Effective Communication Methods for Many-core Architectures with on-chip Networks in the Absence of Cache Coherence

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
Pablo Reble
aus Düsseldorf

Berichter: Universitätsprofessor Dr. rer. nat. Rudolf Mathar
Universitätsprofessor Dr. rer. nat. Matthias Müller

Tag der mündlichen Prüfung: 25. September 2015

Acknowledgements

First, I would like to express my sincere gratitude to my thesis advisors Professor Dr. rer. nat. Rudolf Mathar and Professor Dr. rer. nat. Matthias Müller for their efforts and guidance.
The main research part of this work has been realized at the Chair for Operating Systems at RWTH Aachen University. Special thanks go to Dr. Stefan Lankes for his advice and mentoring. In addition, I would like to thank my colleagues at RWTH for their help and enlightening discussions, the LfBS team, in particular Carsten, Georg, and Simon, the HPC group, in particular Christian, and Dieter. Additionally, I would like to thank the students I worked with, especially Fabian, Florian, Jacek, Marian, and Ole.
I would also like to thank Intel Labs, in particular Michael Riepen and Michael Konow, for supporting my research with experimental hardware, technical mentoring and funding.
I wish to thank my family for their support. My parents Bruno and Ulrike, my siblings Nina and Janosch.
Finally, my biggest thanks go to Eva for her love, support, and inspiration.

Aachen, October 2015 Pablo Reble

Abstract

Since the beginning of the multicore era, parallel processing has become prevalent across the board. Accordingly, the development of processors with multiple computing cores per chip was caused by the fact that fundamental limits in single core performance had been reached. If the trend of integrating more and more cores to a single die continues, general purpose processors with hundreds of cores may be expected in the near future. Further in the future, many-core technology will potentially lead to processors with thousands of cores. The result of this development would be massive parallel-processor architectures that will inevitably create new challenges for the scalability of common software synchronization and communication methods.

It is commonly believed that a reevaluation of established concepts is needed to address such research challenges. One important aspect is the reevaluation of hardware support for effective communication methods. Given a particular power budget per chip, this aspect will be of significant importance. A related question is the demand of changes for established architectures and for redesign of components to support thousands of cores. Similar challenges exist for the support of new memory technologies, which are emerging trends in the field of parallel computing. For instance, instruction-set extensions have been recently introduced for the established Intel Architecture to support non-volatile memory (NVM) and hardware transactional memory (TSX). Full control of on-chip data movement is a related aspect that will result in the waiving of transparent access – for example, of additional instructions for finer-grained cache control.

Because the importance of efficient communication for processors with many cores cannot be underestimated, the focus of this dissertation is on the analysis of efficient communication methods to exploit locality, and on low latency of architectures that follow the network-on-chip paradigm and provide software-controlled high-performance memory in terms of low latency and high throughput.

In 2010, Intel Corporation founded the Many-core Applications Research Community (MARC) to support many-core software research. As a member of this community, the RWTH Aachen University started the projects *MetalSVM* and *iRCCE* to explore new communication concepts and system-software support for future many-core systems.

These projects – the results of which are presented in this dissertation – include the development of software concepts for on-chip communication and

synchronization in the absence of hardware-cache coherence. Intel's Single-chip Cloud Computer (SCC) is a cluster-on-a-chip architecture that represents the first x86 based many-core processor. It implements a new communication concept by waiving full chip cache-coherence and providing hardware support for on-chip message passing with alternative support for on-chip consistency control and explicit communication. The experimental hardware consists of 48 cores which are connected through a mesh interconnect on a single processor die. This configuration, in combination with a flexible and fine-grained memory control, has enabled experiments and the development and verification of low-level communication concepts today, and it thereby guides the development of future systems.

In order to further explore the scalability of low-level software for this kind of architecture, this dissertation includes consideration of the design of a transparent virtual extension. A major achievement of this project is a full working prototype of a cluster of clusters on a chip, which can emulate a many-core processor with more than two hundred cores. This prototype has enabled a deeper analysis of new many-core communication concepts, and has uncovered potential for optimization. Major performance improvements could be achieved by the combination and further development of well-known mechanism and software techniques.

Moreover, a communication model is essential if we are to analytically explore the limits of many-core processors that follow the network-on-chip paradigm and implement a low-memory abstraction by providing software-controlled on-chip memory.

Fundamental requirements for the efficient communication methods that are developed, parametrized and evaluated in this work include configurable caches for direct on-chip memory access or at least finer-grained cache control for data movement. Moreover, a low-level contention model is developed to evaluate different synchronization concepts and to derive optimizations for many-cores with remote direct memory access. Because on this level a contention model is hardware specific, the decision was made to focus on a packed switched on-chip mesh interconnect. This interconnect is implemented by the experimental SCC architecture and also in the next many-core architecture of the Xeon Phi family: codename Knights Landing (KNL).

The Intel SCC research system has been used to support analysis of methods and to verify the accuracy of our concepts when used to model communication. The research system has directly addressable memory, with a high-performance in terms of low-latency and high throughput in relation to the

computational performance of the cores. The experimental hardware shares basic characteristics with attributes of future many-core architectures that result, for example, in a combination of stacked memory and a tight integration of the fabric interconnect. Such similarities create opportunities for the applicability of the effective communication methods to future architectures.

Kurzfassung

Multicore-Prozessoren haben zu der Verbreitung von paralleler Programmierung maßgeblich beigetragen. Die Entwicklung von Prozessoren mit mehr als einem Rechenkern, liegt vor allem darin begründet, dass mit einer Steigerung der Leistungsfähigkeit von Prozessoren mit nur einem Rechenkern physikalische Grenzen erreicht wurden. Aktuelle Prognosen für die Weiterentwicklung der Anzahl von Rechenkernen pro Prozessor zeigen, dass auch bei moderater Steigerung ihrer Integrationsdichte, Prozessoren mit Tausenden von Rechenkernen pro Chip in näherer Zukunft erwartet werden können. Dieser Trend wird zu massiv-parallelen Prozessoren führen, welche gemeinhin als Manycore Systeme bezeichnet werden. In letzter Konsequenz resultieren daraus neue Herausforderungen an etablierte Kommunikations- und Synchronisations-Konzepte.

Es ist eine weit verbreitete Überzeugung, dass nur durch eine Reevaluation etablierter Konzepte in der Forschung, solchen Herausforderungen begegnet werden kann. Ein wesentlicher Aspekt ist die Neubewertung der Hardwareunterstützung vor dem Hintergrund effektiver Kommunikations-Methoden. Ausgehend von der Tatsache, dass Grenzen in der Leistungsaufnahme von Prozessoren existieren, wird zukünftig effizienter Kommunikation eine steigende Bedeutung zukommen. In diesem Zusammenhang stellt sich die Frage, wie vorhandene Architekturen angepasst und welche Komponenten weiterentwickelt werden müssen, um Tausende von Kernen zu unterstützen. Vergleichbare Herausforderungen ergeben sich bei der Unterstützung von neuen Speicher-Technologien, wie nicht-flüchtigem Speicher (NVM) und Hardware Transactional Memory (TSX). Diese Weiterentwicklung des Instruktionssatzes der Intel Architektur steht beispielhaft für einen Paradigmenwechsel, weg von einer vollständigen Abstraktion einer komplexen Speicher-Hierarchie, hin zu einer umfassenderen Kontrolle des Datentransfers innerhalb eines Chips.

Insbesondere die effiziente Kommunikation der Kerne untereinander hat sich dabei als wichtige Komponente hervorgetan, die in dieser Arbeit als zentrale Forschungsfrage im Detail betrachtet werden soll. Wesentlicher Beitrag sind der Entwurf und die Analyse von effizienten Kommunikations-Methoden, die Eigenschaften von Network-on-Chip basierten Architekturen in Kombination mit direkt adressierbarem Speicher ausnutzen.

Im Jahr 2010 gründete Intel die MARC Initiative um Forschungsvorhaben im Bereich parallele Software für Manycore-Systeme zu unterstützen. Als Mitglied der *"Many-core Applications Research Community"* startete die

RWTH Aachen University die Projekte MetalSVM und iRCCE um neue Kommunikations-Konzepte und System-Software Unterstützung für zukünftige Manycore-Systeme zu erforschen.

Ergebnisse dieser Projekte sind unmittelbar in diese Dissertation eingeflossen, insbesondere die beschriebenen Software-Aspekte zur Kommunikation und Synchronisation von Manycore-Systemen ohne Cache Kohärenz. Intel's *"Single-chip Cloud Computer"* (SCC) ist ein Beispiel für eine *"Cluster on a Chip"* Architektur die ein solches Kommunikations-Konzept ermöglicht. Das Forschungs-System besteht aus 48 Rechenkernen auf Basis einer Intel Architektur, welche als Gitter angeordnet in einem einzelnen Prozessor-Chip integriert sind. Diese Architektur verzichtet vollständig auf Cache-Kohärenz und setzt stattdessen auf Hardware Unterstützung für explizite Kommunikation. Diese experimentelle Hardware ermöglicht bereits heute Grundlagenforschung für zukünftige Prozessor-Generationen.

Zur weiteren Betrachtung der Skalierbarkeit von "low-level Software" für eine solche Architektur, wird in dieser Dissertation zudem eine virtuelle Erweiterung beschrieben. Ein wesentlicher Beitrag dieses Projektes ist ein vollfunktionsfähiger Prototyp eines Cluster von *"Cluster on a Chip"* Prozessoren, welcher in der Lage ist ein Manycore-System mit über 200 Kernen zu emulieren. Durch diese Arbeit konnten, durch die Kombination von klassischen Software-Techniken und die Weiterentwicklung der resultierenden Methoden, wesentliche Verbesserungen im Bezug auf die Kommunikation zwischen den Kernen erreicht werden.

Um allgemein die Grenzen der Leistungsfähigkeit einer solchen Architektur analytisch zu bestimmen wird in dieser Arbeit ein Kommunikations-Modell vorgestellt. Die Grundannahme dieses Modells liegt in einer geringeren Abstraktion des Speicher-Zugriffs, so dass Speicher innerhalb eines Chips Speicher direkt adressiert oder über direkte Kontrolle von Caches der Datenfluss gesteuert werden kann. Das Modell beinhaltet Kommunikation für Intel basierte Mehrkern-Prozessoren mit direkt adressiertem integriertem Speicher für konkurrierende Zugriffsmustern und damit die Evaluation von verschiedenen Synchronisations-Mechanismen und deren Optimierung.

Insbesondere für die Entwicklung von zukünftigen Many-core Architekturen sind die Konzepte zur Analyse und Modellierung von Kommunikations-Prozessen wertvoll, die in dieser Arbeit vorgestellt werden. Beispielsweise verfügt das nächste Produkt der Xeon Phi Produktfamilie mit dem Codenamen Knights Landing (KNL) über wesentliche Gemeinsamkeiten mit dem SCC bezogen auf den grundsätzlichen Aufbau der Architektur, wie die

Netzwerk-Topologie und einem relativ schnellem Speicher im Vergleich zu den Rechenkernen, welcher direkt adressiert werden kann.

Contents

1

"Synchronization is a key challenge for the Many-core era" [SS08]

Introduction

The demand of computing power for various kinds of applications is growing beyond the status quo of technology. This observation has been made throughout the history of computing systems until today, especially in scientific computing where computationally intense applications play a predominant role. To fulfill the requirements, computer architectures have changed over the years with a strong emphasis on parallel execution. As a consequence, state-of-the-art computer systems include processors, which are based on multi-core technology. Besides this duplication of functional processor units, a modern architecture implements different levels of parallelism. Parallelism at instruction level, data level, and thread level includes a specific degree of abstraction, which has to be considered for the software-development process.

One challenge for programming that resulted from this growing parallelism involves the concept of concurrency. Herb Sutter formulates the problem in a fundamental article that was published in 2005 on the revolution of concurrency: *"The Free Lunch is over: A Fundamental Turn toward Concurrency in Software"* [Sut05].

Today's importance of parallel-computing systems cannot be underestimated as their pervasion ranges from mobile phones to supercomputers. In 2004, Intel's CEO Ottelini communicated a sea change in computing and speculated that the company will dedicate all of its future product development to multicore designs. This trend towards multi-core architectures has manifested for decades and now dominates the market.

The first microprocessor architectures based on multi-core technology were a consequence of reaching the physical limits. A limitation of power consumption first manifested in cooling issues for single-chip processors when the limit of air cooling was reached in 2004 [HP12]. Whereas an Intel 80386 processor chip consumed about 2 W, the Pentium 4, which was produced ten years later, consumed more than 100 W with a comparable die size of about 1 cm^2. This development necessitated a paradigm shift, because a further frequency scaling, which implies a growing power consumption, was not possible.

These days, the growing importance of energy efficiency can be observed especially in the field of high-performance computing where performance has long been the dominating criterion for the components of a computer system. This observation is closely connected to the Exascale challenge, which envisions supercomputers with at least 10^{18} floating point operations per second (FLOPS) in the near future [SDM11].

It is most likely that to reach this goal – not only for processor chips or system nodes but also on cluster level – that a specific power budget will have to be taken into account [Esm+13]. For future systems, because communication has been identified as a potential bottleneck, fundamental changes in processor design are expected such as the integration of scalable interconnects and sophisticated memory organization [Don+11]. This will pose challenges to system software; therefore, new operating-system designs are in the focus of recent research projects. Related research questions include the role of caches in such an environments, the necessity of full-chip coherent memory and the impact to established programming models. The compatibility with new massive parallel-processor architectures to existing programming models becomes more and more complex as the level of parallelism grows.

Assuming that Moore's Law – that the integration of transistors per chip doubles every two years – will hold for at least the next decades [Moo65], the possibility will arise to integrate thousands of cores per processor die in the near future. For a chip vendor, such a development is of a high importance, as design and verification costs are steadily increasing [NO97].

1.1 Motivation

In current research, the hypothesis that cache coherency represents a general limitation for the scalability of processor architectures is controversial. Research contributions exist that, by promising the end of multicore scaling, attempt to answer the question posed in: *"Dark Silicon and the End of Multicore Scaling"* [Esm+11]. Other contributions exist which argue the opposite assumption that, on the contrary, *"on-chip cache coherence is here to stay"* [MHS12].

The Intel TeraScale computing research program started in 2006 [Intel06] and is aiming at scalable processor architectures. The focus of this program is to explore the programability of future many-core architectures. This focus has lead to the development of design concepts and unconventional architectures. Besides this program, tiled processor architectures that follow the network-on-chip paradigm represent an active field of research.

The term *coherency wall* has been introduced to describe the fact that a limitation exists because of hardware cache-coherence, which introduces additional architectural overhead [Mat11]. Dependent on the access pattern, the cost in time and memory can grow to a point beyond which additional cores are not useful in a single parallel program. Figure 1.1 illustrates this connection. Costs for communication are constant for the local neighborhood in the best case and grow at a linear slope in worst case because of the diameter of the network for a many-core system with explicit on-chip communication.

For a scalable, directory-based scheme, hardware-cache coherency always incurs an N-body effect [Kum+11]. At best, the cost for this technology scales linearly regarding a fixed memory size as cores are added. At worst it scales quadratically, if memory grows linearly with the number of cores. Amdahl's Law, which is commonly used to predict the maximum speedup of parallel applications, predicts a limit of scalability regarding relatively small overheads [HM08]. The general assumption is that relatively small overheads (e.g., 1 % of overhead) can harm scalability [Bor07].

Furthermore, hardware validation costs represents a future challenge for processor design, because growing chip complexity will influence verification costs, which can become significant. It also remains an important and open question whether full-chip cache coherence can be provided for future processors with a core count in the order of thousand cores.

Future processor architectures with many cores may waive full-chip cache coherence. This class of many-core processors, with low memory abstraction

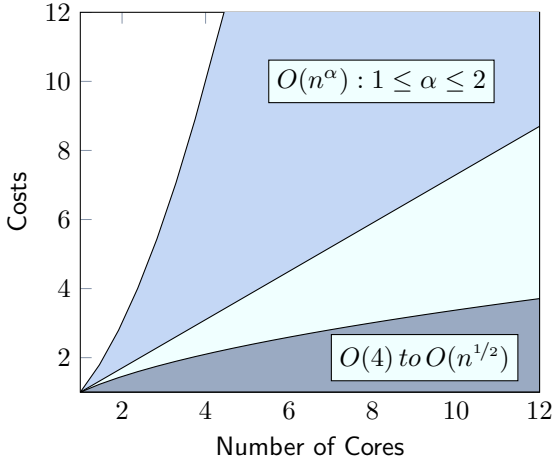


Figure 1.1: Comparison of costs in hardware and time for cache coherent and distributed on-chip memory [Mat11]

and implicit data movements, is referred to as a cluster-on-a-chip processor. Such a fundamental change will influence programming models and system software. Major open issues for this type of architectures include resource allocation (for example, of software controlled on-chip memory) and access granularity.

The *MetalSVM* project started to explore shared memory programming on non-coherent memory coupled cores at the RWTH Aachen University in 2010[1]. The basic idea behind this project is the integration of a Shared Virtual Memory (SVM) management system into a bare-metal hypervisor. As a result, a shared-memory application can be executed in a partly virtualized environment. Such an abstraction does not reduce the demand for high-performance synchronization means regarding scalability and low latency. The observed research trend toward many-core architectures which target the integration of hundreds to thousands of cores per processor die will lead to massive software-controlled on-chip parallelism.

