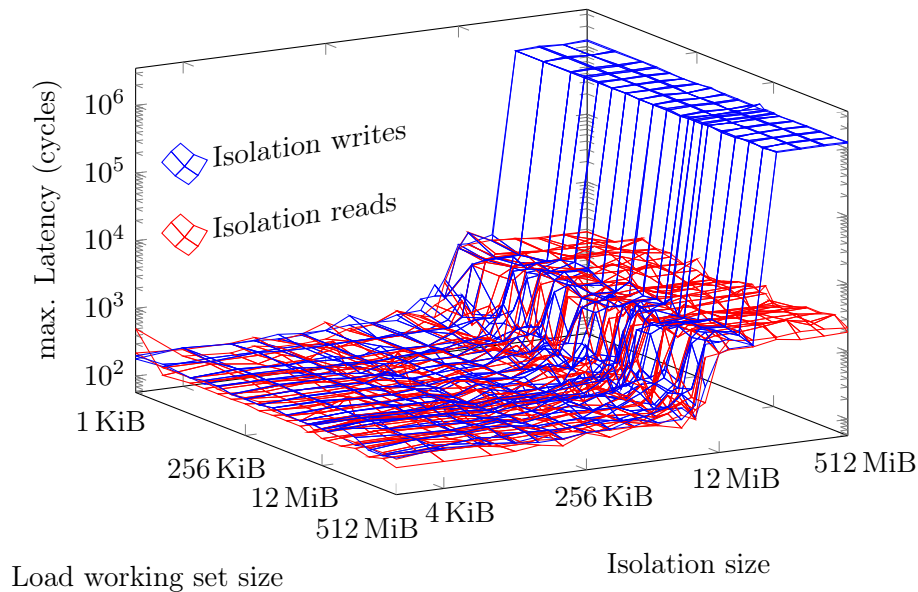




Figure 7.7.: Maximum latency of an isolated task using various memory sizes under varying load scenarios.

The full results are displayed in Fig. 7.7. The red meshes  are generated by the data sets where the isolation only reads and the load either only reads or reads and writes. Since they are not significantly different, they are both drawn in red. Similar for the blue meshes  where the isolation reads and writes. The most important observation is the independence of the results from the load working set size for the complete range from 1 KiB up to 512 MiB.

The minimum and average execution time are independent from the isolation buffer range as illustrated for three different load working set sizes in Fig. 7.8. Only the maximum execution time (WCET) increases if the memory range of the isolated task uses a large part of the Last-Level Cache. When writing large buffers, an isolation is subject to occasional delays of 10^6 CPU cycles (Millisecond range). This is the same effect of cache misses observed in multi-core systems only that in this case, the

7.4. Non-Uniform Memory Access

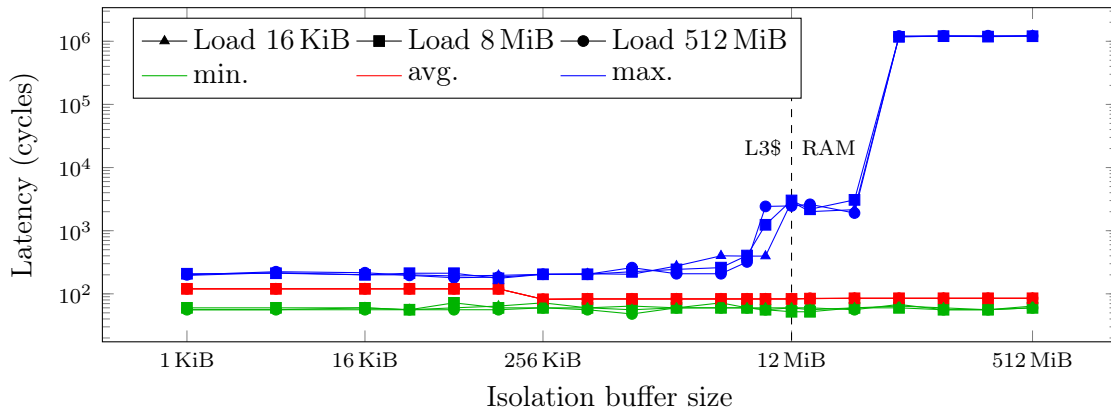


Figure 7.8.: Jitter experienced by the isolated task depending on its memory range for three different load sizes (that do not influence the result significantly).

isolated task itself causes the eviction of its own data if its working set reaches the cache size.

The distribution of the isolated task's execution times for various memory sizes is displayed in Fig. 7.9. The load working set does not influence the result as showed before, but the largest size of 512 MiB read/write access was still selected for these figures. If the isolated task uses less memory than fitting in its private caches (the Westmere CPU has 256 KiB L2\$), the execution times are very narrow. An isolated task using 8 MiB (the size of the L3\$ is 12 MiB) still executes 10^{10} iterations at the same latency as a smaller task, but 10^6 iterations (which is only a fraction below $1/1000^{\text{th}}$ of all iterations) experience a delay of up to $1 \mu\text{s}$. In the last figure representing an isolation working set size of 512 MiB, the histogram is similar to the previous case, but it includes additionally 224 events above $1.2 \mu\text{s}$ up to a maximum of approx. $500 \mu\text{s}$.

Since real applications require some shared variables for the synchronization of the application and Inter-Process Communication (Section 6.1), the access to cache lines not currently stored in the cache is inevitable. Those memory regions are placed on one of the nodes and the processes located on the other node must access it remotely. To estimate the risk of large delays, Fig. 7.10 displays the number of extreme delays above $2 \mu\text{s}$ depending on the isolation size. It can be seen, that the likelihood of extreme delays grows with the size of the buffer. Therefore, it is advisable to keep the working set of isolated tasks in the private caches and keep the shared buffers for IPC as small as possible. Consequently, the shared buffers for IPC between partitions should be placed in the real-time partition. This allows the time-sensitive tasks to access them locally while the less critical tasks must access remotely. The results of these short benchmark series are supported by tests with an application up to 72 Hours that show the same characteristic.

7. Architectural Influence

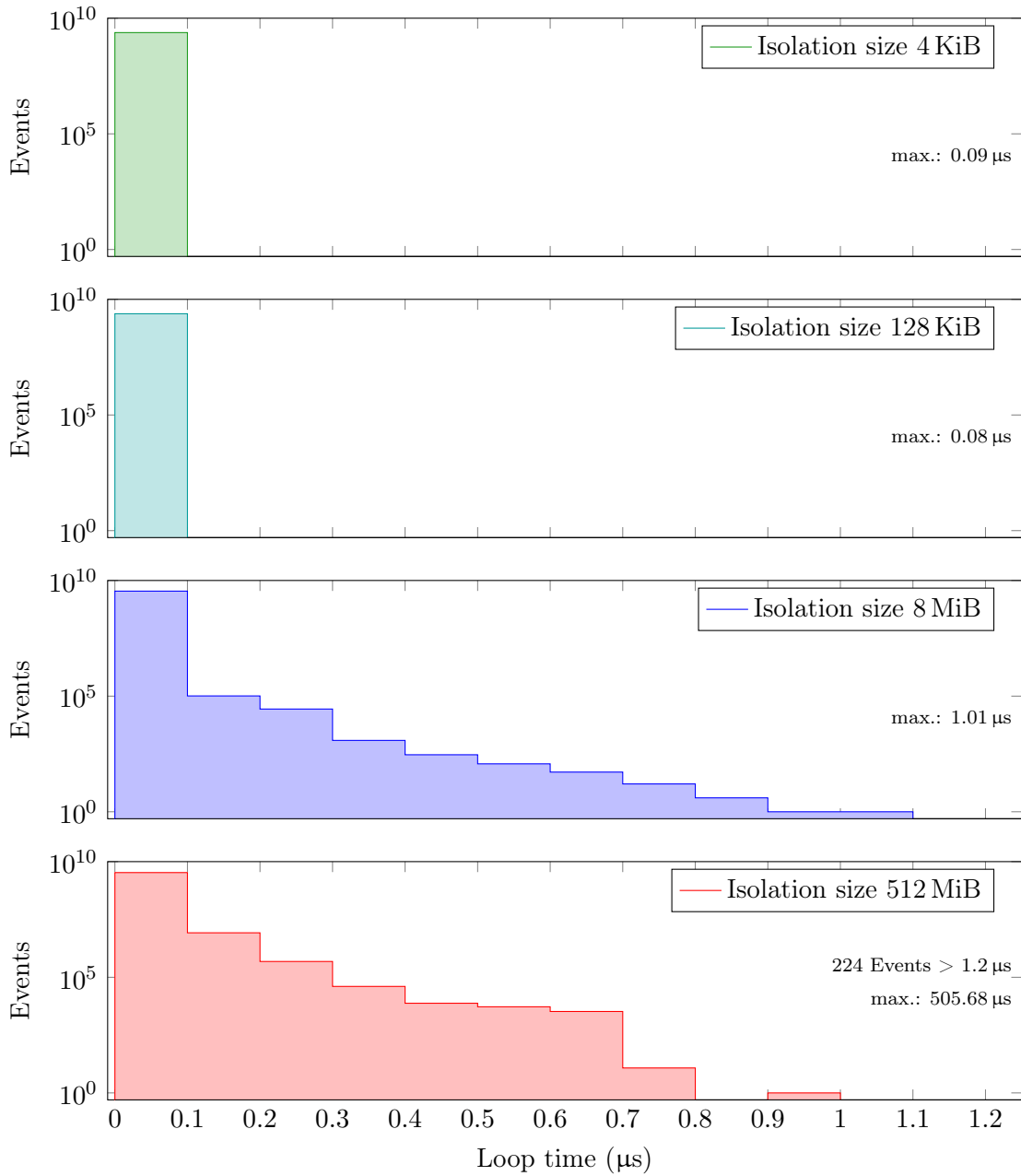


Figure 7.9.: Histograms with largest load (512 MiB read/write buffer size).

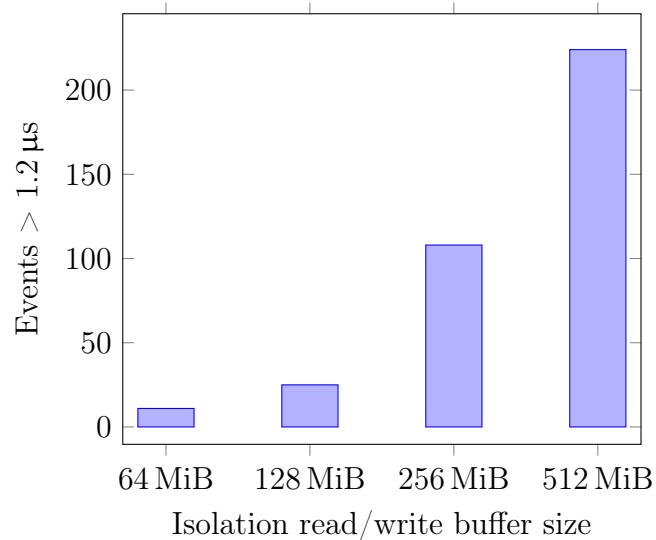


Figure 7.10.: Number of extreme delays above $1.2 \mu\text{s}$ depending on isolation size.

Not only the memory access is separated per socket, also the I/O of the system partition does no longer interfere with memory transfers of the real-time tasks because the MC is in the processor package and does not use the same system interconnect as the device access. This will be further evaluated in Section 7.6.

7.5. Symmetric Multi-Threading

The Pentium IV generation introduced Simultaneous Multi-Threading (SMT) (called “Hyper-Threading”) to the x86 architecture by dividing a physical processor into two logical execution units (Section 2.1.9). They execute independently and are presented to the OS like real processors but share some parts. In fact, their instructions are executed interlaced on the same functional units to cover waiting times due to memory loads. Earlier in Section 4.2.1 was argued to deactivate SMT altogether because the sharing of functional units such as the decoding unit and the caches may introduce unpredictable jitter. With the consideration of NUMA systems and their partitioning along NUMA node borders, the partial use of SMT becomes desirable. The system partition on dedicated NUMA nodes could exploit the additional compute power and usage efficiency while CPUs with isolated tasks would not be influenced by logical processors using the same functional units.

To avoid a negative impact of logical threads onto isolated tasks on the same physical processor, the configuration of an N processor system to boot $1.5 \times N$ logical processors was analyzed. As illustrated in Fig. 7.11 for the test system *haixagon* (Appendix A.3.1), the numbering of the logical threads places the Thread IDs i and $i + N$ on the same physical processor. If the system with $N = 2 \times 6 = 12$

7. Architectural Influence

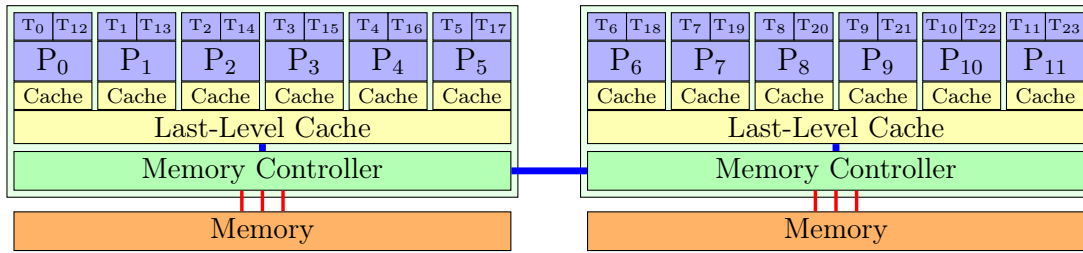


Figure 7.11.: Symmetric Multi-Threading example on two-socket Intel Westmere (test system `haixagon`, Appendix A.3.1) with physical processor numbers (P_i) and logical thread numbers (T_j).

physical processors is started with 18 processors, the first socket carries the activated Threads 0–5 and 12–17 with activated SMT while the second socket executes only Threads 6–11, i. e. one Thread per physical processor.

Although the activation of hardware threads on system nodes should not influence the execution on the real-time node, a different experience was made. If at least one physical processor executes two logical CPUs, the Linux kernel starts kernel threads that are not active without SMT. The existence of these kernel threads causes an instability with the isolated task concept. The effect is isolated tasks not terminating gracefully after they ran for more than 5 Minutes. This is probably due to the Kernel modifications described in Chapter 5. This problem could not be solved so far and the deactivation of SMT is still be advised unless further experiments prove successful.

7.6. Input and Output

Many real-time systems (e. g. for Programmable Logic Controller, PLC) require direct Input/Output (I/O) of digital (General-Purpose I/O, GPIO) or analogue signals (Analog to Digital Converter, ADC or Digital to Analog Converter, DAC) and field buses (e. g. RS-232, I²C, CAN). While most Micro-Controllers (μ Cs) integrate those functional units, COTS systems must be extended with peripheral devices connected for example via Peripheral Component Interconnect (PCI) or PCI Express (PCIe). This is very flexible but the access to those devices uses the system buses and is subject to greater jitter than on μ Cs.

A wide range of publications analyzes the effects of bus load on real-time applications on single-processor [Sch03; Lew+07] and multi-processor systems [SBF05]. On multi-processor systems, other processor's data streams may interfere with the I/O access of hard real-time tasks increasing the unpredictability of their timing [Sch+10]. Stohr et al. propose to confine the I/O of the system partition to a level not having too much impact on the real-time processors [SBF04]. To solve the problem of unpredictable jitter, co-scheduling of tasks [Pel+08] and hardware extensions

[Bak+09; Bet+13] are proposed. But since the concept of isolated tasks should allow using arbitrary load on the remaining processors with only minor changes to the OS, this is not a viable solution. All evaluations of multi-processor I/O so far only regard UMA systems and the older PCI bus [Nam+09]. Stohr includes PCIe in his analysis of using the x86 architecture for real-time systems [Sto06] but does not cover NUMA and non-uniform I/O. The details of PCIe are covered in the standard [PCIe10] and the book of Budruk et al. [BAS03]. A quick overview is given by Bhatt that mentions the support of prioritized data streams and that switched point-to-point connections can be transferred concurrently to other transfers without interfering [Bha02].

In High Performance Computing, NUMA systems are regarded as the solution to overcome limitations in the memory throughput and I/O congestion [BF11]. As shown above, NUMA systems have also advantages in real-time systems by allowing to separate the data transfers between processors and memory because they replace the system bus as unique point of congestion with multiple point-to-point links (e. g. QuickPath Interconnect [Intel09]). In the same way, the shared PCI bus is replaced by its successor PCIe using point-to-point links. In the following is shown how this architecture also improves the predictability of real-time I/O. Today, the trend is to integrate the MCs in the processor chips instead in the classical north bridge. This removes local memory transfers from the system bus leaving more capacity for (and less influence onto) I/O. But the transfers of other processors must be considered. In the most recent generation, also some I/O lanes of PCIe are directly connected to each processor and no longer need a chip set component (north bridge).

7.6.1. Basic Access

Reading or writing to I/O ports on recent Intel micro-architectures takes 0.5 to 1.5 μs in average (Table 7.3). These measurements were executed on available test systems during normal operation. The minimum time results from the architecture-dependent latency, the maximum is influenced by scheduling and other operation. Therefore, the average may be biased, but it allows a first estimation of the jitter. The important result is that the access on a given system depends on the location of the device on the internal buses and if it is placed directly in the chip set or in a peripheral device on the PCI or PCIe connectors. Further, the access latency may depend on the device, as is visible in the `haixagon` (Appendix A.3.1) example of a Parallel port and a Serial port both on the same adapter connected to the PCIe bridge that show different latencies. Generally, the I/O access is very slow compared to other execution in the processors. On recent super-scalar CPUs, multiple instructions can be executed every clock cycle while an `IN` or `OUT` instruction takes between 1500 and more than 4000 cycles. Some devices map their state registers into the memory address space (memory-mapped I/O) and applications can load and store to these locations similar to normal memory. This form of I/O shows a similar access latency like I/O ports.

7. Architectural Influence

System	Device	Connection	Latency (μ s)		Notes
			min.	avg.	
poodoo	VGA	PCIe	0.55	0.96	Core 2 Dual
	IDE	PCIe	1.28	1.29	
	USB	south bridge	0.79	0.97	
octopus	SCSI	PCIe	0.84	0.85	Core 2 Quad UMA 2×4
	USB	south bridge	0.80	0.81	
xaxis	VGA	PCIe	0.82	0.82	Nehalem 1×4
	Parport	PCIe	6.32	10.49	
	USB	south bridge	0.89	0.91	
haixagon	Parport	PCIe direct	1.27	1.33	Westmere NUMA 2×6
	Parport	PCIe cross	1.39	1.46	
	Parport	PCI (PCIe bridge)	1.56	1.61	
	Serial	PCI (PCIe bridge)	1.36	1.42	
	Parport	PCI (south bridge)	1.40	1.47	
	USB-Port	south bridge	1.01	1.03	
pandora	SCSI	PCIe	1.26	1.55	Sandy Bridge NUMA 2×8
	SATA	south bridge	0.99	1.01	

Table 7.3.: I/O latencies of different PCI and PCIe devices (Systems described in Appendix A).

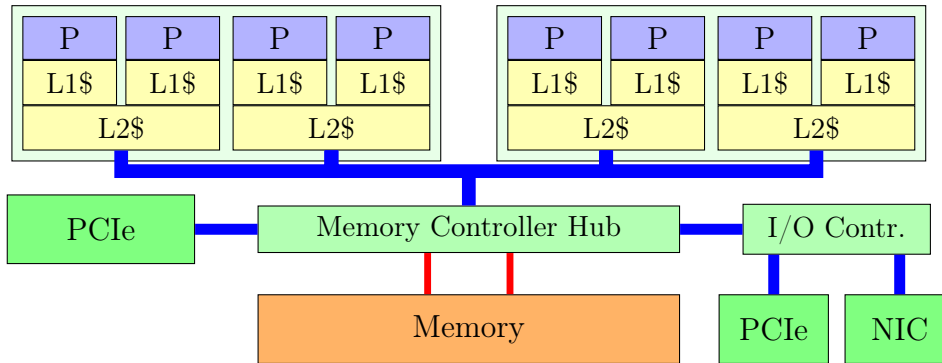


Figure 7.12.: Peripheral I/O capabilities of an UMA system with two sockets (Test system *octopus*, Server board Intel S5000PSL, Appendix A.1.2).

Consequently, the x86 architecture is not capable of executing tasks that require I/O access below some Microseconds (i. e. above approx. 500 kHz). If a feedback control task needs to read multiple sensor values, calculate an answer and then set multiple actors outputs, the latencies of these I/O accesses must be taken into account. A potential optimization is the combination of multiple values into registers which can be done on programmable FPGA-I/O adapters.

7.6.2. Non-uniform I/O Access

The typical peripheral architecture of an UMA system is displayed in Fig. 7.12. All transfers between the CPUs and the chip set may collide on the system bus between the CPUs and the Memory Controller Hub (i. e. the north bridge). Some PCIe ports are directly provided by the Memory Controller Hub and additional PCIe as well as PCI ports are made available by the Interconnect Controller Hub (ICH) (i. e. south bridge). Concurrent memory and I/O accesses from different processors may contend for the chip set inducing unpredictable latencies on hard real-time tasks.

Since NUMA systems behave more predictable than UMA architectures with relation to the memory and cache effects (Section 7.4), the I/O transfers are also assumed to be less affected by other data and I/O transfers. Similar to the separation of memory transfers into a system and a real-time partition (Fig. 7.6 on page 165), NUMA systems allow to separate also the I/O transfers (Fig. 7.13).

In the following experiments, the OS and the compute-intensive tasks are confined to the CPU socket and I/O devices displayed on the left side of Fig. 7.13. On the 12 processor test system *haixagon* (Appendix A.3.1), this are the CPUs #0 to #5. The right side is the dedicated real-time partition with the CPUs #6 to #11 and the PCIe slots connected to the I/O Controller Hub (IOH) directly connected to these CPUs (referenced as “PCIe direct”). The PCIe slots in the system partition are referenced as “PCIe cross” because they are accessed from the real-time CPUs over

7. Architectural Influence

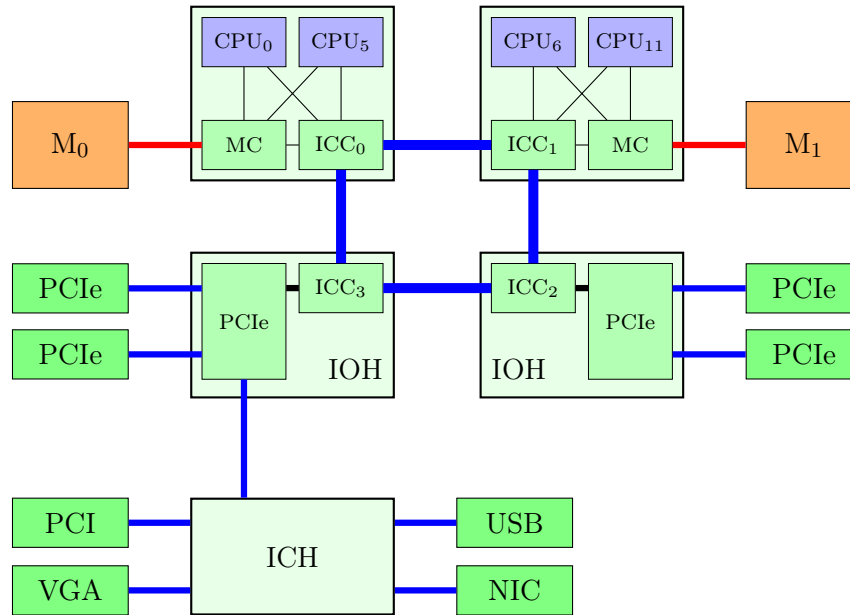


Figure 7.13.: Peripheral I/O capabilities of a NUMA system with two sockets (Simplified test system `haixagon`, Server board Tyan S7025, Appendix A.3.1).

an additional QuickPath connection and may be interfered with system transfers between socket #0 and the ICH. The I/O devices in the ICH (so-called *legacy I/O*) and adapter cards installed in the PCI slot are also assigned to the system partition.

Several Parallel port expansion cards are used as exemplary peripheral devices. Two identical PCIe adapters are placed in the “direct” and “cross” locations. Two PCI adapters are installed for comparison, one in the PCI slot connected via the south bridge and the other in a PCIe-PCI-bridge connected to a “PCIe direct” slot. It is expected, that the latency of devices on the “PCI direct” connection will not be influenced by load in the system partition. The “PCIe cross” devices may be influenced and all devices connected to the south bridge are assumed to be strongly hit by transfer contention.

All following test series were executed on the test system `haixagon` with the isolation patch described in Chapter 5, the isolation methods presented in Chapter 4 and the application architecture proposed in Chapter 6. The Linux system was confined to CPU #0 and the loads were executed on the CPUs #1 to #5. Five concurrent isolated tasks were executed. The first is used for the verification of the isolation quality, it does not execute anything other than the *hourglass* statistics routine (Section 2.3.5 on page 67) and its execution time was in all tests between 0.01 μ s and 0.4 μ s with a constant average of 0.03 μ s. The other four isolated tasks read an I/O port of their assigned Parallel port adapter with the maximum possible frequency.

Load	Min.	1 st Quart.	Average	Median	3 rd Quart.	Max.
idle	0.96	0.99	1.01	1.01	1.03	5.85
sigwait	0.97	0.99	1.01	1.01	1.03	5.53
mmap(4)	0.97	0.99	1.05	1.01	1.03	37.58
mmap(16)	0.97	0.99	1.03	1.01	1.03	5.55
mmap(512)	0.97	0.99	1.33	1.03	1.93	6.32
mmap(1024)	0.97	1.01	1.46	1.05	1.93	6.25
mmap(2048)	0.97	1.03	1.61	1.93	1.93	6.29
mmap(4096)	0.97	1.91	1.73	1.93	1.93	6.35
mmap(16384)	0.97	1.91	1.86	1.93	1.93	6.36

Table 7.4.: Statistic of latencies (in μs) accessing a PCI-device connected via the south bridge under various loads.

Every test was executed for one Minute and generated between 3×10^7 and 2×10^9 samples collected in a histogram. The statistic of median and percentiles was derived from the histogram with an accuracy of $0.02 \mu\text{s}$. The *idle* system scenario does not execute any additional processes. The *system load* was varied with different generators beginning with a memory transferring routine, a forking process and a timer signal benchmark loading the OS, and a file writing routine to generate I/O load. Since those load generators resulted in hardly any difference, the *system load* scenario is the concurrent execution of those loads. Finally, a memory-mapped access to the network card's buffers with varying block size influenced the access to the PCI adapter connected to the south bridge. The *PCI transfer* scenario was executed with a wide range of `mempy()` read transfers that were separated by a sleep time of $1 \mu\text{s}$.

Figure 7.14 shows the latency of the PCI device connected to the south bridge under various load scenarios. By far the most accesses are in the range 0.9 to $1.1 \mu\text{s}$. The maximum as large as approx. $5.5 \mu\text{s}$ causes a very large jitter. Since the delayed events are three to four orders of magnitude more scarce, the average is near to the minimum. The PCI memory mapping data transfer of 4 Byte block size generates a slight increase of the events around 2.5 to $3.5 \mu\text{s}$ and some delays up to $36 \mu\text{s}$. With a PCI transfer block size of 4 KiB, the outliers do not occur, but the median latency increases to $1.9 \mu\text{s}$. The detailed statistic is listed in Table 7.4 (generated from a different benchmark execution resulting in a slight variation of the experienced maximum latencies).

To compare the different connections of peripheral devices, the histograms of all four tested devices under maximum load (*system load* and *PCI transfers* with 16 KiB block size) is compared in Fig. 7.15. The PCI device connected to the south bridge experiences a very large jitter with occasional outliers with more than $30 \mu\text{s}$. The histograms of the lower three figures for devices connected to the PCIe slots are the same for all different load scenarios. This proves that neither the PCIe devices in the

7. Architectural Influence

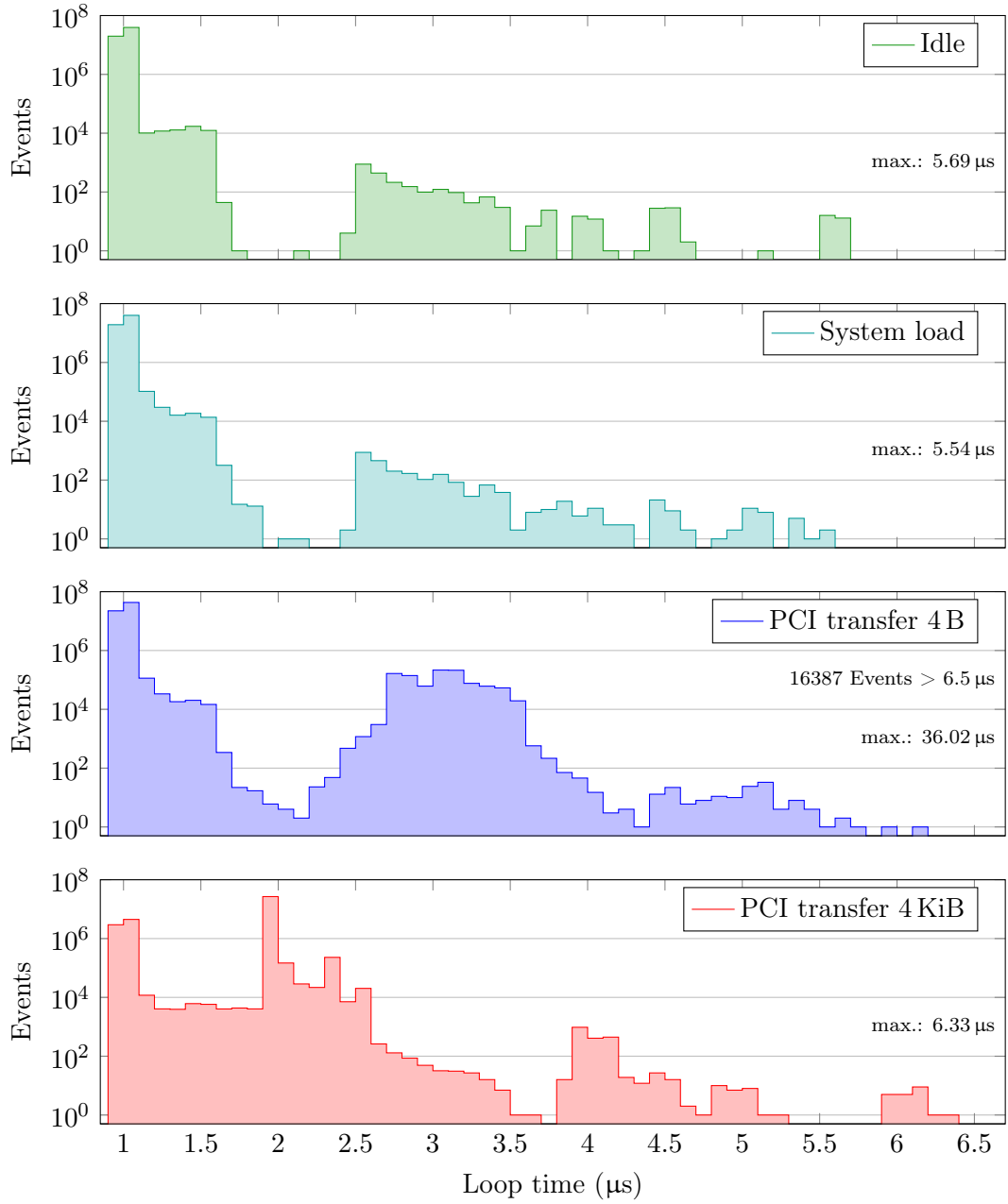


Figure 7.14.: Distribution of latencies of a PCI device connected via the south bridge under various loads.

7.7. Interpretation and Recommendations

real-time partition (right side in Fig. 7.13, as assumed) nor the devices connected to the ICH that forwards the PCI transfers from socket #0 to the south bridge (left side, rather not recommended) are influenced by arbitrary load on the system CPUs. If the device is connected to the PCIe slot of the real-time partition, the jitter of I/O port accesses on this test system is $0.8 \mu\text{s}$.

Real applications will most likely require reading and writing multiple I/O ports. If the worst-case latencies are added, the possible jitter increases with every I/O access. In practice, the experience of extended test series is much better than that. Figure 7.16 displays a histogram of an application benchmark that was modeled after a proprietary hardware-in-the-loop simulator. The test routine executes 6 read and 5 write I/O accesses as well as exchanging 2000 Bytes with a manager process. The run-time of the test series was 5 days, the test was repeated several times and yielded similar results. The jitter of this test routine is $1.6 \mu\text{s}$ which is far better than the expected $11 \times 0.8 = 8.8 \mu\text{s}$ obtained by adding the worst-case latencies. This observation can be explained with the causes for the delays being spread or at least never aggregating and hitting subsequent accesses. Additionally, the very long execution time supports the measurement-based approach for a tight WCET estimation.

7.7. Interpretation and Recommendations

While the control flow of a real-time application can be formally verified, the execution time of basic blocks on current x86 multi-processor systems depends on many factors. Some can be controlled (such as the influence of processes on other CPUs and I/O data transmissions), others are caused or at least influenced by hardware features that are hardly documented. Generally, the resulting execution time can only be measured under (heavier than) practical load. Since NUMA systems perform real-time capable both in memory and cache jitter as well as in I/O predictability, this architecture – initially developed to enhance HPC applications – is well suited for the isolated task concept. A wide range of matching systems is available and can be selected according to the needs (number of processors, proportion of system to real-time partition, number and type of peripheral connectors).

In this Chapter, only a single exemplary NUMA system based on the Intel Westmere micro-architecture with a Tyan S7025 main board was used for the practical measurements. The newer Intel micro-architectures Sandy Bridge and Haswell might have introduced different behavior and must be evaluated in a similar way before using them for hard real-time systems. But generally, the NUMA architecture with PCIe devices is a promising solution for jitter problems on x86 systems. However, these are hardly included in real-time research, so far.

The concluding recommendations are:

7. Architectural Influence

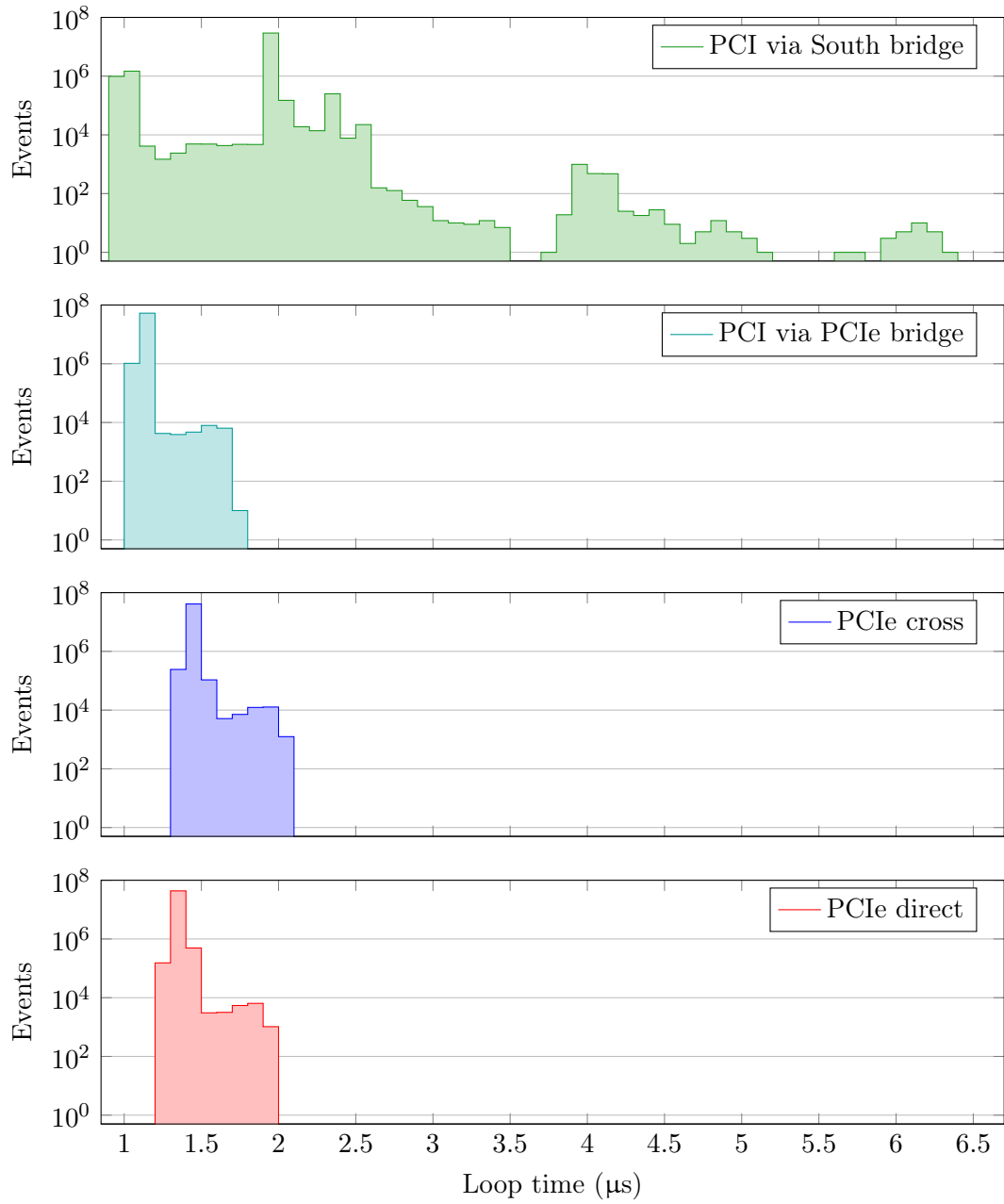


Figure 7.15.: Distribution of latencies of different PCI and PCIe devices under maximum load.

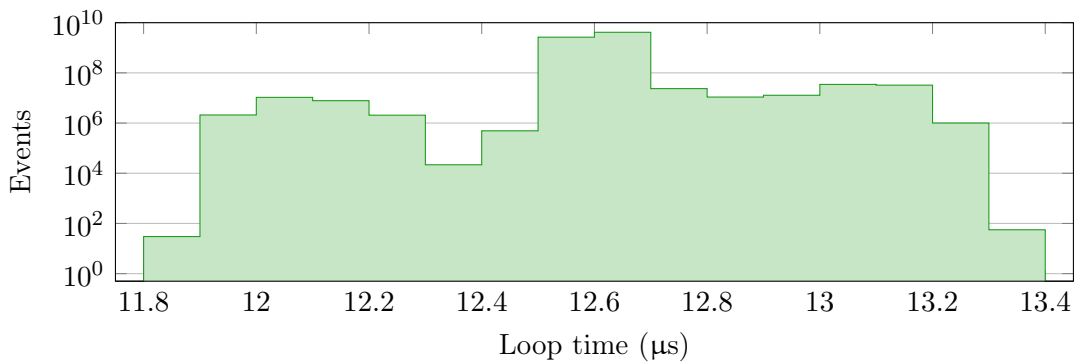


Figure 7.16.: Distribution of execution times of an application benchmark task with 11 I/O accesses and 2000 Byte data exchange after an execution of 5 days (120 h).