A scalable synchronization becomes even more important, and any software-based approach to control memory coherence – such as Shared Virtual Mem-

[1]The project was initially supported by a research grant of Intel Labs Braunschweig

ory (SVM) – will require fast synchronization methods. For a many-core architecture with non-coherent memory-coupled cores, established synchronization methods that were designed for shared memory systems have to be reevaluated. This implies that the communication performance of a many-core architecture must be characterized.

1.2 Contribution

This work covers research on system-software and operating-system support for x86 based many-core systems with low on-chip memory abstraction. The resulting cluster of processors would be assembled without full chip coherence – so called cluster-on-a-chip (CoC) processors – which creates the possibility, for example, to flexibly handle memory coherence in software. Related tasks, such as hardware support for software-controlled coherence or on-chip message passing, pose new challenges to system software. The contribution of this work is the investigation of a software abstraction for communication and synchronization that is performed for the purpose of determining which kind of hardware support is required to implement an efficient communication layer [RCL13; Reb+12c].

A software layered approach is the basis of our project. Related structures consist of a basic communication layer that abstracts the hardware details of communication by controlling the consistency of access to on-chip memory. On top of this hardware-abstraction layer, message-passing libraries can be implemented with less effort that feature more complex protocols for point-to-point and collective communication. The use of such an environment as an inter-kernel communication layer has been treated in a previous publication [Reb+11]. In addition to the efficient implementation of a bare-metal hypervisor, this communication layer builds the base of the project *MetalSVM*, which targets a virtualization of a shared-memory system on non-coherent memory-coupled cores. This work has also been previously published [Reb+12b].

A communication model is first presented in this work, which takes contention into account as a result of thousands of cores that explicitly access on-chip memory locations. This model is used to explore the limits of a cluster-on-a-chip architecture, and it enables the comparison to cache-coherent many-core architectures which implement a higher memory abstraction in hardware. Parameters for the communication model are derived by means of low-level

timing analysis and statistical analysis is used to verify the contention extensions of the communication model. As a result, the performance of different access patterns can be characterized. Additionally, in this dissertation, aspects of real-time computing are applied to create a fully controlled bare-metal benchmark environment [RW14].

As a result, the performance of classic algorithms can be analytically evaluated for this kind of architecture. Consequently, in this work, classic synchronization methods for distributed memory and shared memory systems are evaluated and optimized for a cluster-on-a-chip architecture. The Intel SCC research processor – an example of this kind of architecture – has been used as an application example. Prior to this work, with RCCE, a light-weight native programming library was available to facilitate the use of the message-passing programming model [MvW11].

We describe a virtual extension of Intel SCC's on-chip network to further explore the scalability of new communication concepts in previous publication [Reb+12a]. A further development of this extension creates the possibility of emulating a many-core system with up to 240 cores [Reb+15]. Such communication environments create new challenges to the existing software stack; therefore, solutions are presented and analyzed in this study. This work includes the analysis of different communication protocols and the exploration of the concept of communication offloading to make use of a scaling grid of cluster-on-a-chip processors.

1.3 Structure of this Work

Figure 1.2 illustrates the structure of this work. The basic structure of this work itself is bottom up, and it is focussed on a layered communication structure.

Specifically, this work is structured in the following way: Chapter 2 describes components of state-of-the-art many-core systems. The basic communication layer of an experimental research platform is covered in this chapter, which includes low-level analysis of its on-chip interconnect. Moreover, a communication model is developed and verified.

A communication-and-synchronization interface is presented in Chapter 3. This includes protocols for efficient point-to-point and collective operations. In addition, this chapter describes the development of a virtual extension of a many-core-processor interconnect.

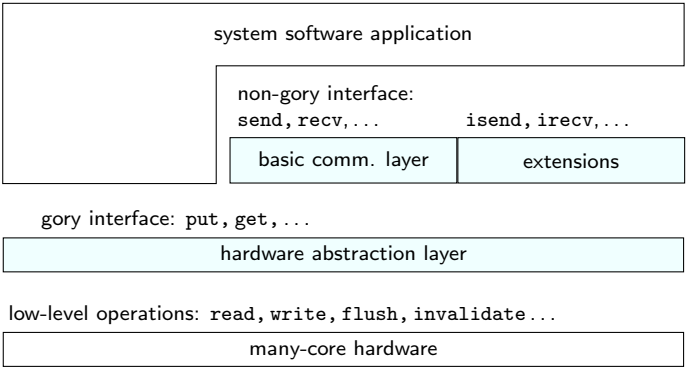


Figure 1.2: Focus of this dissertation regarding a layered communication structure

Chapter 4 targets system-software aspects for many-core software and software-managed coherence. One aspect is the realization of shared memory programming on non-coherent memory-coupled cores through Shared Virtual Memory management. The efficient implementation of a bare-metal hypervisor is detailed. In this context, the communication extension described in Chapter 3 is evaluated as an inter-kernel communication layer.

2

"Caches make the memory state difficult to predict. This greatly complicates software optimization." [Mat11]

Many-core Systems

The integration of multiple processor cores per chip has become an emerging trend within the last decade. Accordingly, this time is also known as the multi-core era in the field of parallel computing. In recent years, a steadily rising core count per processor package can be observed. From dual-cores to quad-cores, processors based on multi-core technology with a core count in the order of 10 cores per processor die are available on the market today.

The key motivation for the integration of multiple processor cores per chip has been reaching the physical limits of manufacturing processors around 2004. Until this time, rising serial compute performance could be observed, which was mainly enabled by increasing the processor frequency. In the field of high-performance computing, and not only for embedded devices, power consumption has become an increasingly hot topic. Processors based on many-core technology provide a better performance-to-watt ratio than processors based on multi-core technology. Accelerators and Coprocessors are options that enable better energy efficiency in high-performance computing.

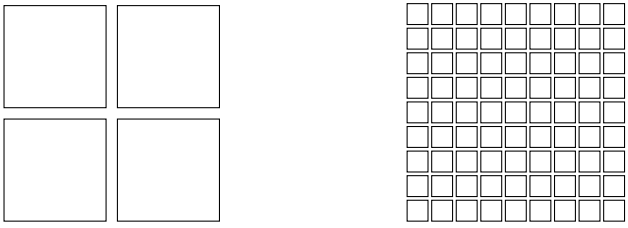
A sharp definition of a many-core architecture does not exist. In addition to a categorization based on architectural features, another classification aspect is programmability. In his book on programming many-core processors, Vajda formulates the widespread assumption that the definition of many-cores architectures is loose [Vaj11].

In common understanding, it is a chip with at least several but more likely hundreds or thousands of cores. These represent many small computing units, which includes a moderate frequency and static-scheduling in combination

with 2-way super-scalar or scalar execution of instructions. In 2012, Reinders communicated[1] his view on the key differences between many-core and multi-core [gop12]. In his opinion, further differences exist between many-core and multi-core architectures in addition to differences in core count. For example, he points out the importance of interconnects for the many-core era.

Another distinction can be made with respect to the manner in which chip design handles physical constraints: e.g., the available number of transistors and the given power budget per chip. Figure 2.1 sketches the utilization of available chip size for many-core and multi-core processor architectures.

As a concrete example in 2015, available high-end processor that implement multi-core design – such as Intel Haswell – provide large shared caches ($< 50\,\text{MB}$) and a moderate core count (< 18) running with a high frequency ($< 3.6\,\text{GHz}$). Many-core processors such as the Xeon Phi target a large core count (> 60) with a moderate frequency ($\sim 1\,\text{GHz}$). The design concepts can be distinguished between latency optimized and throughput optimized.



(a) multi-core (b) many-core

Figure 2.1: Comparison of different architectural concepts regarding the utilization of chip size

Mattson identifies[2] the 48 core SCC research processor by Intel Labs with two cores per tile as one example for the future of many-core computing [Mat11]. If the number of cores grows, the level of software controlled on-chip parallelism evidently rises. The combination of this massive thread-level parallelism and instruction-level parallelism, in the form of vector instructions,

[1] interview: published on `http://go-parallel.com`

[2] in his talk *"The Future of Many-Core Computing: a Tale of two Processors"*

demonstrates that sheer computational performance will become less important, especially for many-core architectures. The ongoing evolution of multi to many-core technology raises the question what a future many-core system may look like, including the interconnect and memory organization.

Message passing and shared memory are established programming models for parallel programs. Hybrid concepts exist that combine both models; however, many-core architectures increase the demand for new programming concepts. For established programming models, limitations exist regarding the mapping of tasks to computing resources that are embedded into complex memory hierarchies, dynamic load balancing and overhead of operating systems and runtimes.

Rather than just focus on applications that are limited by memory bandwidth, synchronization is an open challenge for the many-core era, especially for architectures without full-chip memory coherence. These architectures will provide predictable low latencies to on-chip memory. Influence of specific attributes to synchronization methods, which were intentionally designed for classic distributed shared memory (DSM) systems, have to be reevaluated.

In addition to challenges for established communication and synchronization methods, a loss of full-chip cache coherence will create opportunities for software optimization of many-core memory access. A common assumption is that software-controlled coherence needs additional hardware support. In this work, this hypothesis is studied and evaluated.

Organization of this Chapter

This chapter is organized as follows. The next section covers components of a processor architecture which includes multi-core as well as many-core technology. Section 2.1 is structured into subsections according to the components of a tiled many-core architecture, computing cores, memory organization, and interconnect. Section 2.2 describes the Single-chip Cloud Computer (SCC) that represents Intel's first many-core processor which implements an example of a cluster-on-a-chip. The SCC is a tiled architecture, that combines 48 x86-based cores on a single chip and builds an excellent and flexible research platform for many-core software research.

A communication-and-contention model for the investigated architectures with a low-memory abstraction is discussed in Section 2.3. The remainder of this chapter is on experimentation. It uses a low-level timing analysis with synthetic benchmarks to explore predictability and fairness of on-chip

memory access. Our experiments make possible a better understanding of the preferred communication patterns for a tiled many-core architecture.

Parts of this chapter consist of or are based on previous publications. A study of the predictability of operating-system supported communication was published previously. My contribution was the analysis of on-chip memory access for tightly coupled Intel SCCs within a controlled environment [RW14]. The hardware-synchronization support of the SCC research platform was considered in previous publication [Reb+12c]. My contribution to this work was an analysis of SCC's hardware synchronization support and design of methods to relax contention.

2.1 Basic Components

In this section, the concept of a many-core architecture with a tiled structure is detailed. The design concept of a many-core architecture has been successfully implemented as a product, the Intel Xeon Phi coprocessor, and besides the Intel SCC [Mat+10], as several research projects such as Tilera [Bel+08] and SCORPIO [Day+14].

The terminology, which can be used to describe the components of a many-core architecture are illustrated in Fig. 2.2. A tile represents the basic unit of a network-on-chip (NoC) based processor. Each tile can consist of a combination of computing cores and local memory. The different components, computing cores, on-chip memory organization, and interconnects are described in the following of this section.

2.1.1 Computing Core

Over the years, two key concepts have been developed which can be categorized according to the complexity of a single machine instruction, RISC and CISC.

A modern processor natively works on instructions with a low complexity, Reduced Instruction Set Computer (RISC). For historical reasons, primarily due to backwards compatibility, processors by Intel are the only modern processors that retain a Complex Instruction Set Computer (CISC) architecture. Complex instructions are translated to microcode, which includes a certain overhead. Intel has shown, with the existence of x86 based, also called intel based, processors in the field of embedded computing as well as in the

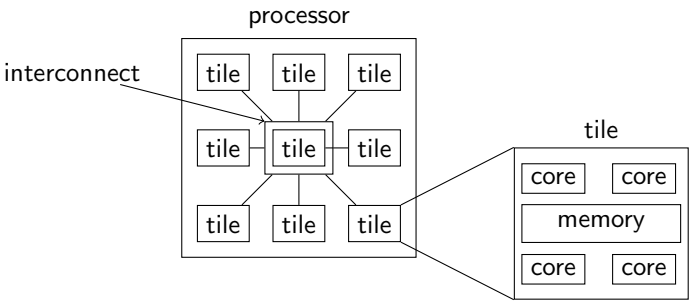


Figure 2.2: Basic structure of a tiled processor with the used terminology

field of high-performance computing, that this overhead represents no general limitation building popular general purpose processors.

Intel based processors represent a classic load and store architecture with a flat memory model. Although memory is physically distributed, a transparent view on memory is provided by a cache coherency protocol on node level. The well established instruction set of Intel's load and store CISC architecture, supports the move operation in different variations, with several limitations for the operands. Explicit on-chip data movement is not supported, for instance to communicate between computing cores.

2.1.2 Memory Organization

For decades, a trend can be observed that the computational performance is faster growing than the memory performance in computing systems. Since over 30 years an exponential grow in computational performance can be observed, whereas the memory performance has grown at least linear in this period of time. Figure 2.3 shows proportional memory and processor performance, in terms of byte moved per second and computational operation per second. Absolute values for the last three decades are referenced to the year 1980, when computing was as fast as data movement. As a result of this development, an issue known as the memory performance gap manifests for computing systems.

Such characteristics of modern computing systems can have a major impact to the performance of applications. For instance, for a scientific application

with a specific data access pattern, which potentially includes a low ratio of computation per data, the processor performance is not important in contrast to the memory performance. A direct connection between the performance of specific applications and the performance of main memory, is known as memory wall and resulting of the development described above [WM95].

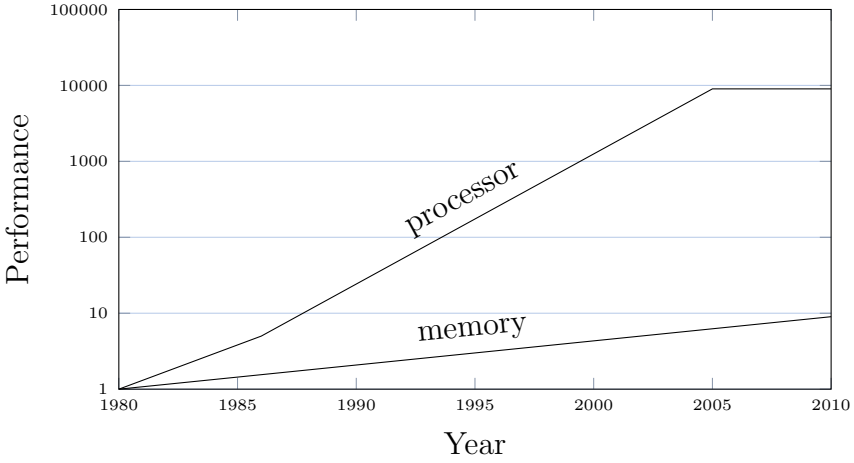


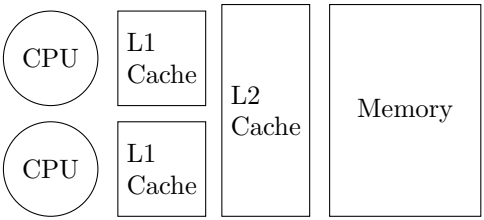
Figure 2.3: Development of memory performance and processor performance over the last 30 years [HP12]

Memory organization of modern processor architectures is designed to hide latencies, and provide a flat memory abstraction. For state-of-the-art large scale shared memory systems with hundreds to thousands of processor cores, based on multi-core technology, a programmer has to take architectural details into account.

Traditionally, modern processors use caches to bridge the memory performance gap. A processor cache memory, shortened to cache, can be defined as a transparent on-chip storage. To hide memory latency, a cache replicates data in on-chip memory. This method results in the existence of copies within the memory systems, for cache memory and main memory.

Commonly for a modern processor, caches with different size and latency are organized in a hierarchy, as illustrated in Fig. 2.4 on Page 15. Between

levels of the hierarchy, costs[3] differ between fast memory that provides data within few processor cycles and slow memory with a latency in the order of hundreds of processor cycles. If a memory request cannot be served by the cache, it is termed as a cache miss. Respectively, if the memory request can be served it is termed as a cache hit. Commonly different types of cache hits and cache misses are distinguished. Those types are influenced by the overall amount of on-chip memory and the displacement strategy. In general, it is distinguished between capacity, conflict and compulsory misses.



size:	1000 B	32 kB	2 MB	16 GB
latency:	300 ps	2 ns	15 ns	50 ns

Figure 2.4: Example of a memory hierarchy of a multi-core processor

Besides the obvious *displacement* strategy, the hardware controller implements a protocol, with the following attributes:

- Granularity
- Write policy
- Associativity
- Replication
- Coherency

[3]Here, costs can be related to price of the hardware and performance in terms of latency and bandwidth

Specific attributes are discussed in the following paragraphs.

Granularity

Caches use the principle of locality, such as temporal and spatial locality to speedup memory access on average [SY05]. *Temporal locality* is based on the assumption of a higher probability that a memory location is accessed at a point in time and again in the near future. The term *spatial locality* includes a higher probability of accessing a near memory location than accessing a far memory location. In other words, it is more likely to access a memory location with a small difference of its address compared to last accessed address.

To support spatial locality, modern processor caches are organized in a certain granularity. The basic entity is called a cache-line, which is equal to the minimum data transfer between cache memory and main memory. A common size of a cache-line is 64 B, which is a constant value for all levels of caches. For a hardware cache coherency protocol a cache-line represents the basic unit of maintaining coherence.

Associativity

A cache controller has to perform a lookup operation to decide whether a memory access results in a cache hit or a cache miss. As the cost of this lookup operation is related to the overall size of a cache, modern processor caches are organized in different degrees of associativity. This organization is commonly a tradeoff between 1-way for latency and hardware cost optimization and full associativity for effective cache utilization. Except for full associativity, whereas the location of a specific cache-line within a cache memory is not restricted, the effective useable cache-size can appear smaller to an application, than the physical size of a cache.