- Group isolated processes on dedicated chips/sockets to avoid sharing their Last-Level Cache with non-isolated processes.
- Isolated tasks should limit their working set as far as possible to remain in the private cache.
- Avoid false sharing.
- Use as few shared variables as possible because they introduce jitter.
- Separate I/O transfers in a similar way as the memory placement.
- Select a main board with PCIe lanes connected to the dedicated hard real-time socket to avoid conflicting data transfers with the rest of the system.

It is emphasized again that this should not soften the requirement for a formal verification of hard real-time tasks. The code must be free of unbounded loops, uncontrolled synchronization and other sources of unpredictability such as system calls to a GPOS. Only the estimation of the WCET of basic blocks can either be too pessimistic and hence, not accepting a solution, or too tight and missing latencies ruining deadlines in the real application. This hardware experience also applies to other real-time approaches like RTOS or sub-kernels (e. g. Xenomai, Jailhouse).

8. Application Examples

Several example and demonstration applications were developed to examine the concept of isolated tasks. Bruns evaluated an early implementation to generate an I²S signal in software that was transmitted to an electronic circuit that generated an audio output [Bru10]. The average output latency was 470 μ s, the audio output frequency was selected to 22 kHz 16-Bit stereo, hence one frame of 32 Bits (up to 64 serial I/O operations) had to be below 45 μ s. That implementation on an Intel Core2 Dual-Core processor suffered to periodic System Management Interrupts (SMIs) issued by the system's firmware [Wal10a] but was a successful demonstration apart from that. Alt implemented a demonstration application to fully exploit the concept with multiple cooperating tasks on a system with twelve processors [Alt13]. His system transmits data by low-level I/O port access and generates an image on an oscilloscope via Digital to Analog Converters (DACs). In a confidential cooperation, the concept was implemented in a commercial product for an hardware-in-the-loop simulator. That application realizes multiple concurrent isolated tasks with a hard real-time constraint below 20 μ s running multi-day tests. Multiple systems are in productive use and conform fully to the expectations.

The application benchmark used for the test series in the previous Chapters was developed as template for a real application but replaces real tasks with synthetic payloads. It features the start of several processes with various configurable loads that support a continuous feedback on their performance. A manager task controls a variable number of isolated tasks each capable of executing different payloads and statistically recording their execution times. The benchmark implements most features described in Chapter 6.

Several different approaches to technical challenges were implemented and compared. These experiences build the base for the following suggestion for an application architecture for the chemical experiment control system introduced in Section 1.1 and further outlined in Chapter 6. This description is used to connect all techniques presented in the previous Chapters. Concluding, it is shown how some application examples from the literature can be ported to the isolated task concept.

8.1. Hardware Selection

A capable base system that was also used for the non-uniform memory and I/O access benchmarks in the previous Chapter (Test system *haixagon*, Appendix A.3.1)

8. Application Examples

is a system with two sockets, each with a six core Intel Xeon E5645 processor with 2.4 GHz on a TYAN S7025 main board. The system has 48 GiB of main memory distributed over two memory nodes corresponding to the processor sockets. The PCI Express (PCIe) Input/Output (I/O) is connected via two I/O Controller Hubs (IOHs) forming a QuickPath ring with the Central Processing Unit (CPU) sockets. Other peripheral devices are provided by an Interconnect Controller Hub (ICH) that is connected to the left IOH (Fig. 7.13) making it possible to divide the resources into a system and a real-time partition. Each CPU socket has local access to half of the installed memory and must access the remaining memory remotely. This platform does not execute time-triggered SMIs which is a prerequisite for its eligibility as a hard real-time system. Basic tests have revealed the need for the following BIOS settings to minimize I/O latencies and to reduce jitter:

Hyperthreading must be *deactivated* to avoid interference from another hardware thread (logical processor) using the same CPU and private cache.

Intel Virtualization Tech is *deactivated* because it introduces an additional layer of indirection tables increasing the latency.

I/O Virtualization is *disabled* for the same reason.

Legacy USB Support should be *deactivated* because it uses SMIs to simulate a PS-2 keyboard from one connected to Universal Serial Bus (USB). This setting disables the use of a USB keyboard in the boot menu and should be used only after the system is fully installed.

Power Management is *deactivated* to avoid automatic frequency switching or power-down triggered by the firmware.

This main board offers six PCIe slots, three connected to each IOH (north bridge). The single Peripheral Component Interconnect (PCI) connector is served by the ICH (south bridge) and is subject to large jitter (Section 7.6.2). This connector should not be used for real-time I/O but can be used by applications in the system partition. Digital I/O (GPIO) and analogue converters (DAC and ADC) can be installed as PCIe expansion cards. If they are used by hard real-time tasks, they should be installed in the three connectors assigned to the real-time partition. Those used by soft real-time tasks or that are not time-critical can safely be installed in the other PCIe slots. If a Data Acquisition (DAQ) peripheral device is only available with PCI interface, a PCIe-PCI bridge can be installed in the real-time partition.

For the I/O, the suitability of available adapters must be evaluated depending on the application. The selection depends on the requirement for the number of General-Purpose I/O (GPIO), Analog to Digital Converter (ADC), and DAC connections. Further, the number and layout of PCIe and PCI slots must be balanced with the isolated tasks. Ideally, each peripheral device is controlled by a single isolated task. I/O adapters for real-time systems are often equipped with user-mode libraries that access the hardware without operating system (OS) interaction for the lowest latencies. It is crucial to either select devices, for which the vendor offers such libraries or to carefully verify, if the documentation allows to implement them.

8.2. Software Architecture

The software architecture for the control system of a chemical experiment is presented as guideline how such a system can be designed. This application was not implemented but it demonstrates how to apply the experience that was made during the implementation of the various concepts, benchmarks and demonstration systems. Some integration experience was made during the benchmark implementation and project consultation.

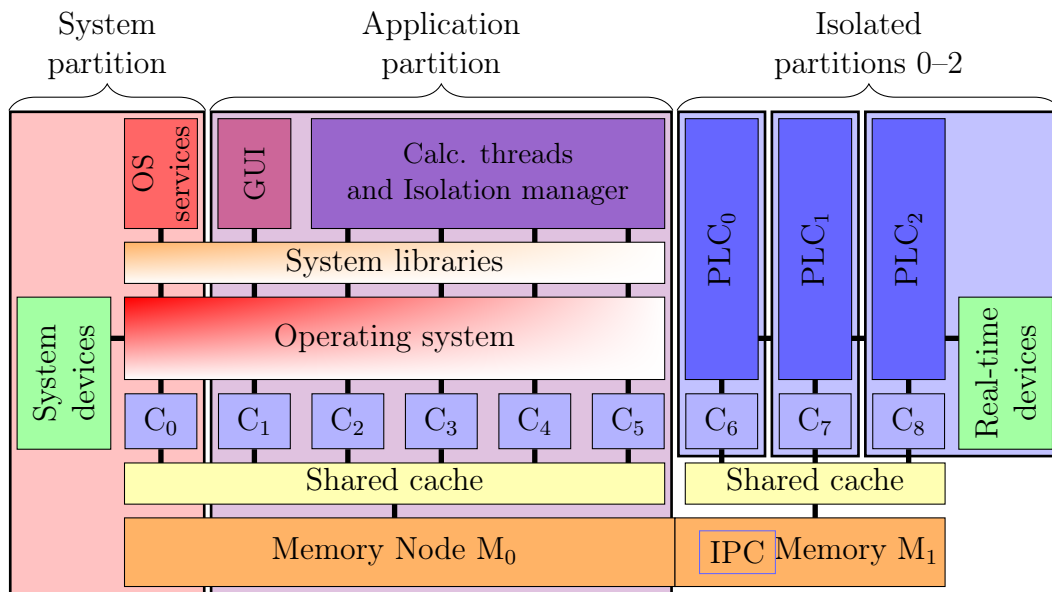


Figure 8.1.: Layer diagram of example application.

The proposed software architecture of an application using the isolated task concept is displayed in Fig. 8.1. The processors are divided into a *system* partition on processor C₀, an *application* partition spanning processors C₁ to C₅ and three distinct *isolated* partitions on the processors C₆ to C₈. With the experience from Chapter 7, the boundary between the *system* and *application* partition and the *isolated* partitions should be aligned to the Non-Uniform Memory-Access (NUMA) nodes. On the example hardware presented in the previous Section with two processor sockets, the system is divided in two halves. The processors C₉ to C₁₁ are not used and remain idle.

The *system* and *application* partitions execute the Linux OS with the Real-Time Preemption Patch (RT-Patch). All interrupts and system services are assigned to the *system* partition on processor C₀. The main part of the application is executed on the five CPUs of the *application* partition. For further optimization, the Graphical User Interface (GUI) can be assigned a dedicated processor while the calculation and manager threads are managed with preemptive priority scheduling by the OS on the

8. Application Examples

remaining CPUs of that partition. The application may use all services and libraries that are offered by the installed Linux distribution. If required, the application can directly use Kernel interfaces or use the abstraction offered by the C library. The devices required by soft real-time tasks or without timing confirmation are controlled by the OS's drivers and accessed by the application using the standard interfaces of the Linux kernel.

All Programmable Logic Controller (PLC) tasks having hard real-time and low latency requirements are executed fully isolated and bare-metal on up to six dedicated CPUs of the real-time partition. In the example, three such bare-metal tasks are started in three dedicated *isolated* partitions. These tasks are implemented as threads of the main application which eases the sharing of memory buffers. If the application requires a higher security level, the unintentional access to shared variables can be impeded by implementing the isolated tasks as separate processes that share only the required Inter-Process Communication (IPC) buffers using e.g. System V Shared Memory.

The *system* and *application* partitions use only local memory on their NUMA node. The isolated tasks restrict themselves as much as possible to their private caches. They also directly access their assigned peripheral devices for I/O. The single access out of these partitions is the IPC buffer located in the memory local to the *isolated* partitions. The isolated tasks using this buffer will cause cache misses that can be served locally. Only the manager task running in the *application* partition accesses the IPC buffer remotely which is slightly slower but can be tolerated since it is not under hard real-time conditions.

The communication functions between the main application and the isolated tasks should be implemented as statically linked library that can be used by all members. This reduces the risk of accidentally accessing unprotected shared variables, violating the communication protocol, or the chance of a deadlock. The user-mode I/O device access of the isolated tasks should also be implemented as simple linked library.

The twelve CPUs of the system are partitioned with CPU-Sets. The *system* set confines all kernel threads and system services to processor C_0 . The *application* partition spans the processors C_1 to C_5 intended for the user interface and the performance-critical threads. The manager task is also executed in that partition since it has less work during the time-critical application phase. The second socket with processors C_6 to C_{11} is reserved for completely isolated tasks executed in up to six dedicated CPU-Sets. The *system* and *application* CPU-Sets use memory node M_0 and the *isolated* partitions are assigned to memory node M_1 . The IPC buffers are allocated and initialized by the isolated tasks to place them in their memory node M_1 .

8.2.1. Graphical User Interface

The main part of the program is implemented as Qt application. During start-up, the command line arguments are checked and stored in an internal data structure. Additionally, default settings are read from a configuration file.

The main window displays the state of the application. It allows to configure and start an experiment. All tasks are executed in other threads or processes to keep the interface responding. The GUI and calculation tasks are executed together in the application partition spanning five CPUs. Their scheduling is managed by the OS and the compute-intensive threads may use priorities to employ soft real-time scheduling.

The calculation tasks are started dynamically depending on the experiment configuration. IPC mechanisms from Qt (Signals and Slots¹) or the C and UNIX libraries (e. g. Pipes [Ker10, Chap. 44] or Message Queues [Ker10, Chap. 46 and 52]) are used to interact with the child processes. By doing so, they can be given updated parameters, receive data to process and send back results. Depending on the requirements of the current experiment, new calculation tasks can be started or their scheduling settings can be adapted to the needs.

8.2.2. Manager and Isolated Tasks

The manager task is a thread started from the main *process* at initialization time. It is executed in the *application* partition and configured to real-time scheduling together with the calculation tasks, but with a lower priority to give them precedence. If the manager needs to transfer high-volume data from its clients, it could also be placed in a dedicated *manager* partition to get a maximum execution time.

The timing diagram of the setup, warm-up, experiment and shut-down phases is shown in Fig. 8.2. When the application is started, it moves itself to the *application* partition, sets up itself and starts the Manager task. (1) After user interaction to start the experiment, the GUI notifies the Manager. (2) That creates the isolated threads successively and moves them to their partitions. They initialize themselves and then go into warm-up mode. The Manager starts also all other tasks. (3) When all is set up, the Manager activates the experiment phase and notifies the GUI that displays the new state to the user. (4) The end of the experiment is either initiated by the user in the GUI or after an abort condition. (5) The isolated tasks and calculation threads are notified to terminate, the latter just exit. (6) The isolated tasks wait to be synchronized to terminate one after the other. During the whole run-time, various system processes may be executed in the *System* partition. In the experiment phase, the Manager task monitors the activity of the isolated tasks. It

¹Online: <http://qt-project.org/doc/qt-4.8/signalsandslots.html> (visited November 12, 2015)

8. Application Examples

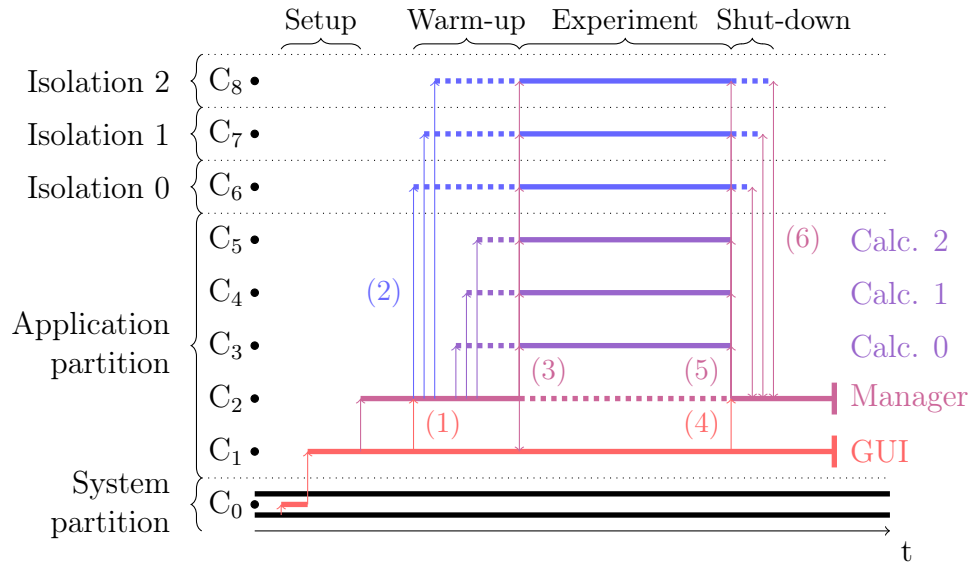


Figure 8.2.: Timing example: (1) User interaction to start. (2) Thread creation and isolation. (3) Activation of experiment. (4) User interaction to terminate. (5) Notification to exit. (6) Synchronized termination of isolations.

also forwards control points to and statistical data from them. These jobs have a lower priority, therefore, the Manager could share its CPU with other tasks.

The isolated tasks started as POSIX threads wait for the Manager task to notify them when they have arrived to their CPU. Then, they initialize their memory, IPC buffers and I/O resources. Memory buffers are by default allocated on the local NUMA node when they are first accessed (Affinity on First Touch, Section 2.2.4). Therefore, it is important to initialize all memory buffers when running on the target CPU. After initialization, the isolated tasks notify the Manager that in turn continues to set up the experiment. The isolated tasks start to warm up and stand by to start their real-time jobs until the manager signals them. The warm-up phase is important to load all used data into the private caches and to keep it there. The linked control routines need a warm-up mode to execute a calculation that is very similar to the real work but that does not have any side effects. I/O accesses should also be included in the warm-up routine but it must be ensured that this does not have any negative impact on the external physical system (plant). When the manager has registered all isolated tasks and all calculation tasks standing by, it releases them to start the real-time experiment.

8.2.3. Closed-Loop Control and I/O

The code for the closed-loop control will be modeled with graphical algorithm design tools. These tools generate C code that is compiled before application start. The

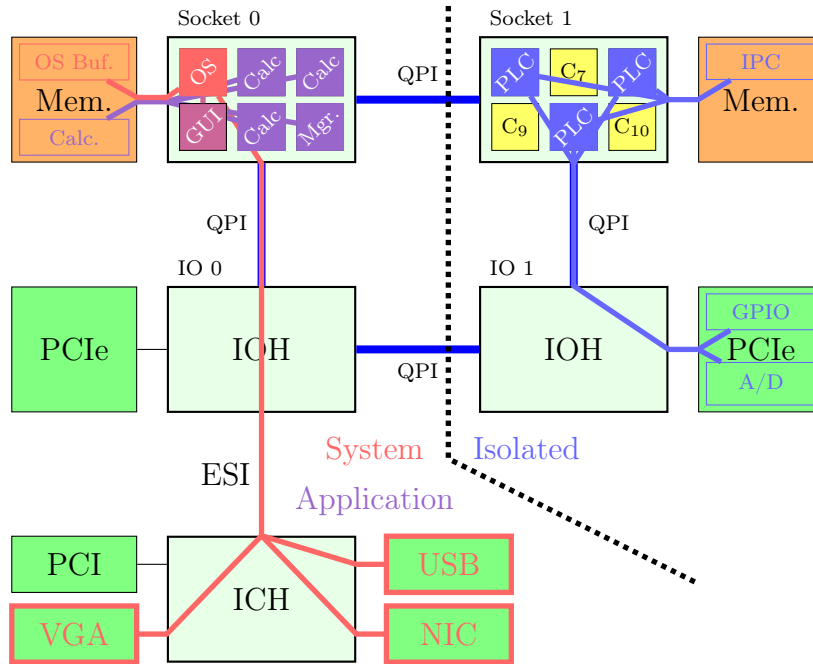


Figure 8.3.: Example application I/O partitioning.

isolated tasks are not allowed to use system calls, their run-time must be predictably short and their memory foot-print (instructions and data) must be small enough to be kept in the private caches. Therefore, the calculation routines are evaluated during the compilation with static code checkers to preclude actions violating the isolated tasks concept. The linker can generate a map file to analyze the memory usage of different parts of the application. This can be helpful for reducing the cache footprint of hard real-time tasks [Fog13b, Chap. 16]. Some of these properties are clearly detectable, others can only generate a warning to be analyzed more carefully in the run-time. The application offers a mode to evaluate the linked control functions. Therefore, the functions are executed in a similar way as in the real-time experiment mode and supplied with test data. The execution time is analyzed statistically to be able to rate their suitability.

During application start, this code is linked as shared library. The isolated PLC tasks execute the closed-loop control function as selected in the GUI. Parameters changed by the user must be handed to the isolated tasks in a synchronized way. At the end of an experiment, the isolated closed-loop control tasks are notified by a flag that they should terminate their work and unwind their isolation mode. In a coordinated manner, they deactivate the Kernel patch, revert the partitioning settings and quit.

The partitioning is shown in Fig. 8.3 to illustrate the separation of data paths. The OS services, the GUI, Manager and Calculation threads use memory on their

8. Application Examples

node and devices in their partition (i. e. the left IOH and the ICH. These devices are controlled by OS drivers. The PLC threads in the isolated partitions use peripheral Data Acquisition adapters connected to the right ICH. They access the I/O devices with user-mode functions implementing the direct access. The only transfer across the partition limit is the Manager task accessing the IPC buffers.

8.2.4. Observation of Execution-Times

In this setup, the occurrence of timing violations must be detected because it can render the results of an experiment invalid. Therefore, all execution times are observed and compared to their allowed latencies. In case of timing errors, a counter will track them. The Manager reads these values from the IPC buffer and reports them to the GUI.

To support the evaluation of user-defined control functions, the isolated tasks generate a statistic of their execution times. This statistic must influence the execution times as little as possible. Therefore it is kept light-weight with regard to memory foot-print and calculation overhead. The simplest solution is the tracking of only maximum and minimum execution times, possibly adjoined with the average. A viable compromise of overhead and informative value is a histogram. The offset, number, and width of recorded intervals is configurable to match the expected execution time and the resulting outliers. This allows to flexibly activate the histogram for evaluation and remove its impact in time-critical situations.

8.2.5. Error Recovery

Generally, a real-time application must be implemented error-free because a failure would be the ultimate timing violation. But apart from every complex code being error-prone, this application supports linking user-supplied code for the closed-loop control functions. Since this user-provided code may contain faults or unbounded loops, the application needs a strategy to handle them.

Two main classes of programming errors can occur: *Signaled error conditions* and *unbounded loops*. Usual program errors such as illegal instructions, division by zero, or access to memory regions without permission are referred to the application by a UNIX signal. Some signals can be handled and the program may continue, other signals allow only to shut down gracefully. With active isolated tasks, these should be shut down similar to the end of a real-time experiment. Since this usual shutdown contains complex synchronization, the emergency shut-down follows similar steps but omits the waiting for other tasks. This is intended to restore the system with highest likelihood to be able to restart another real-time experiment without the need to reboot the whole system.

If an unbounded loop is executed in a control function, the time-triggered task is blocked and will no longer execute or respond. But since the isolated task is not

influenced by other processors, a special recovery is required. With the isolation patch described in Chapter 5 and especially with the deactivation of the timer interrupt (Section 5.3), the Interrupt Flag (IF) has not to be cleared and the isolation of a CPU can be activated from a different processor. This also allows to deactivate the isolation of a task that was detected as stuck and to terminate this thread with the usual functions such as `pthread_cancel()`.

8.3. Other Applications

This Section shortly outlines the migration path for applications presented as examples in other publications.

8.3.1. Application-Specific Integrated Circuit

Application-Specific Integrated Circuits (ASICs) are specifically designed Integrated Circuits (ICs) that can contain multiple processors, caches, I/O units and communication paths. The processors can be of different performance or even different architectures (heterogeneous system). The communication may use hardware queues and synchronization means. The architecture is usually optimized for a single application and the software for the various processors is adapted to them. The extra development costs for an ASIC is only sensible, if the performance of general-purpose processors is not sufficient or if a very high number of units sold levels them out.

During development (before the hardware is available), for small series, or for lower price, the migration to Commercially off-the-Shelf (COTS) systems is reasonable. The concept of isolated tasks allows to implement the individual processing elements of an ASIC in bare-metal tasks with a General-Purpose Operating System (GPOS) running on the remaining CPUs. A further advantage of full-featured processors over Digital Signal Processor (DSP) and Field-Programmable Gate Array (FPGA) is that floating-point arithmetic is available at full execution speed and it is not required to circumvent it with fixed-point algorithms [JCW12].

Figure 8.4 shows an example ASIC [Sch13, Fig. 3.10]. Its purpose is not revealed, but the architecture demonstrates the different jobs of the four processing elements. Processor 4 accesses only the Input and can communicate with processor 3. It could serve as hard real-time input stream processing. The processors 2 and 3 access part of the memory using a common bus. They appear the most versatile. Processor 2 has hardware queues both for the input and output to compensate for not being ready to process pending input and to send bursts of output faster than the receiver can process. The memory is split and only processor 2 can access the whole range.

Independently from the real application domain of this ASIC, the hardware units can be mapped to the isolated task concept as follows. The software running on processors 1 and 4 is executed definitely in isolated partitions with assigned DAQ

buffers in the IPC section to communicate and distribute the packages. More complex handling or firewall rules can be assisted by threads in the application partition. That builds a slow path for packets that disqualify for simpler fast path rules. This is for example usable for the intrusion detector Snort [Dom08, Chap. 8] to include flexible user-provided filtering rules. In an extreme form, data parallelism and pipelining can be interlaced depending on the throughput of the tasks. A single first stage could buffer incoming data, distribute it to N tasks of the second stage (which take longer for each packet, e. g. to decode), then two tasks of the last stage multiplex the packets and send them using two network adapters [Lal13, Fig. 15.13 and 15.14]. However, if the data moved between processors is too large, this can impose a too large contention on the caches [DAR12]. In this case, it should be evaluated if executing the stages of the pipeline on the same CPU or multi-core lowers the cache contention.

8.3.3. Medical Imaging

The combination of Positron Emission Tomography (PET) and Magnetic Resonance Imaging (MRI) allows to improve the image quality because both technologies cover different material of the human body [Wei+12]. The data processing poses challenges in terms of real-time and compute power to present a live image [Gol+13]. Another example is Amide² (A Medical Imaging Data Examiner), a three dimensional image processor. Domeika presents how to parallelize this software using the technique Data Decomposition [Dom08, Chap. 7].

Those applications can generally be implemented using the isolated task concept to improve the timing predictability and get the most performance from the hardware. Since data is processed in streams, the tasks do not need OS services if the communication is based on shared-memory IPC. The Data Acquisition can be realized either with PCIe adapter cards or via NICs that receive data from preprocessing hardware [Gol+13]. The isolated tasks benefit from low-jitter execution and can use the compute time that otherwise is used by the OS itself.

8.3.4. More Examples

In robotic systems, increasingly high compute requirements are driven by more complex image recognition and behavior algorithms to make robots more environment-conscious. In mobile systems, the energy is limited by the battery weight and power consumption of installed controllers. A multi-processor system can benefit from consolidation effects. Bäuml and Hirzinger present the software architecture of such a complex mechatronic system and how it is implemented on a distributed system [BH08, Fig. 2]. They use multiple Linux systems for soft real-time and GUI jobs

²Online: <http://amide.sourceforge.net> (visited November 12, 2015)

8. Application Examples

and a two-processor QNX system for the hard real-time tasks. The communication is very complex and the partitioning should be optimized to reduce the crossing of partition limits [Moy13d, Sect. 6.3.1, p. 210]. Using the isolated task concept, the distributed system can be migrated to a single multi-processor system with different partitions. The systems shown in their picture can directly be mapped to partitions where the Linux systems map to a single or multiple *application* partitions and the two processors of the QNX system map to two according *isolated* partitions.

A further use-case for isolated tasks is computing hardware research. As shown in Sections 2.1.8 and 2.1.10, the behavior of undocumented processor features is difficult to derive in the presence of concurrent execution [Fog13b]. Instead of special micro-kernels such as *smp.boot* (used in Sections 4.6 and 7.2), those benchmarks can also be executed in isolated tasks on arbitrary systems during normal operation.

9. Conclusion

In the area of hard real-time and high-performance computing on multi-processor systems, this work presents a new approach for realizing hard real-time tasks executing on the same system as an application requiring performance and versatility. Further, a solution for the jitter problem on the x86 architecture is presented. The performance isolation [Des13a, Sect. 2.4.2, p. 129] of multiple partitions is realized by reserving some CPUs of a multi-processor system for bare-metal execution and partitioning this system along NUMA nodes to reduce mutual influence. This allows to utilize modern COTS systems, for example x86 servers with 8 to 20 processors and multiple PCIe connectors for Data Acquisition (DAQ) peripheral expansion cards. An arbitrary GPOS can be installed that supports the compute-intensive part of the application and its soft real-time requirements. The implementation of complex embedded systems must regard the hardware architecture as it was always the case when developing embedded systems based on Micro-Controllers (μ Cs) and especially Application-Specific Processors (ASPs) [Moy13a]. The x86 architecture tempts employing the same methods as in general-purpose computing, but to realize hard real-time, advanced methods of such systems must be applied. In the previous Chapters was presented, how this concept differs from previous approaches and how it was realized on an x86 NUMA system. This includes the modification of the employed Linux kernel to tolerate the isolation of some processors and a detailed analysis of the hardware behavior to confine the execution time jitter caused by other processor's execution and contending data transfers.

The advantages of using a commodity multi-processor server system with isolated processors over embedded μ Cs and DSPs is the availability of high computing performance with a superior cost ratio on the well established x86 environment allowing a homogeneous development tool-chain. Existing multi-threaded applications can easily be extended with hard real-time tasks on their dedicated processors without needing to reevaluate the timing of the base application. All available third-party libraries for GUI (e.g. Qt), image and video manipulation and analysis (e.g. Open CV), network services and many more can be used in the application partition. During development, the flexible configuration and interactivity allows a fast turn-around without complex cross building and remote debugging. Applications can be adapted to the user's needs by loading modifiable code generated with graphical tools like Simulink (e.g. to change PLC feedback loops or in Hardware-in-the-Loop (HiL) simulators). The IPC between multiple isolated tasks and with the soft real-time threads in the application partition is realized with low-latency and high-throughput

9. Conclusion

shared memory buffers and efficient spinlock or wait-free algorithms. Public services (e.g. web server, database access) can be realized with established and proved software increasing the security without impact on the timing of the hard real-time partition. This also allows to redistribute task groups to other systems (e.g. in a fail-over situation) without the need to reevaluate the timing.

The assignments listed in Chapter 1 are addressed as follows. The standard Linux OS is used on x86 hardware. A twelve processor NUMA system was selected as final evaluation hardware. Its two NUMA nodes are separated into system and application partitions on one side and multiple isolated partitions on the other. The system partition executes all usual OS services and internal maintenance jobs as well as handles the devices used by soft real-time tasks. The application partition supports the Bound Multi-Processing (BMP) approach of partitioning by reserving processors for selected soft real-time tasks. They may use all OS services but the system designer has finer control what executes concurrently. The isolated partitions are single processors that are isolated from the OS. This means that a single process is executed without interruption or interference. The bare-metal execution prohibits using system calls forcing a pure user-mode execution. The number and assignment of processors to partitions is flexible and can be modified between executions without restarting the whole system. Since the isolation of some CPUs from the Linux kernel is not tolerated by an unchanged OS, the causes of the observed problems were analyzed and addressed with slight modifications. The resulting Kernel patch allows a stable execution as intended. Since the bare-metal execution environment of isolated tasks limits their applicability, the most important functions of communication and accessing peripheral devices are restored by using shared memory based lock- and wait-free message queue algorithms and direct device access based on I/O ports and memory-mapped I/O. This supports the time-triggered paradigm that is the base of many control processes. Additionally, a way to execute interrupts in the isolated bare-metal environment is presented in Section 6.4 that allows to employ also the event-triggered paradigm and even preemptive scheduling in an embedded Real-Time Operating System (RTOS).

The timing behavior of this realization was analyzed theoretically and evaluated practically. In the strict sense of its definition, hard real-time requires a formal verification. This includes the decomposition of a task into its basic blocks that build the Control Flow Graph (CFG). Based on branch probability, loop detection and count prediction, the paths through the CFG can be analyzed. For the execution time of each path, the execution time of the basic blocks must be determined. Although the blocks do not contain any branches, their run time is difficult to predict on such complex hardware as the x86 architecture. Many different approaches exist, but in the presence of shared functional units where contention imposes varying interference, this work settles on evaluating the Worst-Case Execution Time (WCET) of basic blocks practically. Systems using the x86 architecture use a range of structures to connect multiple processors to the same memory. In Chapter 7, multi-core, Uniform

Memory-Access (UMA) and NUMA systems are compared for their predictability imposed by memory and I/O accesses. So far, no other related work has presented the NUMA architecture capable of very predictable hard real-time execution in the presence of uncontrolled concurrent execution on other processors. For an Intel Westmere based system with twelve processors, it is shown how memory and I/O accesses can be regarded in the partition design to realize hard real-time tasks in isolated bare-metal tasks with minimum jitter and high predictability while the other partition executes an unrestricted GPOS with arbitrary applications.

9.1. Summary

Multi-processor systems have long been employed for hard real-time systems with exceptionally high performance requirements using special OSs. Since the broad availability of multi-processor systems in all scales of computing, their application for real-time systems accelerates and triggers an increasing research interest. Chapter 3 presents a wide range of related research. The most important concepts are symmetric and Asymmetric Multi-Processing (AMP) software architectures employing either a single OS or multiple instances for different partitions. The Symmetric Multi-Processing (SMP) concept requires a special multi-processor-capable RTOS to support hard real-time. The drawback is often limited support of third-party libraries and portability of existing applications. The AMP concept uses multiple OS instances that are either managed by an Hardware Abstraction Layer (HAL) or an hypervisor. In this setting, one partition can use a full-featured GPOS and the hard real-time tasks are executed on their own RTOS instances. The coordination and communication between the OSs must be provided by the lower layer. A related approach is the sub-kernel where the HAL layer is provided by the RTOS that executes the GPOS as its lowest prioritized task. Those multi OS instance approaches are not considered in this work because of their limited portability.

The concept of isolated tasks initially aimed for an unmodified Linux system. Even if this proposition was breached by slightly patching the Linux kernel, any standard Linux distribution can be used as base system as presented in Chapter 4. Some settings should be configured in the firmware (BIOS or UEFI) to reduce their influence or to avoid problems by non-responding CPUs. The system is set up for the partitioning using CPU-Sets by moving all running system services into a *system* partition on a dedicated CPU. Further, all Interrupt Request (IRQ) handlers are bound to CPU #0 to ensure their execution when other CPUs block all interrupts. The task intended for isolation can be a process or thread of a main application. It is started, moved to the target CPU-Set and isolated by clearing its IF. This effectively inhibits all external interrupts so that the process itself runs continuously. By adhering to general hard real-time coding standards (e. g. allocating and initializing

9. Conclusion

all required memory in advance to avoid page faults) and not using system calls, it is executed bare-metal on its own CPU.

Extensive experiments have revealed system instabilities during isolated execution lasting longer than several Minutes. Those were analyzed and their causes are addressed in Chapter 5. The circumstances of system blocking are difficult to record because of the instability and unreliability of the system in that condition. The recording was finally done from a remote system that collected extensive information for automated evaluation. The main problems could be located in synchronously waiting Inter-Processor Interrupts (IPIs) and the Read-Copy-Update (RCU) service. It is shown how the IPI sending routines can be located and how they should be modified to inhibit sending IPIs to currently isolated processors. The RCU system and other subsystems are notified by using the same mechanism that prepares hot-plugging a CPU (i. e. physically turning it off or restarting it). With changing the Kernel anyway, a possibility to block all interrupts to an isolated processor without clearing the IF becomes feasible. After routing all IRQs to a different CPU and deactivating all IPIs, only the timer interrupt remains. It can easily be deactivated which allows to let the IF unchanged to allow again selected interrupts or to recover a hung up task. With the presented user-mode interrupt handling, they do not execute Linux code which preserves the timing predictability. This enables event-triggered tasks and even preemptive scheduling in an embedded RTOS. The modifications and infrastructure to activate and support debugging realized for various versions between Linux 2.6.31 and Linux 3.12 in total require a modification of six files.

The bare-metal execution environment is very limited in its applicability. In Chapter 6 is presented, how management, IPC and access to peripheral devices can be realized from the user-mode of an isolated task. Shared memory is a hardware feature that imposes no overhead or OS interaction after set-up. Threads can naturally share global variables within their process, distinct processes can allocate shared memory regions (e. g. System V shared memory). During initialization, those IPC buffers are initialized and allow to manage multiple isolated tasks from a task of the main application to synchronize their start and shut-down. Using wait-free queue algorithms, tasks of different real-time capabilities can exchange messages without the risk of blocking a higher prioritized one. Further is shown how DAQ expansion cards can be accessed from isolated tasks to interact with physical signals.

The applicability of the isolated task concept is presented in practical examples in Chapter 8. A chemical experiment control system is used as prototypical example that integrates most methods presented in this work. A variety of smaller examples complements how the concept can be exploited.

9.2. Evaluation

Together with the fundamental principles of hardware, OSs and real-time computing in Chapter 2, some micro-benchmarks and their results on current x86 hardware are presented. This allows to set the following results into proportion. A first evaluation of the isolated task concept on a commodity desktop Personal Computer (PC) is presented in Section 4.6. Together with a formal verification, this demonstrates the feasibility of the approach. In course of analyzing the causes of system blocking in Chapter 5, further methods of practically measuring the application performance and real-time quality are presented.

The CFG of isolated tasks can be verified formally. This results also in the CFG of the entire CPU because an isolated task is neither interrupted nor is it allowed to call unknown functions (such as system calls). Missing for a complete WCET estimation is the (worst-case) run-time of the basic blocks. Previous publications on determining the execution time of basic blocks on multi-processor systems either restricted the applied hardware or required to control tightly what is executed concurrently on other CPUs that share functional units. Otherwise, the estimation becomes overly pessimistic. However, all related work so far misses to evaluate the NUMA architecture for this use. The extensive evaluation of architectural influences in Chapter 7 includes multi-core, Simultaneous Multi-Threading (SMT), UMA and NUMA systems. Multi-core processors contain multiple CPUs in the same package that usually share the Last-Level Cache (LLC). It is shown, how the inclusive caching mechanism of Intel processors causes the eviction of cached items from the private cache of another processor which increases the unpredictability. To allow arbitrary applications to be executed on the other processors, the approach of controlling their execution presented in other publications is not followed here. Instead, UMA and NUMA systems providing multiple sockets each with distinct LLCs are evaluated. In the former, the cache management of the employed micro-architecture (Intel Core2) takes unacceptable influence on the other CPUs. The solution is to use a NUMA architecture that includes the Memory Controller (MC) in the processor package and connects multiple sockets with the I/O chip set using point-to-point interconnects (Intel processors are based on the QuickPath Interconnect since the Nehalem generation and AMD processors use HyperTransport since 2003). Those systems can be partitioned along NUMA node limits. By restricting the partitions to using their local memory, they do not interfere with the other partition. It is still possible to share memory regions since all memory is available in a global address space only remote access is slightly slower and imposes a greater risk of unpredictable jitter. Extensive benchmarks have shown, that the execution in the application partition does not influence the hard real-time tasks in the isolated partitions.

In a related approach was tried if the logical processors (SMT) of the application partition can be activated without interfering with the isolated tasks. But this

9. Conclusion

imposes functional risks probably caused internal to the Linux kernel. This was not further traced.

After successfully separating the memory transfers to avoid contention, the NUMA architecture does also allow to separate I/O accesses. The interconnect between the sockets supports a ring structure that is implemented in some systems with multiple chip set bridges. The major test system `haixagon` has two IOHs that provide three PCIe connectors each. They are assigned to the adjacent partitions so that the isolated tasks access their DAQ expansion cards without crossing I/O transfers of the system and application partitions. The ICH for other I/O devices (e.g. USB, NIC, graphics) is located in the system partition so that the OS does not interfere with the isolated tasks.

In a company cooperation, the isolated task concept was adjusted for a commercial product. An existing multi-threaded application with sensible soft real-time requirements based on Linux-rt was moved to the application partition without increasing the missed deadlines. It was then extended with multiple isolated tasks that access PCIe DAQ devices with hard real-time demand and a period below 20 μs . The access to the memory-mapped I/O in the range of 12 to 13 μs takes the major amount of time. The control algorithms and data exchange with the main application fit into the remaining time span. The product is successfully deployed and executes multi-day simulations. Advantages of bare-metal tasks related to other approaches are:

- Use of the well-known x86 environment (tool-chain, optimization, debugging).
- Availability of a wide range of third-party libraries (in soft real-time parts of the application), e.g. Open CV (image and video processing), Qt (Graphical User Interfaces).
- Adaptability (during development, adaptive algorithms, prototypes).
- Easy replacement of code (using available and familiar tools), even during runtime (or in very short downtimes).
- Communication between multiple processors via shared memory (high throughput and low latency, manageable application architecture).
- Integration of digital control and interactive services (e.g. via a web server) on the same system using established software (security) while avoiding impact on the timing of real-time tasks.
- Sharing resources is more economical than dedicated resources [MFC01].
- Fail-over: Redistribution of task groups (applications) to other systems without the need of re-evaluation [MFC01].