Replication

Replication of data – with a certain granularity – is a basic method of cache memories to hide latency of the main memory. Dependent on the distribution of copies within different cache levels, a cache memory is specified exclusive, if a cache-line is uniquely located at a single level of the cache hierarchy. In contrast to exclusive caching, a full inclusive cache organization holds multiple copies of a single cache-line in all cache levels.

The main advantage of inclusive caching is a simplification of the displacement process. In contrast to exclusive caching, where a cache-line is only located in a single cache level. For exclusive caching, a copy operation is necessary within the cache hierarchy, if a cache-line is displaced.

The main disadvantage of an inclusive cache organization is that the use of the overall cache capacity is not optimal, as copies exist in different levels. Additionally, an inclusive organization of a shared last level cache can cause side effects. For example, a single core can implicitly invalidate the cache content of another core's higher level private cache by intensively using the shared cache.

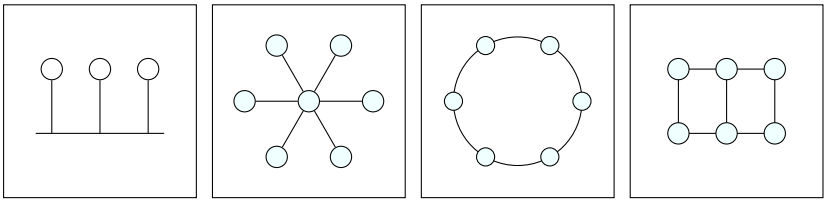
Coherency

False sharing describes a performance issue, that occurs if cores in a shared memory system hold references to different data objects, which results in unnecessary coherency operations because the data objects are sharing a coherency block [BS93; TLH94]. False sharing is one example, when characteristics of a cache organization have to be taken into account to avoid a performance loss, which contradicts a full memory abstraction.

For a processor with multiple cores, the existence of caches poses additional challenges. Assuming a shared access of data from the programming level, also called SMP, additional support is needed to ensure its coherence, because different copies of data can exist in multiple caches. The common cache organization of modern multi-core architectures, separates private and shared caches [CS06]. Commonly the highest level cache memory, the so called Last Level Cache (LLC) is shared, whereas cache memories that are located closer to the core are typically private.

In addition to the replication of data, a cache coherency protocol adds information to each cache-line to track its status. MESI represents a simple and prevalent cache coherency protocol, which is named after its state: *modified*, *exclusive*, *shared*, and *invalid*. The protocol is based on snooping to identify remote memory transactions, that may trigger changes to local cache-lines.

The protocol invalidates all other existing copies in *shared* status to guarantee coherency of cached data rather than a reload of modified cache-lines. Especially for large shared memory systems, which typically have non-uniform memory access (NUMA) characteristics, it is a disadvantage that the four state MESI protocol can not support cache-to-cache transfer of data within a coherency domain. In order to overcome this limitation, AMD Opteron



(a) bus (b) star (c) ring (d) mesh

Figure 2.5: Examples of Network Topologies

microarchitecture implemented with MOESI an extension to MESI. This extension introduced a so called *owned* state that allows a single copy of a cache-line being modified without invalidation of other copies. Obviously the protocol has to broadcast the modification to other shared copies, but this method enables cache-to-cache transfer of data.

A similar approach of extending the classic MESI protocol was first implemented the Intel Nehalem microarchitecture. The resulting extension, called MESIF, adds a so called *forward* state. One copy of a cache-line may exist in this state, which can then be used to serve requests from other processor caches.

2.1.3 Interconnect

A classic terminology to describe network topologies are graphs and edges [DT04]. Apart from this abstract terminology, *graphs* can represent devices, such as computing nodes, cores or organization of memory and *edges* the connections between the devices. Dependent on the connection between the *graphs*, networks can be classified into basic topologies or combinations of them. Examples for different topologies of network interconnects are illustrated in Fig. 2.5 [TvS13].

Figure 2.5a illustrates a bus topology, a single physical connection to connect all nodes. For a communication between the nodes, the connection is typically shared through *time multiplexing*. If one node represents the central connection point, a so called root node, the topology represents a star,

as shown in the next figure. All other nodes are connected to the root by point-to-point connections. The next figure illustrates a ring topology, where every node has exactly two neighbors, which leads to a circular path. Last, an example of a mesh topology is shown, which consists of two or more nodes with redundant paths in between. Consequently a routing mechanism is necessary for the available point-to-point connections. Specific examples for the abstract network topologies, such as described in this paragraph, can be found in classic as well as state-of-the-art processor architectures.

A classic example for the implementation of a bus topology is the front-side bus (FSB) by Intel, which was integrated into a huge number of processors from Pentium to Pentium 4 to connect computing cores and the chipset. AMD was the first chip vendor that replaced the established bus by a crossbar in 2002 for their server processors to increase scalability of the architecture. This technology was adopted by Intel for its Nehalem microarchitecture in 2008. Since 2013, the interconnect of the state of the art multi-core micro architecture by Intel named Haswell, relies on a ring topology. To connect multiple processor chips to a single coherency domain recent Intel processors include with QuickPath Interconnect (QPI) a point-to-point interconnect which includes a cache coherency protocol.

QPI is organized in different layers with the following attributes [QPI09]:

- Physical layer: handles phits, 20 bit physical units
- Routing: Describes the path, that a packet will traverse
- Link layer: Granularity of link layer is a flit with 80 bit length
- Transport layer: Dependent on the architecture
- Protocol layer: Supports different message classes

Intel corporation recently released the Many Integrated Core (MIC) processor architecture, which is the first commercial version of a many-core architecture in the x86 landscape. The MIC (pronounced: Mike) architecture is based on research of the Larrabee project [Sei+08], the Teraflops Research Chip Polaris [Van+08] and the Single-chip Cloud Computer (SCC) [Mat+10]. In 2010, early prototypes with the codename *Knights Ferry* have been released to developers. The first generation product with the codename *Knights Corner* (22 nm) has been announced in 2011 and released in 2013. The second

generation product codenamed *Knights Landing* (14 nm) has been announced in 2013 for a release in 2015.

The Intel Xeon Phi first generation product is an example of a many-core processor that implements a ring interconnect. The interconnect of the many-core processor consists of three bidirectional rings to transfer data, command and address, and coherence as well as credits information, and fulfill their different requirements regarding latency and throughput. All basic processor components, specifically the PCIe logic, processor cores, and memory controllers providing GDDR5 are directly connected to the interconnect. Here, multiple processors represent different coherency domains because PCIe does not provide the feature of maintaining coherent caches, which is supported by QPI.

Another example for a processor architecture that includes an interconnect based on a ring topology, is the Cell processor that has been developed by Sony, Toshiba and IBM. The architecture is of heterogeneous nature. It combines general purpose PowerPC processor cores and special purpose SPU elements [Hof05]. This heterogeneity posed challenges to software development for the processor in its programability. Binary compatibility between the cores was not supported because the SPU elements used a different instruction set that targets efficient execution of SIMD floating point operations.

Intel Labs founded in 2009 the Many-core Applications Research Community (MARC) to support research in the field of many-core software research [MARC]. Through this community with a strong academic part, the SCC has been provided to researchers as a vehicle for many-core software research. A major target of this experimental architecture has been an understanding of the programmability and application scalability of many-core chips. Furthermore, fine grained voltage and frequency control enables new possibilities regarding exploration of energy efficient methods for software development.

2.2 Single-chip Cloud Computer (SCC)

The Single-chip Cloud Computer (SCC) is a *concept vehicle* created by Intel Labs as a platform for many-core software research [SCC10]. The full research platform consists of the experimental processor, which has been developed under the project name RockCreek, the development board with a coprocessor implementation, and a new many-core software stack.

As a PCIe coprocessor, the SCC cannot be used as a standalone system. The processor is connected to a standard server hardware called Management Console PC (MCPC). A standard Linux operating system runs on the MCPC with SCC's software stack, which has been developed by Intel Labs to setup and debug the connected SCC processor.

Figure 2.6 illustrates the tiled architecture of the experimental architecture, consisting of 48 P54C based cores, that are arranged in a 6×4 on-die mesh of tiles, with two cores per tile.

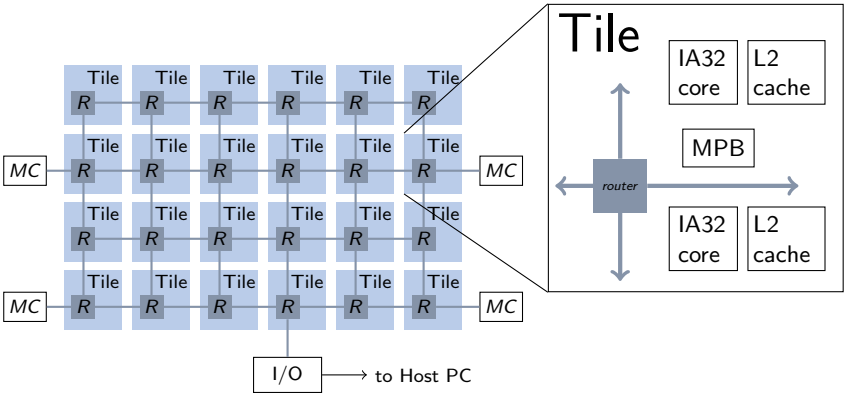


Figure 2.6: Layout of the Intel SCC, focusing on the tiled processor architecture with 24 tiles and 4 memory controllers (MCs) in the corners of the mesh interconnect (Figure based on: [Mat+10])

2.2.1 Research System

The SCC chip has four on-die memory controllers to address the external main memory. The supported DRAM type is *DDR3-800*, where frequencies of the cores and the routers of the mesh are configurable. The routers support frequencies of 800 MHz and 1.6 GHz, while the cores use a frequency between 100 and 800 MHz. The power consumption of the full chip depends on the configuration (frequency and voltage of the mesh and cores) and is between 25 and 125 W.

For a many-core processor that follows the network-on-chip paradigm, each remote memory access can be logically split into two parts: *request* and *acknowledge* of a memory operation. Read and write requests from computing cores are transparently performed by the Mesh Interconnect Unit (MIU), which is part of each tile and directly connected to the FSB of each core (cf. scale-up from Fig. 2.6). Thus, these read and write requests can be a result of a cache-miss or a memory access to an uncached region. For a read operation, the acknowledge holds the actual data, whereas for a write operation the request package holds the data. Regarding a single blocking access, a symmetric pattern can be assumed, when reads and writes on same granularity have identical costs.

The full SCC System consists of the many-core processor that shares a special main board in a common form factor with an FPGA. The FPGA implements a System Interface (SIF) and tunnels the native protocol of the on-chip interconnect to the MCPC through PCIe. For this purpose, the FPGA is directly connected to the on-chip interconnect. Due to the fact that a retargetable component has been chosen for implementation of a system interface, new firmware can add new functionality, such as a set of synchronization registers or a global interrupt controller.

For all benchmark results presented in this chapter[4], the SCC has been used in its default configuration with a frequency of 533 MHz for the cores, and a frequency of 800 MHz for the mesh and the memory.

2.2.2 Processor Cores

Two classic P54C based cores are located on each tile, which are fully compatible to IA32, the 32 bit x86 instruction set. The classic architecture has been chosen for the implementation of the Intel SCC, because it could be fully synthesized [Lu+07]. Positive consequences for software development are that the software stack for x86 based processors, such as a standard compilers (icc,gcc) and a standard operating system (Linux) can be re-used with minor modifications.

In contrast to its successor, a classic Pentium single-core processor of 1994 with a maximum frequency below 200 MHz, the SCC is implemented in 45 nm CMOS technology, which results in a maximum frequency of 1 GHz. Both processors have an identical two level cache hierarchy, with a Level 1 Cache

[4]Unlike noted otherwise.

(L1) size of 32 kB and a Level 2 Cache (L2) size of 256 kB. For the SCC processor cores, hardware cache coherence is provided not even between two cores that share a tile. The design resembles an on-chip distributed shared memory system, which has been defined as a cluster-on-a-chip architecture.

For embedded systems, software controlled on-chip memory is usually termed as scratch-pad memory [Ban+02]. In addition to local processor caches, the on-chip memory organization of the SCC consists of a software controlled on-chip SRAM with a size of 16 kB per tile, which is called Local Memory Buffer (LMB). This memory represents a low-latency infrastructure for communication and synchronization. If the architecture does not provide cache coherence in hardware, the use of caches is per default restricted to *private* memory regions. For efficient communication, hardware support is needed to selectively flush and invalidate shared data, if *shared* memory regions are used in combination with caches or *write-combining buffers*. The P54C instruction set has been extended for the SCC to support a selective invalidation of cache-lines. A software workaround, has been implemented to flush modified data by writing dummy data. These extensions in combination with software controlled on-chip memory represent the hardware support of the SCC for message passing.

Especially, if on-chip memory can be explicitly controlled, distributed shared memory can be categorized according to its location and resulting latency. The SCC has two different types of shared memory that can be accessed by all cores without any coherency: *off-chip* memory which is represented by four DDR3 memory controllers with a supported size of up to 64 GB as well as *on-chip* memory of 16 kB per tile.

Figure 2.7 illustrates different software layers of a cluster-on-a-chip communication environment. RCCE, such as its extension iRCCE, enables message passing communication through `send` and `recv` functions [MvW11; Cla+13b].

The existing communication libraries can be logically separated into two parts: In this chapter the focus is on a basic communication layer, which abstracts a specific hardware and represents the lower part of these two-sided communication layers. The upper part and its further development is described in the next chapter.

A software layer like RCCE can be used to abstract communication by means of software controlled on-chip memory, which is called Local Memory Buffer (LMB) in the scope of Intel SCC research projects. For related projects, the Message Passing Buffer (MPB) has been established as the LMB that is assigned to each core on a tile for low-latency communication. Explicit

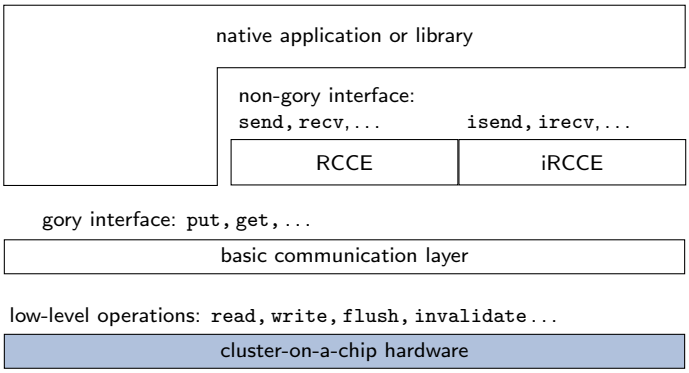


Figure 2.7: Overview of a layered communication structure for a cluster-on-a-chip processor

access to the MPBs and the synchronization of concurrent access can be abstracted by RCCE. However, part of this buffer is typically used for flags for synchronization purpose.

As cores of the SCC are based on 32 bit architecture – each core has 4 GB as maximum amount of addressable memory – further mechanism is needed to access the 64 GB of physical system memory. This support is an extension, which has been realized as another level of memory indirection. Therefore, each core holds a separate Lookup Table (LUT) to translate the 32 bit core addresses into 46 bit system addresses. Each LUT has 256 entries. Its entries are configurable and point to specific types of memory regions, such as off-chip memory, on-chip memory, configuration-and-synchronization register, with a maximum size of 16 MB. In detail, each LUT entry holds the upper 32 bit of a system address, which consists of mesh coordinates, a target router port, and a specific memory segment. To complete the translation process, the lower 14 bit of a core address are added as an offset, whereas the upper 18 bit select the LUT entry.

2.2.3 Interconnect

The tiles of the SCC are connected by a 2D mesh. The mesh consists of routers, where a pair of routers is connected through bidirectional point-to-point connections. One router is located at each tile with four external ports. Consequently, a router has five ports, local, north, east, south, and west. Each router uses static round robin scheduling between the ports to forward packages. Virtual channels are used to avoid the blocking of incoming packets that target a free output port. The reason of such a blocking would be that another output port is blocked, because a packet that has been previously enqueued [Dal92].

A memory request is generated by reading or writing a memory location within the physical address space of a core. A request is split by the Mesh Interconnect Unit (MIU) into basic units, so called flits. The on-chip mesh interconnect of the SCC uses a fixed x-y wormhole routing to transfer data,

Table 2.1 shows average latencies for a memory access of a core to an on-chip and off-chip memory location with a certain distance on the mesh, which is detailed in the Intel SCC technical documentation [SCC12]. These numbers represent latencies of a tile boundary which can be used as a lower bound, because effects resulting from architectural overhead or collisions on the interconnect are not taken into account. To determine a more realistic communication performance, a communication and contention model is derived in the following in this chapter.

Table 2.1: Theoretical memory latencies [SCC12]

Latency Table	Approximate latency to read a cache line (output from the core to input back to core)
L2 access	18 core cycles
MPB access	45 core cycles + $8 * n$ mesh cycles
DDR3 access	40 core cycles + $8 * n$ mesh cycles + 30 cycles (400 MHz on-die memory controller)+ 16 cycles (400 MHz off-die DDR3 latency)
	n=number of hops to the MPB or the memory controller ($0 < n < 8$)

2.2.4 Basic Memory Types

Modern processors use virtual memory management to abstract details of main memory management. The operating system provides a separate address space for each process, with the result that virtual addresses are managed which are translated to physical addresses in hardware. A hardware unit, the Memory Management Unit (MMU), is responsible for the translation of virtual addresses to physical addresses. Page tables, which are commonly organized in multiple levels, hold the mapping of virtual to physical addresses as well as additional meta information, namely page table attributes. Due to the organization in multiple levels, the translation process is called table walk, a time consuming process consisting of multiple memory accesses and commonly accelerated by a Translation Look aside Buffer (TLB). As TLBs are critical for system performance, hardware solutions exist and optimization techniques have been evaluated, for example prefetching [KS02].