- Control of every detail of the system due to the open source GPOS.
- Execution of threads of the same application as isolated tasks easing the application development.
- Provision of bare-metal tasks with hard real-time execution for repeatable execution of control tasks, but also for development, test, and debugging.

9.3. Future Perspectives

Generally, the concept of isolated bare-metal tasks can similarly be realized on other multi-processor architectures and other OSs. The features required by an alternate GPOS are the availability of interfaces for process and interrupt partitioning (equivalent to CPU and IRQ Affinity), basic real-time support (e. g. memory locking to avoid the eviction of pages), user-mode enabling of IF manipulation and I/O access, availability of shared-memory or at least threads, and the configuration of other interfering services such as deactivating NMI watchdogs. The use of synchronous IPIs in other GPOSs is very likely, therefore it is most probably required to have the source code available to be able and allowed to modify the OS itself. If certain features are not available, some can be realized with direct hardware access from user-mode (e. g. the interrupt routing can be set directly in the I/O APIC).

In Section 5.4.2, an alternate approach based on the Hotplugging system is described. This is a hardware-independent feature of Linux that allows to physically deactivate a processor, e. g. to replace a defective unit. By imitating the proceeding of waking up a processor during system start or after Hotplugging, a CPU can also be started and initialized by another function without notifying the OS resulting in this processor executing an isolated task bare-metal without OS interference. This should be portable to every OS as kernel module (hardware driver) and would mostly base on hardware features and configuration.

The current realization of the concept is a C library that includes system setup and isolated task configuration. For a more flexible implementation and to simplify the porting of existing applications, a POSIX interface could be implemented that provides the POSIX.4 real-time extensions to isolated tasks. The Kernel feature RCU is also available as user-mode implementation [MDL13]. It supports five modes with different overhead/memory/real-time trade-offs. The variants not using system calls during run-time (i. e. after setup) can be applied to isolated tasks. It is further possible, to use scripting languages to control hard real-time tasks [Yod99]. Klotzbücher and Bruyninckx work on implementing hard real-time tasks in Lua [KSB10] and present an example to apply it to robotics [KB11] as *component based system* [TLH97; Kop98; Kop97, Chap. 2.2]. Further, the runtime of existing real-time programming languages like Ada or Pearl could probably be implemented in an

9. Conclusion

isolated bare-metal task. This would further ease the flexible implementation or transfer of existing applications.

Kiszka and Wagner are modeling *security risks* in a RTOS. Processes have separate address spaces which protects them from other processes but not from the OS [KW07]. As every isolated partition has full control for its CPU, the hard real-time guarantee also protects from malicious or error-prone code in other processes and the OS. So far, the framework is *trustful*: Every process (with required privilege) can isolate itself or terminate isolated hard real-time tasks. The required privilege is to be executed as root user. As long as an *evil user* has no such privilege, the isolated tasks should be secure, but this must be evaluated in further detail [Lei+07]. Besides the computational protection by partitioning the processors and address spaces of isolated tasks, the I/O access could be protected by using the IOMMU to prevent other processes, applications and devices (e.g. via Direct Memory Access (DMA)) from accessing memory and I/O ports that are assigned to isolated tasks [Des13a, Sect. 2.4.2, p. 128].

9.3.1. Hardware

The x86 architecture is very common and available in flexible configurations. It is difficult to predict where the future development of the x86 architecture will head to (apart from the high probability of more processors per system). Current shared-memory systems (UMA and NUMA) are cache-coherent which means that the caches of all processors are held in a consistent state by hardware transparently to the executed programs. They can access memory as if no caches were present. This imposes a great overhead that rises with the number of processors. Therefore, Intel has searched for alternatives with the Single-Chip Cloud Computer (SCC) research processor that waives cache coherence [SCC10]. In such a system, every processor has a private cache and accesses only its own memory partition forming effectively a distributed system (No Remote Memory-Access (NoRMA) cluster). To cooperate, a Message-Passing mechanism is realized based on special scratchpad memories (called message-passing buffer, MPB). This cluster of single-processor systems requires special programming. It is currently unknown, if this architecture will be implemented in future x86 systems. If so, it would provide natural performance partitioning providing a convenient base for the isolated task concept.

A desirable feature of future x86 based processors is a cache-locking mechanism to avoid the eviction of cache lines of other cores as observed in inclusive caches on Intel processors. The possibility to deactivate the Last-Level Cache would require a memory access on cache miss but would avoid the influence of other processors. Offering weaker consistency models would provide a flexible control when caches are synchronized between processors. Fog predicts the implementation of features currently inherent to SoCs such as programmable logic (FPGA) in general-purpose

processors some day. This would enable an application to define specific instructions coded in a hardware definition language [Fog13b, Sect. 2.1].

Alternate hardware must provide multiple processors that can be assigned directly to certain tasks. Capable replacements include the ARM and MIPS architectures that both provide multi-processor support and that are both increasingly offered for server systems. To realize DAQ, the systems must either provide electrical interfaces (General-Purpose I/O, Digital to Analog and Analog to Digital Converters) or allow to install expansion cards with modern point-to-point interfaces such as PCIe. The positive experience made with the NUMA architecture is most probably transferable to other systems with separated memory and I/O nodes that allow to partition the transfers on the system interconnect to avoid conflicts and contention.

An important factor for the implementation of predictably timed IPC algorithms is the availability of atomic operations for user-mode applications. Architectures such as ARM provide only the Load-Linked/Store-Conditional (LL/SC) operation that does not necessarily allow to implement wait-free queues. Similarly, transactional memory as introduced with the Intel Haswell micro-architecture is not suited for hard real-time. A conditional store as well as a transaction may fail and must be repeated which may occur arbitrarily often.

The Intel Atom Silvermont¹ (released end of 2013) is advertised to be more powerful at the same power consumption than competing ARM processors currently found in smartphones and tablets. This emphasizes the applicability of the x86 architecture in performance-critical embedded systems. However, currently ARM succeeds not only in typical μ Cs, but also as processor (or SoC) in general-purpose computers (like Netbooks and cheap computers like the Raspberry Pi and the BeagleBone Black etc.) and most mobile systems (smartphones and tablets). The isolated task concept was tested on the two-processor Panda Board [Ras11]. The ARM big.LITTLE concept integrates multiple powerful processors with the same number of power-efficient weaker (slower) processors. The simplest approach to OS support is to use either group of them and to switch depending on performance requirements. But a heterogeneous system could also be realized with a load-balancer that moves each task to a processor serving best its performance-to-efficiency requirements. Such a system could be used for bare-metal tasks selecting either the powerful or the efficient processors (or a mix of them). Fast interrupts [Tho14] on ARM could be used to implement a low-overhead user-mode interrupt mechanism.

Generally, for all embedded systems with sensitive requirements (in reliability, timing predictability, performance, etc.), any new hardware must be thoroughly evaluated. In x86 systems, the SMI poses a challenge or even renders a system inapplicable. Besides hardware jitter, the documentation of processor and chip set must be consulted for other risks. In the likely case of too scarce documentation, a full evaluation using benchmarks as described in this work is advisable.

¹<http://heise.de/-1857164>

9. Conclusion

9.3.2. Linux

Since the implementation of the Kernel patch up to Linux 3.12 described in Chapter 5, the development of the Linux kernel paced very fast with a new version every 10 to 12 weeks approximately. The tickless feature was already presented in Section 3.2.1. It was introduced in Linux 3.14 and extended later [McK14a]. This modification has similar requirements of targeted processors as this work. This mode is activated whenever only a single process is active on a CPU and it aims for minimum interruption by the timer interrupt and other Kernel systems. These changes might replace the modifications described in Chapter 5 if it can be guaranteed, that only a single task is placed on a CPU and that this task does not issue unwanted system calls. It must be evaluated especially, if the Inter-Processor Interrupts are securely and completely eliminated.

Since Linux 3.7, the Kernel contains a park facility for kernel threads [Gle12] that is used by the Hotplugging system. Instead of terminating those kernel threads on isolated processors, the park mechanism could probably be used to avoid problems when recreating them after an isolation (Section 4.2.3).

Currently, the semantic of memory locking in Linux just avoids memory being swapped out to background memory (i. e. commonly to a hard disk) so that no major page fault caused by a memory access can result in hard-disk reading or writing. But locking a memory region does not guarantee that the pages remain mapped to fixed physical addresses. Therefore, the Kernel might move the physical page frames for compacting or to optimize NUMA placing. Beside the requirement of fixed physical addresses in DMA transfers, this can result in minor page faults as well as cache and Translation Look-aside Buffer (TLB) misses unsuited for hard real-time tasks. It is currently discussed to implement *pinning* to fix the physical placement of memory regions [Cor14b]. Thus, with the current Linux kernel, the advice to `mlock()` all memory used by isolated tasks still holds true, but with future versions, the locking could be modified to allow moving pages in the physical address space. This could result in unpredictable latency. When this will be changed in the future, the notation of *pinning* will be required to guarantee the current feature of not moving a memory region.

Since Linux 3.11, work queues can contain tasks, that may be scheduled on another CPU. So far, if an isolated CPU does not issue work queue jobs, it will not have to handle any [Cor13h]. That may change in the future and a configuration must be found to avoid executing work queue tasks on isolated CPUs. Linux control groups are currently implemented as successor to CPU-Sets. The partitioning of future Linux versions will therefore be based on control groups which may impose other side-effects.

The porting to a new Linux version depends strongly on the integrated new features. Some transfers were trivial, other versions required a completely new evaluation of new interrupt sources that had to be handled by the Kernel modification. If the

9.3. Future Perspectives

implementation for a Linux kernel is available, the concept is however well suited for applications that require the lowest latency that a bare-metal execution can provide as well as multiple cooperating tasks demanding for soft real-time and high compute performance.

A. Test Systems

The following test systems were used in the course of this work. The hardware details are listed further below.

Name	Micro-architecture	Processors
poodoo	Core 2	Multi-Core 1 × 2
octopus	Core 2	UMA 2 × 4
xaxis	Nehalem	Multi-Core 1 × 4
aix	Westmere	NUMA 2 × 4
haixagon	Westmere	NUMA 2 × 6
pandora	Sandy Bridge	NUMA 2 × 8
devon	AMD K10	NUMA 2 × 4

Table A.1.: Test system overview.

A.1. Intel Core 2

The Core micro-architecture was introduced in 2006 as successor of the Netburst (Pentium IV) micro-architecture that became too complex and power inefficient at high frequencies above 4 GHz. The Core 2 micro-architecture is a minor enhancement that gained a wide distribution. Some previous processors included Simultaneous Multi-Threading (SMT) providing two logical processors, but beginning with the Core 2 generation, most general-purpose processors contained two or four physical cores. The quad-core processors are assembled from two double-core silicon chips, hence they have two separated Last-Level Caches.

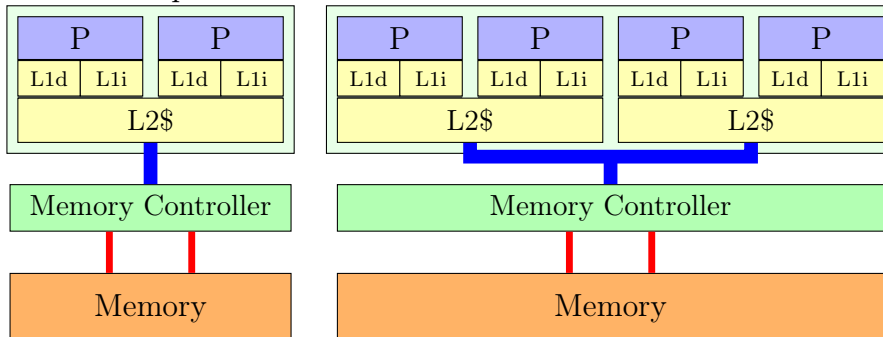


Figure A.1.: Core 2 double- and quad-core cache and memory structure.

A. Test Systems

A.1.1. Test system poodoo

This system is a desktop PC with double-core processor (Fig. A.1, left).

Description	Per entity	Total
CPU type	Intel Core2 Duo E6750	2.66 GHz
Physical cores	1×2	2
L1 Cache	2×32 KiB	64 KiB
L2 Cache	1×4 MiB	4 MiB
Memory	1×6 GiB	6 GiB
Main board	ASUS P5KR	
Memory Controller Hub	Intel P35	
Interconnect Controller Hub	Intel ICH9R	

Table A.2.: Test system *xaxis*.

A.1.2. Test system octopus

The test system *octopus* is a server system with two quad-core chips, thus a total of eight processors connected in a Uniform Memory-Access (UMA) architecture (Fig. 7.4 on page 163).

Description	Per entity	Total
CPU type	$2 \times$ Intel Xeon X5355	2.66 GHz
Physical cores	$2 \times 2 \times 2$	8
L1 Cache	8×32 KiB	64 KiB
L2 Cache	$2 \times 2 \times 4$ MiB	16 MiB
Memory	1×4 GiB	4 GiB
Main board	Intel X3420GP	
Memory Controller Hub	Intel 5000P	
Interconnect Controller Hub	Intel ESB2-E	

Table A.3.: Test system *octopus*.

A.2. Intel Nehalem

The Nehalem micro-architecture contains four processor cores on a chip and integrates the Memory Controller with three lanes on the same die. It supports three levels of caches with a shared L3 cache. Each core can be split into two logical processors (Intel HyperThreading), but this feature is deactivated on our systems.

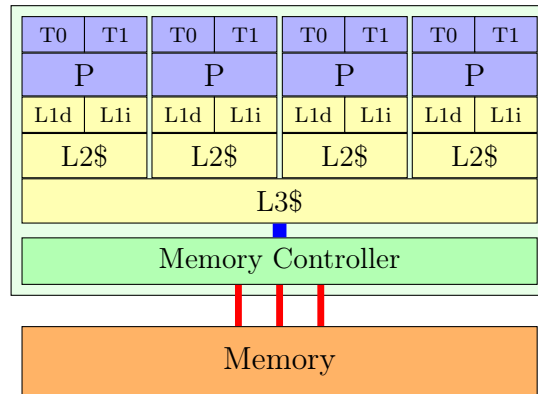


Figure A.2.: Nehalem cache and memory structure.

A.2.1. Test system xaxis

This is a desktop computer with the first-generation Intel Core i7 processor.

Description	Per entity	Total
CPU type	Intel Core i7 Quad-Core 920 2.667 GHz	
Physical cores	1 × 4	4
L1 Cache	4 × 32 KiB	128 KiB
L2 Cache	4 × 256 KiB	1 MiB
L3 Cache	1 × 8 MiB	8 MiB
Memory	1 × 3 GiB	3 GiB
Main board	Intel DX58SO	
I/O Controller Hub	Intel X58	
Interconnect Controller Hub	Intel 82801JI (ICH10 Family)	

Table A.4.: Test system xaxis

A.3. Intel Westmere

The Westmere micro-architecture is a minor enhancement of Nehalem, most notably a shrink from 45 nm to 32 nm die structure. Our test systems with this processor generation are all Non-Uniform Memory-Access (NUMA) systems with two sockets.

A. Test Systems

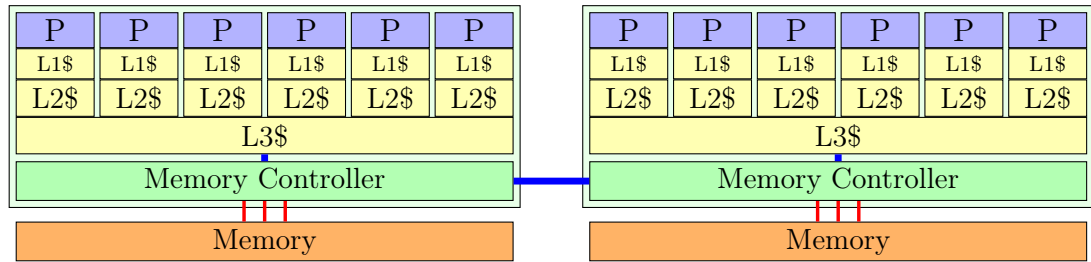


Figure A.3.: Westmere Non-Uniform Memory-Access (NUMA) cache and memory structure.

A.3.1. Test system haixagon

The test system `haixagon` was used for most analyzes. Notable system parameters are listed in Table A.5. The block diagram of the system architecture is shown in Fig. A.4.

Description	Per entity	Total
CPU type	Intel Xeon Hex-Core E5645 2.4 GHz	
Physical cores	2×6	12
L1 Cache	12×32 KiB	384 KiB
L2 Cache	12×256 KiB	3 MiB
L3 Cache	2×12 MiB	24 MiB
Memory	2×24 GiB	48 GiB
Main board	Tyan S7025	
I/O Controller Hub	2× Intel 5520	
Interconnect Controller Hub	Intel ICH10R	

Table A.5.: Test system `haixagon`.

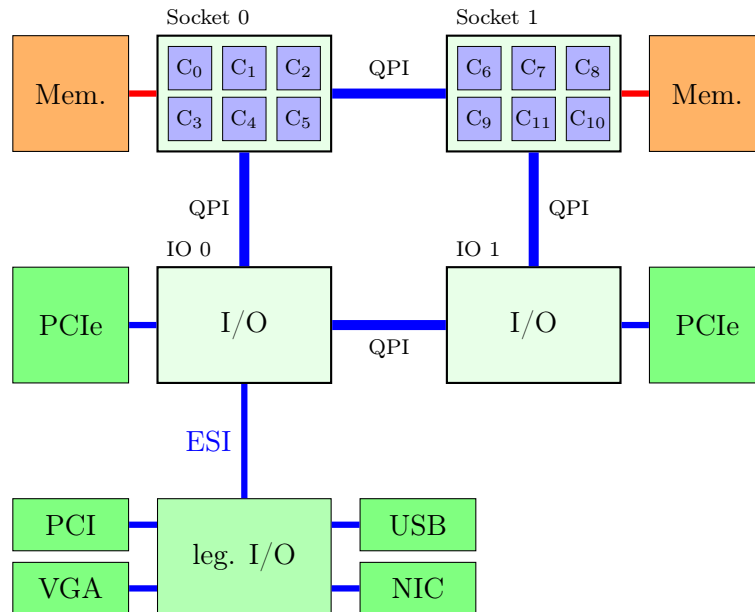


Figure A.4.: Block diagram of haixagon.

A.4. Intel Sandy Bridge

The Sandy Bridge micro-architecture is a major improvement over Nehalem and Westmere but the modifications are in the areas of Instruction-Level Parallelism (ILP) and instruction set extensions (e. g. AVX) that are of minor relevance to this work. The most important change is that Sandy Bridge processors provide more PCIe connections. The previous generations started this trend, but the number of point-to-point links did not suffice for an average system and was extended by the chip set (IOH). The Sandy Bridge micro-architecture allows to build systems with only a single chip set component that provides the *legacy* I/O (ICH, former south bridge, see Fig. A.5).

A.4.1. Test system pandora

The test system *pandora* is a server with two NUMA nodes. The main board provides seven PCIe connectors for peripheral expansion cards and should be suitable for the partitioning of I/O transfers along NUMA nodes.

A. Test Systems

Description	Per entity	Total
CPU type	Intel Xeon Eight-Core E5-2650 2.0 GHz	
Physical cores	2×8	16
L1 Cache	16×32 KiB	512 KiB
L2 Cache	16×256 KiB	4 MiB
L3 Cache	2×20 MiB	40 MiB
Memory	2×32 GiB	64 GiB
Main board	Supermicro X9DRG-QF	
Interconnect Controller Hub	Intel C602	

Table A.6.: Test system *pandora*

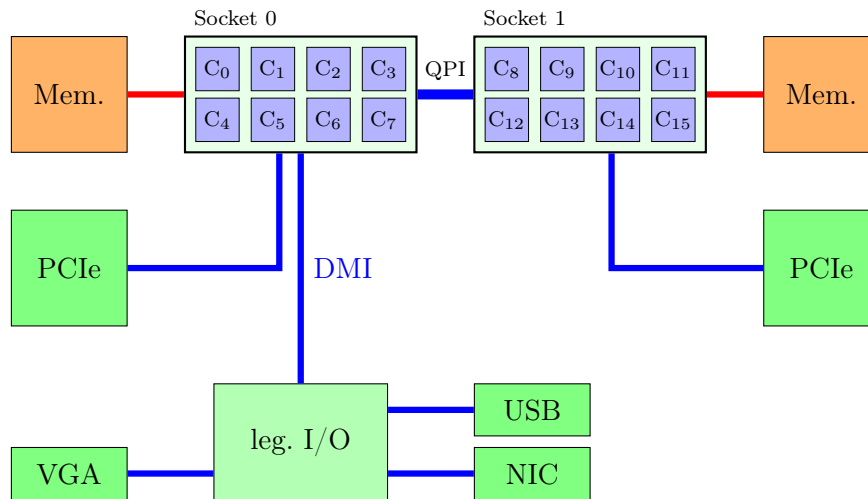


Figure A.5.: Block diagram of *pandora*.

A.5. AMD

A.5.1. Test system *devon*

This is a server system for compute- or memory-bound loads with only a single PCIe expansion slot. The micro-architecture is AMD K10. This was an early test system for the NUMA architecture and only used for the comparison of caching strategies.

Description	Per entity	Total
CPU type	AMD Opteron Quad-Core 2376 2.3 GHz	
Physical cores	2×4	8
L1 Cache	8×64 KiB	512 KiB
L2 Cache	8×512 KiB	4 MiB
L3 Cache	2×6 MiB	12 MiB
Memory	2×16 GiB	32 GiB
Main board	Dell Powerededge SC1435	
Chip set	Broadcom HT-2100/HT-1100	

Table A.7.: Test system *devon*.

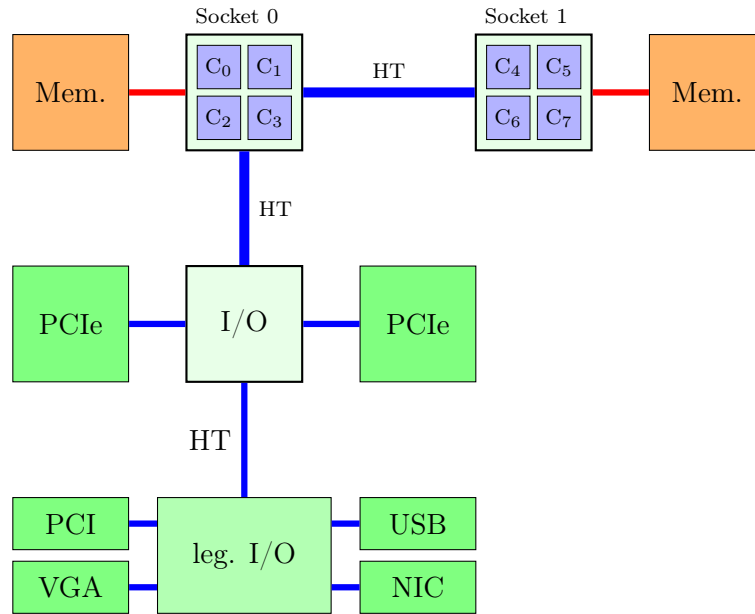


Figure A.6.: Block diagram of *devon*.

B. *smp.boot* Kernel

In embedded systems, the term bare-metal execution describes the situation, that a program runs directly on a processor, i. e. without operating system. In that setting, the program has to manage the hardware itself and no comfort of OSs like files, input/output driver, or network is available. On the other hand, the environment is very controllable and the overhead of unused functionalities can be avoided.

Architectures for embedded systems (μ Cs) are usually very easy to set up (boot and configure) for *bare-metal* programs. However, the x86 architecture – although backwards compatible to the 8086 – is far more difficult to initialize and operate. To support benchmarking low-level code in a strictly controllable environment, the lightweight *smp.boot* Kernel was implemented.

This Kernel boots and initializes a multi-processor system with 32 or 64 Bit protected mode. Paging is set up, all processors can be started (configurable) to execute a benchmark payload. The system does not activate interrupts, therefore it is well suited for research on the hardware behavior in a strictly controllable environment. All execution is in kernel-mode.

The initialization of the hardware is based on *Bran's Kernel Development* tutorial [Fri05] and the web sites OSDev.org¹ and Lowlevel.eu². The multi-processor initialization is mainly based on the processor documentation [Intel13c]. The *smp.boot* Kernel is published as open source Software³.

¹Online: http://wiki.osdev.org/Main_Page (visited November 12, 2015)

²Online: <http://lowlevel.eu/wiki/Hauptseite> (visited November 12, 2015)

³Online: <https://github.com/RWTH-OS/smp.boot> (visited November 12, 2015)

C. Examples for Real-Time systems

Here are listed some more examples that are suited for the concept of isolated tasks on x86 multi-processor systems because they require hard real-time and have a high demand for compute performance [Tan12].

- Robotics: wide range of jobs from fine grained control of movements up to compute-intensive routing, map generation and image analysis [YSL09; KB11; SSL09; BH08]. In mobile robots, an additional challenge is the energy efficiency and weight reduction.
- Driver support (assisted driving) and self-driving cars [GF07; SFR12].
- Industrial control system for time-critical control of a machine and visualization [Kis09].
- Video-based sorting (e.g. quality check of produced goods, separation of unattractive vegetables, routing of letters and parcels) [SFR12].
- Baggage transportation at airports: Read bar-code, query database for flight information (gate), redirect to correct conveyor belt.
- Engine Control Unit: 1500 RPM, one degree of crankshaft rotation: 111 μ s [McK08]. More realistic in current car engines: up to 10 000 RPM: 16 μ s.
- Flight simulation: Distributed simulation [PB01; Rob+03; XX11] can be realized on multi-processor systems with multiple concurrent isolated tasks synchronizing via shared memory.
- Steam-Boiler [ABL96]: A formal problem specification [Abr96] was given to start a competition to advance specification and validation. The result is a Programmable Logic Controller (PLC), but could be part of a larger system.
- Hardware-in-the-Loop (HiL) integration of electric ships [Paq+09] and HiL simulation [PB01; LQ12].
- Medical imaging e.g. X-Ray [TLH97; Intel08b] or Positron Emission Tomography (PET) [Sch+11]: 3D image processing [Dom08, Chap. 7] or real-time number crunching for live images during surgical intervention.

C. Examples for Real-Time systems

- High-frequency trading [[Cor13d](#)].
- Controlling a Radio Telescope [[Bül+04](#)].
- Renewable Energy Research Hardware-in-the-Loop testbed [[Mol+13](#)].
- Software-defined Radio [[Wan+13](#)].
- Behavior research: Systems for interactive test series presenting stimuli and timing the reaction of probands [[Fin01](#)]. Often, the timing constraints are in the range of 1 ms [[PHW02](#); [Ste06](#)], but if the latency must be lower and the compute performance higher, this concept could be helpful.
- Control of mechatronic systems under consideration of friction and mass inertia for rapid prototyping or educational purposes [[Wei11](#)].
- Active Noise Control: Cancellation or reduction of audio noise by generating a phase-inverted signal. For multiple sources, the math is complex and must be real-time (below the speed of sound).
- Direct control of telecommunications and networking devices e.g. during development or for research purposes.
- Avionic systems requiring hard real-time control tasks and a high compute performance with low weight and energy consumption [[BS97](#)].
- Applications currently realized with specially designed multi-processor System-on-a-Chip (SoC) (Application-Specific Processor, ASP) based on Digital Signal Processor (DSP) or Field-Programmable Gate Array (FPGA) [[Sch13](#), Fig. 3.10 and 3.19].
- Gaming consoles: The Playstation 3 was based on a special processor (Cell BE) that was tried to be placed in High Performance Computing (HPC) but did not succeed widely because of its complicated programming model. The current generation of HPC clusters is based on PC processors. This serves as example that Application-Specific Processors (ASPs) are pushed aside by general-purpose processors.
- 3D printing: instead of pre-computing the casting process, the path of the printing head could be determined on the fly based on the input model and a simulation of the heat dissipation in the created structure.
- Embedded multimedia systems that traditionally have been implemented with custom hardware are increasingly build with Commercially off-the-Shelf (COTS) hardware and using a General-Purpose Operating System (GPOS) [[Fue+12](#)].

- Hidden Markov Models for speech recognition, lip-reading and gesture recognition [Nak+12]. These applications are usually classified as soft real-time, but if the performance of a processor shared with the operating system (OS) does not suffice, it can be implemented as bare-metal task.
- Cryptography: encrypt or decrypt a data stream before transmitting over high-throughput connections (e.g. ocean cable, satellite) [Tan+12].

Acronyms

μC Micro-Controller

μs Microsecond

ADC Analog to Digital Converter

ALU Arithmetic Logical Unit

AMP Asymmetric Multi-Processing

AP Application Processor

API Application Programming Interface

APIC Advanced Programmable Interrupt Controller

ARM Advanced RISC Machines

ASIC Application-Specific Integrated Circuit

ASP Application-Specific Processor

BCET Best-Case Execution Time

BMP Bound Multi-Processing

BP Base Pointer

BSP Bootstrap Processor

CAS Compare-and-Swap

CFG Control Flow Graph

CISC Complex Instruction-Set Computer

COTS Commercially off-the-Shelf

CPL Current Privilege Level

Acronyms

CPU	Central Processing Unit
CU	Control Unit
DAC	Digital to Analog Converter
DAQ	Data Acquisition
DI	Device Interrupt
DMA	Direct Memory Access
DPL	Descriptor Privilege Level
DSP	Digital Signal Processor
EDF	Earliest Deadline First
FAD	Fetch-and-Add
FIFO	First-In, First-Out
FPGA	Field-Programmable Gate Array
GHz	Gigahertz
GiB	Gibi-Byte
GPIO	General-Purpose I/O
GPOS	General-Purpose Operating System
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HiL	Hardware-in-the-Loop
HMI	Human-Machine Interface
HPC	High Performance Computing
HPET	High-Precision Event Timer
HT	HyperTransport
Hz	Hertz

I²C Inter-IC Communication
I/O Input/Output
I/O APIC Input/Output Advanced Programmable Interrupt Controller
IC Integrated Circuit
ICC Interconnect Controller
ICH Interconnect Controller Hub
IDT Interrupt Descriptor Table
IF Interrupt Flag
ILP Instruction-Level Parallelism
IOH I/O Controller Hub
IP Instruction Pointer
IPC Inter-Process Communication
IPI Inter-Processor Interrupt
IRQ Interrupt Request
kB Kilobyte
kHz Kilohertz
KiB Kibi-Byte
L1\$ First Level Cache
L2\$ Second Level Cache
L3\$ Third Level Cache
LAN Local Area Network
LIFO Last-In, First-Out
LL/SC Load-Linked/Store-Conditional
LLC Last-Level Cache
localAPIC Local Advanced Programmable Interrupt Controller

Acronyms

MB Megabyte

MC Memory Controller

MCH Memory Controller Hub

MES Manufacturing Execution Systems

MHz Megahertz

MiB Mebi-Byte

MIMD Multiple Instructions Multiple Data

MIPS Microprocessor without Interlocked Pipeline Stages

MMU Memory Management Unit

MP Message-Passing

MPMD Multiple Programs Multiple Data

ms Millisecond

NIC Network Interface Card

NMI Non-Maskable Interrupt

NoRMA No Remote Memory-Access

ns Nanosecond

NUMA Non-Uniform Memory-Access

OS Operating system

PC Personal Computer

PCI Peripheral Component Interconnect

PCIe PCI Express

PI Platform Interrupt

PLC Programmable Logic Controller

PMC Performance Measurement Counter

ppm Parts per million

QPI QuickPath Interconnect

RCU Read-Copy-Update

RDMA Remote Direct-Memory Access

RISC Reduced Instruction-Set Computer

RMA Remote Memory Access

RPM Revolutions per Minute

RT-Patch Real-Time Preemption Patch

RTOS Real-Time Operating System

SCADA Supervisory Control and Data Acquisition

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data

SMI System Management Interrupt

SMM System Management Mode

SMP Symmetric Multi-Processing

SMT Simultaneous Multi-Threading

SoC System-on-a-Chip

SP Stack Pointer

SPARC Scalable Processor Architecture

SPMD Single Program Multiple Data

SSH Secure Shell

TAS Test-and-Set

TDMA Time-Division Multiple Access

TI Timer Interrupt

TID Task ID

TLB Translation Look-aside Buffer

Acronyms

UMA Uniform Memory-Access

USB Universal Serial Bus

WAN Wide Area Network

WCET Worst-Case Execution Time

x86 80x86

x86_32 80x86 (32 Bit)

x86_64 80x86 (64 Bit)

XCHG Exchange

Glossary

80x86 (x86) Architecture of all Intel 80x86 compatible processors. It can further be separated into the 32 Bit and the 64 Bit sub-architectures.

80x86 (32 Bit) (x86_32) Sub-architecture of x86 with 32 Bit instruction set.

80x86 (64 Bit) (x86_64) Sub-architecture of x86 with 64 Bit instruction set.

Actor Allows to influence a plant triggered by a signal (e. g. magnetic valve, motor, heater), i. e. converts numerical values into physical quantities. Input of a physical system, output of a controller.

Advanced Programmable Interrupt Controller (APIC) Part of the x86 architecture that handles interrupts from peripheral devices and between CPUs. It consists of a localAPIC in each CPU and an I/O APIC in the chip set.

Advanced RISC Machines (ARM) Processors using the Reduced Instruction-Set Computer (RISC) architecture provided by ARM Holdings and produced by various companies. Very common in mobile systems and multimedia devices.

Affinity Binding a process or interrupt handler to a specific processor (or set of processors).

Aligned A memory region is aligned, if its offset is a multiple of its size. Some instructions require or work faster with aligned data.

Analog to Digital Converter (ADC) Voltage measurement that generates a digital number proportional to the input. Required to process sensor data in a computer.

Application Software serving a greater purpose, consisting of possibly multiple processes and threads.

Application Binary Interface (ABI) The definition explains how compiled programs interact with the system and with each other.

Application Processor (AP) The processors waiting during system start to be initialized by the Bootstrap Processor.

Glossary

Application Programming Interface (API) The definition includes function names, data structures and rules how programs interact with libraries or operating systems.

Application-Specific Integrated Circuit (ASIC) Special IC developed for a specific application.

Application-Specific Processor (ASP) Special processor developed for a specific application possibly including multiple CPUs and other functional units such as buses, hardware message queues, caches, etc.

Architecture The processor type distinguished by instruction set. The term “architecture” is also applied to software and to the entire hardware system with buses, memory and devices.

Arithmetic Logical Unit (ALU) The functional unit of a CPU that actually calculates. It can be instructed to execute various arithmetic and logic operations on given arguments stored in registers. It is controlled by the Control Unit.

Asymmetric Multi-Processing (AMP) Multiple operating systems instances run concurrently on a multi-processor system on dedicated CPUs.

Atomic A single instruction or function is always executed inseparable and no intermediate states are visible to other processors.

Automation pyramid Leveled model of components in a complex automated system.

Bare-metal Execution of low-level software directly on the hardware, i. e. without other software layers such as libraries or operating system below.

Base Pointer (BP) Register in x86 systems commonly used to manage the current stack frame.

Basic block Part of code with single entry and exit points that does not contain loops and branches, thus that is executed linearly.

Best-Case Execution Time (BCET) The minimum time, a function or basic block can technically require to execute.

Bit mask Variable or register where every bit is interpreted as on/off switch. Allows to set arbitrary combinations of bits.

Bootstrap Processor (BSP) The processor that starts first to initialize a system. It wakes up the Application Processors.

Bound Multi-Processing (BMP) Symmetric Multi-Processing (SMP) system with tasks bound to dedicated CPUs.

Branch Low-level instruction to conditionally continue the execution elsewhere or with the subsequent instruction. See also: jump.

Bus Type of interconnect with all devices connected to the same link. Only a single device may send at any time. It places the target address and data on the bus and the receiving device reads the data. All others should ignore that transmission (but could “snoop”).

Cache A fast but rather small memory located in the CPU that transparently holds the most recently used data to accelerate subsequent accesses.

Cache coherence Hardware feature in multi-processor systems to synchronize all private caches to present a view on shared data as if no caches were involved (transparency). If two caches hold a copy of the same variable and one changes, the other must either update or invalidate.

Cache line Smallest entity that the caches are organized in. Typically 32 or 64 Bytes. Every access to smaller elements still requires a full cache line to be transferred to/from the lower level cache or main memory.

Cache miss If a requested cache line is not present in the cache, the data must be loaded from the level below or from main memory.

Cell level Middle level of the automation pyramid supervising multiple Programmable Logic Controllers of the field level.

Central Processing Unit (CPU) A single processor that executes code independently from others.

Chip set The ICs supporting the CPU, e.g. north bridge and south bridge.

Commercially off-the-Shelf (COTS) This is a computer assembled by easily available commodity parts.

Company level Top level of the automation pyramid managing the components of the cell level.

Compare-and-Swap (CAS) Atomic operation: Compare variable with given constant and swap the variable with a new value if they are equal. Generalization of Test-and-Set (TAS).

Complex Instruction-Set Computer (CISC) Processor architectures with a large instruction set. Many operations can be implemented with fewer instructions (than on RISC) but some instructions take longer to execute.

Compute Unit (CmU) A functional unit containing one or more tightly integrated processors sharing some resources (nomenclature of AMD).

Context switch Changing between user- and kernel-mode, usually for interrupt handling or to switch to a different task.

Control Flow Graph (CFG) Directed graph representation of a computer program. The nodes are basic blocks and the edges represent the possible flow of control from block to block. The CFG is used in compiler optimization and execution time estimation.

Control Unit (CU) The functional unit of a CPU that reads the program instructions and manages the registers, Arithmetic Logical Unit (ALU), memory, and I/O.

Core A single processor in a multi-core package, usually sharing the Last-Level Cache with others. This term is used explicitly for multi-core processors, otherwise, “CPU” or “processor” denominates a unit independently from the package.

CPU-Set Linux feature to generate process containers that can be assigned to a group of CPU. Alternative to process affinity.

Critical section A section of code where shared resources are used that must be protected from concurrent access.

Cron Service of the operating system to execute maintenance work periodically (every hour, day, week, etc.).

Current Privilege Level (CPL) The privilege level, a process currently executes in. A CPL of 0 is kernel-mode, 3 is user-mode.

Cycle A CPU cycle is the clock rate driving a processor. Its inverse is the clock frequency. In a current computer with 2 GHz clock frequency, a cycle is 0.5 ns.

Data Aquisition (DAQ) Digital and Analog input and output of physical signals, including General-Purpose I/O (GPIO), Analog to Digital Converter (ADC) and Digital to Analog Converter (DAC).

Deadline scheduling Real-time scheduling algorithm based on defined deadlines and run-times for every task.

Deadlock Situation where multiple tasks wait for a Semaphore and block each other mutually.

Deadlock avoidance Pessimistically grant access to a Semaphore only, if all following code paths can not induce a deadlock.

Deadlock detection Detect a deadlock situation to recover subsequently.

Deadlock prevention Design a system in way, that deadlocks are not possible.

Descriptor Privilege Level (DPL) The privilege level of a memory page or segment. A process with Current Privilege Level below or equal can access this resource. The DPL of 0 is commonly used for kernel-space, the DPL 3 is for user-space.

Device Interrupt (DI) Interrupt triggered by a peripheral device, generally an IRQ.

Digital Signal Processor (DSP) Integrated Circuit (IC) for signal processing, simpler than a general-purpose processor but highly optimized for specific applications.

Digital to Analog Converter (DAC) Configurable Voltage generator proportional to a digital number. Required to drive an actor.

Direct Memory Access (DMA) A functional unit able to copy memory regions independently from the CPU. Typically used for devices like communication adapters and storage.

Distributed system A group of systems not sharing memory and clock.

Distribution Collection of Linux kernel and system software to provide a functional operating system.

Earliest Deadline First (EDF) Strategy for deadline scheduling: Always selects the task with the nearest deadline.

Event-triggered Paradigm of real-time programming: All tasks are activated on events (interrupts).

Exception Interrupt triggered by the processor on errors during the execution of instructions.

Exchange (XCHG) Atomic operation: Exchange register with variable.

False sharing If two variables are located in the same cache line and modified by different processors, the whole cache line bounces between those CPUs which reduces the throughput drastically although both work on different variables.

Fault Exception that allows to continue after handling the error condition, e.g. swapping a displaced page back in (page fault).

Glossary

Fetch-and-Add (FAD) Atomic operation: Add a given value to a variable and return its previous value.

Field bus Short-distance bus network optimized for time-critical applications used to connect PLCs in a factory.

Field level Lowest level of the automation pyramid directly interacting with physical systems (plants).

Field-Programmable Gate Array (FPGA) An Integrated Circuit reconfigurable by software to create arbitrary logic for special purposes.

First Level Cache (L1\$) First level of the cache hierarchy, typically private for each processor.

First-In, First-Out (FIFO) A buffer strategy where the sequence of the objects is remained, e. g. a message queue.

Flag Binary setting, usually a bit in a bit mask.

Flags register Register holding bit flags like the Interrupt Flag. The flags are set by executing instructions (e. g. comparison, arithmetic) or dedicated instructions and used by conditional branch instructions.

Formal verification Proof of a real-time program by analytical and mathematic methods.

General-Purpose I/O (GPIO) Digital I/O interface that can be configured to be input or output.

General-Purpose Operating System (GPOS) An operating system for usual desktop and laptop usage like Windows, Mac OS and Linux.

Gibi-Byte (GiB) Binary prefix for 2^{30} , circa one billion bytes. $1 \text{ GiB} = 1024 \text{ MiB} = 2^{30} \text{ B} = 1\,073\,741\,824 \text{ B}$.

Gigabyte (GB) One billion bytes, $1 \text{ GB} = 1\,000 \text{ MB} = 1\,000\,000\,000 \text{ B}$.

Gigahertz (GHz) One billion Hertz, $1 \text{ GHz} = 1\,000 \text{ MHz} = 1\,000\,000\,000 \text{ Hz}$. The period is 1 ns

Graphical User Interface (GUI) An interactive program displaying its state and requesting user input.

Hard real-time Variant of real-time where all deadlines must strictly be met. This usually requires a formal verification.

- Hardware Abstraction Layer (HAL)** Software layer that offers a unified interface to a variety of devices.
- Hardware-in-the-Loop (HiL)** Simulation of a plant to test a closed-loop feedback controller. Can be used for development and debugging in case the physical plant is not available or during successive integration.
- Hertz (Hz)** Per second.
- High Performance Computing (HPC)** Throughput-oriented large-scale computing.
- High-Precision Event Timer (HPET)** Functional unit of x86 systems supporting a higher timer interrupt precision.
- Human-Machine Interface (HMI)** Automation component of the Company level, displays the state of the factory and allows to configure general production parameters.
- Hybrid paradigm** Paradigm of real-time programming: Mix of time-triggered and event-triggered.
- HyperTransport (HT)** System interconnect based on point-to-point links used by current AMD processors.
- Hypervisor** Manager of virtualized guest operating systems that have the impression of running directly on their own systems. Allows to execute multiple instances of operating systems that are not capable or aware of sharing a real machine.
- I/O Controller Hub (IOH)** Intel's naming scheme for the chip set part that handles PCIe connections after the Memory Controller was moved to the processor. Former Memory Controller Hub.
- I/O port** Control registers of the x86 architecture accessed with `IN` and `OUT` instructions.
- Input/Output (I/O)** Input and output of data from/to devices or external signals.
- Input/Output Advanced Programmable Interrupt Controller (I/O APIC)** Part of the chip set that receives electrical signals and forwards them as Interrupt Requests to the CPUs.
- Instruction Pointer (IP)** Register holding the address of the currently executing instruction.

Glossary

Instruction-Level Parallelism (ILP) Optimization of single processors to execute multiple instructions out-of-order and/or in parallel. The program must not be adapted to benefit.

Integrated Circuit (IC) A silicon chip with electronic components such as transistors constructed to serve a special purpose. Examples range from simple logic operators and signal amplifiers up to CPUs.

Inter-IC Communication (I²C) A two-wire bus to connect Integrated Circuits.

Inter-Process Communication (IPC) The synchronization and communication of multiple processes.

Inter-Processor Interrupt (IPI) An interrupt, that is issued from one processor for another.

Interconnect Communication structure between processors and devices (chip set, memory) in a computer system.

Interconnect Controller (ICC) Functional unit that manages the physical layer of the interconnect access.

Interconnect Controller Hub (ICH) Intel's naming scheme for the chip set providing legacy I/O devices such as PCI, USB, and Network Interface Card. Former south bridge.

Interrupt Interrupts are functions that can be triggered by external events (hardware or other execution context) and that intercept the currently executing code, store its context before running and restore and continue it afterwards.

Interrupt Descriptor Table (IDT) Table of the interrupt handlers on the x86 architecture.

Interrupt Flag (IF) Bit in the x86 Flags register to mask (block) hardware interrupts.

Interrupt Request (IRQ) A hardware interrupt that requests handling by the operating system.

Intrinsic function Build-in function that is implemented by the compiler and not linked by a library.

Isolation Protecting a process or CPU from effects caused by others.

Jitter Measure for the variability of latencies, difference between minimum and maximum latency.

- Job** Smallest entity of work, a task consists of one or multiple jobs.
- Jump** Low-level instruction to unconditionally continue the execution elsewhere.
See also: branch.
- Kernel** Central program of an operating system that supervises all processes and manages the hardware.
- Kernel module** Dynamically loadable module that allows to add drivers or subsystems to the Linux kernel. Modules can use an Application Programming Interface (API) that is more stable than integrating code directly in the Kernel but they must restrict themselves to the offered interfaces.
- Kernel thread** Thread executing in kernel-mode for internal services and maintenance of the Linux kernel.
- Kernel-mode** Code executing in the context of the kernel with raised privilege.
- Kernel-space** Memory range reserved for use by the kernel.
- Kibi-Byte (KiB)** Binary prefix for 2^{10} , circa one thousand bytes. $1 \text{ KiB} = 2^{10} \text{ B} = 1\,024 \text{ B}$.
- Kilobyte (kB)** One thousand bytes, $1 \text{ kB} = 1\,000 \text{ B}$.
- Kilohertz (kHz)** Thousand Hertz (per second). The period is 1 ms
- Last-In, First-Out (LIFO)** A buffer strategy where the last (topmost) object is removed first, e.g. a stack.
- Last-Level Cache (LLC)** Last level of the cache hierarchy, typically shared between all cores of a package. This term can be used independently from the number of levels.
- Latency** Measure of the timespan between trigger and action. In network transfers, constant time offset for sending small amounts of data.
- Linux kernel** Core (kernel) of the Linux operating system.
- Load-Linked/Store-Conditional (LL/SC)** Pair of instructions for an atomically executed section. Alternative to Atomic instructions if their implementation is too complex on the hardware architecture.
- Local Advanced Programmable Interrupt Controller (localAPIC)** Part of the processor that receives interrupt messages from the I/O APIC and controls their handling in the CPU.

Glossary

Local Area Network (LAN) Network to connect local systems in a building.

Local memory Memory directly connected to a node in a shared memory system.

Lock-free Property of an algorithm that does not require locks but possibly fails and must retry. See also wait-free.

Main board Electronic circuit board hosting the processor(s), the chip set and connectors for peripheral devices.

Mainline Kernel The standard Linux kernel as published on www.kernel.org. Also referred to as 'Vanilla Kernel'. Distributions usually provide extended or specially configured Linux kernels.

Manufacturing Execution Systems (MES) Automation component of the Company level.

Many-core Term to describe future multi-core processors with many more cores than today. The increasing processor number poses a scalability challenge.

Mebi-Byte (MiB) Binary prefix for 2^{20} , circa one million bytes. $1 \text{ MiB} = 1024 \text{ KiB} = 2^{20} \text{ B} = 1\,048\,576 \text{ B}$.

Megabyte (MB) One million bytes, $1 \text{ MB} = 1\,000 \text{ kB} = 1\,000\,000 \text{ B}$.

Megahertz (MHz) One million Hertz, $1 \text{ MHz} = 1\,000 \text{ kHz} = 1\,000\,000 \text{ Hz}$. The period is $1 \mu\text{s}$

Memory consistency Definition or contract how the accesses of different processors to the memory are ordered and detected by others.

Memory Controller (MC) Functional unit that translates bus requests into memory access commands.

Memory Controller Hub (MCH) Intel's naming scheme for the chip set part that contains the Memory Controller and PCIe connections. Former north bridge.

Memory Management Unit (MMU) Translates virtual into physical addresses.

Memory-mapped I/O Control registers embedded in the memory address space accessed with common load/store instructions.

Message queue Data structure and synchronization object for a buffered transfer of asynchronous messages in FIFO order to a recipient (Message-Passing).

- Message-Passing (MP)** Programming paradigm for High Performance Computing where multiple processes are executed on separated systems without access to other system's main memory. The communication and synchronization is done by explicitly exchanging messages via high-performance interconnects.
- Micro-architecture** Subclass of the same architecture to distinguish generations, optimizations, and features.
- Micro-Controller (μC)** Integrated circuit containing a processor, memory, and I/O devices. It allows to implement digital controllers with few external components.
- Micro-kernel** Minimal operating system kernel with the highest privilege, other functions are realized as processes/servers.
- Microprocessor without Interlocked Pipeline Stages (MIPS)** Processor architecture with a RISC instruction set.
- Microsecond (μs)** One-millionth of a second, $1\mu\text{s} = 10^{-3} \text{ ms} = 10^{-6} \text{ s}$.
- Millisecond (ms)** One-thousandth of a second, $1 \text{ ms} = 10^{-3} \text{ s}$.
- Monitor** Synchronization object that allows access to shared resources only by functions and implicitly protects critical sections.
- Multi-core** In the scope of this work: Emphasize multiple processors in a single package. If multi-core processors are combined in a Non-Uniform Memory-Access system, each multi-core becomes a NUMA-node.
- Multi-processor** A computing system with multiple CPUs. In the scope of this work, this term is used independently from the processors being built in a single package or installed in multiple sockets.
- Multiple Instructions Multiple Data (MIMD)** Flynn's classification for computer structures: Multiple autonomous processors execute independently a program processing different data. This includes multi-processor shared-memory systems and clusters (distributed systems).
- Multiple Programs Multiple Data (MPMD)** Programming paradigm for Multiple Instructions Multiple Data (MIMD) systems: Multiple programs or different functions execute different jobs (functional parallelism). On shared-memory systems, this can be realized with threads or multiple processes, on distributed systems, remote procedure calls can be used.
- Mutex** Mutual Exclusion. A binary Semaphore used to allow only one party to enter a critical section.

Glossary

Mutual exclusion The concept of allowing access to a shared resource if no others use it concurrently. A typical implementation is the Mutex.

Nanosecond (ns) One-billionth of a second, $1 \text{ ns} = 10^{-3} \mu\text{s} = 10^{-9} \text{ s}$.

Network Interface Card (NIC) Peripheral device to connect a system to a network (e. g. Ethernet).

No Remote Memory-Access (NoRMA) A cluster architecture where nodes can not directly access other node's memory.

Node Part of a computer system, consisting of one or multiple processors and memory. The nodes can be part of a shared memory system (UMA or NUMA) or be connected to a distributed system.

Non-Maskable Interrupt (NMI) An interrupt that can not be masked (blocked) by the Interrupt Flag. Can be triggered by hardware (Interrupt Request) or another processor (non-maskable Inter-Processor Interrupt).

Non-Uniform Memory-Access (NUMA) A system architecture with memory access timing depending on the location of the physical memory page.

North bridge Classic part of the chip set containing the Memory Controller.

Open source Licence model that provides a software with source code and allows to modify and redistribute it.

Operating system (OS) The system software controlling the hardware and servicing programs. The operating system commonly includes a kernel and system software.

OS noise Deviation in the execution time due interrupts and preemptively scheduled tasks that influence the caches and other functional units.

Out-of-order Property of Instruction-Level Parallelism: a complex pipeline can reorder the instruction stream to better utilize multiple execution units (Arithmetic Logical Unit) if the operations are independent (e. g. not use the result of the previous instruction as input).

Page fault Fault (exception) when accessing an invalid virtual address. The handler has the chance to fix the problem by swapping a displaced page back in or by providing more space.

Page table Data structure used by the Memory Management Unit to resolve a virtual to a physical address. Managed by the operating system to protect the address space of processes.

- Para-virtualization** Type of virtualization where the guest knows about its hypervisor and restricts itself to using the offered interfaces instead of directly manipulating the hardware.
- Partition** Separation of a system into multiple independent parts.
- Parts per million** (ppm) Parts per million.
- PCI Express** (PCIe) Successor of PCI with higher speed.
- Performance Measurement Counter** (PMC) Special function registers of processors to count specified events.
- Peripheral Component Interconnect** (PCI) High-speed bus to connect peripheral devices.
- Personal Computer** (PC) Computer system platform originally presented by IBM and rebuilt by many others. Typically using an x86 processor and in the form of a stand-alone desktop computer or notebook.
- Physical address** Real address in the main memory.
- Plant** Physical system, e. g. a machine or factory, supervised by a controller.
- Platform Interrupt** (PI) Interrupt triggered by the hardware platform and handled by the firmware without interaction or influence of the OS.
- Point-to-point** Type of interconnect where devices have multiple links to other devices and routers may connect more complex structures.
- Portable** A software is considered portable, if it can be transferred easily to other OSs or hardware.
- PowerPC** Performance Optimization With Enhanced RISC – Performance Computing. Processor architecture with a Reduced Instruction-Set Computer (RISC) instruction set.
- Preemption** Capability of an operating system to interrupt a task at any time to activate another one. The *preemptibility* is a measure of how quickly and predictably the preemption can be done. If the preemption can be blocked for certain times, the preemptibility is lowered.
- Priority inversion** A lower-priority task holds a Mutex and blocks a higher-priority task that waits for the same resource.
- Private cache** Cache that is only used by a single processor.

Glossary

Process Execution unit of an operating system with private address-space protected from others.

Processor In the context of this work a single execution unit that runs independently from others, also Central Processing Unit (CPU). A silicon chip (IC) or package may contain multiple processors.

Programmable Logic Controller (PLC) Automation component of the Field level, typically embedded systems for closed-loop control.

QuickPath Interconnect (QPI) System interconnect based on point-to-point links used by current Intel processors.

Read-Copy-Update (RCU) A synchronization mechanism that elides locks to minimize cache misses.

Real-time Computing programs that not only depend on the correct result, but also on it being available in time.

Real-Time Operating System (RTOS) A special class of operating system for real-time systems.

Real-Time Preemption Patch (RT-Patch) Modification of the mainline Linux kernel to improve its predictability and make it more suitable for soft real-time applications.

Reduced Instruction-Set Computer (RISC) Processor architectures with a small, concise instruction set. This requires more instructions to implement algorithms but simplifies the Control Unit and Arithmetic Logical Unit accelerating the execution of a single instruction (compared to CISC).

Remote Direct-Memory Access (RDMA) Use of a DMA engine to transfer data to an RMA buffer of a different system.

Remote memory Memory connected to a different node in a shared memory system. Can be accessed with the same instructions but is slower (increased latency and lower throughput) than local memory.

Remote Memory Access (RMA) Memory Segments of a different system mapped to the local address space can be accessed with load and store instructions like local memory. The transfer is transparently handled by special hardware. Different access times result in a Non-Uniform Memory-Access system.

Revolutions per Minute (RPM) Frequency commonly used for combustion engines; 60 RPM = 1 Hz.

Ring buffer Data structure of equally sized elements used in FIFO order reusing the first element after the last one.

Scalable Processor Architecture (SPARC) Processor architecture for workstations and high-performance servers with a RISC instruction set.

Scheduler Module of an operating system that selects which task to execute. A dynamic scheduler does so during runtime otherwise a static schedule can be pre-defined.

Script Program listing for interpreter execution, usually executed by a shell.

Second Level Cache (L2\$) Second level of the cache hierarchy. If more than two levels are available, then it is typically private for each processor.

Secure Shell (SSH) Provides remote terminal access to a UNIX system using an encrypted network connection.

Semaphore Synchronization Primitive. Applicants must wait until a resource becomes available.

Sensor Measures the state (e. g. temperature, voltage, speed) of a plant, i. e. converts physical quantities into numerical values. Output of a physical system, input of a controller.

Shared cache Cache that is shared by multiple CPUs of a multi-core processor.

Shared memory Region of memory that can be accessed by multiple processes that otherwise can not access each other's memory. Realized by page table entries: The initialization must be done by the operating system but the access is managed by hardware.

Shared-Memory (SHM) Programming paradigm for High Performance Computing where multiple threads are executed on the same system sharing their data. Concurrent access must be protected from data races.

Shell Interactive text terminal to access a UNIX system.

Sign Flag (SF) Bit in the x86 Flags register, equals the sign of the result of an arithmetic operation or a comparison to be used for above/below tests. Usually used by a subsequent branch instruction.

Simultaneous Multi-Threading (SMT) A physical CPU is divided in multiple logical CPUs executing interlaced. By better utilizing internal functional units, nearly parallel execution is possible.

Single Instruction Multiple Data (SIMD) Flynn's classification for computer structures: A processor executing an instruction that processes multiple data elements, e. g. from a vector.

Single Instruction Multiple Threads (SIMT) Used on highly parallel processors such as Graphic Processing Units (GPU).

Single Instruction Single Data (SISD) Flynn's classification for computer structures: A single sequential processor executes code processing a single data stream without parallelism. This is the classical single-processor program.

Single Program Multiple Data (SPMD) Programming paradigm for Multiple Instructions Multiple Data (MIMD) systems: A single program executes on different parts of the data (data parallelism). On shared-memory systems, this can be realized with threads or Open MP, on distributed systems (clusters), a Message-Passing (MP) library like MPI is used.

Slab The Slab is a Linux kernel data structure to manage small memory allocations.

Soft real-time Variant of real-time where most deadlines should be met. Usually, a percentage of met deadlines is given or the degradation function of the result usefulness is defined.

Software interrupt Synchronous interrupt triggered by the running code using an assembly instruction.

South bridge Classic part of the chip set containing I/O devices.

Spinlock A function actively waiting for an event by periodically checking with high frequency.

Stack Last-In, First-Out (LIFO) based data structure, used in a memory region to hold temporary data.

Stack Pointer (SP) Register holding the current top of the stack.

Sub-kernel Special form of an Asymmetric Multi-Processing system with a Hardware Abstraction Layer being an RTOS at the same time and executing a GPOS as task with the lowest priority (idle task).

Super-scalar Property of Instruction-Level Parallelism: multiple execution units (Arithmetic Logical Units) can execute multiple operations in the same cycle.

Supervisory Control and Data Acquisition (SCADA) Automation component of the Cell level that controls multiple PLCs and can also integrate sensors and actors.

- Symmetric Multi-Processing (SMP)** A single OS instance manages all CPUs.
- System call** Function of an operating system with well-defined API to be used by user-mode processes.
- System Management Interrupt (SMI)** An interrupt installed and used by the BIOS which can not be controlled by the operating system (a Platform Interrupt). It is executed in System Management Mode.
- System Management Mode (SMM)** A special operating mode of x86 processors. Only activated for System Management Interrupts and not controllable by the operating system.
- System-call** Well-defined interface for user-mode processes to request services from the operating system.
- System-on-a-Chip (SoC)** Similar to a μC , but with emphasis on a whole computer system integrated on a single Integrated Circuit.
- Task** Execution unit of an operating system where the differentiation of processes and threads does not matter.
- Task switch** Saving the context of a process and restoring another, suspended one.
- Task ID (TID)** A unique identifier for each process and thread.
- Test-and-Set (TAS)** Atomic operation: Write 1 to the given variable and return its previous value. This very basic atomic operation allows a simple implementation of a Mutex.
- Third Level Cache (L3\$)** Third level of the cache hierarchy, typically shared between all cores of a package.
- Thread** Execution unit of an operating system, multiple threads in a process share the same address-space.
- Throughput** Measure for the amount of data per time transferred in a connection. In network transfers, this often depends on the data size and saturates for large data packages.
- Time slice** Small portion of time (typically around 10 ms) assigned to a task for execution before the scheduler switches to another one.
- Time-Division Multiple Access (TDMA)** Reservation of recurrent time-slices for different users, temporal partitioning.

Time-triggered Paradigm of real-time programming: All tasks are activated repeatedly in a loop.

Timer Interrupt (TI) Interrupt triggered by a timer.

Timing anomaly A shorter code requires more execution time (or vice-versa) because of hardware effects on the micro-code level.

Translation Look-aside Buffer (TLB) A cache for mapping logical page addresses to physical page frames that reduces page table walks in the Memory Management Unit.

Uniform Memory-Access (UMA) An architecture with memory access timing similar for all processors.

Universal Serial Bus (USB) A plug and play interface to connect peripheral devices like keyboard, mouse, printer, etc.

User-mode Code executing in the context of a process with reduced privilege. Can access kernel-space only via defined interfaces (system-calls).

User-space Memory range accessible for a process. The kernel can access user-space memory.

Virtual address Logical address in a linear space that must be converted to a physical address for the memory access.

Virtual file Configuration access through a file that can be read and written. Instead of a real file access, an interface function is executed.

Virtualization Simulation of an entire system for a client instance to execute multiple (operating) systems on the same hardware.

Wait-free Property of an algorithm that does not require locks and always proceeds, i. e. always terminates in a limited time. See also lock-free.

Wall clock The operating system's notion of the current time as displayed on a wall clock. Since no hardware unit provides the current time, the operating system uses timer interrupts to keep track of the time passed by.

Wide Area Network (WAN) Network to connect remote sites.

Worst-Case Execution Time (WCET) The maximum time, a function or basic block can possibly require to execute.

Zero Flag (ZF) Bit in the x86 Flags register indicating the result of an arithmetic operation or a comparison was zero. Usually used by a subsequent branch instruction.

List of Figures

1.1. Automation pyramid (simplified) [Sau07]	3
1.2. Layered diagram of the proposed system architecture	9
2.1. Ring structure of privilege levels. The levels 0–3 are exemplary for the x86 architecture, the levels –1 and –2 are optional.	16
2.2. Address space of a process and mapping to physical memory.	17
2.3. Cache levels of Intel P4 processors.	18
2.4. Cache associativity: 2-way.	19
2.5. Classical PC Input/Output (I/O) architecture.	24
2.6. Comparison of simplified Uniform Memory-Access (UMA) and Non-Uniform Memory-Access (NUMA) architectures.	35
2.7. Multi-processor architecture with NUMA characteristic.	35
2.8. Examples for coherent cache lines.	37
2.9. Processor	38
2.10. SMP Operating system on multi-processor system.	49
2.11. Hardware-in-the-Loop Simulation.	56
2.12. Control flow diagram and example code of time-triggered paradigm.	57
2.13. Control flow diagram and example code of event-triggered paradigm.	58
2.14. Control flow diagram of hybrid (time- and event-triggered) paradigm.	59
2.15. Control flow diagram of scheduling.	59
2.16. Real-time Asymmetric Multi-Processing (AMP) system based on a hypervisor.	62
2.17. Operating system based on a sub-kernel.	63
2.18. Hourglass histogram.	69
2.19. Scatter plot (threshold 1000).	70
2.20. Scatter plot (threshold 10 000).	70
4.1. Concept of a strictly isolated task P_5 on a dedicated processor.	88
4.2. Example setup for multiple partitions.	93
4.3. Application Example	97
4.4. First 2000 gaps (above 50 cycles) on idle system, fair scheduling for benchmark.	105
4.5. First 2000 gaps (above 50 cycles) on loaded system, fair scheduling for benchmark.	105

List of Figures

4.6.	First 2000 gaps (above 50 cycles) on loaded system, benchmark executed in its own partition .	106
4.7.	First 2000 gaps (above 50 cycles) on loaded system, benchmark executed fully isolated .	106
4.8.	Histograms on loaded system.	107
5.1.	Example trend of various <code>/proc/meminfo</code> values during six isolations of two Hours each (System: haixagon , Appendix A.3.1 on page 208).	114
5.2.	Inter-Processor Interrupt triggering functions call graph of Linux 3.12.	118
6.1.	Example application: Tasks and interaction	129
6.2.	Simple flag operation.	132
6.3.	Flag used for a handshake operation.	133
6.4.	Control flow for user-mode interrupts.	151
7.1.	Multi-core architecture (Intel Nehalem) with four cores and two logical threads each.	159
7.2.	Example: cache content of a multi-core processor (simplified).	161
7.3.	Comparison of inclusive and exclusive caches. Impact of the load working set size to a small isolated task (max. latency).	162
7.4.	UMA system with two sockets and four cores each (Test system octopus , Appendix A.1.2).	163
7.5.	Maximum latency depending on isolation buffer size with varying load working set sizes (4 KiB to 16 MiB).	164
7.6.	Separation of system and real-time partitions on dedicated NUMA nodes.	165
7.7.	Maximum latency of an isolated task using various memory sizes under varying load scenarios.	166
7.8.	Jitter experienced by the isolated task depending on its memory range for three different load sizes (that do not influence the result significantly).	167
7.9.	Histograms with largest load (512 MiB read/write buffer size).	168
7.10.	Number of extreme delays above 1.2 μ s depending on isolation size.	169
7.11.	Symmetric Multi-Threading example on two-socket Intel Westmere (test system haixagon , Appendix A.3.1) with physical processor numbers (P_i) and logical thread numbers (T_j).	170
7.12.	Peripheral I/O capabilities of an UMA system with two sockets (Test system octopus , Server board Intel S5000PSL, Appendix A.1.2).	173
7.13.	Peripheral I/O capabilities of a NUMA system with two sockets (Simplified test system haixagon , Server board Tyan S7025, Appendix A.3.1).	174
7.14.	Distribution of latencies of a Peripheral Component Interconnect (PCI) device connected via the south bridge under various loads.	176

7.15. Distribution of latencies of different PCI and PCI Express (PCIe) devices under maximum load.	178
7.16. Distribution of execution times of an application benchmark task with 11 I/O accesses and 2000 Byte data exchange after an execution of 5 days (120 h).	179
8.1. Layer diagram of example application.	183
8.2. Timing example: (1) User interaction to start. (2) Thread creation and isolation. (3) Activation of experiment. (4) User interaction to terminate. (5) Notification to exit. (6) Synchronized termination of isolations.	186
8.3. Example application I/O partitioning.	187
8.4. Example Application-Specific Integrated Circuit (ASIC) attributed to a Tensilica presentation [Sch13, Fig. 3.10].	190
A.1. Core2 double- and quad-core cache and memory structure.	205
A.2. Nehalem cache and memory structure.	207
A.3. Westmere NUMA cache and memory structure.	208
A.4. Block diagram of haixagon.	209
A.5. Block diagram of pandora.	210
A.6. Block diagram of devon.	211

List of Tables

2.1. List of micro-architectures from Intel [Intel13d] and AMD [AMD05], Pipeline information estimated by Fog [Fog13d]	25
2.2. Latency (in Central Processing Unit (CPU) cycles) of the RDTSC and serializing instructions (where available).	28
2.3. Memory access latency in CPU cycles.	29
2.4. Latency of selected x86 instructions (in CPU cycles) [Fog13e].	31
2.5. Latency of function and system calls (in CPU cycles).	32
2.6. Hourglass overhead.	67
4.1. First evaluation on Intel Core i7 920 (in CPU cycles).	104
4.2. Maximum gap on the <i>smp.boot</i> Kernel with different cache configurations for varying load ranges (in CPU cycles).	108
5.1. Data sources for logging the system state during isolation	113
5.2. Functions modified to block sending Inter-Processor Interrupts (IPIs) in Linux 3.12.	119
5.3. CPU notification events	121
6.1. Management states as implemented in the benchmark application.	141
6.2. Common hardware extension interfaces of the x86 architecture.	146
6.3. Timings of user-mode interrupts on Intel Core 2 with nearly empty handler function.	152
7.1. Execution time jitter (in CPU cycles) of a small task using 4 KiB buffer (code and data in L1\$) on an idle system after warm-up.	158
7.2. Influence of load using the shared cache on the inclusive private cache of another core (latencies in CPU cycles).	159
7.3. I/O latencies of different PCI and PCIe devices (Systems described in Appendix A).	172
7.4. Statistic of latencies (in μ s) accessing a PCI-device connected via the south bridge under various loads.	175
A.1. Test system overview.	205
A.2. Test system <i>xaxis</i>	206
A.3. Test system <i>octopus</i>	206
A.4. Test system <i>xaxis</i>	207

List of Tables

A.5. Test system haixagon	208
A.6. Test system pandora	210
A.7. Test system devon	211

List of Listings

- 2.1. Hourglass benchmark routine (simplified). 68
- 2.2. Minimal implementation of timestamp recording function. 69

- 4.1. Task Example using the time-triggered paradigm. 100

- 6.1. Semaphore implementation using pseudo atomic functions. 135
- 6.2. Binary Semaphore (Mutex) implementation with GCC intrinsic functions. 136
- 6.3. Simplified wait-free message queue implementation [Lam77]. 137
- 6.4. Implementation and use example of a user-mode interrupt handler. . 150

Index

- C++ programming language, 48
- A/D converter, 129, 145
- Actor, 3, 145
- Ada, 48
- Advanced Programmable Interrupt Controller (APIC), 23
- Advanced Vector Extensions (AVX), 33, 48
- Affinity, 51
 - Interrupt, 94
 - Process, 94
- Affinity on First Touch, 49, 186
- AMD, 5, 13, 21
- Amide, 191
- Android, 2, 7
- Application, 54, 96
- Application benchmark, 26, 39
- Application processor, 89
- Application Programming Interface (API), 48, 145
- Application-specific integrated circuit, 189
- Arithmetic Logical Unit (ALU), 14
- ARM, 1, 5, 13, 22, 38, 66, 72, 139, 201
 - big.LITTLE, 201
- ARTiS, 79
- ASMP-Linux, 80
- Assembler, 42
- Assembly language, 47, 48
- Associativity (cache), 18, 160
- Asymmetric Multi-Processing (AMP), 48, 62
- Asynchronous
 - communication, 46
 - control transfer, 89
 - Device interrupt, 100
 - I/O, 95
 - Inter-processor interrupt, 100
 - Interrupt, 101
 - interrupt, 20, 102
 - Platform interrupt, 101
 - Timer interrupt, 100
- Atmel AVR, 65
- Atomic
 - x86, 38
- Atomic operation, 38, 98
- Automation pyramid, 3, 54, 127
- Availability, 53
- Bare-metal
 - execution, 4, 57, 108, 190
 - task, 9, 87, 111
- Bare-metal execution, 63
- Bare-metal task, 10, 102
- Barrier (synchronization object), 133
- Basic block, 65, 156
- Benchmark, 26
- Best-case execution time, 65, 156
- Big Kernel Lock, 75
- BIOS (Basic Input/Output System), 89
- Blocked (task state), 44
- Blocked waiting, 44
- Bonnie++, 30
- Boot manager, 89, 90

Index

- Boot parameter, 92
- Bootstrap processor, 89
- Bottle-neck (performance), 26
- Bounded multi-processing, 100
- Bounded Multi-Processing (BMP), 49
- Branch, 102
- Branch prediction, 66
- BSD sockets, 73
- Build-time configuration, 92
- Busy waiting, 44, 98, 133, 134
- C programming language, 48
 - C11 standard, 138
- Cache, 17
 - Associativity, 160
- Cache coherence, 37, 130
- Cache line, 17, 159
- Cache miss, 28
- Calibrator (benchmark), 30
- Calling convention, 30
- Capabilities (Linux feature), 94
- Capability, 101
- Cell level, 3, 127
- Centronics printer interface, 146
- Chip, 34
- Chip set, 21
- CISC architecture, 14
- Cluster, 33, 63
- Co-processor, 41
- Code section, 16, 43
- Coherence, 130
- Comedi, 146
- Company level, 3, 127
- Compiler, 42, 47
- Compiler optimization, 47
- Compute Unit, 33, 90
- Concurrency Kit, 132
- Condition variable, 132
- Context, 43
- Context switch, 18, 22, 30, 38, 148, 151, 153
- Control theory, 55
- Control Unit (CU), 14
- Cooperative multi-processing, 58
- Cooperative scheduling, 44, 102
- CPU, 34
- CPU Affinity, 51
- CPU affinity, 94
- CPU-Set (Linux feature), 51, 94, 184
- Critical section, 44, 134
- Criticality, 4, 42, 53
- Cron daemon, 114
- Current Privilege Level (CPL), 15
- Cycle-stealing interrupt, *see* Platform interrupt
- Cyrix, 21
- D/A converter, 129, 145
- Daemon, 42
- Data acquisition, 145, 146
- Data parallelism, 33
- Data section, 43
- Data streaming, 55
- Deadline scheduling, 60, 74, 76
- Deadlock, 45, 134, 184
 - Avoidance, 45, 135
 - Detection, 45, 135
 - Prevention, 45, 135
- Deadlock avoidance, 7
- Deadlock prevention, 7
- Delaying event, 159, 162, 165
- Demand paging, 16, 19, 47, 98
- Descriptor Privilege Level (DPL), 15
- Device driver, 47
- Device interrupt, 19
- Distributed system, 5, 33, 63, 72, 200
- Distribution, 50, 87
- Distribution (Linux), 50
- `dmesg`, 113
- Dynamic scheduling, 60
- Dynticks (Linux feature), 76
- Embedded system, 40
- Emergency shutdown, 142

- Epoch, [132](#)
- Event-triggered paradigm, [58](#)
- Exception, [102](#)
- Exclusive cache, [18](#), [160](#)
- Execution context, [43](#)
- Execution time, [102](#), [103](#), [128](#), [156](#), [157](#)
 - Best-case, [65](#), [156](#)
 - Worst-case, [53](#), [65](#), [156](#)
- Execution time estimation, [74](#)
- Expansion card, [21](#), [25](#), [129](#), [147](#), [174](#), [182](#)

- False sharing, [37](#), [99](#), [131](#), [141](#)
- Field bus, [4](#), [129](#)
- Field level, [3](#), [127](#)
- FIFO priority-based scheduling, [60](#)
- File system, [47](#)
- Finished (task state), [44](#)
- Firm real-time, [75](#)
- Firmware, [42](#)
- First touch
 - affinity, [49](#), [186](#)
 - Memory allocation, [98](#)
- Fixed schedule, [60](#)
- Flag, [132](#)
- Flags register, [14](#)
- Formal verification, [52](#), [53](#), [65](#), [74](#), [77](#), [79](#), [88](#), [99](#), [103](#), [148](#), [155](#), [156](#)
 - Event-triggered, [58](#)
 - Time-triggered, [57](#)
- FreeRTOS, [61](#), [84](#), [153](#)
- Freescale, [72](#)
- Function call, [102](#)
- Functional parallelism, [33](#)

- General-purpose I/O (GPIO), [129](#), [145](#)
- General-Purpose Operating System (GPOS), [42](#)
- Gnu Compiler Collection (GCC), [36](#), [135](#), [138](#), [139](#)
- Graceful shutdown, [142](#)
- Graphical User Interface (GUI), [127](#)

- Handshake, [133](#)
- Hard real-time, [52](#)
- Hardware abstraction, [46](#)
- Hardware Abstraction Layer, [8](#)
- Hardware effects, [6](#), [65](#)
- Hardware jitter, [6](#)
- Hardware-in-the-Loop (HiL) Simulation, [55](#)
- Hawk kernel, [72](#)
- Hazard pointer, [140](#)
- Heap, [43](#)
- Heart beat, [143](#)
- Heterogeneous system, [63](#)
- High-Precision Event Timer (HPET), [51](#), [93](#)
- Homogeneous system, [63](#)
- Horizontal integration, [4](#)
- Hotplugging (Linux feature), [50](#), [93](#)
- Hourglass benchmark, [67](#), [69](#), [92](#), [127](#)
- Huge page, [24](#)
- Huge pages, [99](#)
- Human-Machine Interface, [4](#)
- Hyades, [79](#)
- Hybrid paradigm, [58](#)
- Hyper-Threading, [33](#), [90](#), [169](#)
- Hypervisor, [8](#), [47](#), [48](#), [62](#), [77](#), [82](#)

- I/O APIC, [23](#)
- I/O controller, [25](#)
- I/O Memory Management Unit (IOMMU), [90](#), [200](#)
- I/O port, [20](#)
- Inclusive cache, [18](#), [160](#)
- Input-Process-Output (IPO) model, [57](#)
- Input/Output, [145](#)
- Instruction, [14](#)
- Instruction Pointer (IP), [14](#)
- Instruction set, [14](#)
- Intel, [5](#), [13](#), [21](#)
- Inter-processor interrupt, [19](#)
- Interrupt, [102](#)
- Interrupt abstraction, [74](#), [77](#)

Index

- Interrupt affinity, 94
- Interrupt Flag (IF), 20
- Interrupt Request, 19
- Intrinsic function, 36, 135, 138, 139
- IOzone, 30
- iperf, 30
- Isolated partition, 93
- Isolation, 71, 80, 87

- Jailhouse, 82, 83, 179
- Java, 48
- Jitter, 6, 27, 156, 157, 163
- Job, 52, 54
- Jump, 102

- Kernel, 42
- Kernel log, 113
- Kernel mode, 22
- Kernel modification, 74
- Kernel preemption, 74
- Kernel thread (Linux feature), 50, 95
- Kernel-mode, 46
- Kernel-space memory, 46

- L4 (micro-kernel), 79
- Labview, 4, 143
- Last-Level Cache (LLC), 17
- Latency, 27, 157
- latency, 55
- Legacy I/O, 174
- Library, 42
- Likwid, 51
- Linker, 42
- Linux, 50
- Linux distribution, 50
- Linux kernel module, 76, 78, 82, 121, 123, 125, 149
- Linux User I/O, 146
- Linux-rt, 7
- Little endian, 22
- Load balancing, 48, 49
- Loader, 42
- Local APIC, 23

- Local memory, 36, 164
- Locality, 16
- Lock-free, 7, 64
- Logical processor, 34, 90
- Low-level benchmark, 26
- lwip, 147

- Main board, 21, 25, 34, 41, 83, 146, 177, 182
- Mainline Linux kernel, 7, 50, 75–77, 81
- Major page fault, 98, 202
- Manager task, 96
- Many-core processor, 33, 50
- Master-worker, 33
- Matlab Simulink, 4, 128, 143, 193
- Membench, 28
- Memory consistency, 36, 130
- Memory consistency, 138
- Memory management, 47
- Memory wall, 26
- Memory-mapped I/O, 20
- MERASA project, 73
- MESI (cache synchronization protocol), 37, 164, 165
- Message Passing, 46
- Message queue, 136
 - Wait-free, 137
- Message-Passing, 130
- Micro-Architecture, 25
- Micro-benchmark, 26
- Micro-controller, 41
- Micro-Kernel, 43
- MicroC/OS-II, 61
- Microsoft Windows, 82
- Minor page fault, 98, 202
- MIPS, 13, 20, 22, 80, 201
- Monitor (Synchronization object), 130, 136
- Monitor (synchronization object), 46
- Monolithic Kernel, 43
- Multi-Core Communications API (MCAPI), 48