The following example in C programming language shows indirect access of data with a pointer.

```
1 int * ptr = malloc(sizeof(int));
2 int a = *ptr;
```

If this code example is executed within the scope of a process, the return value of the called function `malloc` is a virtual address. To read the data which is stored at this address, the virtual address is translated to a physical address. Here, the allocation of physical memory can be delayed until data is accessed. For the x86 processor architecture the following memory types can be categorized in the four following categories: *uncacheable*, *cacheable*, *write back*, *write through*. The configuration and the resulting use of caches is crucial in terms of memory performance [Dre07].

If a missing hardware cache coherence is taken into account, data has to be explicitly updated. This includes an implicit data transfer and leads to the question, how cache memories are used. For a write back policy, a valid strategy is to flush locally outdated values. If a write through policy is used, it would be sufficient to invalidate potentially outdated values. Through the method of page table attributes it is possible to tag data, for instance volatile data which is potentially in an inconsistent state. As a result, system software can distinguish between private and shared data.

A processor based on the IA32 compatible Intel P54C, such as the Intel SCC, can distinguish between three cache memory configurations [Intel95]:

- *write back:* In this configuration *reads* from and *writes* to the main memory are cached. A reading memory access can lead to a cache miss. On the basis of its caching policy, a core writes changes back from its cache to memory.
- *write through:* In this configuration, a processor core may use the cache to accelerate *reads* to main memory. *Writes* are written through to the main memory.
- *uncacheable:* In this configuration, access to main memory is not cached for the specific memory region. This behavior leads to the highest performance penalty in terms of throughput compared to other cache memory configurations.

The processor manual says that hardware registers should be accessed in *write through* or *uncacheable* configuration. Due to its direct data manipulation, *uncacheable* is the preferred configuration for hardware registers which are, for example, used for providing atomic operations to the SCC.

Option for Non-coherent Memory Access

Software controlled memory coherence provides certain advantages [LH89]. For example meta information can be used to selectively flush and invalidate data. In order to realize such a behavior, additional hardware support is needed to explicitly move selected cache-lines.

The concept of a cluster-on-a-chip architecture implies that a different access of private and shared memory regions is used. Intel's SCC introduces an additional memory option to handle access to shared memory regions. For selected cache memory configurations, on-chip or off-chip shared memory regions can be tagged.

- Message Passing Buffer Tagged (MPBT): Cache-lines with this option are placed only in Level 1 Cache and bypass all higher level caches. A write combine buffer is used to accelerate writes to regions with this memory type.

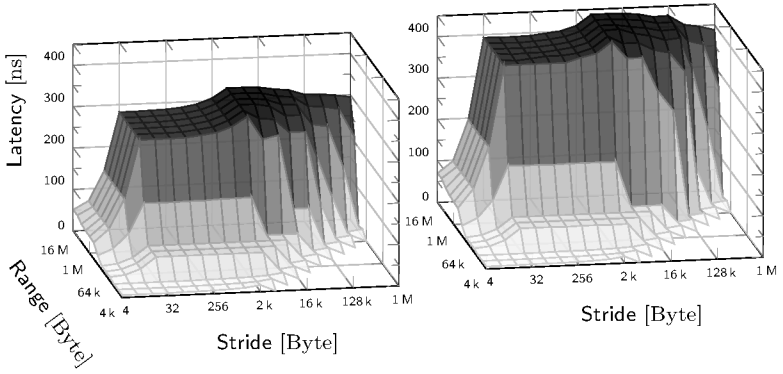
Memory regions in *uncached* or *write through* configuration can be combined with this new memory option for hardware distributed shared memory. As a result of this option, bursts of write operations are accelerated with a write-combining-buffer. The SCC extends IA32 by an instruction to support

fast invalidation of cache-lines which are tagged as MPBT. This is necessary to enable the use of caches for non-coherent shared memory.

Local memory performance

For modern processors with hardware support to hide latencies and to guarantee data consistency – for example out-of-order execution, prefetching, and cache coherency – results from the classic *membench* are no longer sufficient to characterize memory performance [Juc+04]. However, for an in-order architecture without hardware prefetching, such as the Intel SCC, results are sufficient for a characterization of the local memory performance. For shared memory regions, the scenario becomes more complex. A detailed analysis of the relation between memory access and tile distance, respectively memory contention is presented later in this chapter.

Figure 2.8 shows the results of *membench*, a benchmark that measures the average memory access latencies by running iteratively through an allocated array of size `range` with a step width of `stride` [McC95].



(a) SCC Core 0 (south west) (b) SCC Core 47 (north east)

Figure 2.8: Influence of mesh distance to memory performance

The varying latencies for different `range` and `stride` parameters arise from the effect of cache hits within a cache line (`stride` < cache line size) and the reuse of cached data (`range` < cache level size). As shown in Fig. 2.8a and

Fig. 2.8b (in addition to the memory latencies of the SCC platform and its cache sizes) the information that a cache line has the size of 32 B as well as the 4-way associativity of the L2 cache can be extracted. The difference between the two test cases is the distance from a core to its main memory located in off-chip DDR3 RAM. Thus, Fig. 2.8 compares memory latencies of a core located at the closest to a core located at the farthest mesh tile, compared to a single memory controller.

These measurements show a difference of about 100 cycles for the latency to the main memory of a core which is located close to a memory controller compared to a core which is located on the opposite side of the mesh interconnect. The default configuration of the SCC system takes this difference into account by placing private memory regions to the memory controller which has the smallest distance. As a processor core and its private cache is located on the same tile, latencies with a smaller range are independent from the distance of a tile to the memory controller.

2.2.5 Synchronization Support

Architectural support of atomic operations, for example *fetch-and-ϕ*, is a common mean for efficient synchronization of shared memory systems [KRS88]. In general, *fetch-and-ϕ* stands for a *read-modify-write* synchronization operation and can be implemented as: *test-and-set*, *fetch-and-add*, or *compare-and-swap*.

The hardware synchronization support of the SCC architecture is limited to atomic reads and writes on 1, 2, 4 and 32 B granularity. This represents an important attribute to set and get flag values, which is a common mean for the synchronization of distributed shared memory systems. However, regarding efficient implementation of lock-based as well as lock-free synchronization methods, read-modify-write operations provide several advantages [MS91].

In order to overcome this obvious limitation, the SCC provides a set of memory mapped configuration registers dedicated for synchronization. In general, dependent on the behavior and location of the unit that is configured, the access latency can differ in orders of magnitude. For instance, configuration registers to change frequency and voltage of cores, mesh, and memory controllers have a latency in a range of ms to µs. Core frequencies can be changed independently and have a range of 100 MHz to 1 GHz. They are fully configurable and only depend on a selected voltage domain, which spans across four cores.

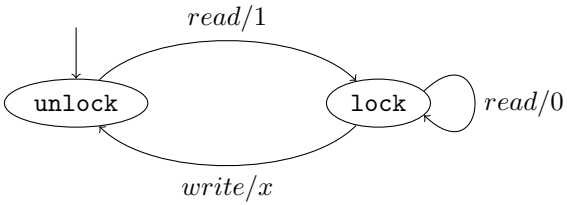


Figure 2.9: State transition of a *test-and-set* register

The interconnect of the Intel SCC does not support an encoding of atomic operations and consequently selected operations have to be emulated by additional units, so called synchronization registers. This method represents an extension to the processor architecture, without modifications to the core and network components. The same way as a configuration register, the access to a synchronization register is memory mapped with one or more memory locations, which is dependent on the complexity of the operation. By the use of synchronization registers, atomic operations, such as *test-and-set*, can be executed on these specific memory locations. Read-and-write requests trigger the synchronization registers to perform a certain operation on an internal value. To illustrate the behavior of the described synchronization register, Fig. 2.9 holds a diagram that describes two internal states of a test-and-set register and its transitions.

As a result busy wait implementations, which are commonly used for low latency synchronization constructs such as locks, can only be used with limitations on the SCC. By switching off or explicitly flushing local caches, on-chip memory as well as the off-chip shared memory can be used for flag based synchronization. Additionally, to emulate atomic operations, the SCC possesses a small set of special memory mapped hardware registers, namely Test and Set Register (TSR) and Atomic Increment Register (AIR). Consequently, synchronization registers can be used to implement a spin-lock.

Figure 2.10 on Page 31 gives an overview on average read latencies for all possible combinations of source and target location of SCC's mesh interconnect. The diagram shows in x direction the variation of target synchronization register and in y direction the cores that access a register. These measurements verify the assumed linear relation between mesh distance and on-chip access latency.

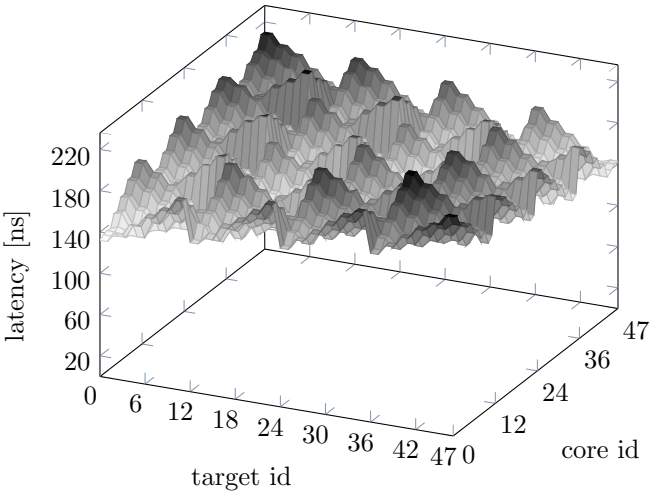


Figure 2.10: Average latencies (in z direction) of synchronization register access from different tiles (core in y direction) to different tiles (target in x direction). [SCC12]

Linear Back-off

For evaluation purpose it can be useful to trace information on a memory access, such as latency or ordering. If the tracing targets a high resolution and therefore the additional memory access is in the critical path, a low and predictable overhead is intended.

Our measurements have verified a linear relation between latency of explicit on-chip memory access and its communication distance. As presented in this chapter, with a low load on the interconnect and memory system of the architecture the latency for a single memory access can be predicted with a high precision. The combination of this predictability with a quantified back-off, can be used to store for instance tracing results including a constant and location independent overhead.

An example to apply this technique, is the realization of a *unified global counter*. Assuming that the access to this counter is defined as a critical section and the communication distance to the counter is constant, dummy operations can be added to each access and thereby unify the costs for an

access from each core. Our concept of a *unified global counter* is valuable for debugging and collecting statistical data, such as record the order of events.

The fact that the cores of the Intel SCC are based on the P54C architecture creates the possibility of a high resolution back-off by using the `nop` instruction. Because of its 2 way superscalarity, an SCC core can perform 2 simple operations, such as a `nop`, per cycle. Dependent on the core frequency, an advantage of the simple architecture is that a precise back-off can be precomputed and realized by executing `nop` instructions in a loop.

2.3 Communication Model

The understanding of memory contention and network congestion for a many-core architecture that follows the network-on-chip paradigm is a primary goal to optimize access patterns for on-chip memory. Contention is a result of multiple cores that access a single on-chip memory location in parallel. It depends on the costs of contention, in terms of increasing latencies, if a synchronization algorithm should rely on a central access pattern or a distributed access pattern, or a combination of both. Consequently, a quantification of the costs of explicit memory accesses to hardware distributed shared memory regions is essential. A communication model for this kind of memory is developed to achieve this goal, with the following assumptions:

- each core has explicit access to a certain amount of hardware distributed on-chip memory
- read and write operations to on-chip memory have identical costs
- on-chip memory locations can be accessed in different memory configurations

As a first step, a simple ping-pong communication scheme is investigated to qualify the best case communication performance without contention. As a second step the synchronization overhead is quantified, and optimizations for a cluster-on-a-chip architecture are discussed. The results have been used for the optimization of a layer that abstracts on-chip data access and realizes efficient synchronization and communication. This upper communication layer is studied in the next chapter.

2.3.1 Related Work

The *LogP* model family names a class of basic communication models with a small set of parameters [Cul+93]. The classic LogP model leaves out effects caused by network contention and uses four parameters to describe communication. The four parameters of LogP are:

- **L**: latency of the communication medium.
- **o**: overhead of sending and receiving a message.
- **g**: gap between successive send or receive operations
- **P**: number of processor and memory modules

Extensions to this model exist, that replace or add parameters to model communication by including the cost of contention C, such as *LoPC* and *LoGPC*. Frank et al. focus in *"LoPC: Modeling Contention in Parallel Algorithms"* on the communication with active messages, that have small message sizes and to execute code at receiver's side [FAV97]. Moritz et al. extend with *LoGPC*, the *LogP* and *LogGP* model by network contention and collision for message processing [MF01]. *LogP* has not been designed to model memory access, nevertheless it is accurate in predicting the on-chip memory access of the Intel SCC without contention [Rot11].

Already in 1987, Yew et al. explored the *Distributing Hot-Spot Addressing in Large-Scale Multiprocessors* [YTL87]. This study targets clusters with thousands of processors. The resulting conclusion is that serious performance problems can occur, if a small percentage of accesses targets a single memory location. Regarding many-core systems with software controlled on-chip memory, this issue can even occur without connecting multiple processors.

The focus of this work is on a tiled many-core system that implements the network-on-chip paradigm and enables explicit on-chip memory access. Studies exists that express research challenges for on-chip interconnects in general and evaluate congestion for specific types of networks [Owe+07; NM93]. Dally studies the performance of networks using virtual channels [Dal92]. This work is interesting for research with the SCC because its interconnect applies the technique of virtual channels.

Few practical approaches exist that follow the LogP communication model with extensions to derive optimizations for communication of real hardware in an analytic way [Hoe+05]. Ramos and Hoefler present a communication

model for cache coherent SMP systems [RH13]. In this work, it is shown that a communication distance is not measurable for the Xeon Phi, if the cache is implicitly used for inter core communication. This situation changes for an architecture with a lower memory abstraction and explicit access to on-chip memory.

For a first evaluation of the SCC's theoretical communication performance, the focus of the subsequent paragraph is on a simple communication pattern. A message is transferred with a one-sided *put* operation of data to local on-chip memory, which implies a remote get communication scheme on receiver's side. A simple synchronization scheme works with a so called *canary* in the last cache-line that is transferred. A *canary* is a unique tag that holds a specific information, for example a completion of the *put* operation.

2.3.2 Analysis

This section covers the detailed analysis of data movement characteristics for a cluster-on-a-chip architecture. We use a low-level benchmark routine to derive parameters for a communication model and to quantify the predictability of memory access [RW14]. Our communication model leads to a deeper understanding of the interconnect and of the identification of bottlenecks. Moreover, we present in this work a statistical analysis for the Intel SCC to verify our communication model.

In addition to the average latency, we are interested in the overall distribution of latencies. This information is important, especially if we want to quantify the predictability of a communication path.

For all x86-based processors since Pentium, a high resolution and low overhead timing measurement is possible. The *time stamp counter* is a 64 bit register, which counts the number of CPU cycles since reset and is accessed by the `rdtsc` instruction. For modern processors a slightly different metric is implemented in order to calculate the elapsed time for a code fragment while scaling the processor frequency. Another invariant can represent the overhead of the instruction to calculate timing differences, for example, for out of-order architectures. We have measured a constant overhead of 13 cycles, which represents the lowest possible granularity for the simple in-order cores of the SCC. The processor frequency for the actual timing interval was set to a constant value for all of our measurements due to comparability reasons, and unless otherwise noted the cores are running at 533 MHz.

The setup of the measurements, which are presented in this section, is that a single process is started at each core. Since each process runs with a separate operating system instance, a pinning of processes to cores is not necessary because a migration is not possible. One method that we have selected for low level timing analysis is to disable all sources of interrupts for the benchmark runtime and thereby emulate bare-metal execution. Otherwise for the identification of outliners, effects of the operating system have to be taken into account.

2.3.3 Multi-Line Ping-Pong

The benchmark which we analyze in this section copies multiple chunks of data between two cores. As this benchmark targets a measurement of the on-chip inter-core communication performance, it transfers data between private-and-shared buffer. Parameters have to be chosen within certain bounds to exclude a pure benchmark of the cache-memory system. We specify those architecture dependent parameters exemplarily for the Intel SCC.

For low latency communication, on-chip memory is the preferred location for allocation of shared buffers. Several constraints exist regarding architecture dependent buffer sizes for this benchmark. As we target the evaluation of hardware support for message passing, a limitation resulting from the private buffer access has to be avoided. Consequently, both buffers should fit into the first level cache of each core. In addition to that limitation, the smallest granularity of a data transfer is the size of a cache-line, which is also architecture dependent.

The Intel SCC has 16 kB of software controlled on-chip memory per tile, respectively 8 kB per core, a cache-line size of 32 B, and a first level cache size of 16 kB. Consequently, 8 kB of memory is allocated for each core as a private buffer and as a shared buffer. The cache for the private buffer memory region is configured as *write back* and for the shared buffer as *write through* with MPBT option.

The *ping-pong* communication pattern, which is illustrated in Fig. 2.11, can be described as follows: First, core A transfers n cache-lines by copying the data from its private to the shared buffer. Second, core B transfers the data from the shared to its private buffer. Next, the procedure is repeated, whereas the cores change their roles: B is the sender and A is the receiver.