- Multi-core processor, [1](#), [32–34](#), [72](#), [158](#)
- Multi-processor, [1](#), [155](#)
- Multi-processor system, [34](#)
- Multi-tasking, [43](#)
- Multiple Instructions Multiple Data (MIMD), [33](#)
- Mutex, [75](#), [134](#)
- Mutual exclusion, [45](#), [134](#)

- NMI Watchdog, [92](#)
- No remote memory access architecture (NoRMA), [34](#)
- no_hz (Linux feature), [76](#)
- Node, [164](#)
- Non-uniform memory architecture (NUMA), [34](#)
- North bridge, [24](#), [173](#)
- Notifier chain (Linux feature), [116](#), [121](#), [124](#)

- OpenMP, [129](#)
- openSuSE, [7](#)
- Operating system, [42](#)
- Optimization, [47](#)
- OS noise, [6](#)
- Out-of-order architecture, [64](#)
- Out-of-order execution, [15](#), [27](#), [30](#), [36](#), [66](#), [103](#), [138](#)

- Package, [34](#)
- Page fault, [16](#), [19](#), [196](#)
 - Major, [98](#), [202](#)
 - Minor, [98](#), [202](#)
- Paging, [15](#), [47](#)
- Para-virtualization, [63](#), [78](#), [79](#)
- Parallax Propeller, [58](#)
- Parallel port, [146](#)
- parMERASA project, [73](#)
- Partitioning, [62](#), [63](#), [73](#), [87](#), [97](#)
- PCI, [20](#)
- PCI Express, [21](#)
- Peripheral device, [129](#), [170](#)
- Physical address, [43](#)
- Physical processor, [34](#), [90](#)
- Pinned kernel thread, [50](#), [95](#), [121](#)
- Pipelining, [33](#)
- Plant, [52](#), [186](#)
- Platform interrupt, [19](#), [101](#)
- Polling, [44](#), [98](#)
- Portability, [4](#), [46](#)
- POSIX, [48](#), [73](#)
 - Real-time extensions, [80](#)
 - Shared memory, [98](#)
 - Threads, [186](#)
- PowerPC, [13](#)
- Pre-faulting, [98](#)
- Pre-fetching (cache), [17](#), [29](#)
- Predictability, [52](#)
- Preemptibility, [74](#)
- Preemptive multi-processing, [60](#)
- Preemptive scheduling, [44](#)
- Printer interface, [146](#)
- Priority
 - Ceiling, [7](#), [64](#)
 - Inheritance, [7](#), [64](#)
- Priority inversion, [7](#), [64](#), [75](#), [133](#), [136](#)
- Priority-based scheduling, [60](#)
 - First-in, first-out, [60](#)
 - Round-robin, [60](#)
- Private cache, [36](#)
- Privilege level, [15](#)
- Process, [43](#)
- Process affinity, [94](#)
- Processor, [34](#)
- Profiling, [39](#)
- Programmable Logic Controller, [3](#)
- Programmable Logic Controller (PLC), [55](#)
- Protection level, [15](#)
- Python, [48](#)

- QNX, [61](#), [192](#)
- Qt, [10](#), [127](#), [129](#), [185](#), [193](#), [198](#)

- Rapid prototyping, [127](#)

Index

- Reactivation, [43](#)
- Read-Copy-Update (RCU), [51](#), [120](#), [121](#), [139](#), [140](#)
- Ready (task state), [43](#)
- Real-time, [42](#)
 - Firm, [75](#)
 - Hard, [52](#)
 - Soft, [52](#)
- Real-Time Ethernet, [147](#)
- Real-time execution, [97](#)
- Real-Time system, [52](#)
- Realfeel, [67](#)
- Red Hat, [7](#)
- Redhawk Linux, [81](#)
- Remaining system, [93](#)
- Remote invocation, [46](#)
- Remote memory, [164](#)
- Return (from function and interrupt handler), [102](#)
- Ring buffer, [137](#)
- RISC architecture, [14](#)
- Roofline model, [26](#)
- Round-robin priority-based scheduling, [60](#)
- RS-232, [146](#)
- RT-Patch, [7](#), [10](#), [67](#), [74](#)
- RTAI Linux, [78–82](#)
- RTEMS, [73](#)
- RTLinux, [77–79](#), [81](#), [82](#)
- Run-time setting, [92](#)
- Running (task state), [43](#)

- Safe state, [53](#), [55](#)
- Sand box, [144](#)
- Scalability, [50](#), [63](#)
- SCC, [156](#), [200](#)
- Schedulability, [60](#), [65](#)
- Scheduling algorithm, [44](#)
 - Deadline, [60](#), [74](#), [76](#)
 - First-in, first-out, [60](#)
 - Priority-based, [60](#)
 - Round-robin, [60](#)

- Segmentation, [47](#)
- Segmentation fault, [145](#)
- Semaphore, [75](#), [134](#)
- Sensor, [3](#), [145](#)
- Serial port, [146](#)
- Setup phase, [97](#)
- Shared cache, [36](#)
- Shared library, [143](#)
- Shared memory, [16](#), [64](#), [67](#), [88](#), [96](#), [98](#), [130](#), [198](#)
- Shared-memory multi-processor system, [33](#), [48](#)

- Shell
 - Script, [112](#)
- Shielded CPU, [8](#), [80](#), [81](#)
- Signal processing, [55](#)
- Simulink, [4](#), [128](#), [143](#), [193](#)
- Simultaneous multi-threading (SMT), [33](#), [90](#), [169](#)
- Single Instruction Multiple Data (SIMD), [33](#)
- Single Instruction Single Data (SISD), [33](#)
- Single Program Multiple Data (SPMD), [33](#), [164](#)
- Slab (Linux feature), [52](#), [113](#), [120](#)
- Sleep, [44](#)
- Slot, [21](#)
- smp.boot* kernel, [108](#)
- Socket, [34](#)
- Soft real-time, [52](#)
- Softlockup Detector, [92](#)
- Software interrupt, [19](#)
- Software tracing, [128](#)
- South bridge, [24](#), [173](#)
- Spacial partitioning, [82](#), [83](#)
- SPARC processor, [73](#)
- Spatial partitioning, [74](#), [87](#)
- Spin-lock, [45](#), [75](#), [133](#)
- Spinlock, [64](#), [130](#), [134](#), [135](#)
- Spinning, [44](#)
- Spring kernel, [72](#)

- Springboard function, 149
- Stability theory, 55
- Stack, 16, 43
- Stack Pointer (SP), 14
- Stream (benchmark), 30
- Streaming, 33
- Streaming SIMD Extension (SSE), 33, 48
- Sub-kernel, 8, 63, 77
- Super-scalar execution, 15, 27, 30, 66, 103, 171
- Suspend, 43
- Suspend (task), 44
- Swapping, 47
- Symmetric Multi-processing, 62
- Symmetric Multi-Processing (SMP), 48
- Synchronous
 - communication, 46
 - control transfer, 89
 - Exception, 101, 102
 - Interrupt, 101
 - interrupt, 20, 102
 - System call, 100
- `sysfs` virtual file system, 115
- System call, 19, 47
- System partition, 93
- System V, 48
 - Shared Memory, 98, 130, 196
- System-Management Interrupt, 91
- System-Management Interrupt (SMI), 91, 101
- System-Management Mode (SMM), 23
- Task, 43, 54
- Task state, 43
- Task switch, 40, 44, 45, 69, 76, 87, 93, 152
- Temporal partitioning, 71, 87
- Texas Instruments, 72
- Thread, 43
- Threshold, 68, 70, 103
- Throughput, 27
- tickless (Linux feature), 76
- Tickless mode (Linux feature), 76, 82, 202
- Time slice, 43, 60, 71
- Time-Triggered Paradigm, 102
- Time-triggered paradigm, 57, 99
- Time/utility function, 53
- Timer interrupt, 19
- Timing analysis, 65
- Timing anomalies, 6, 64
- Timing anomaly, 155
- TLB shutdown, 120
- Tool (software), 42
- Transactional memory, 39, 201
- Translation Look-aside Buffer (TLB), 15
- Tri-Core micro-controller, 73
- UEFI (Unified Extended Firmware Interface), 89
- Uniform memory architecture (UMA), 34
- UNIX signal, 152
- User mode, 22
- User-mode, 46
- User-space memory, 46
- Userspace Read-Copy-Update, 132
- Vectorization, 33, 48
- Vertical integration, 4
- Via, 21
- Virtual address, 43, 47
- Virtual machine, 48
- Von Neumann architecture, 13
- VxWorks, 61
- Wait free, 190
- Wait-free, 7, 64, 136
- Wall clock time, 51, 93, 109, 111, 112, 122
- Wall-clock time, 46
- Watchdog timer, 92
- Working set, 16, 17

Index

Worst-case

 Latency, [103](#)

Worst-case execution time, [53](#), [65](#), [156](#)

Worst-case execution time (WCET), [54](#)

Write-back, [161](#)

Write-Back cache, [18](#), [108](#)

Write-through, [161](#)

Write-Through cache, [18](#), [108](#)

x86, [21](#)

x86 architecture, [21](#)

 Atomic operations, [38](#)

 Cache coherence, [130](#)

 Shared memory, [130](#)

 Time-stamp counter, [27](#)

Xenomai, [78–80](#), [82](#), [179](#)

Zero-copy protocol, [138](#)

Bibliography

- [AB98] Luca Abeni and Giorgio Buttazzo. “Integrating Multimedia Applications in Hard Real-Time Systems.” In: *Proc. 19th IEEE Real-Time Systems Symp. (RTSS)*. (Madrid, Spain, Dec. 2–4, 1998). Washington, DC, USA: IEEE Computer Society, 1998, pp. 4–13. DOI: [10.1109/REAL.1998.739726](https://doi.org/10.1109/REAL.1998.739726).
- [ABB09] James H. Anderson, Sanjoy K. Baruah, and Björn B. Brandenburg. “Multicore Operating-System Support for Mixed Criticality.” In: *Proc. Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*. (San Francisco, CA, USA, Apr. 16, 2009). 2009.
- [Abb13] Doug Abbott. *Linux for Embedded and Real-Time Applications*. Waltham, MA, USA: Elsevier, 2013. ISBN: 978-0-12-415996-9.
- [Abe+02] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. “A Measurement-Based Analysis of the Real-Time Performance of Linux.” In: *Proc. of 8th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*. (San Jose, CA, USA, Sept. 24–27, 2002). Washington, DC, USA: IEEE Computer Society, 2002, pp. 133–142. DOI: [10.1109/RTAS.2002.1137388](https://doi.org/10.1109/RTAS.2002.1137388).
- [Abe01] Luca Abeni. “Coping With Interrupt Execution Time in RT Kernel: a Non-Intrusive Approach.” In: *Proc. Work-in-Progress Session 22nd IEEE Real-Time Systems Symposium (RTSS-WiP)*. (London, England, Dec. 3–6, 2001). 2001.
- [ABL96] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, eds. *Formal Methods for Industrial Applications. Specifying and Programming the Steam Boiler Control*. Vol. 1165. Lect. Notes in Comp. Sci. The book grew out of a Dagstuhl Seminar, June 1995. Berlin, Germany: Springer, 1996. ISBN: 3-540-61929-1. DOI: [10.1007/BFb0027227](https://doi.org/10.1007/BFb0027227).
- [Abr96] Jean-Raymond Abrial. “Steam-boiler control specification problem.” In: *Formal Methods for Industrial Applications. Specifying and Programming the Steam Boiler Control*. Ed. by Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. Vol. 1165. Lect. Notes in Comp. Sci. The book grew out of a Dagstuhl Seminar, June 1995. Berlin, Germany: Springer, 1996, pp. 500–509. ISBN: 3-540-61929-1. DOI: [10.1007/BFb0027252](https://doi.org/10.1007/BFb0027252).

Bibliography

- [ACD06] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. “Real-Time Scheduling on Multicore Platforms.” In: *Proc. of 12th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*. (San Jose, CA, USA, Apr. 4–7, 2006). Washington, DC, USA: IEEE Computer Society, 2006, pp. 179–190. DOI: [10.1109/RTAS.2006.35](https://doi.org/10.1109/RTAS.2006.35).
- [AEM07] Siro Arthur, Carsten Emde, and Nicholas Mc Guire. “Assessment of the Realtime Preemption Patches (RT-Preempt) and their impact on the general purpose performance of the system.” In: *Proc. 9th Real-Time Linux Workshop (RTLWS)*. (Linz, Austria, Nov. 2–4, 2007). 2007.
- [AG96] Sarita V. Adve and Kouros Gharachorloo. “Shared Memory Consistency Models: A Tutorial.” In: *Computer* 29.12 (Dec. 1996), pp. 66–76. ISSN: 0018-9162. DOI: [10.1109/2.546611](https://doi.org/10.1109/2.546611).
- [AH00] James H. Anderson and Philip Holman. “Efficient Pure-buffer Algorithms for Real-time Systems.” In: *Proc. 7th Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. (Cheju Island, South Korea, Dec. 12–14, 2000). Washington, DC, USA: IEEE Computer Society, 2000, pp. 57–64. DOI: [10.1109/RTCSA.2000.896371](https://doi.org/10.1109/RTCSA.2000.896371).
- [AL07] Anant Agarwal and Markus Levy. “The KILL Rule for Multicore.” In: *Proc. 44th ACM/IEEE Design Automation Conf. (DAC)*. (San Diego, CA, USA, June 4–8, 2007). Washington, DC, USA: IEEE Computer Society, 2007, pp. 750–753. ISBN: 978-1-59593-627-1.
- [AIB13] Samy Al Bahra. “Nonblocking Algorithms and Scalable Multicore Programming.” In: *ACM Queue* 11.5 (May 2013), 40:40–40:64. ISSN: 1542-7730. DOI: [10.1145/2488364.2492433](https://doi.org/10.1145/2488364.2492433).
- [ALL12] Hakan Akkan, Michael Lang, and Lorie M. Liebrock. “Stepping towards noiseless Linux environment.” In: *Proc. 2nd Int. Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. (Venice, Italy, June 29, 2012). New York, NY, USA: ACM Assoc. for Computing Machinery, 2012, 7:1–7:7. ISBN: 978-1-4503-1460-2. DOI: [10.1145/2318916.2318925](https://doi.org/10.1145/2318916.2318925).
- [Alt13] Stephan Alt. “Kooperation von rechenintensiven und isolierten Tasks unter Echtzeitbedingungen.” German. Bachelor thesis. Aachen, Germany: RWTH Aachen University, Faculty of Electrical Engineering and Information Technology, Chair for Operating Systems, Sept. 2013.
- [AM12] Hemant Agrawal and Ravi Malhotra. “Device Drivers in User-Space: A Case for Network Device Driver.” In: *Int. J. Inf. Educ. Technol.* 2.5 (2012). Ed. by Steve Thatcher et al., pp. 461–463. ISSN: 1947-3826. DOI: [DOI:10.7763/IJIET.2012.V2.179](https://doi.org/10.7763/IJIET.2012.V2.179).

- [AM95] James H. Anderson and Mark Moir. “Universal Constructions for Large Objects.” In: *Proc. 9th Int. Workshop Distributed Algorithms (WDAG)*. (Le Mont-Saint-Michel, France, Sept. 13–15, 1995). Ed. by Jean-Michel Hélary and Michel Raynal. Vol. 972. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 1995, pp. 168–182. ISBN: 3-540-60274-7. DOI: [10.1007/BFb0022146](https://doi.org/10.1007/BFb0022146).
- [AMD05] *Software Optimization Guide for AMD64 Processors*. 25112. Sunnyvale, CA, USA: Advanced Micro Devices, Inc., Sept. 2005.
- [AMD10] *CPUID Specification*. 25481. Sunnyvale, CA, USA: Advanced Micro Devices, Inc., Sept. 2010.
- [AMD12a] *AMD64 Architecture Programmer’s Manual*. Vol. 1: *Application Programming*. 24592. Sunnyvale, CA, USA: Advanced Micro Devices, Inc., Mar. 2012.
- [AMD12b] *AMD64 Architecture Programmer’s Manual*. Vol. 2: *System Programming*. 24593. Sunnyvale, CA, USA: Advanced Micro Devices, Inc., Sept. 2012.
- [AMD12c] *AMD64 Architecture Programmer’s Manual*. Vol. 3: *General-Purpose and System Instructions*. 24594. Sunnyvale, CA, USA: Advanced Micro Devices, Inc., Sept. 2012.
- [AMD12d] *AMD64 Architecture Programmer’s Manual*. Sunnyvale, CA, USA: Advanced Micro Devices, Inc., 2012.
- [AMD13] *BIOS and Kernel Developer’s Guide for AMD Family 15h Models 00h-0Fh Processors*. 42301. Sunnyvale, CA, USA: Advanced Micro Devices, Inc., Jan. 2013.
- [AMR10] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. “Resilience Analysis: Tightening the CRPD bound for set-associative caches.” In: *Proc. ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. (Stockholm, Sweden, Apr. 12–16, 2010). New York, NY, USA: ACM Assoc. for Computing Machinery, 2010, pp. 153–162. ISBN: 978-1-60558-953-4. DOI: [10.1145/1755888.1755911](https://doi.org/10.1145/1755888.1755911).
- [And+08] Alexandru Andrei, Petru Eles, Zebo Peng, and Jakob Rosen. “Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip.” In: *21st Int. Conf. on VLSI Design (VLSID)*. (Hyderabad, India, Jan. 4–8, 2008). Washington, DC, USA: IEEE Computer Society, 2008, pp. 103–110. DOI: [10.1109/VLSI.2008.33](https://doi.org/10.1109/VLSI.2008.33).

Bibliography

- [And+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, et al. “Continuous Profiling: Where Have All the Cycles Gone?” In: *ACM Trans. Comput. Syst.* 15.4 (Nov. 1997), pp. 357–390. ISSN: 0734-2071. DOI: [10.1145/265924.265925](https://doi.org/10.1145/265924.265925).
- [AVR10] *8-bit AVR Instruction Set*. Rev. 0856I-AVR-07/10. San Jose, CA, USA: Atmel Corporation, July 2010.
- [BA09a] Björn B. Brandenburg and James H. Anderson. “Joint Opportunities for Real-Time Linux and Real-Time Systems Research.” In: *Proc. 11th Real-Time Linux Workshop (RTLWS)*. (Dresden, Germany, Sept. 28–30, 2009). 2009, pp. 1–12.
- [BA09b] Björn B. Brandenburg and James H. Anderson. “Reader-Writer Synchronization for Shared-Memory Multiprocessor Systems.” In: *Proc. 21st Euromicro Conf. on Real-Time Systems (ECRTS)*. (Dublin, Ireland, July 1–3, 2009). Washington, DC, USA: IEEE Computer Society, 2009, pp. 184–193. DOI: [10.1109/ECRTS.2009.14](https://doi.org/10.1109/ECRTS.2009.14).
- [Bai+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, et al. “The NAS Parallel Benchmarks.” In: *Int. J. High Perform. Comput. Appl.* 5.3 (1991), pp. 63–73. ISSN: 1741-2846. DOI: [10.1177/109434209100500306](https://doi.org/10.1177/109434209100500306).
- [Bak+09] Stanley Bak, Emiliano Betti, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. “Real-Time Control of I/O COTS Peripherals for Embedded Systems.” In: *Proc. 30th IEEE Real-Time Systems Symp. (RTSS)*. (Washington, DC, USA, Dec. 1–4, 2009). Washington, DC, USA: IEEE Computer Society, 2009, pp. 193–203. DOI: [10.1109/RTSS.2009.41](https://doi.org/10.1109/RTSS.2009.41).
- [Bak06] Theodore P. Baker. “A Comparison of Global and Partitioned EDF Schedulability Tests for Multiprocessors.” In: *Int. Conf. on Real-Time and Network Systems (RTNS)*. (Poitiers, France, May 30–31, 2006). Ed. by Guy Juanole and Pascal Richard. Poitiers, France: Laboratoire d’Informatique Scientifique et Industrielle, 2006, pp. 119–127. URL: http://www.lisi.ensma.fr/rtns06/fichiers/rtns06_proceedings.pdf (visited on Sept. 4, 2014).
- [Bar+07] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, et al. “Performance Comparison of VxWorks, Linux, RTAI and Xenomai in a Hard Real-time Application.” In: *15th IEEE-NPSS Real-Time Conference*. (Batavia, IL, USA, Apr. 29–May 4, 2007). Washington, DC, USA: IEEE Computer Society, May 2007. ISBN: 1-4244-0867-9. DOI: [10.1109/RTC.2007.4382787](https://doi.org/10.1109/RTC.2007.4382787).

- [Bar+96] Sanjoy K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. “Proportionate Progress: A Notion of Fairness in Resource Allocation.” In: *Algorithmica* 15.6 (1996), pp. 600–625. ISSN: 0178-4617. DOI: [10.1007/BF01940883](https://doi.org/10.1007/BF01940883).
- [Bar97] Michael Barabanov. “A Linux-based Real-Time Operating System.” Master’s thesis. Socorro, NM, USA: New Mexico Institute of Mining and Technology, 1997.
- [BAS03] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI Express System Architecture*. Boston, MA, USA: Addison-Wesley, 2003. ISBN: 0-321-15630-7.
- [Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. “Mathematizing C++ Concurrency.” In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 55–66. ISSN: 0362-1340. DOI: [10.1145/1925844.1926394](https://doi.org/10.1145/1925844.1926394).
- [BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel. From I/O Ports to Process Management*. 3rd ed. Sebastopol, CA, USA: O’Reilly, 2005. ISBN: 978-0-59600-565-8.
- [BCA08] Björn B. Brandenburg, John M. Calandrino, and James H. Anderson. “On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study.” In: *Proc. 29th IEEE Real-Time Systems Symp. (RTSS)*. (Barcelona, Spain, Nov. 30–Dec. 3, 2008). Washington, DC, USA: IEEE Computer Society, 2008, pp. 157–169. DOI: [10.1109/RTSS.2008.23](https://doi.org/10.1109/RTSS.2008.23).
- [BCB09] Wolfgang Betz, Marco Cereia, and Ivan Cibrario Bertolotti. “Experimental evaluation of the Linux RT Patch for real-time applications.” In: *IEEE Conf. on Emerging Technologies Factory Automation (ETFA)*. (Mallorca, Spain, Sept. 22–25, 2009). Washington, DC, USA: IEEE Computer Society, Sept. 2009, pp. 1–4. DOI: [10.1109/ETFA.2009.5347056](https://doi.org/10.1109/ETFA.2009.5347056).
- [BcD08] BSDaemon, coideloko, and D0nAnd0n. “System Management Mode Hack Using SMM for “Other Purposes”.” In: *Phrack mag.* 0x0c (0x41 Apr. 2008). URL: <http://www.phrack.org/issues.html?issue=65&id=7> (visited on May 7, 2014).
- [BDJ06] Anne Bracy, Kshitij Doshi, and Quinn Jacobson. “Disintermediated Active Communication.” In: *IEEE Comput. Archit. Lett.* 5.2 (July 2006). ISSN: 1556-6056. DOI: [10.1109/L-CA.2006.15](https://doi.org/10.1109/L-CA.2006.15).
- [Bet+08] Emiliano Betti, Daniel Pierre Bovet, Marco Cesati, and Roberto Gioiosa. “Hard Real-Time Performances in Multiprocessor-Embedded Systems Using ASMP-Linux.” In: *EURASIP J. Embed. Syst.* 2008.1 (2008), 10:1–10:16. ISSN: 1687-3955. DOI: [10.1155/2008/582648](https://doi.org/10.1155/2008/582648).

Bibliography

- [Bet+13] Emiliano Betti, Stanley Bak, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. “Real-Time I/O Management System with COTS Peripherals.” In: *IEEE Trans. Comput.* 62.1 (2013), pp. 45–58. ISSN: 0018-9340. DOI: [10.1109/TC.2011.202](https://doi.org/10.1109/TC.2011.202).
- [BF11] Sergey Blagodurov and Alexandra Fedorova. “User-level scheduling on NUMA multicore systems under Linux.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, June 13–15, 2011). Ed. by Ralph Siemsen. 2011, pp. 81–91. URL: <http://kernel.org/doc/mirror/ols2011.pdf> (visited on May 7, 2014).
- [BGP97] Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. “Fast Scheduling of Periodic Tasks on Multiple Resources.” In: *Proc. Workshop Job Scheduling Strategies for Parallel Processing (IPPS)*. (Geneva, Switzerland, Apr. 5, 1997). Ed. by Dror G. Feitelson and Larry Rudolph. Vol. 1291. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 1997, pp. 280–288. ISBN: 3-540-63574-2. DOI: [10.1109/IPPS.1995.395946](https://doi.org/10.1109/IPPS.1995.395946).
- [BH08] Berthold Bäuml and Gerd Hirzinger. “When hard realtime matters: Software for complex mechatronic systems.” In: *Robot. Auton. Syst.* 56.1 (2008). Ed. by T. Arai, R. Dillmann, and R. Grupen, pp. 5–13. ISSN: 0921-8890. DOI: [10.1016/j.robot.2007.09.017](https://doi.org/10.1016/j.robot.2007.09.017).
- [Bha02] Ajay V. Bhatt. *Creating a PCI Express Interconnect*. White paper. Santa Clara, CA, USA: Technology and Research Labs, Intel Corp., 2002. URL: http://www.pcisig.com/specifications/pciexpress/resources/PCI_Express_White_Paper.pdf (visited on May 7, 2014).
- [BJ95] Gregory Bollella and Kevin Jeffay. “Support For Real-Time Computing Within General Purpose Operating Systems. Supporting Co-Resident Operating Systems.” In: *Proc. of 1st IEEE Real-Time Technology and Applications Symp. (RTAS)*. (Chicago, IL, USA, May 15–17, 1995). Washington, DC, USA: IEEE Computer Society, 1995, pp. 4–14. DOI: [10.1109/RTTAS.1995.516189](https://doi.org/10.1109/RTTAS.1995.516189).
- [Bla+10] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. “A Case for NUMA-aware Contention Management on Multicore Systems.” In: *Proc. 19th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. (Vienna, Austria, Sept. 11–15, 2010). New York, NY, USA: ACM Assoc. for Computing Machinery, 2010, pp. 557–558. ISBN: 978-1-4503-0178-7. DOI: [10.1145/1854273.1854350](https://doi.org/10.1145/1854273.1854350).
- [BLA09] Björn B. Brandenburg, Hennadiy Leontyev, and James H. Anderson. “Accounting for Interrupts in Multiprocessor Real-Time Systems.” In: *Proc. 15th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. (Beijing, China, Aug. 24–26, 2009).

- Washington, DC, USA: IEEE Computer Society, 2009, pp. 273–283. DOI: [10.1109/RTCSA.2009.37](https://doi.org/10.1109/RTCSA.2009.37).
- [BLA11] Björn B. Brandenburg, Hennadiy Leontyev, and James H. Anderson. “An overview of interrupt accounting techniques for multiprocessor real-time systems.” In: *J. Syst. Archit.* 57.6 (2011), pp. 638–654. ISSN: 1383-7621. DOI: [10.1016/j.sysarc.2010.05.011](https://doi.org/10.1016/j.sysarc.2010.05.011).
- [Bli+04] Martin J. Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. “Linux on NUMA Systems.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, July 21–24, 2004). Ed. by John W. Lockhart. Vol. 1. 2004, pp. 89–101. URL: <http://kernel.org/doc/ols/> (visited on May 7, 2014).
- [BM10] Jeremy H. Brown and Brad Martin. “How fast is fast enough? Choosing between Xenomai and Linux for real-time applications.” In: *Proc. 12th Real-Time Linux Workshop (RTLWS)*. (Nairobi, Kenya, Oct. 25–27, 2010). 2010, pp. 1–17.
- [Boc+09] Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. “Parallel Programming Must Be Deterministic by Default.” In: *1st USENIX Workshop on Hot Topics in Parallelism (HotPar)*. (Berkeley, CA, USA, Mar. 30–31, 2009). Berkeley, CA, USA: USENIX, 2009.
- [Bon+12] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. “Deterministic Execution Model on COTS Hardware.” In: *Proc. 25th Int. Conf. Architecture of Computing Systems (ARCS)*. (Munich, Germany, Feb. 28–Mar. 2, 2012). Ed. by Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte. Vol. 7179. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 2012, pp. 98–110. ISBN: 978-3-642-28292-8. DOI: [10.1007/978-3-642-28293-5_9](https://doi.org/10.1007/978-3-642-28293-5_9).
- [Bon94] Jeff Bonwick. “The Slab Allocator: An Object-Caching Kernel Memory Allocator.” In: *USENIX Summer Technical Conf.* (Boston, MA, USA, June 6–10, 1994). Berkeley, CA, USA: USENIX, 1994. URL: <http://static.usenix.org/publications/library/proceedings/bos94/bonwick.html> (visited on May 7, 2014).
- [Boy+10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, et al. “An Analysis of Linux Scalability to Many Cores.” In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. (Vancouver, BC, Canada, Oct. 4–6, 2010). Berkeley, CA, USA: USENIX, 2010. URL: http://www.usenix.org/event/osdi10/tech/full_papers/Boyd-Wickizer.pdf (visited on July 18, 2014).

Bibliography

- [BR03] Steve Brosky and Steve Rotolo. “Shielded Processors: Guaranteeing Sub-millisecond Response in Standard Linux.” In: *Proc. 2003 IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*. (Nice, France, Apr. 22–26, 2003). Washington, DC, USA: IEEE Computer Society, 2003. DOI: [10.1109/IPDPS.2003.1213237](https://doi.org/10.1109/IPDPS.2003.1213237).
- [Bra+08] Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, and James H. Anderson. “Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?” In: *Proc. of 14th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*. (St. Louis, MO, USA, Apr. 22–24, 2008). Washington, DC, USA: IEEE Computer Society, 2008, pp. 342–353. DOI: [10.1109/RTAS.2008.27](https://doi.org/10.1109/RTAS.2008.27).
- [Bre93] Timothy Brecht. “On the importance of parallel application placement in NUMA multiprocessors.” In: *4th Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS)*. 1993, pp. 1–18.
- [Bro04] Steve Brosky. “Shielded CPUs: Real-Time Performance in Standard Linux.” In: *Linux J.* (Issue 121 May 2004). ISSN: 1075-3583. URL: <http://www.linuxjournal.com/article/6900> (visited on Sept. 9, 2014).
- [Bru10] Raphael Bruns. “Design and Implementation of a Real-Time Interface.” Diploma thesis. Aachen, Germany: RWTH Aachen University, Faculty of Electrical Engineering and Information Technology, Chair for Operating Systems, Dec. 2010.
- [BS97] Tim Boggess and Fred Shirley. “High-performance Scalable Computing for Real-Time Applications.” In: *Proc. 6th Int. Conf. on Computer Communications and Networks (ICCN)*. (Las Vegas, Nevada, USA, Sept. 22–25, 1997). Washington, DC, USA: IEEE Computer Society, 1997, pp. 332–335. ISBN: 0-8186-8186-1. DOI: [10.1109/ICCN.1997.623332](https://doi.org/10.1109/ICCN.1997.623332).
- [BSF04] Alexander von Bülow, Jürgen Stohr, and Georg Färber. “Towards an Efficient Use of Caches in State of the Art Processors for Real-Time Systems.” In: *Proc. Work-In-Progress Session 16th Euromicro Conf. on Real-Time Systems (ECRTS-WiP)*. (Catania, Italy, June 30–July 2, 2004). Ed. by Steve Goddard. Tech. rep. TR-UNL-CSE-2004-0010. University of Nebraska-Lincoln, Department of Computer Science and Engineering, 2004. URL: <http://cse.unl.edu/~goddard/ecrts04wip/proceedings/> (visited on May 7, 2014).

- [BT09] Vlastimil Babka and Petr Tůma. “Investigating Cache Parameters of x86 Family Processors.” In: *Computer Performance Evaluation and Benchmarking. Proc. SPEC Benchmark Workshop*. (Austin, TX, USA, Jan. 25, 2009). Ed. by David R. Kaeli and Kai Sachs. Vol. 5419. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 2009, pp. 77–96. ISBN: 978-3-540-93798-2. DOI: [10.1007/978-3-540-93799-9_5](https://doi.org/10.1007/978-3-540-93799-9_5).
- [Bül+04] Alexander von Bülow, Jürgen Stohr, Georg Färber, Peter Müller, and Johann B. Schraml. *Using the RECOMS Architecture for Controlling a Radio Telescope*. Tech. rep. Munich, Germany: Lehrstuhl für Realzeit-Computersysteme, Technische Universität München, 2004.
- [But05] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications*. 2nd ed. New York, NY, USA: Springer, 2005. ISBN: 0-387-23137-4.
- [BW01] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. 3rd ed. Harlow, England: Pearson Education Ltd., 2001. ISBN: 0-201-72988-1.
- [BW88] Jean-Loup Baer and Wenn-Hann Wang. “On the Inclusion Properties for Multi-level Cache Hierarchies.” In: *Conf. Proc. 15th Int. Symp. on Computer Architecture (ISCA)*. (Honolulu, HI, USA, May 30–June 2, 1988). Los Alamitos, CA, USA: IEEE Computer Society, 1988, pp. 73–80. ISBN: 0-8186-0861-7. DOI: [10.1109/ISCA.1988.5212](https://doi.org/10.1109/ISCA.1988.5212).
- [BY97] Michael Barabanov and Victor Yodaiken. “Introducing Real-Time Linux.” In: *Linux J.* (Issue 34 Feb. 1997). ISSN: 1075-3583. URL: <http://www.linuxjournal.com/article/232> (visited on Sept. 9, 2014).
- [BYT12] Lana Brindley, Alison Young, and Cheryn Tan. *Red Hat Enterprise MRG 2. Realtime Tuning Guide*. 3rd ed. Raleigh, NC, USA: Red Hat, Inc., 2012. URL: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_MRG/2/html/Realtime_Tuning_Guide/index.html (visited on May 7, 2014).
- [C11] ISO/IEC JTC1/SC22/WG14 - C. *ISO/IEC 9899:2011: Programming language C*. Technical Standard. Geneva, Switzerland, 2011.
- [CAB07] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. “A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms.” In: *Proc. 19th Euromicro Conf. on Real-Time Systems (ECRTS)*. (Pisa, Italy, July 4–6, 2007). Washington, DC, USA: IEEE Computer Society, 2007, pp. 247–258. DOI: [10.1109/ECRTS.2007.81](https://doi.org/10.1109/ECRTS.2007.81).

Bibliography

- [Cal+06] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. “LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers.” In: *Proc. 27th IEEE Real-Time Systems Symp. (RTSS)*. (Rio de Janeiro, Brazil, Dec. 5–8, 2006). Washington, DC, USA: IEEE Computer Society, 2006, pp. 111–126. DOI: [10.1109/RTSS.2006.27](https://doi.org/10.1109/RTSS.2006.27).
- [Cal+07] John M. Calandrino, Dan Baumberger, Tong Li, Scott Hahn, and James H. Anderson. “Soft Real-Time Scheduling on Performance Asymmetric Multicore Platforms.” In: *Proc. of 13th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*. (Bellevue, WA, USA, Apr. 3–6, 2007). Washington, DC, USA: IEEE Computer Society, 2007, pp. 101–112. DOI: [10.1109/RTAS.2007.35](https://doi.org/10.1109/RTAS.2007.35).
- [Cam+05] A. Martí Campoy, E. Tamura, S. Sáez, F. Rodríguez, and J. V. Busquets-Mataix. “On Using Locking Caches in Embedded Real-Time Systems.” In: *Proc. 2nd Int. Conf. Embedded Software and Systems (ICESS)*. (Xi’an, China, Dec. 16–18, 2005). Ed. by Laurence Tianruo Yang, Xingshe Zhou, Wei Zhao, Zhaohui Wu, Yian Zhu, et al. Vol. 3820. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 2005, pp. 150–159. ISBN: 3-540-30881-4. DOI: [10.1007/11599555_17](https://doi.org/10.1007/11599555_17).
- [Car+04] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, et al. “A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms.” In: *Handbook on Scheduling Algorithms, Methods, and Models*. Ed. by Joseph Y-T. Leung. Boca Raton, FL, USA: Chapman Hall/CRC Press, 2004, pp. 30.1–30.19. ISBN: 1-58488-397-9.
- [CB13] Felipe Cerqueira and Björn B. Brandenburg. “A Comparison of Scheduling Latency in Linux, PREEMPT_RT, and LITMUS^{RT}.” In: *Int. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPert) in conjunction with the 25th Euromicro Conf. on Real-Time Systems*. (Paris, France, July 9, 2013). Ed. by Andrea Bastoni and Shinpei Kato. 2013. URL: http://ert1.jp/~shinpei/conf/ospert13/OSPert13_proceedings_web.pdf (visited on May 7, 2014).
- [Cha+04] Gilles Chanteperdrix, Alexis Berlemont, Dominique Ragot, and Philippe Kajfasz. “Integration of Real-Time Services in User-Space Linux.” In: *Proc. 6th Real-Time Linux Workshop (RTLWS)*. (Singapore, Nov. 3–5, 2004). 2004.
- [Cha02] Lin Chao, ed. *Intel Technology Journal* 6 (1 Feb. 14, 2002): *Hyper-Threading Technology*. ISSN: 1535-766X.

- [Che+09] Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, and Giuseppe Lipari. “Hierarchical Multiprocessor CPU Reservations for the Linux Kernel.” In: *Int. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT) in conjunction with the 16th Euromicro Conf. on Real-Time Systems*. (Dublin, Ireland, July 1–3, 2009). Ed. by Stefan M. Petters and Peter Zijlstra. 2009, pp. 14–22. URL: <http://www.artist-embedded.org/artist/Overview,1750.html> (visited on May 7, 2014).
- [Cor07] Jonathan Corbet. “Clockevents and dyntick.” In: *LWN.net* (Feb. 21, 2007). URL: <http://lwn.net/Articles/223185> (visited on Oct. 7, 2014).
- [Cor10] Jonathan Corbet. “The state of realtime Linux.” In: *LWN.net* (June 15, 2010). URL: <http://lwn.net/Articles/392154> (visited on Oct. 7, 2014).
- [Cor12] Jonathan Corbet. “Relocating RCU callbacks.” In: *LWN.net* (Oct. 31, 2012). URL: <http://lwn.net/Articles/522262> (visited on Oct. 7, 2014).
- [Cor13a] Jonathan Corbet. “3.9 Merge window part 1.” In: *LWN.net* (Feb. 20, 2013). URL: <http://lwn.net/Articles/539179> (visited on Oct. 7, 2014).
- [Cor13b] Jonathan Corbet. “A new direction for power-aware scheduling.” In: *LWN.net* (Oct. 15, 2013). URL: <http://lwn.net/Articles/570353> (visited on Oct. 7, 2014).
- [Cor13c] Jonathan Corbet. “Is the whole system idle?” In: *LWN.net* (July 10, 2013). URL: <http://lwn.net/Articles/558284> (visited on Oct. 7, 2014).
- [Cor13d] Jonathan Corbet. “Low-latency ethernet device polling.” In: *LWN.net* 551284 (May 21, 2013). URL: <http://lwn.net/Articles/551284> (visited on Oct. 7, 2014).
- [Cor13e] Jonathan Corbet. “(Nearly) full tickless operation in 3.10.” In: *LWN.net* (May 8, 2013). URL: <http://lwn.net/Articles/549580> (visited on Oct. 7, 2014).
- [Cor13f] Jonathan Corbet. “Optimizing CPU hotplug locking.” In: *LWN.net* (Oct. 9, 2013). URL: <http://lwn.net/Articles/569686> (visited on Oct. 7, 2014).
- [Cor13g] Jonathan Corbet. “Polling block drivers.” In: *LWN.net* (June 26, 2013). URL: <http://lwn.net/Articles/556244> (visited on Oct. 7, 2014).