A coordination of the shared buffer is enabled by writing a predefined value, a so called canary, into the last byte of the last cache-line that is transferred

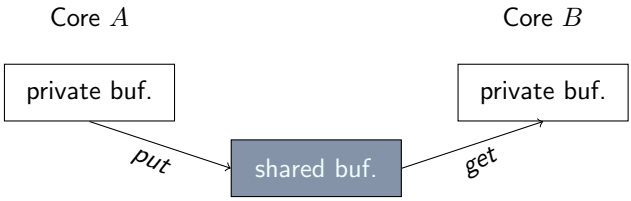


Figure 2.11: Multi-line ping pong communication scheme

between the cores. A requirement of this synchronization scheme for the applied communication protocol is that the architecture provides *strong write ordering*. This is valid for x86 architectures, that implement total store ordering [OSS09].

Figure 2.12 holds round-trip times with 5000 repetitions for different transfer sizes, which are plotted on a logarithmic scale. For the sake of clarity, the diagram holds only measurements for the minimum communication distance, where each packet crosses a single router. A detailed analysis has shown, that latencies of a point-to-point connection without additional load on the mesh interconnect have a low jitter and a high predictability for this kind of architecture [RW14].

As a result of this analysis, we will next estimate the communication times dependent on the message size. The dashed curve of Fig. 2.12 on the previous page shows communication times for n byte which are calculated according to the following formula:

$$t_{comm}(n) = o + n \times t_{transfer} = 204\,cycles + 9.3\,^{cycles}/_{byte} \times n\,byte \qquad (2.1)$$

For the observed ping-pong communication scheme an upper limit for unidirectional communication throughput can be calculated if the round-trip times are divided by two. For large values of n the overhead is negligible and the maximum throughput is about 114 MB/s for a minimum distance between communicating tiles on the mesh. If an increasing communication distance is integrated as an additional parameter to Eq. (2.1) this influences only the factor n. The detailed measurements of the multi-line ping-pong benchmark corresponds to the numbers of Table 2.1 and the factor rises to $12.3\,^{cycles}/_{byte}$ for a maximum communication distance of the SCC of 8, which results in a maximum throughput of 87 MB/s.

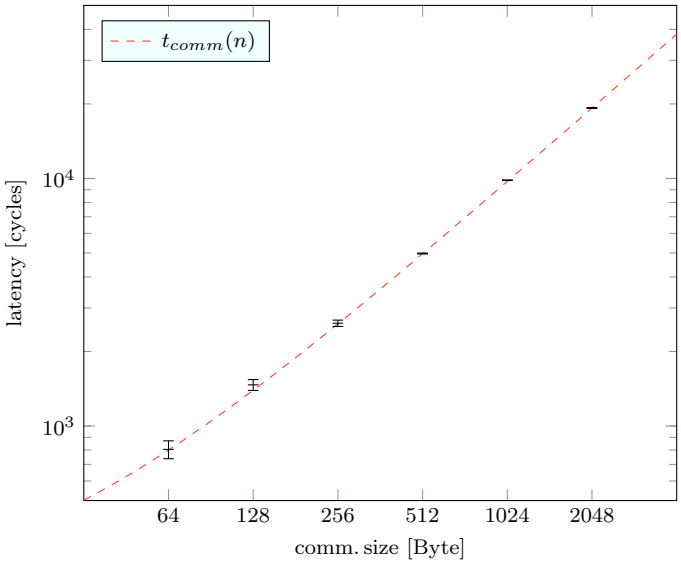


Figure 2.12: Multiline pingpong

For small messages, a major part of the overall latency consists of the synchronization overhead. In the subsequent part of this section, the overhead of using flags for synchronization purposes is analytically derived and quantified.

2.3.4 Quantify synchronization overhead

The preferred location of a flag value is on-chip memory, because *reads* and *writes* to flags are latency sensitive operations. Moreover, continuously reads to remote memory locations stresses the interconnect. In Section 2.2.4, different cache memory configurations have been described, which can be used for a flag. If consistency of a synchronization flag has to be explicitly controlled, the cost of a flag access is directly dependent of the configuration. Resulting costs of reading and writing flags influence the overall communication performance. The cost of a flag access has a major impact to the design of a complex communication protocol and the expected throughput.

The time $t_{op}(d)$ for the *read* and the *write* of a value to a distributed shared memory location can be calculated with the following formulas:

$$\begin{aligned} t_{write}(d) &= t_{mpb}(d) + t_{flush} \\ t_{read}(d) &= t_{mpb}(d) + t_{invalidate} \end{aligned} \tag{2.2}$$

Those latencies can be defined as functions of the parameter d, according to the communication distance in hops. Obviously, the distance is 0, for a flag located in the local buffer.[5]

The terms t_{flush} and $t_{invalidate}$ are dependent on the memory configuration. Assuming MPBT data for the SCC, the cost of data invalidation is constant 1 cycle, whereas the flush operation is dependent on the content of the *write-combining buffer*. For the SCC, the *write-combining buffer* holds exactly one cache-line of 32 B and writing the last byte flushes the buffer implicitly. Because an operation to flush the *write-combining buffer* explicitly is missing, only implicit flushing is possible through a write operation to another memory location with MPB tag.

A technique to minimize the synchronization overhead is discussed in the next paragraph, with different memory configurations for on-chip memory access.

Optimization

According to the different modes of addressing shared memory for a x86 based many-core, that are described in Section 2.2.4, a direct influence on the latency of memory operations is expected.

For a shared memory access in *write through* configuration and with MPBT option, the *write-combining buffer* as well as the first level cache is used for acceleration. The costs of a blocking flush operation of the *write-combining buffer* depends on its state, which can be full, partly filled or empty. A flush operation can either be triggered by a special instruction, or by filling the buffer with dummy data and thereby flush the buffer in an implicit way. Whereas the first method method requires an extension of the instruction set, the second method only requires information on the architecture, such as size and behavior of the *write-combining buffer*.

[5] For the SCC the maximum distance between two tiles is 8.

Regarding the SCC architecture, the state of the buffer is relevant to predict the cost of a flush operation, due to the behavior of the *write-combining buffer*. If the buffer, which holds a single cache-line, is partly filled, each byte is written independently to target memory location. This behavior leads to a maximum number of 31 memory access as a result of an implicit flush operation, which is relevant for latency sensitive operations.

Due to the missing coherency, caching does not represent a benefit for a flag access if a remote memory location can be accessed in an explicit way. The main problem is not the cost of invalidation to force the reload of a flag value, but the flush operation of the *write-combining buffer*, for the investigated scenario. We propose *uncacheable* mapping without *MPBT* configuration to minimize the costs of flag-write operations. A side effect of this optimization is that write operations to flags can pass buffered write operations, for example a write to the communication buffer.

As a consequence, the communication protocol has to ensure that data is written through to memory. This can be done by selectively flushing the *write-combining buffer*. Additionally, the receiver has to ensure that the date is present, for instance polling on the last cache-line for a canary value. Another way to overcome the described issue is a blocking behavior of a flush operation of the *write-combining buffer*. Dependent on the size of the mentioned buffer this blocking time t_{flush} can become significant.

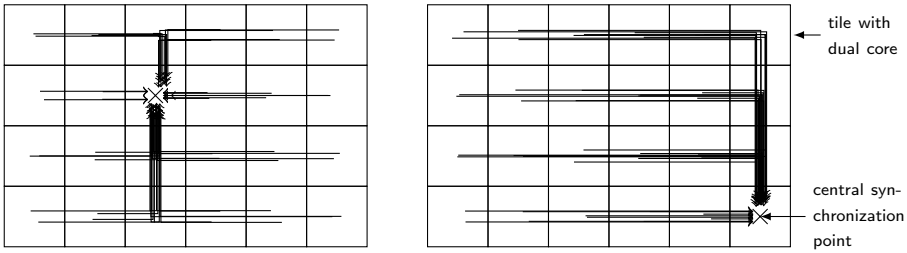
With a bypass of the *write-combining buffer* for latency sensitive transfers, the described optimization can reduce the synchronization overhead of the SCC up to a factor of two. For RCCE family, we have implemented this optimization, the so called *bypass flags*, by mapping the LMB in different memory configurations. Influence of this optimization to point-to-point and collective communication for the Intel SCC is presented in the next chapter.

2.3.5 Contention

For the efficient design of synchronization methods, especially of central synchronization points, it is important to quantify the cost of contention. Contention can be defined as a competition for *shared resources* by parallel *units of execution*. In the context of this work, typical examples for *shared resources* are: processor caches, main memory, and i/o ports. Common examples for parallel *units of execution* are: threads and processes.

Studies have shown that, for cache-coherent systems, a contended shared-memory location mainly stresses the coherency management [Cha+05]. Here,

the cache-coherency controller represents a bottleneck, as a central instance that controls concurrent accesses. Further assuming that this limiting factor is released for many-cores with multiple coherency domains, the access to on-chip memory can partly be handled in an explicit way. This alternative way of accessing low-latency remote memory, will have an impact on the design of synchronization methods. The assumption is that memory contention and resulting network congestion can harm scalability. One important task is to quantify the resulting overhead for a given architecture to achieve good algorithmic performance. For instance methods which are based on central synchronization points can cause contention. Figure 2.13 compares the access pattern of a central synchronization point between a central location and a corner location of a mesh interconnect.



(a) center location (b) corner location

Figure 2.13: Access pattern of a central synchronization point. Arrows illustrate a reading access from a processor core to a central synchronization point.

For low latency operations, the access pattern is of significant importance for architectures that follow the network-on-chip paradigm. A common effect is a linear rising latency because of a larger node distance. Moreover, the routing policy can directly influence the latency of a memory access

The cost model for contention, which is derived in this section, takes attributes of the interconnect into account and is based on the following assumptions:

- Symmetric tile frequencies, which implies an identical request and service rate for each tile

- 2D mesh interconnect without congestion control
- Static routing algorithm between the tiles (x-y routing)
- Routers provide FIFO queues to store incoming flits
- Routers implement a round-robin scheduling to select the next flit which is forwarded to a neighboring router

Up to this point, conflicting memory accesses have not been taken into account. If we want to model such an effect, we have to quantify the number of outstanding memory requests per tile and decide if the network or the target memory location is a bottleneck. As memory requests are an attribute of the basic core architecture, we devise a simple formula to calculate the number of memory requests per row.

$$\frac{\textit{memory requests}}{\textit{row}} = \frac{\textit{tiles}}{\textit{row}} \times \frac{\textit{cores}}{\textit{tile}} \times \frac{\textit{memory requests}}{\textit{core}} \tag{2.3}$$

For instance, the Intel SCC has a *memory-request-per-core* ratio of 1 due to its P54C basic core architecture.

The routing algorithm is also of major importance to derive a contention model. In the scope of this work, we assume that a static x-y routing policy is used to forward flits. Moreover, we assume that if a router is blocked, flits are stored in a buffer. For these buffers, we further assume that the administration of pending flits is handled according to a First In, First Out (FIFO) policy.

First, we assume that the local memory buffer is saturated, which is a valid assumption for this communication scenario. Consequently, the accumulated bandwidth of all accessing tiles is equal to the local MPB bandwidth, which can be normalized to 1.

$$\sum_{x=0}^{n_x} \sum_{y=0}^{n_y} b_{x,y} = b_{LMB} = 1 \tag{2.4}$$

If the condition is met, that the accumulated request rate of all tiles is sufficient to saturate one port, the available bandwidth per router can be divided by the number of incoming ports, due to the round robin scheduling of incoming flits. If this condition is fulfilled, the available bandwidth per tile can be calculated by subsequently dividing the bandwidth by i starting at the router, which is directly connected to target on-chip memory location.

Here, i stands for the number of saturated incoming ports of each router. Furthermore, the assumption is valid that tiles which are connected in the same row share the available bandwidth, if a static x-y routing is used. In this communication scenario, the average latency of a memory request can now be calculated by the inverse of the memory bandwidth.

The factor of rising memory latencies, as a result of contention, is illustrated in Fig. 2.14, exemplarily for a 6×4 2D mesh with 2 cores per tile.

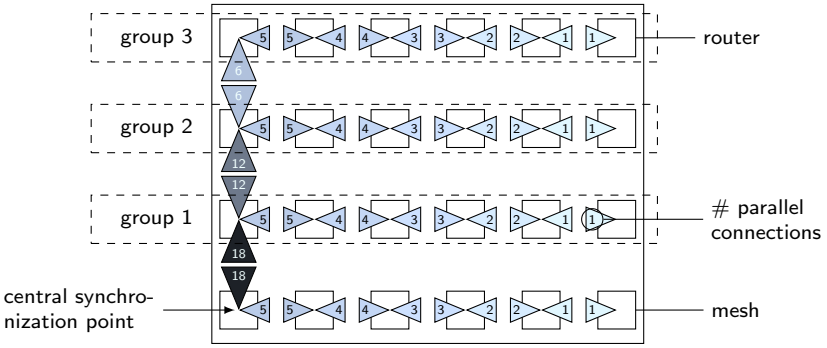


Figure 2.14: Contention and concurrency groups resulting of a hotspot on a 2D mesh with x-y routing. Figure shows the Intel SCC as an example with one active core per tile.

We assume that target on-chip memory is located in a corner of the mesh, which represents the worst case scenario. The numbers of active links are labeled for each router, which means outstanding requests at a certain point in time. The color of each router port indicates the expected degree of contention, with white for low contention and dark gray for high contention. The result for the given architecture is an unfair sharing of available bandwidth, when the accumulated request rate of the computing cores is higher than the service rate of target memory, This means that target memory location is saturated.

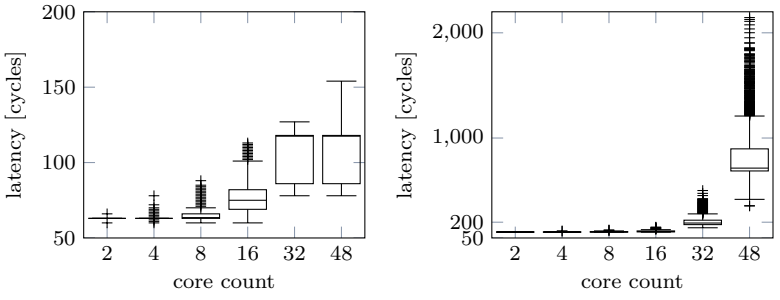
As a result, spinning cores can be categorized into concurrency groups according to its priority, in the case of a static routing algorithm. Of course this mapping of cores to concurrency groups depends on the remote memory location. For instance in case of x-y routing and if the on-chip memory is

located in the corner of the mesh, all cores that share one row have the same priority.

For a representative dataset, 5000 data points have been recorded, for an explicit memory access of each core. Results are presented as classic box plots [MTL78], which is briefly described in the following paragraph.

The median is an upper bound of exactly 50 % of all measurements. In other words, this value divides a set of samples by two, so that 50 % of samples are higher and 50 % of samples are lower than the median. Each data set has a lower and upper quartile, which represents an equivalent border for 25 respectively 75 percent of the measurements. In a box plot, the quartiles are the upper and lower end of the boxes, while the median represents the line that separates the box. The distance between quartile and median is multiplied by a specific factor, commonly 1.5, to obtain the maximum length of a whisker. A whisker spans an interval for all other values outside the quartiles. Data points which are outside this interval, are marked as outliners.

The two diagrams in Fig. 2.15 show the latencies for two different communication distances. The nearest communication distance (1 hop) is plotted in Fig. 2.15a and the largest communication distance (8 hops) is plotted in Fig. 2.15b. The diagram shows the memory latencies, on the y-axis, in relation to the number of cores that access on-chip memory in parallel, on the x-axis.



(a) near (b) far

Figure 2.15: Latencies of on-chip memory access under contention

A significant result of the experiments is the fact that tiles are grouped into domains which do not necessarily depend on their absolute distance to

the memory location. Our experiments have shown, that the tiles of the Intel SCC have to be grouped according to their y-coordinate on the mesh.

In the next paragraph, the effect of mesh distance to fairness of central busy-wait synchronization point is analyzed.

2.3.6 Back-off with Feedback

Smai and Thorelli have targeted congestion in networks through the avoidance of bandwidth saturation by choosing an appropriate timeout, a so called back-off [ST98]. An exponential function is often used as a good approximation to calculate parameters of a back-off function used in networks [KSM05]. The combination of predictability and low-latencies of an on-chip network generates further potential for optimization.

A low memory abstraction for many-core processors implies an explicit access to on-chip memories. Especially for architectures with a memory-access path that crosses the network interconnect, memory contention can cause network congestion. In general, the load on the interconnection network is dependent on the number of outstanding requests at a certain point in time. Furthermore, it is assumed for the analysis of the Intel SCC that only a back pressure mechanism is applied to control the number of outstanding requests.

The routing algorithm and the existence of buffers to temporarily store packets at a router are additional attributes of an on-chip interconnect that can influence the occurrence of network congestion. For the SCC, two outstanding memory requests can exist per tile. The routers have FIFO buffers and virtual channel support.

As a result, according to the definition of concurrency groups in Section 2.3.5 an appropriate back-off can be derived to realize busy waiting to a remote memory location. As mentioned before, the basic approach is to overcome contention of a single synchronization point, for instance resulting of a spin-lock or a barrier which uses a central counter.

2.4 Conclusion

In this chapter we describe and analyze the basic components of a x86-based many-core architecture, which follows a network-on-chip paradigm. This includes an overview on memory organization, processor core and network interconnect, with a focus on their interaction.

The basic organization of this work is bottom-up. The structure is based on a layered approach of memory abstraction for a many-core processor. This chapter covers the basic communication layer, which is used in the remainder of this work as a hardware-abstraction layer. In order to predict the costs of on-chip communication, we have developed a communication model for a cluster-on-a-chip. The research-processor SCC by Intel is used as an example to derive parameters and demonstrate the applicability of the communication model in practice. The development of such a model represents a key contribution to explore the limits of a processor architecture without full chip cache coherence in terms of communication performance.

Moreover, we quantify the costs of contention and identify consequences resulting from explicit on-chip memory access. To overcome limitations of a given many-core architecture, we propose a quantified back-off with feedback to realize central synchronization points. The insights of this low-level analysis can further be used to develop efficient communication methods.

3

"A communications disruption could mean only one thing: invasion" [Star99]

Communication and Synchronization

For a many-core architecture that implements a network-on-chip processor with many small computing cores, programming in various abstraction levels from application to system software development has a tremendous demand for effective communication and synchronization.

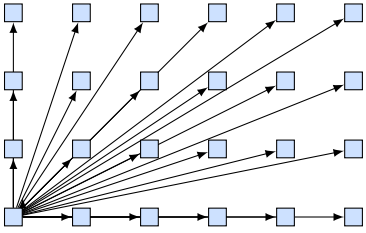


Figure 3.1: Example of a collective-communication pattern for a 6×4 mesh

The focus of this chapter is the design and implementation of a low-level communication and synchronization interface for tiled processor architectures. This interface targets especially processors which follow a many-core design approach, including software controlled on-chip memory or even configurable coherence domains per chip.

In this chapter, a communication model is used to explore the limits for such architectures. Different communication patterns are analyzed, for example as illustrated in Fig. 3.1 on the previous page. As a result of this analysis, we have developed and analyzed alternative communication methods for non-coherent memory coupled clusters with software controlled on-chip memory. In addition to that, we use the communication model to predict different protocols for point-to-point communication and derive optimizations.

At RWTH Aachen University, we have developed with iRCCE a low-level communication library for the Intel SCC. In this chapter, its feasibility is demonstrated and its performance is evaluated with the use of micro-benchmarks. Results of the NPB application benchmarks are discussed in the following chapter.

The main difference of Rock Creek Communication Environment (RCCE) family[1] compared to existing interfaces and libraries for many-core systems is a resource aware communication design for direct on-chip message passing. If future processor architectures provide remote memory access to software controlled on-chip memory for low-latency communication, the word remote will no longer be related to a physical distributed memory location and message passing will become an even more attractive option for on-chip communication. My contribution to the work which is presented in this chapter are in particular the following points:

- Specification and analysis of a communication model for a low-level communication environment.
- Development of sophisticated point-to-point communication that achieves low overhead.
- Implementation of classic software techniques to hide latencies.
- Optimization of synchronization constructs for the Intel SCC.

The approach of implementing memory coherence in software as covered in the next chapter, has a strong demand for a low-level interface, such as iRCCE. Resulting requirements of this specific application are a low latency for messages between coherence domains and the option for asynchronous communication.

[1] Including RCCE and its extensions iRCCE, RCCE_comm, ...

Organization of this Chapter

The main concept of this chapter has been first presented in our paper: *"A Fast Inter-Kernel Communication and Synchronization Layer for MetalSVM"* [Reb+11]. A further development of this concept, especially regarding the analysis of different communication schemes for a cluster-on-a-chip processor and the extension of an existing research system has been published in: *"Connecting the Cloud: Transparent and Flexible Communication for a Cluster of Intel SCCs"* [Reb+12a].

Both concepts have been further developed as presented in this dissertation. This includes the interaction of communication layers, which is illustrated in Fig. 3.2 whereas the focus of this chapter on RCCE family is highlighted.

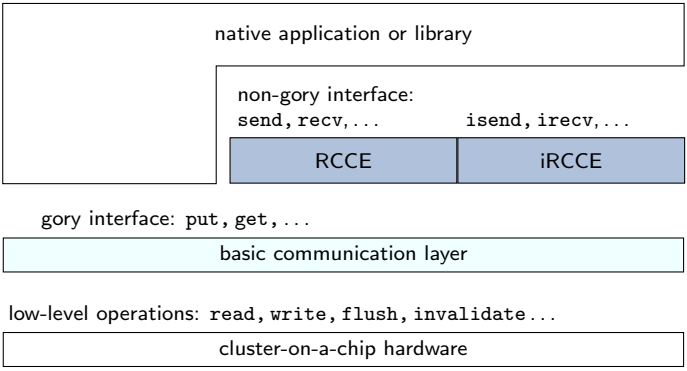


Figure 3.2: Focus of this chapter in relation to the investigated layered communication structure

This chapter is structured as follows: First, related work to the development of a low-latency communication infrastructure is summarized. The second section of this chapter summarizes and analyzes hardware and software extensions for an experimental many-core system. This includes support of atomic operations for the SCC. and an extension of its research framework to emulate a cluster-on-a-chip with more cores.

The third section of this chapter focusses on an extension of the low-level communication environment RCCE. This work is based on the integration of

iRCCE [Cla+13b], a non-blocking and low-latency communication and synchronization layer that features a fully asynchronous mailbox-system, to a bare-metal framework of the Intel SCC [Reb+11].

The fourth section covers optimizations of message passing buffers, such as a dynamic allocation scheme, to enable low latency, high throughput and scalability.

Last section of this chapter covers the implementation of high performance synchronization constructs, in terms of low latency and scalability. Some of the presented results, which are discussed in the second part of this chapter, have been published in our previous work [Reb+12c]. Also, this previous publication covers the analysis of the impact of a high performance on-chip mesh interconnect to common busy-wait synchronization methods.

3.1 Related Work

Computing nodes, which are commonly coupled by a dedicated high-performance interconnect, are called compute cluster, often shortened to cluster. In the common understanding, a cluster consists of a set of loosely or tightly coupled computer systems, that execute the same application. Dependent on the programming model, a communication interface is typically used to abstract hardware details of a specific fabric interconnect, such as Infiniband, PCIe or Ethernet.

An unconventional cluster is a cluster system which consist of components that have not been developed or not intentionally designed for a use in high-performance computing (HPC). One example is the use of small in-order processor cores to build many-core processors with a computing power beyond 1 TFLOP/s. A cluster consisting of processors that provide scratchpad memory, can also be seen as an unconventional cluster, because explicit access to on-chip memory has not been established in the x86-based processor landscape. The SCC is a research processor which can be categorized as a distributed shared memory system. It waives full chip cache coherence and provides hardware support for on-chip message passing. Today, from an HPC perspective such a system can be seen as an unconventional cluster.

For conventional clusters, that consist of nodes with multi-core processors which are connected by a dedicated high-performance interconnect, the established parallel programming model is message passing. This model is based on the assumption that hardware-distributed memory is not shared by default

and that communication is handled explicitly. Besides pure message passing, hybrid models exist that combine message-passing and shared-memory programming models to explore node-level parallelism.

MPI is a de-facto standard for message passing in HPC, by defining a broad set of communication functions, including classic two-sided communication (`send` and `receive`) and one-sided communication (`put` and `get`) behavior. As a full featured API, MPI supports a diverse range of parallel workloads and programming models. The interface and its application is described in one of the subsequent paragraphs in more detail.

Besides MPI, a broad range of low-level communication interfaces exist, which are not necessarily independent of a specific vendor or designed with a specific hardware in mind, especially for embedded systems. Nevertheless, existing low-level APIs typically provide an abstraction of communication-device details.

With MCAPI [MC11], the Multicore Association defines a portable API for message passing to abstract communication and synchronization for distributed embedded systems with closely coupled cores. A two-sided communication interface is specified, which includes support for different communication channels.

GASnet [Bon02] and ARMCI [Nie+06] are based on the approach of a one-sided communication interface. Both approaches assume native direct memory access (DMA), which is supported by network specific drivers or globally addressable shared memory.

Symmetric Communications InterFace (SCIF) defines an API, which exists in two programming styles and has intentionally been developed for communication within an Intel platform between Xeon processor (host) and Xeon Phi coprocessor (device). Dependent on the programming space, such as kernel and user space, specific functionality is provided [SCIF14].

RCCE as a light-weight communication layer has been developed by Intel Labs for their many-core research processor SCC [MvW11]. The related communication interface and its reference implementation, both called RCCE, is described in Section 3.1.2.

In addition to that, RCCE family names the original communication environment and its extensions iRCCE and RCCE_comm [Cla+13b; Cha10], which have been developed by members of the MARC community [MARC]. In the scope of this dissertation, the term RCCE family will be used to describe the general communication concept independent of a concrete implementation.

OpenSHMEM [Poo+11] is a standard for SHMEM libraries, which shares similarities to the RCCE family. This standard targets the issue that each vendor provides a slightly different shared memory programming library. However, optimizations have been applied with a specific hardware in mind, such as the Cray MPP systems.

SISCI names an API by Dolphin, which creates a software infrastructure for programming of clusters with shared memory interconnects, with PCIe as the latest hardware-generation base. Another API in this scope is Shared Memory Interface (SMI) [DSB99], which has been developed at RWTH Aachen University for SCI [SCI93].

3.1.1 Message Passing Interface

Message Passing Interface (MPI) is a well established and widely used standard for programming distributed memory systems, in a way that communication has to be explicitly handled. For instance, communication in parallel programs can be explicitly expressed with message passing between processes. Specifically, communication consists of moving data from one process to another through cooperative operations. Message Passing Interface itself does not represent a programming model and is not limited to one However as the name suggests, it primarily addresses the message-passing parallel-programming model.

MPI-1.0 has been released in 1994 [MPI94]. The further development has followed the concept of extensions, such that MPI-2 mostly represents a superset of MPI-1. At the state of this work, the most recent version of the Standard is 3.0, which has been released in September 2012.

MPI supports the SPMD programming paradigm, which is a prevalent and easy-to-use parallel programming paradigm where multiple processes execute a single program [GLS94]. These processes communicate by means of an MPI communication library.

Communication can be categorized if communication parameters are two sided specified by involved processes, both on sender and receiver side. Since version 2.5 MPI extends communication functions by so called one sided communication, whereas parameters are specified one-sided.

Different libraries exist that implement the MPI standard. Widely used MPI implementations are for instance OpenMPI and MPICH, besides vendor specific implementations. Some behavior is implementation specific, such as the support of low-level API's or network interconnects of a specific vendor.

3.1.2 RCCE

RCCE (pronounced: *rocky*) is a communication environment which has been developed by Intel Labs for the SCC (codename: Rock Creek), to support many-core software research [MvW11]. The abbreviation RCCE stands for Rock Creek Communication Environment, and is used in synonym for a communication interface, as well as the reference implementation of a communication library [vWMH11].

The communication library follows the SPMD paradigm, where a single RCCE process is started on each processor core of the Intel SCC. Because the concept of a small communication layer does not specify sharing of an address space, RCCE uses the abstract term Unit of Execution (UE), which describes an entity that changes the program counter. UEs can represent processes or threads that are executed in parallel, with the common attribute that each UE has its unique rank.

For communication between UEs and their synchronization, RCCE provides two APIs, *gory* and *non-gory*. In contrast to the *gory* interface, which enables direct on-chip memory access with related `put` and `get` functions, the *non-gory* interface abstracts direct access to software-controlled on-chip memory. The connection between both interfaces has been introduced in Chapter 2. The *gory* interface represents a subset of the basic communication layer, that has been described in an abstract way.

The environment is applicable for a many-core system with software-controlled on-chip memory and multiple coherency domains. In the context of the SCC, these on-chip memory regions are called Local Memory Buffers (LMBs) and partly used as Message Passing Buffers (MPBs) for inter-core communication. Figure 3.3, which can be found on the next page, illustrates the memory architecture of the Intel SCC as used by RCCE.

The RCCE library implements a simple communication protocol for low-latency point-to-point communication. For the realization of blocking two-sided communication with the Message Passing Buffers, a *local put, remote get* communication scheme is used. The reference implementation of RCCE is limited to this communication scheme, whereas for its extension iRCCE alternative schemes are provided.

The default scheme of RCCE implies that the MPB of the sender is used to transfer a message, which simplifies the coordination of access. The RCCE library implements a flag-based synchronization scheme, that relies on atomic read and write capabilities of the SCC. In its nature as an x86-based system,

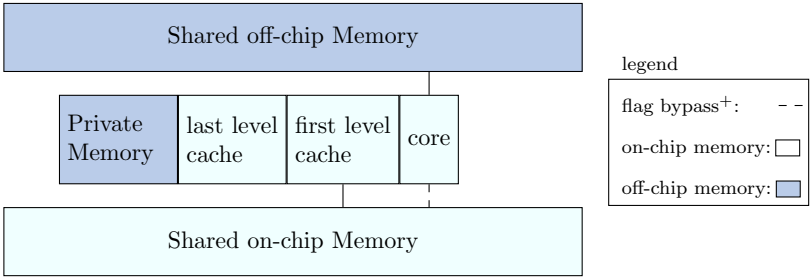


Figure 3.3: Memory architecture of the Intel SCC as used by RCCE [Mat+10] and extended by iRCCE$^+$ (optional)

the SCC architecture supports atomic updates on byte (B), half-word (2 B), word (4 B) granularity to data which is not MPB tagged. For MPB tagged data, a write combining buffer extends this support of atomic updates to cache-line (32 B) granularity. Due to this hardware feature the default flag size of RCCE is 32 B.

With version 1.1, RCCE introduces a software workaround to explicitly flush the *write-combining buffer* of the SCC, which enables flag access with a smaller granularity (1 B). This shrink of flag size has the main advantage that more space of the LMB is left for the MPB. In contrast to the described options with lock-less flag access, RCCE provides a third option which locks access to synchronization flags. This option decreases the communication throughput by 10 to 20 % because it increases the synchronization overhead. This is a result of a reduction of the flag size to a single bit [Mat+10].

The interaction between the communication functions `RCCE_send()` and `RCCE_recv()` are described in the following, if the default protocol is used. First, the sender puts the message to its local MPB. Second, the sender toggles a flag at receivers side to indicate the event. Finally, the sender waits at a synchronization point until the receiver indicates the completion of the pending message-copy operation to its private memory. This implements a two-copy communication scheme, because a message is copied from a private to a shared buffer and back to a private buffer.

Figure 3.4 shows a first estimation of on-chip communication latency between two tiles for the SCC, according to Howard et al. [How+11].

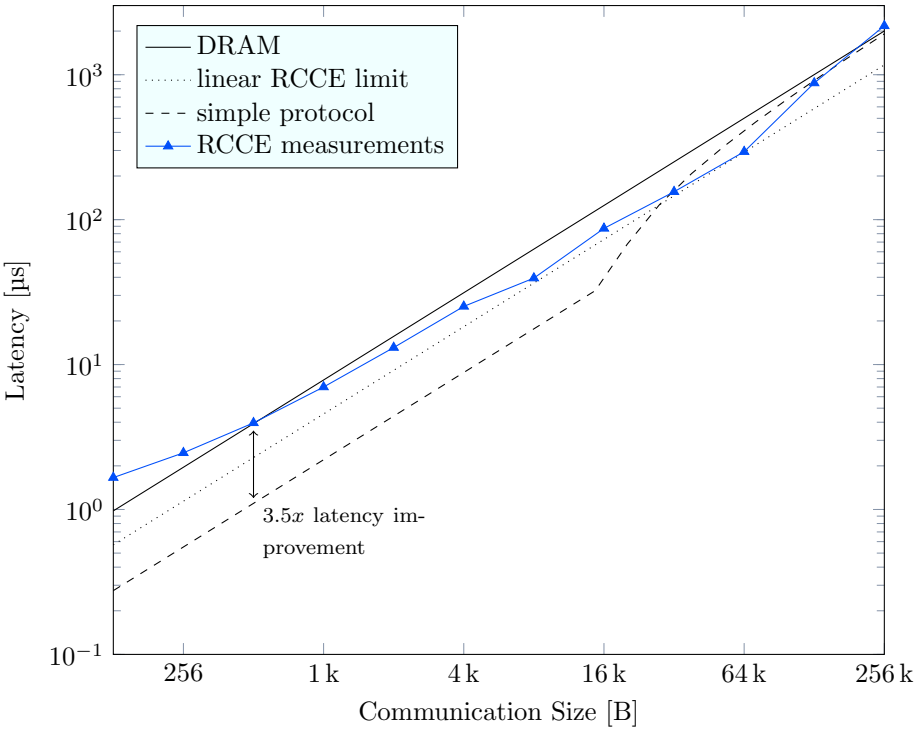


Figure 3.4: Communication performance of RCCE on the Intel SCC

We assume a different latency gap, of 3.5x, between the on-chip SRAM and the off-chip DRAM to compare the effective theoretical communication performance with the communication performance of RCCE in practice. This lower latency gap of on-chip and off-chip memory is more realistic due to a hardware bug of the SCC hardware, which prevents a bypassing of the interconnect for local tile memory access. In fact, this bug degrades the point-to-point communication performance and has to be considered for a comparison to the theoretical communication performance of the Intel SCC.

The diagram holds measurements of the common ping-pong application for the communication library RCCE on the SCC[2]. Our calculation of the theoretical throughput is based on the assumption of a simplified communication protocol for message passing. The on-chip SRAM is used for a data transfer for messages with up to 8 kB, and the off-chip DRAM is used to transfer the remainder of a larger message. Compared to this modeling of communication, RCCE uses the message passing buffer also for a transfer of larger messages by separating the message into chunks of buffer size.