Bibliography

- [Cor13h] Jonathan Corbet. “The 3.11 merge window opens.” In: *LWN.net* (July 3, 2013). URL: <http://lwn.net/Articles/557314> (visited on Oct. 7, 2014).
- [Cor13i] Jonathan Corbet. “The ‘Jailhouse’ hypervisor.” In: *LWN.net* (Nov. 19, 2013). URL: <http://lwn.net/Articles/574274> (visited on Oct. 7, 2014).
- [Cor14a] Jonathan Corbet. “Deadline scheduler merged for 3.14.” In: *LWN.net* (Jan. 21, 2014). URL: <http://lwn.net/Articles/581491> (visited on Oct. 7, 2014).
- [Cor14b] Jonathan Corbet. “Locking and pinning.” In: *LWN.net* (June 4, 2014). URL: <http://lwn.net/Articles/600502> (visited on Oct. 7, 2014).
- [CP00] Antoine Colin and Isabelle Puaut. “Worst Case Execution Time Analysis for a Processor with Branch Prediction.” In: *Real-Time Syst.* 18.2-3 (2000), pp. 249–274. ISSN: 0922-6443. DOI: [10.1023/A:1008149332687](https://doi.org/10.1023/A:1008149332687).
- [CP01] Antoine Colin and Isabelle Puaut. “Worst-case execution time analysis of the RTEMS real-time operating system.” In: *Proc. 13th Euromicro Conf. on Real-Time Systems (ECRTS)*. (Delft, Netherlands, June 13–15, 2001). Washington, DC, USA: IEEE Computer Society, 2001, pp. 191–198. DOI: [10.1109/EMRTS.2001.934029](https://doi.org/10.1109/EMRTS.2001.934029).
- [CRJ07] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. “Synchronization for an optimal real-time scheduling algorithm on multiprocessors.” In: *Proc. 2nd IEEE Int. Symp. on Industrial Embedded Systems (SIES)*. (Lisbon, Portugal, July 4–6, 2007). Washington, DC, USA: IEEE Computer Society, 2007, pp. 9–16. ISBN: 1-4244-0840-7. DOI: [10.1109/SIES.2007.4297311](https://doi.org/10.1109/SIES.2007.4297311).
- [CRK05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. 3rd ed. Sebastopol, CA, USA: O’Reilly, 2005. ISBN: 978-0-596-00590-0.
- [CW76] Harold J. Curnow and Brian A. Wichmann. “A synthetic benchmark.” In: *Comput. J.* 19.1 (1976), pp. 43–49. ISSN: 1460-2067. DOI: [10.1093/comjnl/19.1.43](https://doi.org/10.1093/comjnl/19.1.43).
- [Dak09] Steven Dake. “The Corosync High Performance Shared Memory IPC Reusable C Library.” In: *Proc. Linux Symp. (OLS)*. (Montreal, QC, Canada, July 13–17, 2009). Ed. by Robyn Bergeron, Chris Dukes, Jonas Fonseca, and John Hawley. 2009. URL: <http://linuxsymposium.org/2009/ls-2009-proceedings.pdf> (visited on May 7, 2014).

- [Dan12] Jens Dankert. “Implementation and Evaluation of Real-Time Applications comparing different Paradigms.” Master thesis. Aachen, Germany: RWTH Aachen University, Faculty of Electrical Engineering and Information Technology, Chair for Operating Systems, Dec. 2012.
- [DAR12] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. “Toward Predictable Performance in Software Packet-Processing Platforms.” In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. (San Jose, CA, USA, Apr. 25–27, 2012). Berkeley, CA, USA: USENIX, 2012, pp. 141–154. ISBN: 978-931971-92-8. URL: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/dobrescu> (visited on July 11, 2014).
- [Das+11] Dakshina Dasari, Björn Andersson, Vincent Nélis, Stefan M. Petters, Arvind Easwaran, et al. “Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus.” In: *IEEE 10th Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*. (Changsha, Hunan Province, China, Nov. 16–18, 2011). Washington, DC, USA: IEEE Computer Society, 2011, pp. 1068–1075. ISBN: 978-0-7695-4600-1. DOI: [10.1109/TrustCom.2011.146](https://doi.org/10.1109/TrustCom.2011.146).
- [Das+13] Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters. “Identifying the Sources of Unpredictability in COTS-based Multicore Systems.” In: *Proc. 8th IEEE Int. Symp. on Industrial Embedded Systems (SIES)*. (Porto, Portugal, June 19–21, 2013). Washington, DC, USA: IEEE Computer Society, 2013, pp. 39–48. ISBN: 978-1-4799-0658-1. DOI: [10.1109/SIES.2013.6601469](https://doi.org/10.1109/SIES.2013.6601469).
- [DEG01] Loïc Dufflot, Daniel Etiemble, and Olivier Grumelard. “Using CPU System Management Mode to Circumvent Operating System Security Functions.” In: *Proc. 7th CanSecWest Conf.* 2001.
- [Den+99] Z. Deng, Jane W.-S. Liu, L. Zhang, S. Mouna, and A. Frei. “An Open Environment for Real-Time Applications.” In: *Real-Time Syst.* 16 (2-3 1999), pp. 155–185. ISSN: 0922-6443. DOI: [10.1023/A:1008094905565](https://doi.org/10.1023/A:1008094905565).
- [Des13a] Sanjay R. Deshpande. “Design Considerations for Multicore SoC Interconnections.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 5. ISBN: 978-0-12-416018-7.
- [Des13b] Mathieu Desnoyers. “Proving the Correctness of Nonblocking Data Structures.” In: *ACM Queue* 11.5 (May 2013), 30:30–30:43. ISSN: 1542-7730. DOI: [10.1145/2488364.2490873](https://doi.org/10.1145/2488364.2490873).

Bibliography

- [Dic13] Tom Dickens. “Software Synchronization.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 12. ISBN: 978-0-12-416018-7.
- [Dij63] Edsger W. Dijkstra. “Over seinpalen.” Dutch. Circulated privately. 1963. URL: <http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF> (visited on May 7, 2014).
- [Dij65] Edsger W. Dijkstra. “Solution of a Problem in Concurrent Programming Control.” In: *Commun. ACM* 8.9 (Sept. 1965), p. 569. ISSN: 0001-0782. DOI: [10.1145/365559.365617](https://doi.org/10.1145/365559.365617).
- [Dij68] Edsger W. Dijkstra. “Cooperating sequential processes.” In: *Programming Languages: NATO Advanced Study Institute*. Ed. by F. Genuys. Academic Press, 1968, pp. 43–112.
- [DM03a] Lorenzo Dozio and Paolo Mantegazza. “Linux Real Time Application Interface (RTAI) in low cost high performance motion control.” In: *ANIPLA Conf. on Motion Control*. (Milano, Italy, Mar. 27–28, 2003). 2003.
- [DM03b] Lorenzo Dozio and Paolo Mantegazza. “Real time distributed control systems using RTAI.” In: *6th Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISOCR)*. (Hakodate, Hokkaido, Japan, May 14–16, 2003). Washington, DC, USA: IEEE Computer Society, 2003, pp. 11–18. DOI: [10.1109/ISORC.2003.1199229](https://doi.org/10.1109/ISORC.2003.1199229).
- [DM04] Cort Dougan and Zwane Mwaikambo. “Lies, Misdirection, and Real-Time Measurements. Here are guidelines for evaluating claims made by real-time operating systems vendors.” In: *C/C++ Users J.* (Apr. 2004), pp. 6–13. ISSN: 1075-2838. URL: <http://www.drdoobbs.com/184401780> (visited on May 7, 2014).
- [DNA11] Dakshina Dasari, Vincent Nélis, and Björn Andersson. “WCET Analysis Considering Contention on Memory Bus in COTS-Based Multicores.” In: *IEEE 16th Conf. on Emerging Technologies Factory Automation (ETFA)*. (Toulouse, France, Sept. 5–9, 2011). Washington, DC, USA: IEEE Computer Society, 2011. ISBN: 978-1-4577-0018-7. DOI: [10.1109/ETFA.2011.6059176](https://doi.org/10.1109/ETFA.2011.6059176).
- [DO13] Paul Dubrulle and Emmanuel Ohayon. “A Dedicated Micro-Kernel to Combine Real-Time and Stream Applications on Embedded Manycores.” In: *Procedia Comput. Sci.* 18 (2013), pp. 1634–1643. ISSN: 1877-0509. DOI: [10.1016/j.procs.2013.05.331](https://doi.org/10.1016/j.procs.2013.05.331).
- [Dom08] Max Domeika. *Software Development for Embedded Multi-core Systems. A Practical Guide Using Embedded Intel Architecture*. Burlington, MA, USA: Newness, 2008. ISBN: 978-0-7506-8539-9.

- [Dom13] Max Domeika. “Communication and Synchronization Libraries.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 8. ISBN: 978-0-12-416018-7.
- [Don88] Jack J. Dongarra. “The LINPACK Benchmark: An explanation.” In: *1st Int. Supercomputing Conf. (SC)*. (Athens, Greece, June 8–12, 1987). Ed. by Elias N. Houstis, Theodore S. Papatheodorou, and Constantine D. Polychronopoulos. Vol. 297. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 1988, pp. 456–474. ISBN: 3-540-18991-2. DOI: [10.1007/3-540-18991-2_27](https://doi.org/10.1007/3-540-18991-2_27).
- [Dow08] Allen B. Downey. *The Little Book of Semaphores*. 2nd ed. Online. Needham, MA, USA: Green Tea Press, 2008. URL: <http://www.greenteapress.com/semaphores/> (visited on May 7, 2014).
- [Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory*. Online. Raleigh, NC, USA: Red Hat, Inc., 2007. URL: <http://www.akkadia.org/drepper/cpumemory.pdf> (visited on June 5, 2014).
- [Dre11] Ulrich Drepper. *Futexes Are Tricky*. Tech. rep. Version 1.6. Online. Raleigh, NC, USA: Red Hat, Inc., Nov. 2011. URL: <http://www.akkadia.org/drepper/futex.pdf> (visited on May 7, 2014).
- [Duf+09] Loïc Dufflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. *Getting into the SMRAM: SMM Reloaded*. Presentation slides. 2009. URL: http://www.ssi.gouv.fr/IMG/pdf/Cansec_final.pdf.
- [Duv09] Dominic Duval. “From Fast to Predictably Fast.” In: *Proc. Linux Symp. (OLS)*. (Montreal, QC, Canada, July 13–17, 2009). Ed. by Robyn Bergeron, Chris Dukes, Jonas Fonseca, and John Hawley. 2009, pp. 79–86. URL: <http://linuxsymposium.org/2009/ls-2009-proceedings.pdf> (visited on May 7, 2014).
- [DW05] Sven-Thorsten Dietrich and Daniel Walker. “The Evolution of Real-Time Linux.” In: *Proc. 7th Real-Time Linux Workshop (RTLWS)*. (Lille, France, Nov. 3–4, 2005). 2005.
- [Edg11] Jake Edge. “ELC: A PREEMPT_RT roadmap.” In: *LWN.net* (Apr. 27, 2011). URL: <http://lwn.net/Articles/440064> (visited on Oct. 7, 2014).
- [Edg13a] Jake Edge. “Plans for hot adding and removing memory.” In: *LWN.net* (June 12, 2013). URL: <http://lwn.net/Articles/553199> (visited on Oct. 7, 2014).
- [Edg13b] Jake Edge. “The future of realtime Linux.” In: *LWN.net* (Nov. 6, 2013). URL: <http://lwn.net/Articles/572740> (visited on Oct. 7, 2014).

Bibliography

- [Eid+04] Eric Eide, Tim Stack, John Regehr, and Jay Lepreau. “Dynamic CPU management for real-time, middleware-based systems.” In: *Proc. of 10th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*. (Toronto, ON, Canada, May 25–28, 2004). Washington, DC, USA: IEEE Computer Society, 2004, pp. 286–295. DOI: [10.1109/RTAS.2004.1317274](https://doi.org/10.1109/RTAS.2004.1317274).
- [Els+72] Bernard Elspas, Karl N. Levitt, Richard J. Waldinger, and Abraham Waksman. “An Assessment of Techniques for Proving Program Correctness.” In: *ACM Comput. Surv.* 4.2 (June 1972), pp. 97–147. ISSN: 0360-0300. DOI: [10.1145/356599.356602](https://doi.org/10.1145/356599.356602).
- [Emd10] Carsten Emde. “Long-term monitoring of apparent latency in PRE-EMPT RT Linux real-time systems.” In: *Proc. 12th Real-Time Linux Workshop (RTLWS)*. (Nairobi, Kenya, Oct. 25–27, 2010). 2010.
- [Emd11] Carsten Emde. “How to cope with the negative impact of a processor’s energy-saving features on real-time capabilities?” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011. ISBN: 978-3-0003-6193-7.
- [Eng00] Ralf S. Engelschall. “Portable Multithreading: The Signal Stack Trick For User-Space Thread Creation.” In: *USENIX Annu. Technical Conf.* (San Diego, CA, USA, June 18–23, 2000). Berkeley, CA, USA: USENIX, June 2000.
- [ESZ08] Shawn Embleton, Sherri Sparks, and Cliff Zou. “SMM Rootkits: A New Breed of OS Independent Malware.” In: *Proc. of the 4th Int. Conf. on Security and Privacy in Communication Networks (SecureComm)*. (Istanbul, Turkey, Sept. 22–25, 2008). New York, NY, USA: ACM Assoc. for Computing Machinery, 2008, 11:1–11:12. ISBN: 978-1-60558-241-2. DOI: [10.1145/1460877.1460892](https://doi.org/10.1145/1460877.1460892).
- [EV11] Benjamin Engel and Marcus Völpl. “Turning Krieger’s MCS Lock into a Send Queue. or, a Case for Reusing Clever, Mostly Lock-Free Code in a Different Area.” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011. ISBN: 978-3-0003-6193-7.
- [Fer+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, et al. “Reliable and Precise WCET Determination for a Real-Life Processor.” In: *Proc. 1st Int. Workshop Embedded Software (EMSOFT)*. (Tahoe City, CA, USA, Oct. 8–10, 2001). Ed. by Thomas A. Henzinger and Christoph M. Kirsch. Vol. 2211. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 2001, pp. 469–485. ISBN: 3-540-42673-6. DOI: [10.1007/3-540-45449-7_32](https://doi.org/10.1007/3-540-45449-7_32).

- [Fin01] Steven A. Finney. “Real-time data collection in Linux: A case study.” In: *Behav. Res. Methods Instrum. Comput.* 33 (2 2001), pp. 167–173. ISSN: 0743-3808. DOI: [10.3758/BF03195362](https://doi.org/10.3758/BF03195362).
- [FLO02] Dominique Fober, Stephane Letz, and Yann Orlarey. “Lock-Free Techniques for Concurrent Access to Shared Objects.” In: *Actes des Journées d’Informatique Musicale (JIM)*. (Marseille, France, May 29–31, 2002). Marseille, France: Centre Nationale de Création Musicale, 2002, pp. 143–150. URL: <http://www.grame.fr/ressources/publications/fober-JIM2002.pdf> (visited on May 7, 2014).
- [Fly72] Michael J. Flynn. “Some Computer Organizations and Their Effectiveness.” In: *Computers, IEEE Transactions on C-21.9* (Sept. 1972), pp. 948–960. ISSN: 0018-9340. DOI: [10.1109/TC.1972.5009071](https://doi.org/10.1109/TC.1972.5009071).
- [Fog13a] Agner Fog. *Software optimization manuals*. Copenhagen, Denmark: Technical University of Denmark, 2013. URL: <http://www.agner.org/optimize/> (visited on May 7, 2014).
- [Fog13b] Agner Fog. *Software optimization manuals*. Vol. 1: *Optimizing software in C++*. An optimization guide for Windows, Linux and Mac platforms. Copenhagen, Denmark: Technical University of Denmark, Sept. 23, 2013. URL: <http://www.agner.org/optimize/> (visited on May 7, 2014).
- [Fog13c] Agner Fog. *Software optimization manuals*. Vol. 2: *Optimizing sub-routings in assembly language*. An optimization guide for x86 platforms. Copenhagen, Denmark: Technical University of Denmark, Sept. 28, 2013. URL: <http://www.agner.org/optimize/> (visited on May 7, 2014).
- [Fog13d] Agner Fog. *Software optimization manuals*. Vol. 3: *The microarchitecture of Intel, AMD and VIA CPUs*. An optimization guide for assembly programmers and compiler makers. Copenhagen, Denmark: Technical University of Denmark, Sept. 4, 2013. URL: <http://www.agner.org/optimize/> (visited on May 7, 2014).
- [Fog13e] Agner Fog. *Software optimization manuals*. Vol. 4: *Instruction tables*. Lists of instruction latencies, throughputs and micro-operation break-downs for intel, AMD and VIA CPUs. Copenhagen, Denmark: Technical University of Denmark, Apr. 3, 2013. URL: <http://www.agner.org/optimize/> (visited on May 7, 2014).
- [Fri05] Brandon Friesen. *Bran’s Kernel Development*. A tutorial on writing kernels. Online. 2005. URL: <http://www.osdever.net/bkerndev/Docs/title.htm> (visited on May 7, 2014).

Bibliography

- [FRK02] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. “Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux.” In: *Proc. Ottawa Linux Symp. (OLS)*. (Ottawa, ON, Canada, June 26–29, 2002). Ed. by John W. Lockhart. 2002, pp. 479–489. URL: <http://www.linuxsymposium.org/archives/OLS/Reprints-2002/> (visited on May 7, 2014).
- [Fue+12] David de la Fuente, Jesús Barba, Fernando Rincón, Julio Daniel Dondo, and Juan Carlos López. “Flexible, Open and Efficient Embedded Multimedia Systems.” In: *Embedded Systems. High Performance Systems, Applications and Projects*. Ed. by Kiyofumi Tanaka. Rijeka, Croatia: InTech, 2012. Chap. 7. ISBN: 978-953-51-0350-9. DOI: [10.5772/2684](https://doi.org/10.5772/2684).
- [Gal95] Bill O. Gallmeister. *POSIX.4, Programming for the Real World*. Sebastopol, CA, USA: O’Reilly, 1995. ISBN: 1-56592-070-0.
- [GDA08] Alexander Graf, Olaf Dabrunz, and Stefan Assmann. “Interrupt Handling on x86 (RT) and Boot Interrupt Quirks.” In: *Proc. 10th Real-Time Linux Workshop (RTLWS)*. (Colotlán, Jalisco, Mexico, Sept. 28–30, 2008). 2008.
- [Gee07] David Geer. “For Programmers, Multicore Chips Mean Multiple Challenges.” In: *Computer* 40.9 (Sept. 2007), pp. 17–19. ISSN: 0018-9162. DOI: [10.1109/MC.2007.311](https://doi.org/10.1109/MC.2007.311).
- [Ger+12] Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, and Pascal Sainrat. “Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications.” In: *Design, Automation Test in Europe Conf. Exhibition (DATE)*. (Dresden, Germany, Mar. 12–16, 2012). Washington, DC, USA: IEEE Computer Society, Mar. 2012, pp. 671–676. DOI: [10.1109/DATE.2012.6176555](https://doi.org/10.1109/DATE.2012.6176555).
- [Ger01] Philippe Gerum. “The XENOMAI Project, Implementing a RTOS emulation framework on GNU/Linux.” In: *Proc. 3rd Real-Time Linux Workshop (RTLWS)*. (Milan, Italia, Nov. 26–29, 2001). 2001.
- [Ger04] Philippe Gerum. *Xenomai – Implementing a RTOS emulation framework on GNU/Linux*. White paper. Xenomai, 2004. URL: <http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf> (visited on May 7, 2014).
- [GF07] Matthias Goebel and Georg Färber. “A Real-Time-capable Hard- and Software Architecture for Joint Image and Knowledge Processing in Cognitive Automobiles.” In: *IEEE Intelligent Vehicles Symp.* (Istanbul, Turkey, June 13–15, 2007). Washington, DC, USA: IEEE Computer Society, 2007, pp. 734–740. DOI: [10.1109/IVS.2007.4290204](https://doi.org/10.1109/IVS.2007.4290204).

- [Gio+04] Roberto Gioiosa, Fabrizio Petrini, Kei Davis, and Fabien Lebaillif-Delamare. “Analysis of System Overhead on Parallel Computers.” In: *Proc. 4th IEEE Int. Symp. on Signal Processing and Information Technology (ISSPIT)*. (Rome, Italy, Dec. 18–21, 2004). Washington, DC, USA: IEEE Computer Society, 2004, pp. 387–390. DOI: [10.1109/ISSPIT.2004.1433800](https://doi.org/10.1109/ISSPIT.2004.1433800).
- [GL11] Thomas Gusenleitner and Gerhard Lettner. “Evaluation of RT-Linux on different hardware platforms for the use in industrial machinery control.” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011. ISBN: 978-3-0003-6193-7.
- [Gle12] Thomas Gleixner. “kthread: Implement park/unpark facility.” In: *LWN.net* (June 5, 2012). URL: <http://lwn.net/Articles/500338> (visited on Oct. 7, 2014).
- [GM08] Luis Claudio R. Gonçalves and Arnaldo Carvalho de Melo. “Application Testing under Realtime Linux.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, July 23–26, 2008). Ed. by John W. Lockhart, Gurhan Ozen, Eugene Teo, Kyle McMartin, Jake Edge, et al. Vol. 1. 2008, pp. 143–150. URL: <http://kernel.org/doc/ols/> (visited on May 7, 2014).
- [GN06] Thomas Gleixner and Douglas Niehaus. “Hrtimers and Beyond: Transforming the Linux Time Subsystems.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, July 20–23, 2006). Ed. by John W. Lockhart, David M. Fellows, and Kyle McMartin. Vol. 1. 2006, pp. 333–346. URL: <http://kernel.org/doc/ols/> (visited on May 7, 2014).
- [Goe+02] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. “Supporting Time-Sensitive Applications on a Commodity OS.” In: *SIGOPS Operat. Syst. Rev.* 36.SI (Dec. 2002), pp. 165–180. ISSN: 0163-5980. DOI: [10.1145/844128.844144](https://doi.org/10.1145/844128.844144).
- [Gol+13] Benjamin Goldschmidt, Christoph W. Lerche, Torsten Solf, André Salomon, Fabian Kiessling, et al. “Towards Software-Based Real-Time Singles and Coincidence Processing of Digital PET Detector Raw Data.” In: *IEEE Trans. Nucl. Sci.* 60.3 (June 2013), pp. 1550–1559. ISSN: 0018-9499. DOI: [10.1109/TNS.2013.2252193](https://doi.org/10.1109/TNS.2013.2252193).
- [Gor07] Mel Gorman. *Understanding The Linux Virtual Memory Manager*. 2007. URL: <https://www.kernel.org/doc/gorman/pdf/understand.pdf> (visited on May 7, 2014).

Bibliography

- [GR05] Joël Goossens and Pascal Richard. “Overview of real-time scheduling problems.” In: *9th Int. Workshop on Project Management and Scheduling*. (Nancy, France, Apr. 26–28, 2004). Ed. by Marie-Claude Portmann. Invited session. 2005.
- [Gre14] Brendan Gregg. “Ftrace: The hidden light switch.” In: *LWN.net* (Aug. 13, 2014). URL: <http://lwn.net/Articles/608497> (visited on Oct. 7, 2014).
- [GSC07] Corey Gough, Suresh Siddha, and Ken Chen. “Kernel Scalability—Expanding the Horizon Beyond Fine Grain Locks.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, June 27–30, 2007). Ed. by John W. Lockhart, Gurhan Ozen, John Feeney, Len DiMaggio, and John Poelstra. Vol. 1. 2007, pp. 153–165. URL: <http://kernel.org/doc/ols/> (visited on May 7, 2014).
- [Gua+09] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. “Cache-Aware Scheduling and Analysis for Multicores.” In: *Proc. 7th ACM Int. Conf. on Embedded Software (EMSOFT)*. (Grenoble, France, Oct. 12–16, 2009). New York, NY, USA: ACM Assoc. for Computing Machinery, 2009, pp. 245–254. ISBN: 978-1-60558-627-4. DOI: [10.1145/1629335.1629369](https://doi.org/10.1145/1629335.1629369).
- [Gui+11] Alberto Guiggiani, Michele Basso, Massimo Vassalli, and Francesco Difato. “Realtime Suite: a step-by-step introduction to the world of real-time signal acquisition and conditioning.” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011. ISBN: 978-3-0003-6193-7.
- [Hag05] William von Hagen. “Real-Time and Performance Improvements in the 2.6 Linux Kernel.” In: *Linux J.* (Issue 134 June 2005). ISSN: 1075-3583. URL: <http://www.linuxjournal.com/article/8041> (visited on May 7, 2014).
- [Hal+00] Wolfgang A. Halang, Roman Gumzej, Matjaž Colnarič, and Marjan Družovec. “Measuring the Performance of Real-Time Systems.” In: *Real-Time Syst.* 18 (1 2000), pp. 59–68. ISSN: 0922-6443. DOI: [10.1023/A:1008102611034](https://doi.org/10.1023/A:1008102611034).
- [Her91] Maurice Herlihy. “Wait-Free Synchronization.” In: *ACM Trans. Program. Lang. Syst.* 13.1 (Jan. 1991), pp. 124–149. ISSN: 0164-0925. DOI: [10.1145/114005.102808](https://doi.org/10.1145/114005.102808).
- [Her93] Maurice Herlihy. “A Methodology for Implementing Highly Concurrent Data Objects.” In: *ACM Trans. Program. Lang. Syst.* 15.5 (Nov. 1993), pp. 745–770. ISSN: 0164-0925. DOI: [10.1145/161468.161469](https://doi.org/10.1145/161468.161469).

- [HH89] V. P. Holmes and D. L. Harris. “A designer’s perspective of the Hawk multiprocessor operating system kernel.” In: *SIGOPS Operat. Syst. Rev.* 23.3 (July 1989), pp. 158–172. ISSN: 0163-5980. DOI: [10.1145/71021.71030](https://doi.org/10.1145/71021.71030).
- [HHW98] Hermann Härtig, Michael Hohmuth, and Jean Wolter. “Taming Linux.” In: *Proc. 5th Australasian Conf. on Parallel and Real-Time Systems*. Berlin, Germany: Springer, 1998. ISBN: 9814021229.
- [HM08] Serge E. Hallyn and Andrew G. Morgan. “Linux Capabilities: making them work.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, July 23–26, 2008). Ed. by John W. Lockhart, Gurhan Ozen, Eugene Teo, Kyle McMartin, Jake Edge, et al. Vol. 1. 2008, pp. 163–172. URL: <http://kernel.org/doc/ols/> (visited on May 7, 2014).
- [HMN09] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. “Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems.” In: *Proc. 42nd Annu. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*. (New York, NY, USA, Dec. 12–16, 2009). New York, NY, USA: ACM Assoc. for Computing Machinery, 2009, pp. 413–422. ISBN: 978-1-60558-798-1. DOI: [10.1145/1669112.1669165](https://doi.org/10.1145/1669112.1669165).
- [HP07] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. 4th ed. San Francisco, CA, USA: Morgan Kaufmann, 2007. ISBN: 978-0-12-370490-0.
- [HP11] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2011. ISBN: 978-0-12-383872-8.
- [HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Elsevier Inc., 2008. ISBN: 978-0-12-370591-4.
- [HW10] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton, FL, USA: CRC Press, Inc., 2010. ISBN: 978-1-4398-1192-4.
- [Hwa93] Kai Hwang. *Advanced Computer Architecture. Parallelism, Scalability, Programmability*. New York, NY, USA: McGraw-Hill, Inc., 1993. ISBN: 0-07-113342-9.
- [HX96] Kai Hwang and Zhiwei Xu. “Scalable Parallel Computers for Real-Time Signal Processing.” In: *IEEE Signal Process. Mag.* 13.4 (July 1996), pp. 50–66. ISSN: 1053-5888. DOI: [10.1109/79.526898](https://doi.org/10.1109/79.526898).
- [Intel08a] *Intel Processor Identification and the CPUID Instruction. Application Note 485*. 241618-033. Santa Clara, CA, USA: Intel Corp., Nov. 2008.

Bibliography

- [Intel08b] *How Much Performance Do You Need for 3D Medical Imaging?* 315896. Santa Clara, CA, USA: Intel Corp., 2008. URL: <http://download.intel.com/design/embedded/medical-solutions/315896.pdf> (visited on July 1, 2014).
- [Intel09] *An Introduction to the Intel QuickPath Interconnect*. 320412-001US. Santa Clara, CA, USA: Intel Corp., Jan. 2009.
- [Intel13a] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 1: *Basic Architecture*. 253665-045US. Santa Clara, CA, USA: Intel Corp., Jan. 2013.
- [Intel13b] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 2: *Instruction Set Reference*. 325383-045US. Santa Clara, CA, USA: Intel Corp., Jan. 2013.
- [Intel13c] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3: *System Programming Guide*. 325384-045US. Santa Clara, CA, USA: Intel Corp., Jan. 2013.
- [Intel13d] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 248966-028. Santa Clara, CA, USA: Intel Corp., July 2013.
- [Intel13e] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Santa Clara, CA, USA: Intel Corp., Jan. 2013.
- [Ish+13] Takuya Ishikawa, Toshikazu Kato, Shinya Honda, and Hiroaki Takada. "Investigation and Improvement on the Impact of TLB misses in Real-Time Systems." In: *Int. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT) in conjunction with the 25th Euromicro Conf. on Real-Time Systems*. (Paris, France, July 9, 2013). Ed. by Andrea Bastoni and Shinpei Kato. 2013. URL: http://ert1.jp/~shinpei/conf/ospert13/OSPERT13_proceedings_web.pdf (visited on May 7, 2014).
- [ISO06] ISO/IEC. *Technical Report on C++ Performance*. Tech. rep. ISO/IEC TR 18015:2006(E). Geneva, Switzerland, 2006.
- [Jai13] Gitu Jain. "Programming Languages." In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 9. ISBN: 978-0-12-416018-7.
- [Jal+10] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. "Achieving Non-Inclusive Cache Performance with Inclusive Caches. Temporal Locality Aware (TLA) Cache Management Policies." In: *Proc. 4^{3rd} Annu. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*. (Atlanta, GA, USA, Dec. 4–8, 2010). Washington, DC, USA: IEEE Computer Society, 2010, pp. 151–162. DOI: [10.1109/MICRO.2010.52](https://doi.org/10.1109/MICRO.2010.52).

- [JCW12] Jhing-Fa Wang, Po-Chun Lin, and Bo-Wen Chen. “Design and Applications of Embedded Systems for Speech Processing.” In: *Embedded Systems. High Performance Systems, Applications and Projects*. Ed. by Kiyofumi Tanaka. Rijeka, Croatia: InTech, 2012. Chap. 9. ISBN: 978-953-51-0350-9. DOI: [10.5772/2684](https://doi.org/10.5772/2684).
- [JLT85] E. Douglas Jensen, C. Douglas Locke, and Hideyuki Tokuda. “A Time-Driven Scheduling Model for Real-Time Operating Systems.” In: *Proc. 6th IEEE Real-Time Systems Symp. (RTSS)*. (San Diego, CA, USA, Dec. 3–6, 1985). Washington, DC, USA: IEEE Computer Society, 1985, pp. 112–122. ISBN: 0-8186-0675-4.
- [Jos+10] Adhijaj Joshi, Swapnil Pimpale, Mandar Naik, Swapnil Rathi, and Kiran Pawar. “Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, July 13–16, 2010). Ed. by Robyn Bergeron. 2010, pp. 101–107. URL: <http://kernel.org/doc/mirror/ols2010.pdf> (visited on May 7, 2014).
- [kad02] kad. “Handling Interrupt Descriptor Table for fun and profit.” In: *Phrack mag.* (59 July 2002). URL: <http://www.phrack.org/issues.html?issue=59&id=4> (visited on May 7, 2014).
- [KB11] Markus Klotzbücher and Herman Bruyninckx. “Hard real-time Control and Coordination of Robot Tasks using Lua.” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011. ISBN: 978-3-0003-6193-7.
- [KCS04] Seongbeom Kim, Dhruva Chandra, and Yan Solihin. “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture.” In: *Proc. 13th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. (Antibes Juan-les-Pins, France, Sept. 29–Oct. 3, 2004). Washington, DC, USA: IEEE Computer Society, 2004, pp. 111–122. ISBN: 0-7695-2229-7. DOI: [10.1109/PACT.2004.15](https://doi.org/10.1109/PACT.2004.15).
- [Ker10] Michael Kerrisk. *The Linux Programming Interface*. San Francisco, CA, USA: No Starch Press, 2010. ISBN: 978-1-59327-220-3.
- [KG93] H. Kopetz and G. Grünsteidl. “TTP - A time-triggered protocol for fault-tolerant real-time systems.” In: *Dig. of Papers 23rd Int. Symp. on Fault-Tolerant Computing (FTCS)*. (Toulouse, France, June 22–24, 1993). Washington, DC, USA: IEEE Computer Society, June 1993, pp. 524–533. DOI: [10.1109/FTCS.1993.627355](https://doi.org/10.1109/FTCS.1993.627355).

Bibliography

- [Kis05] Jan Kiszka. “The Real-Time Driver Model and First Applications.” In: *Proc. 7th Real-Time Linux Workshop (RTLWS)*. (Lille, France, Nov. 3–4, 2005). 2005.
- [Kis09] Jan Kiszka. “Towards Linux as a Real-Time Hypervisor.” In: *Proc. 11th Real-Time Linux Workshop (RTLWS)*. (Dresden, Germany, Sept. 28–30, 2009). 2009, pp. 205–214.
- [Kis13a] Jan Kiszka. *Static System Partitioning and KVM*. Online. Presentation slides: KVM Forum, Edinburgh, Scotland, October 21+22. Edinburgh, Scotland, Oct. 21, 2013. URL: <http://www.linux-kvm.org/wiki/images/b/b1/Kvm-forum-2013-Static-Partitioning.pdf> (visited on May 7, 2014).
- [Kis13b] Jan Kiszka. *Virtualizing Real-time Systems with Linux*. Online. Presentation slides: LinuxCon Japan, Tokyo, May 29–31. Tokyo, Japan, May 30, 2013. URL: http://events.linuxfoundation.org/sites/events/files/lcjp13_kiszka.pdf (visited on May 7, 2014).
- [Kle04] Andi Kleen. *An NUMA API for Linux*. White Paper. Online. SUSE Labs, Aug. 2004. URL: <http://halobates.de/numaapi3.pdf> (visited on Sept. 2, 2014).
- [KLW00] Tei-Wei Kuo, Kwei-Jay Lin, and Yu-Chung Wang. “An Open Real-Time Environment for Parallel and Distributed Systems.” In: *Proc. 20th Int. Conf. on Distributed Computing Systems (ICDCS)*. (Taipei, Taiwan, Apr. 10–13, 2000). Washington, DC, USA: IEEE Computer Society, 2000, pp. 206–213. DOI: [10.1109/ICDCS.2000.840931](https://doi.org/10.1109/ICDCS.2000.840931).
- [Kop08] Hermann Kopetz. “The Rationale for Time-Triggered Ethernet.” In: *Proc. 29th IEEE Real-Time Systems Symp. (RTSS)*. (Barcelona, Spain, Nov. 30–Dec. 3, 2008). Invited keynote paper. Washington, DC, USA: IEEE Computer Society, 2008, pp. 3–11. URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4700407> (visited on May 7, 2014).
- [Kop91] Hermann Kopetz. “Event-Triggered versus Time-Triggered Real-Time Systems.” In: *Proc. Int. Workshop Operating Systems of the 90s and Beyond*. (Dagstuhl Castle, Germany, July 8–12, 1991). Ed. by Arthur I. Karshmer and Jürgen Nehmer. Vol. 563. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 1991, pp. 86–101. ISBN: 3-540-54987-0. DOI: [10.1007/BFb0024530](https://doi.org/10.1007/BFb0024530).
- [Kop93] Hermann Kopetz. “Should Responsive Systems be Event-Triggered or Time-Triggered?” In: *IEICE Trans. Inf. Syst.* 76.11 (Nov. 1993), pp. 1325–1332. ISSN: 0916-8532.

- [Kop97] Hermann Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Norwell, MA, USA: Kluwer Academic Publishers, 1997. ISBN: 0-7923-9894-7.
- [Kop98] Herman Kopetz. “Component-based design of large distributed real-time systems.” In: *Control Eng. Pract.* 6.1 (1998), pp. 53–60. ISSN: 0967-0661. DOI: [10.1016/S0967-0661\(97\)10047-8](https://doi.org/10.1016/S0967-0661(97)10047-8).
- [Kra+09] Charles Krasic, Mayukh Saubhasik, Anirban Sinha, and Ashvin Goel. “Fair and Timely Scheduling via Cooperative Polling.” In: *Proc. 4th ACM European Conf. on Computer Systems (EuroSys)*. (Nuremberg, Germany, Mar. 31–Apr. 3, 2009). New York, NY, USA: ACM Assoc. for Computing Machinery, 2009, pp. 103–116. ISBN: 978-1-60558-482-9. DOI: [10.1145/1519065.1519077](https://doi.org/10.1145/1519065.1519077).
- [Kri+93] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. “A Fair Fast Scalable Reader-Writer Lock.” In: *Proc. of the Int. Conf. on Parallel Processing (ICPP)*. (Syracuse, NY, USA, Aug. 16–20, 1993). Vol. 2. Washington, DC, USA: IEEE Computer Society, 1993, pp. 201–204. DOI: [10.1109/ICPP.1993.21](https://doi.org/10.1109/ICPP.1993.21).
- [KRI10] Shinpei Kato, Ragnathan Rajkumar, and Yutaka Ishikawa. “AIRS: Supporting Interactive Real-Time Applications on Multicore Platforms.” In: *Proc. 22nd Euromicro Conf. on Real-Time Systems (ECRTS)*. (Brussels, Belgium, July 6–9, 2010). Washington, DC, USA: IEEE Computer Society, 2010, pp. 47–56. DOI: [10.1109/ECRTS.2010.33](https://doi.org/10.1109/ECRTS.2010.33).
- [KSB10] Markus Klotzbücher, Peter Soetens, and Herman Bruyninckx. “OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages.” In: *Int. Workshop on Dynamic languages for Robotic and Sensors (DYROS)*. (Darmstadt, Germany, Nov. 15–18, 2010). Ed. by Emanuele Menegatti. 2010, pp. 284–289. URL: <https://www.sim.informatik.tu-darmstadt.de/simpar/ws/sites/DYROS2010/03-DYROS.pdf> (visited on May 7, 2014).
- [Küh98] C. Kühnel. *AVR RISC Microcontroller Handbook*. Woburn, MA, USA: Butterworth-Heinemann, 1998. ISBN: 978-0-7506-9963-1.
- [KW05] J. Kiszka and B. Wagner. “RTnet - a flexible hard real-time networking framework.” In: *10th IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*. (Catania, Italy, Sept. 19–22, 2005). Vol. 1. Washington, DC, USA: IEEE Computer Society, Sept. 2005, pp. 449–456. DOI: [10.1109/ETFA.2005.1612559](https://doi.org/10.1109/ETFA.2005.1612559).