The black line shows estimated memory latency, whereas the dashed line illustrates the MPB latencies for small messages below 16 kB and the latencies for larger messages, which are transferred according to the described protocol. For small messages below 512 B the latencies of RCCE are higher than the estimation of memory latencies, because of the synchronization and protocol overhead. If software pipelining is used, such as in the given example, for messages above 32 kB the latencies are smaller than the simple reference communication protocol. Later in this chapter, the communication performance of RCCE is described and analyzed in more detail. However, this brief comparison already uncovers the overhead of the communication environment on the one hand and on the other hand the potential for optimization.

3.2 vSCC: Extending a Research Platform

vSCC architecture[3] consists of hardware and software components, that extend the functionality of the Intel SCC research system. In contrast to a research environment based on hardware components off-the-shelf (COTS),

[2] For this comparison we selected the highest core/mesh/memory frequency of 800/1600/1600 MHz

[3] v stands for virtual extension which provides additional functionality to the Intel Single Chip-cloud Computer

the SCC research system provides flexibility to modify the architecture and thereby explore low-level engineering aspects for many-core software. The first extension as presented in this section has been developed by Intel Labs, whereas the second extension has been developed by RWTH Aachen University in cooperation with Intel Labs [Reb+12a].

Figure 3.5 illustrates the topology of vSCC as an example of five tightly coupled SCC devices.

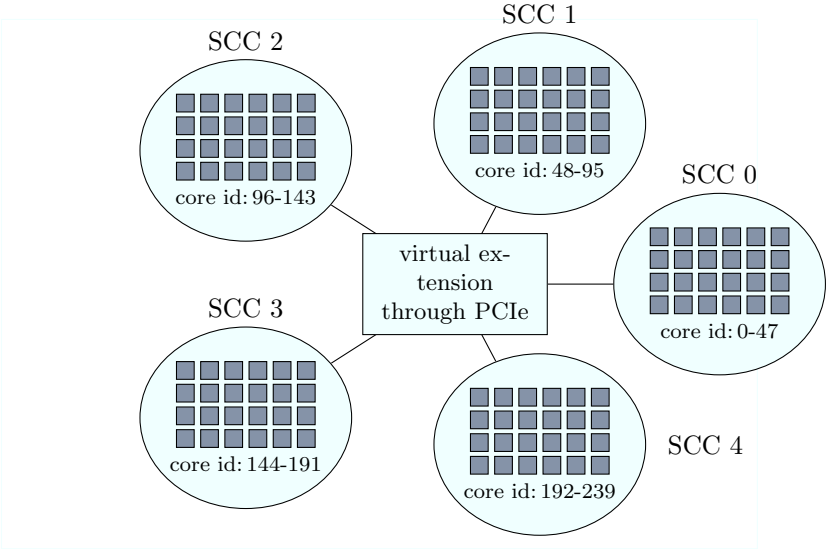


Figure 3.5: Topology of vSCC, which consist in the illustration of five devices. Each device represents an SCC with 48 cores and resulting 240 physical core ids for vSCC

The host system of the SCCs is directly connected to the on-chip interconnect of the device via its system interface. The intended use of this direct on-chip connection between device and host system was to setup and to debug the device.

As the basic architecture of the SCC is 32 bit based, each core can only address 4 GB of memory. Due to the fact that the size of the main memory is 32 to 64 GB, the cores need additional functionality to address the physical memory of the many-core system. Such a functionality has been implemented

by another indirection in hardware, named Lookup Tables (LUTs), in addition to paging. The LUT indirection is programmable and creates the flexibility to address remote memory in a transparent way.

Our method is to redirect this way the access to a certain amount of each core's physical address space via the host system. As a result, the host system can directly respond to memory requests of the cores. This technique introduces a higher latency, because it adds the host system to the critical communication path, but it creates further possibilities, such as tightly connecting two devices.

The approach of using the LUT indirection level to customize the physical address space of the many-core system is similar to extensions of SCC's system interface. Upon version *1.4.0*, *sccKit* provides a set of *Atomic Increment Counter* ($2 \times 48 = 96$ AIC), that emulates atomic operations with a proprietary interface, which is detailed in the subsequent paragraph.

3.2.1 Global Atomic Operations

As a workaround for the missing support of system-wide atomic operations by the instruction set, the SCC research platform provides a set of on-chip synchronization register. Specific register emulate atomic operations for the cores of the SCC, thereby that each core can atomically change the value of a shared counter.

The access to atomic operations works as described in the following: A synchronization register is triggered by reading and writing a machine word to a specific memory location. The synchronization register handles resulting memory request independently of the processor core and modifies an internal value, which mimics execution of an atomic operation. This implementation of atomic operation represents a limitation compared to atomic functionality which is provided by common shared-memory architectures. Hereby, various operations such as *atomic increment* or *compare exchange* up to machine word size are provided with the `LOCK` prefix. In contrast to that flexibility the syntax of test-and-set register does not provide a read access without a possible changed value. Another restriction is the limited amount of synchronization registers and software controlled on-die memory.

However, to determine the value of atomic operations for a tiled many-core architecture the chosen implementation of atomic operations with synchronization register is essential. For the Intel SCC, additional synchronization register have been integrated to the Rocky Lake system FPGA, because it is

directly connected to the on-chip mesh network. As a result of this extension, shared counters can be realized whereas each counter is composed of two synchronization register, *initialization* and *increment*, and can be controlled as follows.

- A write to the *initialization* register loads a 32 bit value to the counter
- A read to the same address simply returns the current value of the counter
- A read to the *increment* register executes a post-increment operation atomically on the counter value
- A write to the same address similar decrements the counter value

The basic approach of the described workaround is that memory locations are mapped to specific operations. For instance, access to a specific address performs an atomic add of x to an internal value, as result of a read and returns the incremented value. Consequently, atomic operations can be supported for a processor architecture that uses simple load and store operations to transfer data through a packet based network.

Dependent on the location of accessing core, latencies without contention to off-chip synchronization resources are about three to five times higher than on-chip register. If contention is taken into account, our measurements have shown that a straight forward busy-wait implementation of a central off-chip synchronization point is not possible for the SCC [Reb+12c]. The design and implementation of related synchronization constructs is described in Section 3.5, for instance the use of a quantified back-off to relax contention and realize a shared counter based barrier.

3.2.2 Increasing the Core Count

An important contribution of adding functionality to the SCC research platform has been developed at RWTH Aachen University in cooperation with Intel Labs. The approach to virtually extend SCC's on-chip network to explore its scalability has been first proposed by Gries et al. as an extension of the system FPGA [Gri+11]. Due to several limitations of the research platform, this extension has not be realized. We have followed a similar approach and developed extensions of the host system driver functionality to transparently connect multiple coprocessor many-core systems [Reb+12a].

Our full working prototype enables the operation of a tightly connected cluster of cluster-on-a-chip processors, which is able to run a RCCE session with up to 240 cores. The setup consists of 5 SCC devices that are connected to a single host [Reb+15].

First, we achieved excellent scalability of SCC's architecture for up to 96 cores with several modifications. To reach this goal an alternative communication scheme and protocol extensions have been developed for iRCCE, which are described in Section 3.4. Due to hardware limitations of the SCC research system, we had to give up transparent emulation of more than two tightly coupled processors. As a result of this extension, the host driver can provide additional functionality, such as routing memory requests to a remote destination or adding new instructions that are controlled through read and write requests.

For the basic communication between two SCC cores, hardware support for message passing is provided. We use this support to accelerate inter processor communication and develop further communication extensions to hide latency. Resulting achievements that provide additional functionality through system software extensions are described in the next chapter.

Results

The ping-pong application has been used, to demonstrate the applicability and restrictions of the vSCC prototype. Regarding the scalability of the new research environment, performance measurements of floating-point-intensive applications are presented in the next chapter.

The default communication scheme of the RCCE family turned out to be a real performance issue for tightly coupled inter-device communication. Main reason for this performance issue is the fact that the remote communication path consists of read operations. As the SCC's design lacks a DMA controller, a communication with software-controlled on-chip memory is mapped to machine operations that target a remote memory location.

With our prototype, such a memory operation is routed through the host system, which can significantly increase its cost in terms of latency. Here, the important attribute of a read operation is that the requested data has to be available before the *acknowledge* can be generated.

For vSCC, the cost of a memory operation mainly depends on the distance of the component, which generates the *acknowledge*. In the described configuration it is generated by the communication task, which is running on the

host. As a result, costs for a *read*, in terms of latency, are about two times higher than the costs for a *write*, because a *write acknowledge* can be immediately generated by the communication task. The option of the SCC research system to generate automatic-write acknowledges on the device magnifies this gap between remote read and write access.[4]

Because the memory subsystem of P54C can only handle one outstanding request, the time to acknowledge memory operations is directly connected to throughput of inter-device communication. With a change of the communication scheme to *remote-put*, the throughput for inter-device communication can be significantly increased from around 1 MB/s up to a maximum of 37.17 MB/s.

The main reason for this significant improvement is the fact that multiple write requests can be on-the-fly. We see a direct correlation between the fast acknowledge of write operations and high inter-device communication throughput, due to the fact that the in-order cores of the SCC, with a simple memory hierarchy and small write-combining buffers, are not good in hiding memory latency. As a consequence, the possibility to generate on-board automatic write acknowledgements leads to highest throughput for vSCC. The maximum throughput which has been achieved by our emulation with 25.7 % of native on-chip communication.

All in all, we have shown that the latency of inter-device communication is effectively hidden due to low-level pipelining of a message-passing payload transfer. Figure 3.6 holds throughput results for the common ping-pong application with a focus on on-chip and off-chip communication.

To obtain these measurements, the amount of data that is transferred in each iteration is divided by the round-trip times. For each message size the ping-pong pattern is repeated multiple times and its average is used to calculate the throughput. The focus of our measurements is on small to medium message sizes between 1 B and 256 kB. We have chosen this specific range because of the cache size of a single SCC core. If a message is larger than the last level cache, access to the off-chip memory represents a bottleneck for the throughput and the application results will not give more insights on the on-chip communication performance.

[4]The option to generate automatic write acknowledgments by the system interface has been identified as a source of transmission errors for the SCC with a high load on the interconnect. In the default configuration the option is switched off.

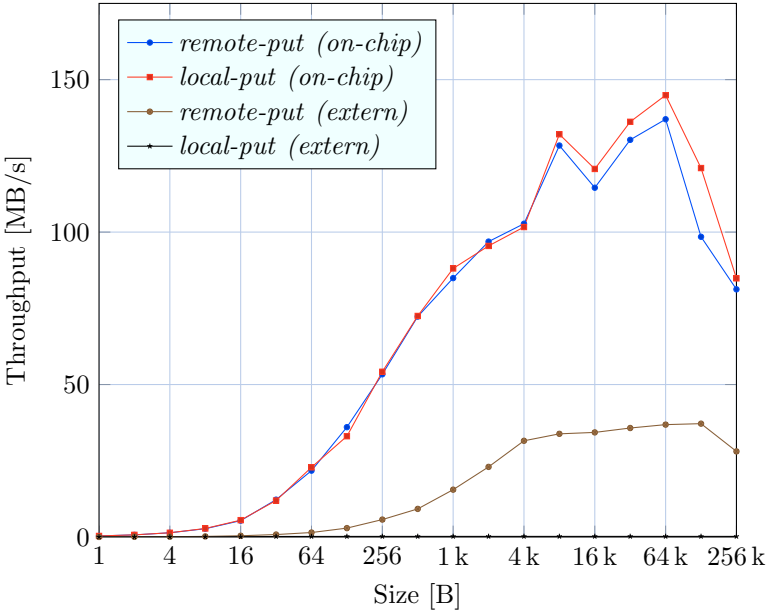


Figure 3.6: Communication throughput for different communication schemes of iRCCE on vSCC: on-chip vs. extern (MCPC routed)

The results show, that for on-chip communication, *remote get* is the preferred communication scheme, whereas for a larger communication distance *remote put* provides a better performance.

3.3 iRCCE: Extending Rock Creek Communication Environment

iRCCE (pronounced: *irocky*) has been developed at RWTH Aachen University as an extension to RCCE, to support a light-weight non-blocking communication environment for the SCC [Cla+13b]. The main intention for its

development has been to overcome limitations regarding the functionality of RCCE.[5]

RCCE's non-gory interface, which abstracts on-chip communication, is limited to blocking communication by means of cooperative functions. Classic `send` and `recv` function pairs are provided for message passing between UEs to enable a blocking communication.

In 2010, version 1.0 of iRCCE has been released as an extension to RCCE version 1.0.13 [MvW11]. Version 2.0 has been published in 2013, which includes support of SCC's extended synchronization functionality. All published versions of iRCCE extend the namespace of RCCE and do not change its basic functionality, such as on-chip memory allocation and access. The main advantage of this naming convention is a reuse of the hardware abstraction of the SCC. Additionally, the further development of RCCE and iRCCE is more or less independent.

Especially regarding its communication performance, the main features of iRCCE are highlighted in the following list:

- improved memcopy (iRCCE 1.0)
- pipelining (iRCCE 1.0)
- non-blocking communication (iRCCE 1.0)
- asynchronous communication[6]
- tagged flags (iRCCE 2.0)
- alternative communication schemes[7]
- bypass flags[7]

The impact of our optimizations to point-to-point communication can be observed if the different throughput results of the ping-pong application are compared. We use the simple ping-pong benchmark in this chapter to qualify protocol optimizations and present synthetic-application benchmark results in the next chapter.

[5] The integration of iRCCE to MetalSVM as inter-kernel communication layer is discussed in the next chapter.
[6] This feature is part of MetalSVM iRCCE
[7] This feature is not part of a release version.

The different graphs of Fig. 3.7 indicate the impact of iRCCE's main features to the communication performance. If all optimizations are enabled, we see the highest throughput over all message sizes. In detail, the features, which can be used independently of each other, are: improved *memory-copy*, *pipelining* as a protocol optimization and finally *bypass flags*. We have discussed, in Section 2.3.4, the option *bypass flags* to decrease the synchronization overhead of the RCCE family.

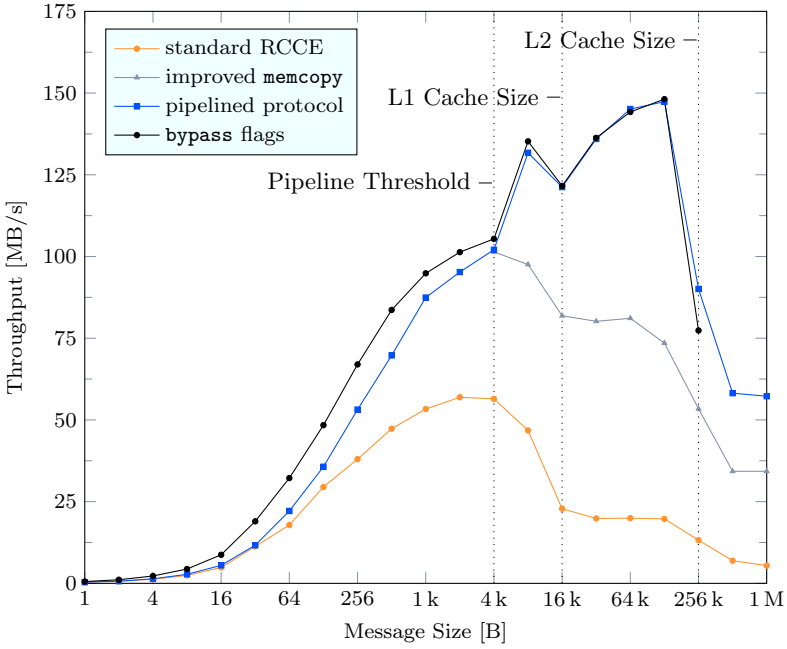


Figure 3.7: Comparison of *RCCE* point-to-point communication performance to optional features of *iRCCE*

vSCC support

A version of iRCCE with extended functionality has been developed at the RWTH Aachen University to improve the support of vSCC. This version

also includes additional inquiring functions, which are beneficial for a tightly coupled cluster with heterogenous interconnect.

The existing inquiring functions, `RCCE_ue()` and `RCCE_num_ues()`, return the current rank of a UE or respectively the total number of ranks of a session. Similar to this behavior, `iRCCE_dev()` and `iRCCE_num_devs()`, return the `id` of current device and the total number of devices, that a session is currently using. Additionally, `RCCE_num_ues_dev()` returns the number of active cores per device. These five additional functions provide enough information, for each UE to determine the position within a hierarchical physical setup of vSCC. As a result, applications can take the hierarchy of a cluster of cluster-on-a-chip processors into account [Reb+12a].

3.3.1 Communication Model

In this paragraph, a communication model for RCCE family is derived, which can be used to predict its costs for point-to-point communication.[8]

The model requires architecture specific numbers as input, which are shared-buffer latencies and private-cache characteristics, such as its size and bandwidth. For the SCC, three categories of private data access have been identified, whereas costs differ in one order of magnitude, specifically first level cache, second level cache, and main memory.

Kielmann et. al describe in their work, *Fast Measurement of LogP Parameters for Message Passing Platforms* [KBV00], a method to derive the parameters of the LogP model family through a simple message passing application. A client-server application is available for MPI to derive the parameters for the *LogGP* communication model.[9] The application consists of two processes, an active measurement process so called server and a passive mirror process so called client, whereas the terms active and passive indicate the role for the benchmark.

For *LogP*, the overhead (o) defines the time when a processor is busy in sending or receiving messages. A simple modification of the classic *LogP* model is to distinguish between send overhead o_s and receive overhead o_r of each message. The gap g for the given communication scenario is equal to the sending overhead o_s, because RCCE uses active waiting to indicate that

[8] This work has not been previously published.