Bibliography

- [KW07] Jan Kiszka and Bernardo Wagner. “Modelling Security Risks in Real-Time Operating Systems.” In: *5th IEEE Int. Conf. on Industrial Informatics*. (Vienna, Austria, July 23–27, 2007). Vol. 1. Washington, DC, USA: IEEE Computer Society, June 2007, pp. 125–130. DOI: [10.1109/INDIN.2007.4384743](https://doi.org/10.1109/INDIN.2007.4384743).
- [KY08] Shinpei Kato and Nobuyuki Yamasaki. “Modular Real-Time Linux.” In: *Proc. 10th Real-Time Linux Workshop (RTLWS)*. (Colotlán, Jalisco, Mexico, Sept. 28–30, 2008). 2008.
- [Lab97] Jean J. Labrosse. “Designing with Real-Time Kernels.” In: *Proc. of Embedded Systems Conf. East*. (Boston, Massachusetts, USA, Mar. 10–12, 1997). San Francisco, CA, USA: Miller Freeman Inc., 1997, pp. 379–389. ISBN: 0-8793-0388-3.
- [Lal13] Sanjay Lal. “Bare-Metal Systems.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 15. ISBN: 978-0-12-416018-7.
- [Lam09] Christopher Lameter. “Shoot first and stop the OS noise.” In: *Proc. Linux Symp. (OLS)*. (Montreal, QC, Canada, July 13–17, 2009). Ed. by Robyn Bergeron, Chris Dukes, Jonas Fonseca, and John Hawley. 2009, pp. 479–489. URL: <http://linuxsymposium.org/2009/ls-2009-proceedings.pdf> (visited on May 7, 2014).
- [Lam77] Leslie Lamport. “Concurrent Reading and Writing.” In: *Commun. ACM* 20.11 (Nov. 1977), pp. 806–811. ISSN: 0001-0782. DOI: [10.1145/359863.359878](https://doi.org/10.1145/359863.359878).
- [LCB00] Guiseppe Lipari, John Carpenter, and Sanjoy Baruah. “A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments.” In: *Proc. 21st IEEE Real-Time Systems Symp. (RTSS)*. (Orlando, FL, USA, Nov. 27–30, 2000). Washington, DC, USA: IEEE Computer Society, 2000, pp. 217–226. DOI: [10.1109/REAL.2000.896011](https://doi.org/10.1109/REAL.2000.896011).
- [Lei+07] Bernhard Leiner, Martin Schlager, Roman Obermaisser, and Bernhard Huber. “A Comparison of Partitioning Operating Systems for Integrated Systems.” In: *26th Int. Conf. Computer Safety, Reliability, and Security (SAFECOMP)*. (Nuremberg, Germany, Sept. 18–21, 2007). Ed. by Francesca Saglietti and Norbert Oster. Vol. 4680. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 2007, pp. 342–355. ISBN: 978-3-540-75100-7. DOI: [10.1007/978-3-540-75101-4_33](https://doi.org/10.1007/978-3-540-75101-4_33).

- [Lew+07] Mark Lewandowski, Mark J. Stanovich, Theodore P. Baker, Kartik Gopalan, and An-I Andy Wang. “Modeling device driver effects in real-time schedulability analysis: Study of a network driver.” In: *Proc. of 13th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*. (Bellevue, WA, USA, Apr. 3–6, 2007). Washington, DC, USA: IEEE Computer Society, 2007, pp. 57–68. DOI: [10.1109/RTAS.2007.18](https://doi.org/10.1109/RTAS.2007.18).
- [Lew91] Donald Lewine. *POSIX Programmer’s Guide. Writing Portable UNIX Programs*. Sebastopol, CA, USA: O’Reilly, 1991. ISBN: 0-937175-73-0.
- [LHH97] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. “OS-Controlled Cache Predictability for Real-Time Systems.” In: *Proc. of 3rd IEEE Real-Time Technology and Applications Symp. (RTAS)*. (Montréal, QC, Canada, June 9–11, 1997). Washington, DC, USA: IEEE Computer Society, 1997, pp. 213–224. DOI: [10.1109/RTAS.1997.601360](https://doi.org/10.1109/RTAS.1997.601360).
- [Li+04] Peng Li, Binoy Ravindran, Syed Suhaib, and Shahrooz Feizabadi. “A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems.” In: *IEEE Trans. Softw. Eng.* 30.9 (Sept. 2004), pp. 613–629. ISSN: 0098-5589. DOI: [10.1109/TSE.2004.45](https://doi.org/10.1109/TSE.2004.45).
- [Lic+08] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, et al. “Predictable Programming on a Precision Timed Architecture.” In: *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. (Atlanta, GA, USA, Oct. 19–24, 2008). New York, NY, USA: ACM Assoc. for Computing Machinery, 2008, pp. 137–146. ISBN: 978-1-60558-469-0. DOI: [10.1145/1450095.1450117](https://doi.org/10.1145/1450095.1450117).
- [Liu00] Jane W. S. Liu. *Real-Time Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2000. ISBN: 0-13-099651-3.
- [LL73] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.” In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: [10.1145/321738.321743](https://doi.org/10.1145/321738.321743).
- [LLX09] Tiantian Liu, Minming Li, and Chun Jason Xue. “Minimizing WCET for Real-Time Embedded Systems via Static Instruction Cache Locking.” In: *Proc. of 15th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*. (San Francisco, CA, USA, Apr. 13–16, 2009). Washington, DC, USA: IEEE Computer Society, 2009, pp. 35–44. DOI: [10.1109/RTAS.2009.11](https://doi.org/10.1109/RTAS.2009.11).
- [Lov10] Robert Love. *Linux Kernel Development*. 3rd ed. Boston, MA, USA: Pearson Education Inc., 2010. ISBN: 978-0-672-32946-3.

Bibliography

- [LQ12] Chai Lina and Zhou Qiang. “The Design and Implementation of Real-time HILS Based on RTX Platform.” In: *10th IEEE Int. Conf. on Industrial Informatics (INDIN)*. (Beijing, China, July 25–27, 2012). Washington, DC, USA: IEEE Computer Society, 2012, pp. 276–280. DOI: [10.1109/INDIN.2012.6301215](https://doi.org/10.1109/INDIN.2012.6301215).
- [LS04] Edya Ladan-Mozes and Nir Shavit. “An Optimistic Approach to Lock-Free FIFO Queues.” In: *Proc. 18th Int. Conf. Distributed Computing (DISC)*. (Amsterdam, Netherlands, Oct. 4, 2004–Oct. 7, 2010). Ed. by Rachid Guerraoui. Vol. 3274. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 2004, pp. 117–131. ISBN: 3-540-23306-7. DOI: [10.1007/978-3-540-30186-8_9](https://doi.org/10.1007/978-3-540-30186-8_9).
- [LS99] Thomas Lundqvist and Per Stenstrom. “Timing Anomalies in Dynamically Scheduled Microprocessors.” In: *Proc. 20th IEEE Real-Time Systems Symp. (RTSS)*. (Phoenix, AZ, USA, Dec. 1–3, 1999). Washington, DC, USA: IEEE Computer Society, 1999, pp. 12–21. DOI: [10.1109/REAL.1999.818824](https://doi.org/10.1109/REAL.1999.818824).
- [Lue12] Kenn R. Luecke. “Software Development for Parallel and Multi-Core Processing.” In: *Embedded Systems. High Performance Systems, Applications and Projects*. Ed. by Kiyofumi Tanaka. Rijeka, Croatia: InTech, 2012. Chap. 3. ISBN: 978-953-51-0350-9. DOI: [10.5772/2684](https://doi.org/10.5772/2684).
- [Lue13] Ken Luecke. “Tools.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 10. ISBN: 978-0-12-416018-7.
- [Man09] Keith Mannthey. *System Management Interrupt Free Hardware*. Presentation slides: Linux Plumbers Conference, Portland, OR, USA, September 23–25, 2009. URL: <http://linuxplumbersconf.org/2009/slides/Keith-Mannthey-SMI-plumbers-2009.pdf>.
- [Mar+04] Philippe Marquet, Eric Piel, Julien Soula, and Jean-Luc Dekeyser. “Implementation of ARTiS, an asymmetric real-time extension of SMP Linux.” In: *Proc. 6th Real-Time Linux Workshop (RTLWS)*. (Singapore, Nov. 3–5, 2004). 2004.
- [Mar10] Maximilian Marx. “Porting of an Embedded RTOS into a Userspace Process.” Bachelor thesis. Aachen, Germany: RWTH Aachen University, Faculty of Electrical Engineering and Information Technology, Chair for Operating Systems, Aug. 2010.
- [Mas+04] M. Masmano, I. Ripoll, A. Crespo, and J. Real. “TLSF: a new dynamic memory allocator for real-time systems.” In: *Proc. 16th Euromicro Conf. on Real-Time Systems (ECRTS)*. (Catania, Italy, June 30–July 2, 2004).

- Washington, DC, USA: IEEE Computer Society, 2004, pp. 79–88. DOI: [10.1109/EMRTS.2004.1311009](https://doi.org/10.1109/EMRTS.2004.1311009).
- [Mas+08] Miguel Masmano, Ismael Ripoll, Patricia Balbastre, and Alfons Crespo. “A constant-time dynamic storage allocator for real-time systems.” In: *Real-Time Syst.* 40.2 (2008), pp. 149–179. ISSN: 0922-6443. DOI: [10.1007/s11241-008-9052-7](https://doi.org/10.1007/s11241-008-9052-7).
- [MČ08] Pavel Moryc and Jindřich Černohorský. “Task jitter measurement under RTLinux and RTX operating systems, comparison of RTLinux and RTX operating environments.” In: *Int. Multiconf. on Computer Science and Information Technology (IMCSIT)*. (Wisła, Poland, Oct. 20–22, 2008). Ed. by M. Ganzha, M. Paprzycki, and T. Pelech-Pilichowski. Vol. 3. Washington, DC, USA: IEEE Computer Society, 2008, pp. 703–709. ISBN: 978-83-60810-14-9. DOI: [10.1109/IMCSIT.2008.4747319](https://doi.org/10.1109/IMCSIT.2008.4747319).
- [MCAPI11] The Multicore Association. *Multicore Communications API*. Technical Standard. Version V2.015. Mar. 25, 2011.
- [McK+01] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, et al. “Read-Copy Update.” In: *Proc. Ottawa Linux Symp. (OLS)*. (Ottawa, ON, Canada, July 25, 2001–July 28, 2002). 2001. URL: <http://www.linuxsymposium.org/archives/OLS/Reprints-2001/> (visited on May 7, 2014).
- [McK+06] Paul E. McKenney, Dipankar Sarma, Ingo Molnar, and Suparna Bhattacharya. “Extending RCU for Realtime and Embedded Workloads.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, July 20–23, 2006). Ed. by John W. Lockhart, David M. Fellows, and Kyle McMartin. Vol. 2. 2006, pp. 123–138. URL: <http://kernel.org/doc/ols/> (visited on May 7, 2014).
- [McK05a] Paul E. McKenney. “A realtime preemption overview.” In: *LWN.net* (Aug. 10, 2005). URL: <http://lwn.net/Articles/146861> (visited on Oct. 7, 2014).
- [McK05b] Paul E. McKenney. *Attempted summary of “RT patch acceptance” thread, take 2*. Message to Linux Kernel Mailing List. July 11, 2005. URL: <http://article.gmane.org/gmane.linux.kernel/317742> (visited on May 7, 2014).
- [McK07] Paul E. McKenney. “SMP and Embedded Real-Time.” In: *Linux J.* (Issue 153 Jan. 2007). ISSN: 1075-3583. URL: <http://www.linuxjournal.com/article/9361> (visited on Sept. 9, 2014).

Bibliography

- [McK08] Paul E. McKenney. “‘Real Time’ vs. ‘Real Fast’: How to Choose?” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, July 23–26, 2008). Ed. by John W. Lockhart, Gurhan Ozen, Eugene Teo, Kyle McMartin, Jake Edge, et al. Vol. 2. 2008, pp. 57–66. URL: <http://kernel.org/doc/ols/> (visited on May 7, 2014).
- [McK09] Paul E. McKenney. “Deterministic Synchronization in Multicore Systems: the Role of RCU.” In: *Proc. 11th Real-Time Linux Workshop (RTLWS)*. (Dresden, Germany, Sept. 28–30, 2009). 2009, pp. 245–254.
- [McK13a] Paul E. McKenney. “Provide infrastructure for full-system idle.” In: *LWN.net* (July 8, 2013). URL: <http://lwn.net/Articles/558229> (visited on Oct. 7, 2014).
- [McK13b] Paul E. McKenney. “Structured Deferral: Synchronization via Procrastination.” In: *ACM Queue* 11.5 (May 2013), 20:20–20:39. ISSN: 1542-7730. DOI: [10.1145/2488364.2488549](https://doi.org/10.1145/2488364.2488549).
- [McK14a] Paul McKenney. *Bare-Metal Multicore Performance in a General-Purpose Operating System*. Online. Presentation slides: Linux Collaboration Summit, Napa Valley, CA, USA, March 26–28. Napa Valley, CA, USA, Mar. 28, 2014. URL: <http://www2.rdrop.com/~paulmck/scalability/paper/BareMetal.2014.03.28a.pdf> (visited on May 15, 2014).
- [McK14b] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* First print edition. Apr. 3, 2014. URL: <https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html> (visited on July 4, 2014).
- [McK96] Paul E. McKenney. “Selecting Locking Primitives for Parallel Programming.” In: *Commun. ACM* 39.10 (Oct. 1996), pp. 75–82. ISSN: 0001-0782. DOI: [10.1145/236156.236174](https://doi.org/10.1145/236156.236174).
- [MDL13] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. “User-space RCU.” In: *LWN.net* (Nov. 13, 2013). URL: <http://lwn.net/Articles/573424> (visited on Oct. 7, 2014).
- [MDP00] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. “RTAI: Real Time Application Interface.” In: *Linux J.* (Issue 72 Apr. 2000). ISSN: 1075-3583. URL: <http://www.linuxjournal.com/article/3838> (visited on Sept. 9, 2014).
- [Meß11] Sebastian Meßingfeld. “Integration eines optischen 3D-Sensors in eine mobile Roboterplattform.” German. Bachelor thesis. Aachen, Germany: Fachhochschule Aachen, Medizintechnik und Technomathematik, July 2011.

- [Meu+] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. *Top 500 Supercomputer Sites. List of June 2014*. URL: <http://www.top500.org> (visited on Sept. 30, 2014).
- [MFC01] Aloysius K. Mok, Xiang Feng, and Deji Chen. “Resource Partition for Real-Time Systems.” In: *Proc. of 7th IEEE Real-Time Technology and Applications Symp. (RTAS)*. (Taipei, Taiwan, May 30–June 1, 2001). Washington, DC, USA: IEEE Computer Society, 2001, pp. 75–84. DOI: [10.1109/RTAS.2001.929867](https://doi.org/10.1109/RTAS.2001.929867).
- [MG11] Zoltan Majo and Thomas R. Gross. “Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead.” In: *Proc. Int. Symp. on Memory Management (ISMM)*. (San Jose, CA, USA, June 4–5, 2011). New York, NY, USA: ACM Assoc. for Computing Machinery, 2011, pp. 11–20. ISBN: 978-1-4503-0263-0. DOI: [10.1145/1993478.1993481](https://doi.org/10.1145/1993478.1993481).
- [MHH02] Frank Mehnert, Michael Hohmuth, and Hermann Härtig. “Cost and benefit of separate address spaces in real-time operating systems.” In: *Proc. 23rd IEEE Real-Time Systems Symp. (RTSS)*. (Austin, TX, USA, Dec. 3–5, 2002). Washington, DC, USA: IEEE Computer Society, 2002, pp. 124–133. DOI: [10.1109/REAL.2002.1181568](https://doi.org/10.1109/REAL.2002.1181568).
- [Mic04] Maged M. Michael. “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects.” In: *IEEE Trans. Parallel Distrib. Syst.* 15.6 (2004), pp. 491–504. ISSN: 1045-9219. DOI: [10.1109/TPDS.2004.8](https://doi.org/10.1109/TPDS.2004.8).
- [Mit+11] Hitoshi Mitake, Thung-Han Lin, Hiromasa Shimada, Yuki Kinebuchi, Ning Li, et al. “Towards Co-existing of Linux and Real-Time OSes.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, June 13–15, 2011). Ed. by Ralph Siemsen. 2011, pp. 55–68. URL: <http://kernel.org/doc/mirror/ols2011.pdf> (visited on May 7, 2014).
- [MM01] Momtchil Momtchev and Philippe Marquet. *An Open Operating System for Intensive Signal Processing*. Tech. rep. LIFL Report #2001-08. Lille, France: Laboratoire d’informatique fondamentale, Université des sciences et technologies de Lille, Oct. 2001.
- [MMU11] Stefan Metzloff, Jörg Mische, and Theo Ungerer. “A Real-Time Capable Many-Core Model.” In: *Work-in-Progress Proc. 32nd IEEE Real-Time Systems Symposium (RTSS-WiP)*. (Vienna, Austria, Nov. 30, 2011). 2011, pp. 21–24. URL: <http://www.cs.wayne.edu/~fishern/Meetings/wip-rtss2011/WiP-RTSS-2011-Proceedings-Post.pdf> (visited on Sept. 5, 2014).

Bibliography

- [Mol+13] Christoph Molitor, Aandrea Benigni, Alexander Helmedag, Kan Chen, Davide Cali, et al. “Multiphysics Test Bed for Renewable Energy Systems in Smart Homes.” In: *IEEE Trans. Ind. Electron.* 60.3 (2013), pp. 1235–1248. ISSN: 0278-0046. DOI: [10.1109/TIE.2012.2190254](https://doi.org/10.1109/TIE.2012.2190254).
- [Moo06] Gordon E. Moore. “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” In: *Solid-State Circuits Society Newsletter, IEEE* 11.5 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: [10.1109/N-SSC.2006.4785860](https://doi.org/10.1109/N-SSC.2006.4785860).
- [Mor+11] Alessandro Morari, Roberto Gioiosa, Robert W. Wisniewski, Francisco J. Cazorla, and Mateo Valero. “A Quantitative Analysis of OS Noise.” In: *Proc. 2011 IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*. (Anchorage, AK, USA, May 16–20, 2011). Washington, DC, USA: IEEE Computer Society, 2011, pp. 852–863. DOI: [10.1109/IPDPS.2011.84](https://doi.org/10.1109/IPDPS.2011.84).
- [Mos93] David Mosberger. “Memory Consistency Models.” In: *SIGOPS Oper. Syst. Rev.* 27.1 (Jan. 1993), pp. 18–26. ISSN: 0163-5980. DOI: [10.1145/160551.160553](https://doi.org/10.1145/160551.160553).
- [Moy13a] Bryon Moyer, ed. *Real World Multicore Embedded Systems. A Practical Approach*. Oxford, England: Newness, 2013. ISBN: 978-0-12-416018-7.
- [Moy13b] Bryon Moyer. “Introduction and Roadmap.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 1. ISBN: 978-0-12-416018-7.
- [Moy13c] Bryon Moyer. “The Promise and Challenges of Concurrency.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 2. ISBN: 978-0-12-416018-7.
- [Moy13d] Bryon Moyer. “Operating Systems in Multicore Platforms.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 6. ISBN: 978-0-12-416018-7.
- [MS05] Paul E. McKenney and Dipankar Sarma. “Towards Hard Realtime Response from the Linux Kernel on SMP Hardware.” In: *linux.conf.au*. (Canberra, Australia, Apr. 18–23, 2005). 2005.
- [MS13] Bryon Moyer and Paul Stravers. “Partitioning Programs for Multicore Systems.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 11. ISBN: 978-0-12-416018-7.

- [MS91] John M. Mellor-Crummey and Michael L. Scott. “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.” In: *ACM Trans. Comput. Syst.* 9.1 (Feb. 1991), pp. 21–65. ISSN: 0734-2071. DOI: [10.1145/103727.103729](https://doi.org/10.1145/103727.103729).
- [MS96a] Larry W. McVoy and Carl Staelin. “lmbench: Portable Tools for Performance Analysis.” In: *USENIX Annu. Technical Conf.* (San Diego, CA, USA, Jan. 22–26, 1996). Berkeley, CA, USA: USENIX, 1996, pp. 279–294.
- [MS96b] Maged M. Michael and Michael L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.” In: *Proc. 19th Annu. ACM Symp. on Principles of Distributed Computing (PODC)*. (Philadelphia, PA, USA, May 23–26, 1996). New York, NY, USA: ACM Assoc. for Computing Machinery, 1996, pp. 267–275. ISBN: 0-89791-800-2. DOI: [10.1145/248052.248106](https://doi.org/10.1145/248052.248106).
- [MU12] Stefan Metzloff and Theo Ungerer. “Impact of Instruction Cache and Different Instruction Scratchpads on the WCET Estimate.” In: *Proc. 14th IEEE Int. Conf. on High Performance Computing and Communication (HPCC) – 9th IEEE Int. Conf. on Embedded Software and Systems (ICESS)*. (Liverpool, England, June 25–27, 2012). Washington, DC, USA: IEEE Computer Society, June 2012, pp. 1442–1449. DOI: [10.1109/HPCC.2012.211](https://doi.org/10.1109/HPCC.2012.211).
- [Mul13] The Multicore Association. *Multicore Programming Practices*. White Paper. 2013.
- [Nak+12] Kazuhiro Nakamura, Ryo Shimazaki, Masatoshi Yamamoto, Kazuyoshi Takagi, and Naofumi Takagi. “A VLSI Architecture for Output Probability and Likelihood Score Computations of HMM-Based Recognition Systems.” In: *Embedded Systems. High Performance Systems, Applications and Projects*. Ed. by Kiyofumi Tanaka. Rijeka, Croatia: InTech, 2012. Chap. 8. ISBN: 978-953-51-0350-9. DOI: [10.5772/2684](https://doi.org/10.5772/2684).
- [Nam+09] Min-Young Nam, Rodolfo Pellizzoni, Lui Sha, and Richard M. Bradford. “ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs.” In: *14th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*. (Potsdam, Germany, June 2–4, 2009). Washington, DC, USA: IEEE Computer Society, June 2009, pp. 11–22. ISBN: 978-0-7695-3702-3. DOI: [10.1109/ICECCS.2009.31](https://doi.org/10.1109/ICECCS.2009.31).
- [NBP96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. Sebastopol, CA, USA: O’Reilly, 1996. ISBN: 1-56592-115-1.

Bibliography

- [Neu93] John von Neumann. “First draft of a report on the EDVAC.” In: *IEEE Ann. Hist. Comput.* 15.4 (1993), pp. 27–75. ISSN: 1058-6180. DOI: [10.1109/85.238389](https://doi.org/10.1109/85.238389).
- [NKN08] Farhang Nemati, Johan Kraft, and Thomas Nolte. “Towards Migrating Legacy Real-Time Systems to Multi-Core Platforms.” In: *IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*. (Hamburg, Germany, Sept. 15–18, 2008). Washington, DC, USA: IEEE Computer Society, Sept. 2008, pp. 717–720. DOI: [10.1109/ETFA.2008.4638477](https://doi.org/10.1109/ETFA.2008.4638477).
- [NP12] Jan Nowotsch and Michael Paulitsch. “Leveraging Multi-Core Computing Architectures in Avionics.” In: *9th European Dependable Computing Conference (EDCC)*. (Sibiu, Romania, May 8–11, 2012). Washington, DC, USA: IEEE Computer Society, May 2012, pp. 132–143. DOI: [10.1109/EDCC.2012.27](https://doi.org/10.1109/EDCC.2012.27).
- [NW12] Dieter Nazareth and Christian Wurm. “Modellbasierte Entwicklung einer Lichtsteuerung für ein Rapid Prototyping System.” German. In: *Herausforderungen durch Echtzeitbetrieb. Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e. V. (GI)*. (Boppard, Germany, Nov. 22–23, 2012). Ed. by Wolfgang A. Halang. Informatik aktuell. Berlin, Germany: Springer, 2012. ISBN: 978-3-642-24658-6.
- [Obe+99] Kevin M. Obenland, Tiffany Frazier, Jin S. Kim, and John Kowalik. “Comparing the Real-time Performance of Windows NT to an NT Real-time Extension.” In: *Proc. of 5th IEEE Real-Time Technology and Applications Symp. (RTAS)*. (Vancouver, BC, Canada, June 2–4, 1999). Washington, DC, USA: IEEE Computer Society, 1999, pp. 142–151. DOI: [10.1109/RTAS.1999.777669](https://doi.org/10.1109/RTAS.1999.777669).
- [Obe00] Kevin M. Obenland. *The Use of POSIX in Real-time Systems, Assessing its Effectiveness and Performance*. Tech. rep. McLean, VA, USA: The MITRE Corporation, Sept. 2000. URL: http://mitre.org/work/tech_papers/tech_papers_00/obenland_posix/ (visited on May 7, 2014).
- [Ode10] Lena Oden. “Examination of Real-Time Optimization Methods for High Performance Computing.” Diploma thesis. Aachen, Germany: RWTH Aachen University, Faculty of Electrical Engineering and Information Technology, Chair for Operating Systems, Nov. 2010.
- [OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. “A Better x86 Memory Model: x86-TSO.” In: *Proc. 22nd Int. Conf. Theorem Proving in Higher Order Logics (TPHOLs)*. (Munich, Germany, Aug. 17–20, 2009). Ed.

- by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 2009, pp. 391–407. ISBN: 978-3-642-03358-2. DOI: [10.1007/978-3-642-03359-9_27](https://doi.org/10.1007/978-3-642-03359-9_27).
- [PAH04] Martin Pohlack, Ronald Aigner, and Hermann Härtig. *Connecting Real-Time and Non-Real-Time Components*. Tech. rep. TUD-FI04-01. Dresden, Germany: Technische Universität Dresden, Fakultät Informatik, Feb. 2004.
- [Pal+11] Jacob Palczynski, Carsten Weise, Stefan Kowalewski, and Daniel Ulmer. “Estimation of Clock Drift in HiL Testing by Property-Based Conformance Check.” In: *IEEE 4th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*. (Berlin, Germany, Mar. 21–25, 2011). Washington, DC, USA: IEEE Computer Society, 2011, pp. 590–595. ISBN: 978-0-7695-4345-1. DOI: [10.1109/ICSTW.2011.101](https://doi.org/10.1109/ICSTW.2011.101).
- [Pao+13] Marco Paolieri, Jörg Mische, Stefan Metzloff, Mike Gerdes, Eduardo Quinones, et al. “A Hard Real-Time Capable Multi-Core SMT Processor.” In: *ACM Trans. Embed. Comput. Syst.* 12.3 (Apr. 2013), 79:1–79:26. ISSN: 1539-9087. DOI: [10.1145/2442116.2442129](https://doi.org/10.1145/2442116.2442129).
- [Pao10] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. White paper. Santa Clara, CA, USA, Sept. 2010.
- [Paq+09] Jean-Nicolas Paquin, Wei Li, Jean Bélanger, Loïc Schoen, Irène Pérès, et al. “A Modern and Open Real-Time Digital Simulator of All-Electric Ships with a Multi-Platform Co-Simulation Approach.” In: *IEEE Electric Ship Technologies Symp. (ESTS)*. (Baltimore, MD, USA, Apr. 20–22, 2009). Washington, DC, USA: IEEE Computer Society, 2009, pp. 28–35. DOI: [10.1109/ESTS.2009.4906490](https://doi.org/10.1109/ESTS.2009.4906490).
- [PB00] Peter Puschner and Alan Burns. “Guest Editorial: A Review of Worst-Case Execution-Time Analysis.” In: *Real-Time Syst.* 18.2-3 (2000), pp. 115–128. ISSN: 0922-6443. DOI: [10.1023/A:1008119029962](https://doi.org/10.1023/A:1008119029962).
- [PB01] Marco Papini and Paul Baracos. “Real-Time Simulation, Control and HiL with COTS Computing Clusters.” In: *AIAA Modeling and Simulation Technologies Conf.* (Denver, Colorado, USA, Aug. 14–17, 2000). Reston, VA, USA: American Inst. of Aeronautics and Astronautics, 2001. DOI: [10.2514/6.2000-4593](https://doi.org/10.2514/6.2000-4593).
- [PC07] Rodolfo Pellizzoni and Marco Caccamo. “Toward the Predictable Integration of Real-Time COTS based Systems.” In: *Proc. 28th IEEE Real-Time Systems Symp. (RTSS)*. (Tucson, AZ, USA, Dec. 3–6, 2007).

Bibliography

- Washington, DC, USA: IEEE Computer Society, 2007, pp. 73–82. DOI: [10.1109/RTSS.2007.15](https://doi.org/10.1109/RTSS.2007.15).
- [PCI02] PCI Special Interest Group. *PCI Local Bus Specification*. Technical Standard. Version 2.3. Mar. 29, 2002.
- [PCIe10] PCI Special Interest Group. *PCI Express Specification*. Technical Standard. Version 3.0. Nov. 10, 2010.
- [PDL06] Martin Pohlack, Björn Döbel, and Adam Lackorzyński. “Towards Runtime Monitoring in Real-Time Systems.” In: *Proc. 8th Real-Time Linux Workshop (RTLWS)*. (Lanzhou, China, Oct. 12–15, 2006). 2006.
- [Pel+08] Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. “Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems.” In: *Proc. 29th IEEE Real-Time Systems Symp. (RTSS)*. (Barcelona, Spain, Nov. 30–Dec. 3, 2008). Washington, DC, USA: IEEE Computer Society, 2008, pp. 221–231. DOI: [10.1109/RTSS.2008.42](https://doi.org/10.1109/RTSS.2008.42).
- [Pel+10] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. “Worst Case Delay Analysis for Memory Interference in Multicore Systems.” In: *Design, Automation Test in Europe Conf. Exhibition (DATE)*. (Dresden, Germany, Mar. 8–12, 2010). Washington, DC, USA: IEEE Computer Society, Mar. 2010, pp. 741–746. DOI: [10.1109/DATE.2010.5456952](https://doi.org/10.1109/DATE.2010.5456952).
- [Pel+11] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, et al. “A Predictable Execution Model for COTS-based Embedded Systems.” In: *Proc. of 17th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*. (Chicago, IL, USA, Apr. 11–14, 2011). Washington, DC, USA: IEEE Computer Society, 2011, pp. 269–279. DOI: [10.1109/RTAS.2011.33](https://doi.org/10.1109/RTAS.2011.33).
- [Pet02] Stefan M. Petters. “Worst Case Execution Time Estimation for Advanced Processor Architectures.” Doctoral dissertation. Munich, Germany: Technische Universität München, 2002.
- [PF00] Stefan M. Petters and Georg Färber. “Bounding the Execution Time of Real-Time Tasks on Modern Processors.” In: *Proc. 7th Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCISA)*. (Cheju Island, South Korea, Dec. 12–14, 2000). Washington, DC, USA: IEEE Computer Society, 2000, pp. 498–502. DOI: [10.1109/RTCISA.2000.896433](https://doi.org/10.1109/RTCISA.2000.896433).
- [PH11] David A. Patterson and John L. Hennessy. *Computer Organization and Design. The Hardware/Software Interface*. 4th ed. San Francisco, CA, USA: Morgan Kaufmann, 2011. ISBN: 978-0-12-374750-1.

- [PHW02] Richard R. Plant, Nick Hammond, and Tom Whitehouse. “Toward an Experimental Timing Standards Lab: Benchmarking precision in the real world.” In: *Behav. Res. Methods Instrum. Comput.* 34 (2 2002), pp. 218–226. ISSN: 0743-3808. DOI: [10.3758/BF03195446](https://doi.org/10.3758/BF03195446).
- [Pie+04] Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. *Load-Balancing for a Real-Time System Based on Asymmetric Multi-Processing*. Research report LIFL Report #2004-06. Lille, France: Laboratoire d’informatique fondamentale, Université des sciences et technologies de Lille, Apr. 2004.
- [Pie+06] Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. “Asymmetric Scheduling and Load Balancing for Real-Time on Linux SMP.” In: *Revised Selected Papers 6th Int. Conf. Parallel Processing and Applied Mathematics (PPAM)*. (Poznan, Poland, Sept. 11–14, 2005). Ed. by Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski. Vol. 3911. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 2006, pp. 896–903. ISBN: 3-540-34141-2. DOI: [10.1007/11752578_108](https://doi.org/10.1007/11752578_108).
- [Pop+04] Paul Pop, Petru Eles, Zebo Peng, and Traian Pop. “Analysis and Optimization of Distributed Real-Time Embedded Systems.” In: *ACM Trans. Des. Autom. Electron. Syst.* 11.3 (June 2004), pp. 593–625. ISSN: 1084-4309.
- [POSIX08] IEEE Portable Applications Standards Committee. *Portable Operating System Interface (POSIX.1-2008)*. Technical Standard. 2008. URL: <http://pubs.opengroup.org/onlinepubs/9699919799/> (visited on Nov. 29, 2013).
- [PP07] Isabelle Puaut and Christophe Pais. “Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison.” In: *Design, Automation Test in Europe Conf. Exhibition (DATE)*. (Nice, France, Apr. 16–20, 2007). Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–6. DOI: [10.1109/DATE.2007.364510](https://doi.org/10.1109/DATE.2007.364510).
- [PS08] Peter Puschner and Martin Schoeberl. “On Composable System Timing, Task Timing, and WCET Analysis.” In: *8th Int. Workshop on Worst-Case Execution Time Analysis (WCET)*. Ed. by Raimund Kirner. Vol. 8. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2008. ISBN: 978-3-939897-10-1. DOI: [10.4230/OASICS.WCET.2008.1662](https://doi.org/10.4230/OASICS.WCET.2008.1662).
- [Rag+04] Dominique Ragot, Yulen Sadourny, Denis Foueillassar, Philippe Couvee, Léonard Sibille, et al. “Linux for High Performance and Real-Time Computing on SMP Systems.” In: *Proc. 6th Real-Time Linux Workshop (RTLWS)*. (Singapore, Nov. 3–5, 2004). 2004, pp. 105–116.

Bibliography

- [Ras11] Josef Raschen. “Hardware-Independent Inter-Process Communication for Real-Time Applications.” Diploma thesis. Aachen, Germany: RWTH Aachen University, Faculty of Electrical Engineering and Information Technology, Chair for Operating Systems, Nov. 2011.
- [Reb09] Jan Pablo Reble. “Development and Evaluation of Scaling Mechanisms for Synchronisation in Multi-Core Real Time Systems.” Diploma thesis. Aachen, Germany: RWTH Aachen University, Faculty of Electrical Engineering and Information Technology, Chair for Operating Systems, Oct. 2009.
- [Reg02] John Regehr. “Inferring Scheduling Behavior with Hourglass.” In: *USENIX Technical Conf., FREENIX Track*. (Monterey, CA, USA, June 10–15, 2002). Berkeley, CA, USA: USENIX, 2002. URL: <http://static.usenix.org/publications/library/proceedings/usenix02/tech/freenix/regehr.html> (visited on May 7, 2014).
- [Rei+06] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, et al. “A Definition and Classification of Timing Anomalies.” In: *6th Int. Workshop on Worst-Case Execution Time Analysis (WCET)*. (Dresden, Germany, July 4, 2006). Ed. by Frank Mueller. Vol. 4. Open-Access Series in Informatics (OASISs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2006. ISBN: 978-3-939897-03-3. DOI: [10.4230/OASISs.WCET.2006.671](https://doi.org/10.4230/OASISs.WCET.2006.671).
- [Rei+07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. “Timing predictability of cache replacement policies.” In: *Real-Time Syst.* 37.2 (2007), pp. 99–122. ISSN: 0922-6443. DOI: [10.1007/s11241-007-9032-3](https://doi.org/10.1007/s11241-007-9032-3).
- [RH07] Steven Rostedt and Darren V. Hart. “Internals of the RT Patch.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, June 27–30, 2007). Ed. by John W. Lockhart, Gurhan Ozen, John Feeney, Len DiMaggio, and John Poelstra. Vol. 2. 2007, pp. 161–172. URL: <http://kernel.org/doc/ols/> (visited on May 7, 2014).
- [RLA07] Mohan Rajagopalan, Brian T. Lewis, and Todd A. Anderson. “Thread Scheduling for Multi-Core Platforms.” In: *11th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. (San Diego, CA, USA, May 7, 2007–May 9, 2005). Berkeley, CA, USA: USENIX, 2007. URL: <https://www.usenix.org/legacy/events/hotos07/tech/> (visited on May 7, 2014).
- [Rob+03] Andrew Robbie, Graeme Simpkin, John Fulton, and David Craven. “Experiences in developing a Linux Cluster for Real-Time Simulation.” In: *SimTecT 2003*. (Adelaide, Australia, May 26–29, 2003). 2003. URL:

- <https://web.archive.org/web/20060821224125/http://siaa.asn.au/get/2395356957.pdf> (visited on May 7, 2014).
- [Roc08] Marc J. Rochkind. *Advanced UNIX Programming*. 2nd ed. Boston, MA, USA: Pearson Education Inc., 2008. ISBN: 0-13-141154-3.
- [Ros+07] Jakob Rosén, Alexandru Andrei, Petru Eles, and Zebo Peng. “Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip.” In: *Proc. 28th IEEE Real-Time Systems Symp. (RTSS)*. (Tucson, AZ, USA, Dec. 3–6, 2007). Washington, DC, USA: IEEE Computer Society, 2007, pp. 49–60. DOI: [10.1109/RTSS.2007.24](https://doi.org/10.1109/RTSS.2007.24).
- [RS01] John Regehr and John A. Stankovic. “Augmented CPU Reservations: Towards Predictable Execution on General-Purpose Operating Systems.” In: *Proc. of 7th IEEE Real-Time Technology and Applications Symp. (RTAS)*. (Taipei, Taiwan, May 30–June 1, 2001). Washington, DC, USA: IEEE Computer Society, 2001, pp. 141–148. DOI: [10.1109/RTAS.2001.929880](https://doi.org/10.1109/RTAS.2001.929880).
- [RTX09] *Hard Real-Time with InvervalZero RTX on the Windows Platform*. White paper DOC-RTX-004. Waltham, MA, USA: IntervalZero, Inc., Sept. 2009. URL: <http://www.directinsight.co.uk/products/venturcom/RTXWhitePaper-6-09.pdf> (visited on May 7, 2014).
- [RTX12] *InvervalZero RTX 2012 Product Brief*. Data sheet. Waltham, MA, USA: IntervalZero, Inc., 2012. URL: <http://www.directinsight.co.uk/downloads/evc/031/RTX2012ProductBrief.pdf> (visited on May 7, 2014).
- [RW14] Pablo Reble and Georg Wassen. “Towards Predictability of Operating System Supported Communication for PCIe Based Clusters.” In: *EuroPar 2013: Parallel Processing Workshops. Revised Selected Papers BigDataCloud, DIHC, FedICI, HeteroPar, HiBB, LSDVE, MHPC, OMHI, PADABS, PROPER, Resilience, ROME, and UCHPC*. (Aachen, Germany, Aug. 26–27, 2013). Ed. by Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, et al. Vol. 8374. Lect. Notes in Comp. Sci. Berlin, Germany: Springer, 2014, pp. 833–842. ISBN: 978-3-642-54419-4. DOI: [10.1007/978-3-642-54420-0_81](https://doi.org/10.1007/978-3-642-54420-0_81).
- [RWK11] Stefan Richter, Michael Wahler, and Atul Kumar. “A Framework for Component-Based Real-Time Control Applications.” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011. ISBN: 978-3-0003-6193-7.

Bibliography

- [Sau07] Thilo Sauter. “The Continuing Evolution of Integration in Manufacturing Automation.” In: *IEEE Ind. Electron. Mag.* 1.1 (2007), pp. 10–19. ISSN: 1932-4529. DOI: [10.1109/MIE.2007.357183](https://doi.org/10.1109/MIE.2007.357183).
- [SBF04] Jürgen Stohr, Alexander von Bülow, and Georg Färber. “Using State of the Art Multiprocessor Systems as Real-Time Systems – The RECOMS Software Architecture.” In: *Proc. Work-In-Progress Session 16th Euromicro Conf. on Real-Time Systems (ECRTS-WiP)*. (Catania, Italy, June 30–July 2, 2004). Ed. by Steve Goddard. Tech. rep. TR-UNL-CSE-2004-0010. University of Nebraska-Lincoln, Department of Computer Science and Engineering, 2004. URL: <http://cse.unl.edu/~goddard/ecrts04wip/proceedings/> (visited on May 7, 2014).
- [SBF05] Jürgen Stohr, Alexander von Bulow, and Georg Färber. “Bounding Worst-Case Access Times in Modern Multiprocessor Systems.” In: *Proc. 17th Euromicro Conf. on Real-Time Systems (ECRTS)*. (Palma de Mallorca, Spain, July 6–8, 2005). Washington, DC, USA: IEEE Computer Society, 2005, pp. 189–198. DOI: [10.1109/ECRTS.2005.10](https://doi.org/10.1109/ECRTS.2005.10).
- [SCC10] *SCC External Architecture Specification (EAS)*. Revision 1.1. Intel Corp. Santa Clara, CA, USA, Nov. 2010. URL: <http://communities.intel.com/docs/DOC-5852> (visited on Aug. 18, 2014).
- [Sch+09] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, et al. “Parallel, Hardware-Supported Interrupt Handling in an Event-Triggered Real-Time Operating System.” In: *Proc. Int. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*. (Grenoble, France, Oct. 11–16, 2009). New York, NY, USA: ACM Assoc. for Computing Machinery, 2009.
- [Sch+10] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. “Worst-Case Response Time Analysis of Resource Access Models in Multi-core Systems.” In: *Proc. 47th Design Automation Conference (DAC)*. (Anaheim, CA, USA, June 13–18, 2010). New York, NY, USA: ACM Assoc. for Computing Machinery, 2010, pp. 332–337. ISBN: 978-1-4503-0002-5. DOI: [10.1145/1837274.1837359](https://doi.org/10.1145/1837274.1837359).
- [Sch+11] Volkmar Schulz, Bjoern Weissler, Pierre Gebhardt, Torsten Solf, Christoph W. Lerche, et al. “SiPM based preclinical PET/MR Insert for a human 3T MR: first imaging experiments.” In: *IEEE Nuclear Science Symp. and Medical Imaging Conf. (NSS/MIC)*. (Valencia, Spain, Oct. 23–29, 2011). 2011, pp. 4467–4469. DOI: [10.1109/NSSMIC.2011.6152496](https://doi.org/10.1109/NSSMIC.2011.6152496).
- [Sch03] Sebastian Schonberg. “Impact of PCI bus load on applications in a PC architecture.” In: *Proc. 24th IEEE Real-Time Systems Symp. (RTSS)*. (Cancun, Mexico, Dec. 3–5, 2003). Washington, DC, USA:

- IEEE Computer Society, 2003, pp. 430–439. DOI: [10.1109/REAL.2003.1253290](https://doi.org/10.1109/REAL.2003.1253290).
- [Sch13] Frank Schirrmeyer. “Multicore Architectures.” In: *Real World Multi-core Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 3. ISBN: 978-0-12-416018-7.
- [SE11] Simon Schliecker and Rolf Ernst. “Real-time Performance Analysis of Multiprocessor Systems with Shared Memory.” In: *ACM Trans. Embed. Comput. Syst.* 10.2 (Jan. 2011), 22:1–22:27. ISSN: 1539-9087. DOI: [10.1145/1880050.1880058](https://doi.org/10.1145/1880050.1880058).
- [Seb01] Filip Sebek. *Cache Memories and Real-Time Systems*. Tech. rep. MRTC 01/37. Västerås, Sweden: Mälardalen University, Dept. of Computer Engineering, 2001.
- [SFR12] Michael Schmidt, Dietmar Fey, and Marc Reichenbach. “Parallel Embedded Computing Architectures.” In: *Embedded Systems. High Performance Systems, Applications and Projects*. Ed. by Kiyofumi Tanaka. Rijeka, Croatia: InTech, 2012. Chap. 1. ISBN: 978-953-51-0350-9. DOI: [10.5772/2684](https://doi.org/10.5772/2684).
- [SG94] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. 4th ed. Reading, MA, USA: Addison-Wesley, 1994. ISBN: 0-201-59292-4.
- [SGD09] Joachim Schröder, Tilo Gockel, and Rüdiger Dillmann. *Embedded Linux – Das Praxisbuch*. Dordrecht, Netherlands: Springer, 2009. ISBN: 978-3-540-78619-1.
- [Sin+04] Mohit Sindhvani, Tim F. Oliver, Douglas L. Maskell, and T. Srikanthan. “RTOS Acceleration Techniques – Review and Challenges.” In: *Proc. 6th Real-Time Linux Workshop (RTLWS)*. (Singapore, Nov. 3–5, 2004). 2004, pp. 131–136.
- [SL06] Claudio Scordino and Giuseppe Lipari. “Linux and Real-Time: Current Approaches and Future Opportunities.” In: *ANIPLA International Congr.* (Rome, Italy). 2006.
- [SM08] Vivy Suhendra and Tulika Mitra. “Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores.” In: *Proc. 45th ACM/IEEE Design Automation Conf. (DAC)*. (Anaheim, CA, USA, June 8, 2008–June 13, 2007). New York, NY, USA: ACM Assoc. for Computing Machinery, 2008, pp. 300–303. ISBN: 978-1-60558-115-6. DOI: [10.1145/1391469.1391545](https://doi.org/10.1145/1391469.1391545).
- [SMZ11] Sandeep Shukla, Prabhat Mishra, and Zeljko Zilic. “A Brief History of Multiprocessors and EDA.” In: *IEEE Design & Test of Computers* 28.3 (2011), p. 96. ISSN: 0740-7475. DOI: [10.1109/MDT.2011.50](https://doi.org/10.1109/MDT.2011.50).

Bibliography

- [SO12] Renan Augusto Starke and Romulo Silva de Oliveira. “A Heterogeneous Preemptive and Non-preemptive Scheduling Approach for Real-Time Systems on Multiprocessors.” In: *2nd Brazilian Conf. on Critical Embedded Systems (CBSEC)*. (Campinas, SP, Brazil, May 21–25, 2012). Washington, DC, USA: IEEE Computer Society, 2012, pp. 70–75. ISBN: 978-1-4673-1912-6. DOI: [10.1109/CBSEC.2012.9](https://doi.org/10.1109/CBSEC.2012.9).
- [SO13] Renan Augusto Starke and Rômulo S. Oliveira. “System-Management-Mode in Real-Time PC-Based Control Applications.” In: *J. Contr. Autom. Electr. Syst.* 24.4 (2013), pp. 430–438. ISSN: 2195-3880. DOI: [10.1007/s40313-013-0053-y](https://doi.org/10.1007/s40313-013-0053-y).
- [Sou+11] Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, and Björn Andersson. “On the implementation of real-time slot-based task-splitting scheduling algorithms for multiprocessor systems.” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011, pp. 207–218. ISBN: 978-3-0003-6193-7.
- [SR89] John A. Stankovic and Krithi Ramamritham. “The Spring kernel: a new paradigm for real-time operating systems.” In: *SIGOPS Operat. Syst. Rev.* 23.3 (July 1989), pp. 54–71. ISSN: 0163-5980. DOI: [10.1145/71021.71024](https://doi.org/10.1145/71021.71024).
- [SRH99] David Stepner, Nagarajan Rajan, and David Hui. “Embedded Application Design Using a Real-Time OS.” In: *Proc. 36th Annu. Design Automation Conf. (DAC)*. (New Orleans, LA, USA, June 21, 1999–June 25, 1998). New York, NY, USA: ACM Assoc. for Computing Machinery, 1999, pp. 151–156. ISBN: 1-58113-092-9. DOI: [10.1109/DAC.1999.781301](https://doi.org/10.1109/DAC.1999.781301).
- [Sri+98] Balaji Srinivasan, Shyamalan Pather, Robert Hill, Furquan Ansari, and Douglas Niehaus. “A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software.” In: *Proc. of 4th IEEE Real-Time Technology and Applications Symp. (RTAS)*. (Denver, CO, USA, June 3–5, 1998). Washington, DC, USA: IEEE Computer Society, 1998, pp. 112–119. DOI: [10.1109/RTAS.1998.683194](https://doi.org/10.1109/RTAS.1998.683194).
- [SRL90] Lui Sha, Raguathan Rajkumar, and John P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization.” In: *IEEE Trans. Comput.* 39.9 (Sept. 1990), pp. 1175–1185. ISSN: 0018-9340. DOI: [10.1109/12.57058](https://doi.org/10.1109/12.57058).
- [SS11] Philippe Stellwag and Wolfgang Schröder-Preikschat. “Challenges in Real-Time Synchronization.” In: *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*. (Berkeley, CA, USA, May 26–27, 2011). Ed. by

- Michael McCool and Mendel Rosenblum. Berkeley, CA, USA: USENIX, 2011.
- [SS72] Michael D. Schroeder and Jerome H. Saltzer. “A Hardware Architecture for Implementing Protection Rings.” In: *Commun. ACM* 15.3 (Mar. 1972), pp. 157–170. ISSN: 0001-0782. DOI: [10.1145/361268.361275](https://doi.org/10.1145/361268.361275).
- [SS95] Rafael H. Saavedra and Alan Jay Smith. “Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes.” In: *IEEE Trans. Comput.* 44.10 (Oct. 1995), pp. 1223–1235. ISSN: 0018-9340. DOI: [10.1109/12.467697](https://doi.org/10.1109/12.467697).
- [SSL09] Philippe Stellwag, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “An Asynchronous Nonblocking Coordination and Synchronization Protocol for a Parallel Robotic Control Kernel.” In: *Proc. 2nd Workshop on Isolation and Integration in Embedded Systems (IIES)*. (Nuremberg, Germany, Mar. 31, 2009). New York, NY, USA: ACM Assoc. for Computing Machinery, 2009, pp. 7–12. ISBN: 978-1-60558-464-5. DOI: [10.1145/1519130.1519132](https://doi.org/10.1145/1519130.1519132).
- [SSP08] Vaidyanathan Srinivasan, Gautham R Shenoy, and Venkatesh Pallipadi. “Energy-aware task and interrupt management in Linux.” In: *Proc. Linux Symp. (OLS)*. (Ottawa, ON, Canada, July 23–26, 2008). Ed. by John W. Lockhart, Gurhan Ozen, Eugene Teo, Kyle McMartin, Jake Edge, et al. Vol. 2. 2008, pp. 187–198. URL: <http://kernel.org/doc/ols/> (visited on May 7, 2014).
- [ST93] Stefan Savage and Hideyuki Tokuda. “Real-Time Mach Timers: Exporting Time to the User.” In: *3rd USENIX MACH III Symp. (MACH)*. (Santa Fe, NM, USA, Apr. 1993). Berkeley, CA, USA: USENIX, 1993, pp. 221–232. URL: <http://www-cse.ucsd.edu/~savage/papers/Machnix93.pdf> (visited on May 7, 2014).
- [Sta09] William Stallings. *Operating Systems. Internals and Design Principles*. 6th ed. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2009. ISBN: 978-0-13-603337-0.
- [Sta88] John A. Stankovic. “Misconceptions About Real-Time Computing. A Serious Problem for Next-Generation Systems.” In: *Computer* 21.10 (Oct. 1988), pp. 10–19. ISSN: 0018-9162. DOI: [10.1109/2.7053](https://doi.org/10.1109/2.7053).
- [Ste+95] Thomas L. Sterling, Danies Savarese, Donald J. Becker, John E. Dorband, Udaya A. Ranawake, et al. “BEOWULF: A Parallel Workstation for Scientific Computation.” In: *Proc. of the Int. Conf. on Parallel Processing (ICPP)*. (Urbana-Champaign, IL, USA, Aug. 14–18, 1995). Vol. 1. Washington, DC, USA: CRC Press, 1995. ISBN: 0-8493-2615-X.

Bibliography

- [Ste06] Neil Stewart. “A PC parallel port button box provides millisecond response time accuracy under Linux.” In: *Behav. Res. Methods* 38 (1 2006), pp. 170–173. ISSN: 1554-351X. DOI: [10.3758/BF03192764](https://doi.org/10.3758/BF03192764).
- [Sto06] Jürgen Stohr. “Auswirkungen der Peripherieanbindung auf das Realzeitverhalten PC-basierter Multiprozessorsysteme.” German. Doctoral dissertation. Munich, Germany: Fakultät für Elektrotechnik und Informationstechnik, TU München, Mar. 2006.
- [Sto13] Neal Stollon. “Multicore Debug.” In: *Real World Multicore Embedded Systems. A Practical Approach*. Ed. by Bryon Moyer. Oxford, England: Newness, 2013. Chap. 16. ISBN: 978-0-12-416018-7.
- [Sut05] Herb Sutter. “The Free Lunch Is Over. A Fundamental Turn Toward Concurrency in Software.” In: *Dr. Dobbs’s Journal* 30.3 (Mar. 2005). ISSN: 1044-789X.
- [SysV10] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell, eds. *System V Application Binary Interface. AMD64 Architecture Processor Supplement*. Technical Standard. Sept. 3, 2010.
- [SysV95a] *System V Interface Definition*. 4th ed. Vol. 1. Provo, UT, USA: Novell, Inc., June 15, 1995. URL: <http://www.sco.com/developers/devspecs/> (visited on May 7, 2014).
- [SysV95b] *System V Interface Definition*. 4th ed. Vol. 2. Provo, UT, USA: Novell, Inc., June 15, 1995. URL: <http://www.sco.com/developers/devspecs/> (visited on May 7, 2014).
- [SysV95c] *System V Interface Definition*. 4th ed. Vol. 3. Provo, UT, USA: Novell, Inc., June 15, 1995. URL: <http://www.sco.com/developers/devspecs/> (visited on May 7, 2014).
- [SZ13] Hang Su and Dakai Zhu. “Scheduling of Elastic Mixed-Criticality Tasks in Multiprocessor Real-Time Systems.” In: *19th IEEE Real-Time and Embedded Technology and Applications Symp., Work-in-Progress Proc. (RTAS-WiP)*. (Philadelphia, PA, USA, Apr. 9–12, 2013). Ed. by Marko Bertogna. 2013. URL: <http://www.cister.isep.ipp.pt/rtas2013/wip> (visited on May 7, 2014).
- [Tam+04] E. Tamura, F. Rodríguez, J. V. Busquets-Mataix, and A. Martí Campoy. “High Performance Memory Architectures with Dynamic Locking Cache for Real-Time Systems.” In: *Proc. Work-In-Progress Session 16th Euromicro Conf. on Real-Time Systems (ECRTS-WiP)*. (Catania, Italy, June 30–July 2, 2004). Ed. by Steve Goddard. Tech. rep. TR-UNL-CSE-2004-0010. University of Nebraska-Lincoln, Department of Computer Science and Engineering, 2004. URL: <http://cse.unl.edu/~goddard/ecrts04wip/proceedings/> (visited on May 7, 2014).

- [Tan+12] Camel Tanougast, Abbas Dandache, Mohamed Salah Azzaz, and Said Saboudi. “Hardware Design of Embedded Systems for Security Applications.” In: *Embedded Systems. High Performance Systems, Applications and Projects*. Ed. by Kiyofumi Tanaka. Rijeka, Croatia: InTech, 2012. Chap. 12. ISBN: 978-953-51-0350-9. DOI: [10.5772/2684](https://doi.org/10.5772/2684).
- [Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd ed. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2007. ISBN: 978-0138134594.
- [Tan12] Kiyofumi Tanaka, ed. *Embedded Systems. High Performance Systems, Applications and Projects*. Rijeka, Croatia: InTech, 2012. ISBN: 978-953-51-0350-9. DOI: [10.5772/2684](https://doi.org/10.5772/2684).
- [Tho11] Michael E. Thomadakis. *The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms*. Research report. College Station, TX, USA: Texas A&M University, Supercomputing Facility, Mar. 17, 2011. URL: <http://sc.tamu.edu/systems/eos/nehalem.pdf> (visited on Sept. 9, 2014).
- [Tho14] Daniel Thompson. “Debugging ARM kernels using fast interrupts.” In: *LWN.net* (May 29, 2014). URL: <http://lwn.net/Articles/600359> (visited on Oct. 7, 2014).
- [TLH97] Vu Tran, Dar-Biau Liu, and Brad Hummel. “Component-based Systems Development: Challenges and Lessons Learned.” In: *Proc. 8th IEEE Int. Workshop on Software Technology and Engineering Practice (STEP)*. (London, England, July 14–18, 1997). Ed. by David Budgen, Gene Hoffnagle, and Jos Trienekens. Washington, DC, USA: IEEE Computer Society, 1997, pp. 452–462. DOI: [10.1109/STEP.1997.615535](https://doi.org/10.1109/STEP.1997.615535).
- [Tsa+05] Dan Tsafir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. “System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications.” In: *Proc. 19th Annu. Int. Conf. on Supercomputing (ICS)*. (Cambridge, MA, USA, June 20–22, 2005). New York, NY, USA: ACM Assoc. for Computing Machinery, 2005, pp. 303–312. ISBN: 1-59593-167-8. DOI: [10.1145/1088149.1088190](https://doi.org/10.1145/1088149.1088190).
- [TVU98] Martin Timmerman, Bart Van Beneden, and Laurent Uhres. “Windows NT Real-Time Extensions: better or worse?” In: *Real-Time Magazine* 3 (1998), pp. 11–19. URL: http://redwood.cs.ttu.edu/~andersen/cs5355/1998q3_p011.pdf (visited on Sept. 3, 2014).
- [TZ99] Philippas Tsigas and Yi Zhang. “Non-blocking Data Sharing in Multiprocessor Real-Time Systems.” In: *Proc. 6th Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. (Hong Kong, China, Dec. 13–15, 1999). Washington, DC, USA: IEEE Computer Society, 1999, pp. 247–254. DOI: [10.1109/RTCSA.1999.811240](https://doi.org/10.1109/RTCSA.1999.811240).

Bibliography

- [Uhl11] Robert Uhl. “Analyse und Manipulation der Linux Interrupt-Behandlung auf der x86-Architektur.” German. Bachelor thesis. Aachen, Germany: RWTH Aachen University, Faculty of Electrical Engineering and Information Technology, Chair for Operating Systems, Nov. 2011.
- [Ung+10] Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, et al. “Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability.” In: *IEEE Micro* 30.5 (2010), pp. 66–75. ISSN: 0272-1732. DOI: [10.1109/MM.2010.78](https://doi.org/10.1109/MM.2010.78).
- [Ung+13] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, et al. “par-MERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability.” In: *Proc. 16th Euromicro Conf. on Digital System Design (DSD)*. (Santander, Spain, Sept. 4–6, 2013). Ed. by José Silva Matos and Francesco Loporati. Washington, DC, USA: IEEE Computer Society, 2013, pp. 363–370. ISBN: 978-0-7695-5074-9. DOI: [10.1109/DSD.2013.46](https://doi.org/10.1109/DSD.2013.46).
- [Vad+09] Srivatsa Vaddagiri, Bharata B. Rao, Vaidyanathan Srinivasan, Anithra P. Janakiraman, Balbir Singh, et al. “Scaling software on multi-core through co-scheduling of related tasks.” In: *Proc. Linux Symp. (OLS)*. (Montreal, QC, Canada, July 13–17, 2009). Ed. by Robyn Bergeron, Chris Dukes, Jonas Fonseca, and John Hawley. 2009, pp. 287–295. URL: <http://linuxsymposium.org/2009/ls-2009-proceedings.pdf> (visited on May 7, 2014).
- [Vaj11] András Vajda. *Programming Many-Core Chips*. New York, NY, USA: Springer, 2011. ISBN: 978-1-4419-9738-8.
- [VGR98] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. “Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors.” In: *SIGOPS Operat. Syst. Rev.* 32 (5 Oct. 1998), pp. 181–192. ISSN: 0163-5980. DOI: [10.1145/384265.291044](https://doi.org/10.1145/384265.291044).
- [Vir12] Al Viro. “Al Viro’s new execve/kernel-thread design.” In: *LWN.net* (Oct. 16, 2012). URL: <http://lwn.net/Articles/520227> (visited on Oct. 7, 2014).
- [VN13] Viktor Vafeiadis and Chinmay Narayan. “Relaxed Separation Logic: A Program Logic for C11 Concurrency.” In: *SIGPLAN Not.* 48.10 (Oct. 2013), pp. 867–884. ISSN: 0362-1340. DOI: [10.1145/2544173.2509532](https://doi.org/10.1145/2544173.2509532).
- [VT10] Vaidehi M. and T. R. Gopalakrishnan Nair. “Multicore Applications in Real Time Systems.” In: *Comp. Res. Repos.* (2010). arXiv: [1001.3539](https://arxiv.org/abs/1001.3539) [cs.SE].

- [Wal10a] Gerhard Wallraf. “In-depth Analysis of x86’s System Management Mode.” Diploma thesis. Aachen, Germany: RWTH Aachen University, Faculty of Electrical Engineering and Information Technology, Chair for Operating Systems, Dec. 2010.
- [Wal10b] Colin Walls. *A Case for MCAP: CPU-to-CPU Communications in Multicore Designs*. White Paper. Wilsonville, OR, USA: Mentor Graphics Cop., 2010. URL: <http://www.mentor.com/embedded-software/request?selected=58911> (visited on Sept. 9, 2014).
- [Wan+13] Zhen Wang, Limin Xiao, Xin Su, Sin Qi, and Xibin Xu. “On the Optimization of Real Time Performance of Software Defined Radio on Linux OS.” In: *J. Commun. Netw.* 5.3B (2013). Ed. by Bharat Bhargava et al., pp. 292–297. ISSN: 1947-3826. DOI: [10.4236/cn.2013.53B2054](https://doi.org/10.4236/cn.2013.53B2054).
- [Was+12] Georg Wassen, Pablo Reble, Stefan Lankes, and Thomas Bemmerl. *Real-Time on Many-Core Systems. Where real-time and high-performance can benefit from each other*. Poster presentation. MARC Symposium, Aachen, Germany, Nov. 29, 2012. 2012.
- [WB11] Wolfgang Wallner and Josef Baumgartner. “openPOWERLINK in Linux Userspace: Implementation and Performance Evaluation of the Real-Time Ethernet Protocol Stack in Linux Userspace.” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011, pp. 155–164. ISBN: 978-3-0003-6193-7.
- [WDL11] Hannes Weisbach, Björn Döbel, and Adam Lackorzynski. “Generic User-Level PCI Drivers.” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011, pp. 91–100. ISBN: 978-3-0003-6193-7.
- [Web02] Andrew Webber. *Realfeel Test of the Preemptible Kernel Patch*. Seattle, WA, USA, Oct. 2002. URL: <http://www.linuxjournal.com/article/6405> (visited on May 7, 2014).
- [Wec09] Filip Wecherowski. “A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers.” In: *Phrack mag.* 0x0d (0x42 June 2009). URL: <http://www.phrack.org/issues.html?issue=66&id=11> (visited on May 7, 2014).
- [Wei+12] Björn Weissler, Pierre Gebhardt, Peter Diippenbecker, Benjamin Goldschmidt, André Salomon, et al. “Design concept of world’s first preclinical PET/MR insert with fully digital silicon photomultiplier technology.” In: *Nuclear Science Symposium and Medical Imaging Conference*

Bibliography

- (*NSS/MIC*), 2012 *IEEE*. (Anaheim, CA, USA, Oct. 27–Nov. 3, 2012). 2012, pp. 2113–2116. DOI: [10.1109/NSSMIC.2012.6551484](https://doi.org/10.1109/NSSMIC.2012.6551484).
- [Wei11] Klaus Weichinger. “A Nonlinear Model-Based Control realized with an Open Framework for Educational Purposes.” In: *Proc. 13th Real-Time Linux Workshop (RTLWS)*. (Prague, Czech Republic, Oct. 20–22, 2011). Open Source Automation Development Lab (OSADL) eG, 2011. ISBN: 978-3-0003-6193-7.
- [Wei84] Reinhold P. Weicker. “Dhrystone: A Synthetic Systems Programming Benchmark.” In: *Commun. ACM* 27.10 (Oct. 1984), pp. 1013–1030. ISSN: 0001-0782. DOI: [10.1145/358274.358283](https://doi.org/10.1145/358274.358283).
- [Wil+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, et al. “The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools.” In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: [10.1145/1347375.1347389](https://doi.org/10.1145/1347375.1347389).
- [Wil+09] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, et al. “Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems.” In: *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 28.7 (July 2009), pp. 966–978. ISSN: 0278-0070. DOI: [10.1109/TCAD.2009.2013287](https://doi.org/10.1109/TCAD.2009.2013287).
- [Wil02] Clark Williams. *Linux Scheduler Latency*. Tech. rep. Raleigh, NC, USA: Red Hat, Inc., Mar. 2002.
- [Wil06] Rob Williams. *Real-Time Systems Development*. Oxford, England: Elsevier, 2006. ISBN: 978-0-7506-6471-4.
- [Wil08] Samuel Webb Williams. “Auto-tuning Performance on Multicore Computers.” Tech. rep. no. UCB/EECS-2008-164. PhD Thesis. Berkeley, CA, USA: Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 17, 2008. ISBN: 978-1-109-09814-3.
- [WL04] Peter Wurmsdobler and Suvendhu Laha. “A Real-Time Compliant Implementation of Linear Time-Invariant Digital Filters in the C Programming Language.” In: *Proc. 6th Real-Time Linux Workshop (RTLWS)*. (Singapore, Nov. 3–5, 2004). 2004, pp. 81–89.
- [WLB12] Georg Wassen, Stefan Lankes, and Thomas Bemmerl. “Harte Echtzeit für Anwendungsprozesse in Standard-Betriebssystemen auf Mehrkernprozessoren.” German. In: *Herausforderungen durch Echtzeitbetrieb. Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V. (GI)*. (Boppard, Germany, Nov. 22–23, 2012). Ed. by Wolfgang A. Halang. Informatik aktuell. Berlin,

- Germany: Springer, 2012, pp. 39–48. ISBN: 978-3-642-24658-6. DOI: [10.1007/978-3-642-24658-6_5](https://doi.org/10.1007/978-3-642-24658-6_5).
- [WM95] William A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious.” In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: [10.1145/216585.216588](https://doi.org/10.1145/216585.216588).
- [Won+08] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, et al. “Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor.” In: *Proc. 17th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. (Toronto, ON, Canada, Oct. 25–29, 2008). New York, NY, USA: ACM Assoc. for Computing Machinery, 2008, pp. 52–61. ISBN: 978-1-60558-282-5. DOI: [10.1145/1454115.1454125](https://doi.org/10.1145/1454115.1454125).
- [WR09] Rafal Wojtczuk and Joanna Rutkowska. *Attacking SMM Memory via Intel CPU Cache Poisoning*. Tech. rep. Warsaw, Poland: Invisible Things Lab, Mar. 2009. URL: <http://invisiblethingslab.com/itl/Resources.html> (visited on May 7, 2014).
- [WR12] Reinhard Wilhelm and Jan Reineke. “Embedded Systems: Many Cores – Many Problems.” In: *Proc. 7th IEEE Int. Symp. on Industrial Embedded Systems (SIES)*. (Karlsruhe, Germany, June 20–22, 2012). Washington, DC, USA: IEEE Computer Society, 2012, pp. 176–180. ISBN: 978-1-4673-2684-1. DOI: [10.1109/SIES.2012.6356583](https://doi.org/10.1109/SIES.2012.6356583).
- [WWP09] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures.” In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: [10.1145/1498765.1498785](https://doi.org/10.1145/1498765.1498785).
- [XX11] Yan Xiaodong and Zhou Xiang. “Design of Rapid Prototyping System Based on RTX Realtime Kernel.” In: *IEEE Int. Conf. on Signal Processing, Communications and Computing (ICSPCC)*. (Xi’an, China, Sept. 14–16, 2011). Washington, DC, USA: IEEE Computer Society, 2011. ISBN: 978-1-4577-0894-7. DOI: [10.1109/ICSPCC.2011.6061551](https://doi.org/10.1109/ICSPCC.2011.6061551).
- [Yag01] Karim Yaghmour. *Adaptive Domain Environment for Operating Systems*. Tech. rep. Sherbrooke, QC, Kanada: Opersys Inc., 2001. URL: <http://ftp.opersys.com/ftp/pub/Adeos/adeos.pdf> (visited on May 7, 2014).
- [Yan+05] Jian Yang, Yu Chen, Huayong Wang, and Bibo Wang. “A Linux Kernel with Fixed Interrupt Latency for Embedded Real-Time System.” In: *2nd Int. Conf. on Embedded Software and Systems*. (Xi’an, China, Dec. 16–

Bibliography

- 18, 2005). Washington, DC, USA: IEEE Computer Society, Dec. 2005. DOI: [10.1109/ICSS.2005.3](https://doi.org/10.1109/ICSS.2005.3).
- [YAW05] S. Yamagiwa, K. Aoki, and K. Wada. “Active Zero-copy: A performance study of non-deterministic messaging.” In: *4th Int. Symp. on Parallel and Distributed Computing (ISPDC)*. (Lille, France, July 4–6, 2005). Washington, DC, USA: IEEE Computer Society, July 2005, pp. 325–332. DOI: [10.1109/ISPDC.2005.11](https://doi.org/10.1109/ISPDC.2005.11).
- [YL06] Joshua J. Yi and David J. Lilja. “Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations.” In: *Computers, IEEE Transactions on* 55.3 (Mar. 2006), pp. 268–280. ISSN: 0018-9340. DOI: [10.1109/TC.2006.44](https://doi.org/10.1109/TC.2006.44).
- [Yod99] Victor Yodaiken. “The RTLinux Manifesto.” In: *Proc. 5th Linux Expo*. (Raleigh, NC, USA, May 22, 1999). 1999. URL: <http://vyodaiken.com/papers-and-talks/> (visited on Sept. 2, 2014).
- [You07] Matt T. Yourst. “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator.” In: *IEEE Int. Symp. on Performance Analysis of Systems Software (ISPASS)*. (San Jose, CA, USA, Apr. 25–27, 2007). Washington, DC, USA: IEEE Computer Society, 2007, pp. 23–34. DOI: [10.1109/ISPASS.2007.363733](https://doi.org/10.1109/ISPASS.2007.363733).
- [YPS05] Kamen Yotov, Keshav Pingali, and Paul Stodghill. “Automatic Measurement of Memory Hierarchy Parameters.” In: *SIGMETRICS Perform. Eval. Rev.* 33.1 (June 2005), pp. 181–192. ISSN: 0163-5999. DOI: [10.1145/1071690.1064233](https://doi.org/10.1145/1071690.1064233).
- [YSL09] Hobin Yoon, Jungmoo Song, and Jamee Lee. “Real-Time Performance Analysis in Linux-Based Robotic Systems.” In: *Proc. Linux Symp. (OLS)*. (Montreal, QC, Canada, July 13–17, 2009). Ed. by Robyn Bergeron, Chris Dukes, Jonas Fonseca, and John Hawley. 2009, pp. 331–339. URL: <http://linuxsymposium.org/2009/ls-2009-proceedings.pdf> (visited on May 7, 2014).
- [YZ08] Jun Yan and Wei Zhang. “WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches.” In: *Proc. of 14th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*. (St. Louis, MO, USA, Apr. 22–24, 2008). Washington, DC, USA: IEEE Computer Society, 2008, pp. 80–89. DOI: [10.1109/RTAS.2008.6](https://doi.org/10.1109/RTAS.2008.6).
- [Zen+09] Hui Zeng, Matt Yourst, Kanad Ghose, and Dmitry Ponomarev. “MPTLsim: A Simulator for X86 Multicore Processors.” In: *Proc. of the 46th Annu. Design Automation Conf. (DAC)*. (San Francisco, CA, USA, July 26–31, 2009). New York, NY, USA: ACM Assoc. for Com-

- puting Machinery, 2009, pp. 226–231. ISBN: 978-1-60558-497-3. DOI: [10.1145/1629911.1629974](https://doi.org/10.1145/1629911.1629974).
- [Zha+05] J. Zhang, R. Lumia, J. Wood, and G. Starr. “Achieving Deterministic, Hard Real-time Control On An IBM-Compatible PC: A General Configuration Guideline.” In: *IEEE Int. Conf. on Systems, Man and Cybernetics*. (Hawai’i, HI, USA, Oct. 10–12, 2005). Vol. 1. Washington, DC, USA: IEEE Computer Society, Oct. 2005, pp. 293–299. ISBN: 0-7803-9298-1. DOI: [10.1109/ICSMC.2005.1571161](https://doi.org/10.1109/ICSMC.2005.1571161).
- [Züp13] Alexander Züpke. “Deterministic Fast User Space Synchronization.” In: *Int. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT) in conjunction with the 25th Euromicro Conf. on Real-Time Systems*. (Paris, France, July 9, 2013). Ed. by Andrea Bastoni and Shinpei Kato. 2013. URL: http://ert1.jp/~shinpei/conf/ospert13/OSPERT13_proceedings_web.pdf (visited on May 7, 2014).

Referenced URLs

- <http://amide.sourceforge.net> (visited on November 12, 2015)
- <http://concurrencykit.org> (visited on November 12, 2015)
- <http://gpgpu.org> (visited on November 12, 2015)
- <http://heise.de/-2063812> (visited on November 12, 2015)
- <http://homepages.cwi.nl/~manegold/Calibrator/> (visited on November 12, 2015)
- <http://lowlevel.eu/wiki/Hauptseite> (visited on November 12, 2015)
- <http://ltnng.org/urcu/> (visited on November 12, 2015)
- <http://media.freescale.com/investor-relations/press-release-archive/2007/16-10-2007-d.aspx> (visited on November 12, 2015)
- <http://newscenter.ti.com/index.php?s=32851&item=123723> (visited on November 12, 2015)
- <http://openmp.org> (visited on November 12, 2015)
- <http://qt-project.org> (visited on November 12, 2015)
- <http://qt-project.org/doc/qt-4.8/signalsandslots.html> (visited on November 12, 2015)

Bibliography

- <http://real-time.ccur.com/home/products/redhawk-linux> (visited on November 12, 2015)
- <http://rt.wiki.kernel.org> (visited on November 12, 2015)
- <http://savannah.nongnu.org/projects/lwip/> (visited on November 12, 2015)
- https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/_005f_005fsync-Builtins.html (visited on November 12, 2015)
- <https://github.com/esnet/iperf> (visited on November 12, 2015)
- <https://github.com/RRZE-HPC/likwid> (visited on November 12, 2015)
- <https://github.com/RWTH-OS/smp.boot> (visited on November 12, 2015)
- <https://github.com/siemens/jailhouse> (visited on November 12, 2015)
- <https://rt.wiki.kernel.org> (visited on November 12, 2015)
- https://rt.wiki.kernel.org/index.php/Cpuset_management_utility/tutorial (visited on November 12, 2015)
- https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application (visited on November 12, 2015)
- <https://www.kernel.org> (visited on November 12, 2015)
- <https://www.rtai.org> (visited on November 12, 2015)
- https://www.rtai.org/?Archive_announcements&id=32 (visited on November 12, 2015)
- http://wiki.osdev.org/Main_Page (visited on November 12, 2015)
- <http://www2.rdrop.com/~paulmck/scalability/> (visited on November 12, 2015)
- <http://www.android.com> (visited on November 12, 2015)
- <http://www.atmel.com> (visited on November 12, 2015)
- <http://www.bdti.com/Services/Benchmarks> (visited on November 12, 2015)
- <http://www.coker.com.au/bonnie++/> (visited on November 12, 2015)
- <http://www.comedi.org> (visited on November 12, 2015)
- <http://www.cs.virginia.edu/stream/> (visited on May 6, 2014)

- <http://www.directinsight.co.uk/products/venturcom/soft-control-architecture.html> (visited on November 12, 2015)
- <http://www.eembc.org> (visited on November 12, 2015)
- http://www.etas.com/de/products/rtpro_pc.php (visited on November 12, 2015)
- <http://www.freertos.org> (visited on November 12, 2015)
- <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-overview.html> (visited on November 12, 2015)
- <http://www.intel.de/content/www/de/de/processors/pentium/pentium-d-processor-brief.html> (visited on November 12, 2015)
- <http://www.iozone.org> (visited on November 12, 2015)
- <http://www.kithara.de/de/produkte/realtime-suite> (visited on November 12, 2015)
- <http://www.lifl.fr/west/artis/> (visited on November 12, 2015)
- <http://www.nematron.com/products/legacy/hyperkernel.html> (visited on November 12, 2015)
- <http://www.opensuse.org> (visited on November 12, 2015)
- <http://www.parallax.com/microcontrollers/propeller> (visited on November 12, 2015)
- <http://www.qnx.com/products/rtos/> (visited on November 12, 2015)
- <http://www.raspberrypi.org> (visited on November 12, 2015)
- <http://www.real-time.de/preise/grad2012.html> (visited on November 12, 2015)
- <http://www.redhat.com> (visited on November 12, 2015)
- <http://www.rtems.org> (visited on November 12, 2015)
- <http://www.spec.org> (visited on November 12, 2015)
- <http://www.sprg.uniroma2.it/asmlinux/> (visited on November 12, 2015)
- <http://www.tenasys.com/index.php/overview-ifw> (visited on November 12, 2015)
- <http://www.windriver.com> (visited on November 12, 2015)
- <http://www.windriver.com/products/vxworks/> (visited on November 12, 2015)

Bibliography

- <http://www.xenomai.org> (visited on November 12, 2015)
- <http://xenomai.org/2014/06/configuring-for-x86-based-dual-kernels/> (visited on November 12, 2015)

Cited Linux source and documentation files

- `linux-3.12/arch/x86/kernel/entry_{32,64}.S`
- `linux-3.12/COPYING`
- `linux-3.12/Documentation/cgroups/cpusets.txt`
- `linux-3.12/Documentation/cputopology.txt`
- `linux-3.10/Documentation/lockup-watchdogs.txt`
- `linux-3.12/Documentation/RCU/RTFP.txt`