[9] The application uses message passing through send and receive functions, a port to native RCCE was relatively simple.

a message has been completely handled on receiver's side. This is a typical result for hardware without DMA facilities.

For small messages (below 16 kB) the copy operation to a local MPB (o_s-o_r) and the waiting time for completion, represent a significant part of the entire communication process. If a message is larger than 8 kB, it does not fit into the message passing buffer of the SCC any more. The communication protocol can split the message into chunks that fit into the MPB and successively transfer these chunks. If software pipelining is used to interleave basic one-sided operations, which are used to transfer the message, the gap ($o_s - o_r$) becomes smaller compared to the total communication time because continuous copy operations are used to transfer the message payload.

Another effect that degrades communication throughput of larger messages is that the private buffer does no longer fit into private caches of the communicating processors. Consequently, the private buffer starts to become a bottleneck.

Williams et. al assume in their work, *"Roofline: An insightful visual performance model for multicore architectures"* [WWP09], that for the foreseeable future, off-chip memory bandwidth will often be the constraining resource in system performance. Roofline is a simple model, which relates processor performance to off-chip memory traffic. The result is a model for the performance of computer systems which predicts dependent on the computational intensity of applications an upper limit with the memory bandwidth and the peak-computing performance. The communication of a cluster-on-a-chip architecture can be described, similar to this approach.

Instead of using the term computational intensity, we use the term communication intensity, which describes the ratio of on-chip communication per memory transaction. Following the concept of different memory subsystems (on-chip and off-chip) that limit communication in combination with *LogGP*, creates a model which can be effectively used to estimate point-to-point communication with RCCE.

Figure 3.8 compares the predicted performance to the measured performance for point-to-point communication between two neighboring tiles. For small messages (below 8 kB) a LogGP model without modifications is accurate for the estimation of communication time. As long as the private buffer completely fits into the first level cache, the shared buffer access is a bottleneck, which leads to an exact prediction of the communication time. For a prediction of messages above 8 kB, a linear slope has to be introduced to take the limitation of private buffer access into account. For message sizes

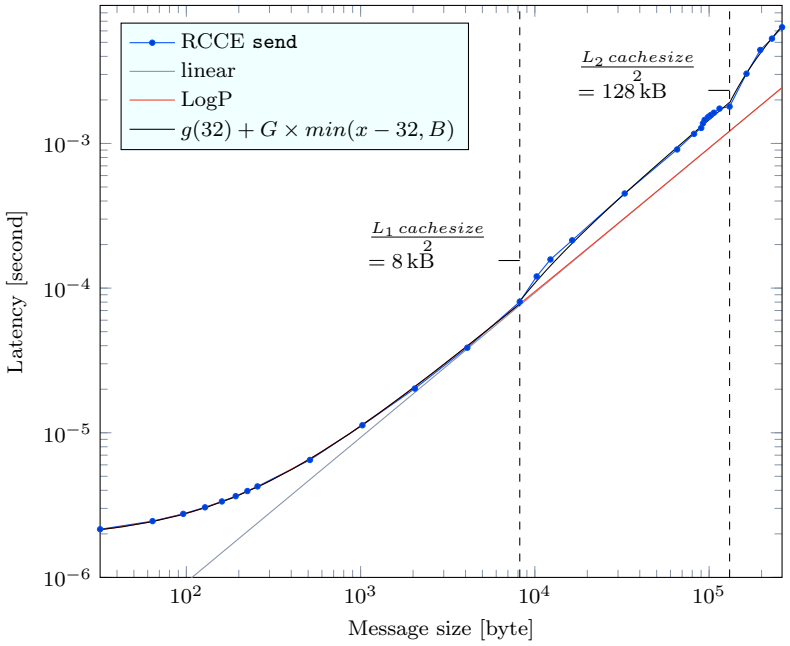


Figure 3.8: Communication model for RCCE point-to-point communication without pipelining

between 8 kB and 128 kB, the two private buffers, *send* and *receive*, fit into the Level 2 Cache. In this scenario, private buffer access can be served by the cache memory, if we assume a warm cache. The estimated communication latencies can be shown as a graph with a linear slope, because of the constant bandwidth of memory which represents the bottleneck for a given message size in the ping-pong scenario. For messages with a size larger that 128 kB the private buffer does no longer fit entirely into the L2 cache. As a result memory requests are served by the next level of the memory hierarchy with a lower bandwidth. In the diagram of Fig. 3.8 this effect is shown thereby the graph has the same gradient but a different offset.

In the diagram of Fig. 3.7 on Page 64, which shows ping-pong results for the SCC, this effect manifests in plateaus of the graphs that plot the communication throughput of RCCE.

3.3.2 Communication Modes

Cooperative communication functions, that are used by message passing applications, typically consist of communication and synchronization elements. Different communication modes can be distinguished for point-to-point communication. In the context of many-core systems, important attributes of communication are in general: *blocking* or *non-blocking*, *synchronous* or *asynchronous*, and *buffered* or *unbuffered*. The attributes are defined next and briefly described, to categorize the protocol optimizations which are discussed in the following.

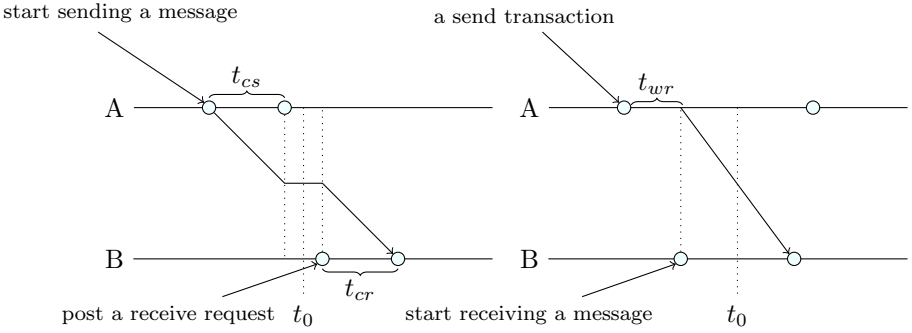
Blocking communication behavior can be defined, in a way that a sending process remains within the communication function until communication is completed. A blocking communication interface prevents interleaving of cooperating function calls, which includes an explicit communication and an implicit synchronization.

Similar to the definition of blocking communication, asynchronous communication can be defined according to the chronological order of basic operations. In particular, the sender can complete the send operation independent of receiver's communication progress.

Buffered and unbuffered communication modes differ, as their names suggest, if an intermediate buffer is used to transfer data. For instance, if a sender can not immediately complete a non-blocking communication, it has to copy data to a local buffer which is typically managed by the communication library. This additional copy operation can increase the communication latency.

For a many-core system, every kind of asynchronous communication requires a management of the communication progress. For example, signals are a common mean to pass the information that an event has occurred and thereby indicate the arrival of a message. Either interrupts can be used to signal the event of an incoming message or a background thread can handle incoming messages.

The RCCE communication library has been designed for a use without operating system support. Specific design concept targets a light-weight communication environment as well as bare-metal execution of RCCE applications. However, support for asynchronous control is not available for this environment, such as multithreading and active-message functionality. Active messages are a concept to execute functions remotely, which needs some support by the runtime or the operating system.



(a) Buffered and Asynchronous (b) Blocking and Synchronous

Figure 3.9: Comparison of two communication mode examples (Figure based on [Cla+12])

3.4 Message Passing Buffer

The new communication concept of the Intel SCC is based on direct access to low-latency and high-bandwidth memory. This memory is allocated as a message passing buffer in a distributed manner and used for on-chip communication with explicitly controlled consistency. In any case, such a message passing buffer is a scarce resource because, it has to be placed in memory that is located close to each processor core. For the realization of message passing, a communication protocol is required that specifies the allocation of buffers, their coordination and accessing schemes.

RCCE is a light-weight communication environment that uses *busy-waiting* to indicate the arrival of messages. This strategy avoids dependencies on the operating system, so that a bare metal execution of RCCE applications can be natively supported.

For the realization of point-to-point communication, RCCE allocates a dedicated communication buffer for each rank. As implemented by the reference implementation, a single threaded execution of RCCE processes simplifies the allocation of message passing buffers and the access to these buffers for each core of a many-core system. For a communication protocol, this attribute represents a mutual exclusion of access to software controlled on-chip mem-

ory, for example by the sender and the receiver of a message. By removing this limit, we expect a positive effect for the communication performance.

Towards the realization of message passing in an efficient manner, we discuss alternative methods for a coordination of software-controlled on-chip memories and their allocation schemes. In this context, efficiency is defined as a high throughput, low latency and resource awareness. Consequently, new communication protocols are described in this chapter that extend the RCCE family. In fact, the presented work is based on detailed analysis and resulting optimizations, which have been discussed in Chapter 2.

The term Message Passing Buffer (MPB) has been established as a common term to describe the part of the LMB, which is used for point-to-point communication [Mat+10; Gri+11; CRK11; Cla+11]. MPB also describes the memory concept which is implemented by RCCE. However, many related publications use the term MPB to describe the software controlled on-chip memory of the SCC. This is not exactly the same because RCCE has several restrictions regarding allocation and access of on-chip memory regions. Because in this dissertation alternatives to this concept are discussed, the term MPB is only used to describe the allocation of on-chip memory as a communication buffer in combination with SCC new cache memory configuration as described in Section 2.2.4 of Chapter 2.

If software-controlled on-chip memory is used as a buffer for message passing, a coordination scheme for this buffer is mandatory. This assumption is also valid for a protocol which enable data transfer between sender and receiver. A common abstraction for this scenario of concurrency is a classic *single producer, single consumer* problem, when the sender of a message represents a producer and the receiver represents a consumer [Tan07].

For a more efficient transfer of large messages compared to RCCE, the communication extension iRCCE uses a classic software pipelining. Here, the buffer is split into multiple regions that are controlled separately for read and write access, even if the message could be transferred en bloc. This technique targets at interleaving local memory-buffer access and thereby increase the communication performance. A static pipeline threshold[10] of 4 kB has been experimentally derived as a good value for the given configuration [Cla+13b].

However, this limit depends on the size of the MPB and costs of read and write operations to the on-chip memory location. If optimizations, such as a

[10]Pipeline threshold defines a message size. Larger messages are not transferred en-block but split into smaller parts and transferred continuously

bypass of all caches for flag access, are applied to the basic communication layer, a finer granularity can become more attractive. In this section the use of classic software techniques, such as the implementation of a ring buffer, is discussed and a communication model is derived. For a given many-core architecture, we specify parameters to analyze the communication in detail.

3.4.1 Communication Schemes

The use of software-controlled on-chip memory as a communication buffer for message passing gives the possibility of different communication schemes. For the investigated communication environment, two basic communication schemes are possible, which can be distinguished by the location of the shared buffer. Either data is put in the local MPB of the sender or of the receiver. Both schemes are possible but require a different synchronization of the MPB.

Figure 3.10 illustrates the access scheme of the two alternative communication schemes, *local put* and *remote put*. On the left hand side of the diagram, the communication buffer is local to the sender, which implies a *local put* of data for the sender. Consequently, the receiver gets the message from a remote memory location.

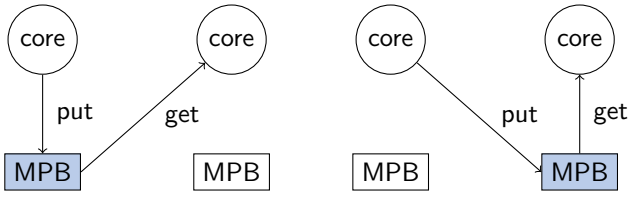


Figure 3.10: Comparison of on-chip Communication Schemes

Our analysis underlines, that a local-put, remote-get (lprg) communication scheme, or short *local put*, is well suited for low-latency on-chip communication. It is no surprise that the basic communication protocol of RCCE is based on this communication scheme. For point-to-point communication through software-controlled on-chip memory, several advantages are resulting. The fact that write operations of each core can be limited to its local communication buffer simplifies the synchronization. Another advantage of *local put*

is that RCCE API specifies a non-blocking probe functionality, which means a receiver can determine if a message is pending or not without further sender invocation. This functionality is guaranteed by *local put*, because a receiver can probe for an outstanding message by accessing the local communication buffer of the sender.

The virtual extension of a many-core interconnect, as described in Section 3.2.2, generates the possibility to access on-chip memory that is located on another processor the same way as local on-chip memory. As a result, RCCE family can be used for on-chip and remote processor communication on the SCC platform. To support a session with multiple virtual devices, iRCCE extends the communication ranks in a linear way. By porting iRCCE to the vSCC architecture, it turned out that the maximum throughput is limited to 1 MB/s. The main reason for this poor performance is the combination of default *local put* communication scheme and transparent routing of remote packages. Here, a message transfer is realized by copying its payload from remote shared to private buffer, which results in remote read operations. Since the latency of this operation has a direct influence to the throughput, the remote communication path degrades the performance.

In general, the acknowledge of a read operation can not be generated before the requested data is available. This behavior is different for a write operation, where the acknowledgement can be immediately generated, because no additional information is necessary. A processor core that follows the network-on-chip paradigm with multiple outstanding memory requests represents an alternative solution for this general problem. However, dependent on the routing policy, additional support may be necessary for such a core to guarantee the strict ordering of memory operations.

The use of a remote-put, local-get (rplg) communication scheme, or short *remote put* represents an option to improve inter-device communication performance for the Intel SCC. This alternative scheme changes the remote memory access from read to write, respectively get to put in terms of the one-sided communication system. A drawback of this scheme is that it creates a multiple writer scenario, so that a core has to pass exclusive write access to its local communication buffer. This requires additional synchronization which is not needed for *local put*, because write access for each UE is restricted to the local buffer plus dedicated flags. Consequently, an exchange of the communication scheme from *local put* to *remote put* requires modifications to the communication protocol.

In the following, we propose a new protocol for the alternative communication scheme, which recovers non-blocking probe functionality. In order to realize this new protocol, we introduce probe flags as described in a subsequent paragraph.

3.4.2 Flags

In general, attributes of busy wait synchronization are well understood, and have been intensively discovered in research and practice. The main advantage of busy waiting is a low latency. Disadvantages are a wasting of CPU cycles, especially if a short spinning time can not be guaranteed and the potential of unfairness regarding the order of request grants and resulting starvation. Such an issue clearly depends on the specific implementation. Alternatives to busy waiting, are operating system or runtime supported synchronization primitives that typically introduce a certain administration overhead. However, a common combination is busy waiting and blocking schemes to achieve best performance.

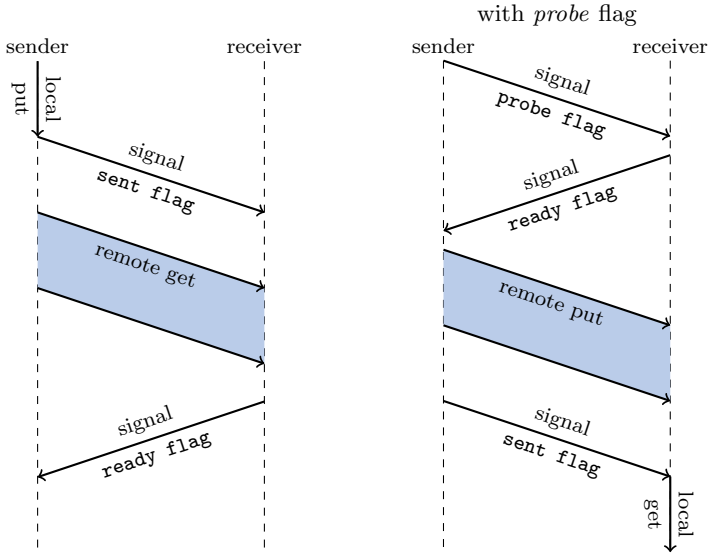
Shared variables are named flags, which hold a binary value to indicate if an event has occurred. By default, RCCE uses two flags, `sent` and `recv`, to control the access to the Message Passing Buffer for each rank of a session for the realization of point-to-point communication. Collective operations, such as a low-level barrier implementation can introduce additional flags. In the following, other kind of flags are described which are necessary for the implementation of an alternative communication scheme.

Probe Flags

A use of alternative communication schemes (e.g. *remote put*) can break direct access to the Local Memory Buffer to initiate a send operation. If multiple senders can access a shared buffer, the receiver has to grant access. For *local put*, which is used as a default by RCCE's communication protocol, two flags per rank are sufficient to coordinate access to the MPB, as illustrated in Fig. 3.11. Changing the scheme contradicts a non-blocking probe functionality. A behavior which is specified by RCCE, because it is potentially required by communication environments that use RCCE for hardware abstraction and implement a higher level communication, such as MPI.

We have developed a new communication protocol, to reconstruct the non-blocking probe functionality of RCCE. If n is the total number of ranks, $n-1$

additional flags are necessary for each rank to realize *probe* flags. Obviously, this solution introduces another synchronization step with additional overhead, as illustrated in Fig. 3.11. Tagged flags, which are described in the subsequent paragraph, represent an option to recover low latency for small messages.



(a) *local put/remote get* (b) *remote put/local get*

Figure 3.11: Timely behavior of Communication Protocols [Reb+12a]

Tagged Flags

Piggyback messages are used by many tools in the scope of MPI in a certain variety [SBdS08]. Messages are extended in a transparent fashion on top of a communication library to transfer additional information. *Tagged flags* is a feature that was introduced by iRCCE version *2.0* to provide low latency for small messages. The basic idea of this feature is to pack a flag value with a small message payload in a piggyback fashion up to a certain granularity that can be atomically transferred [Reb+12a].

Regarding communication schemes, this low-level approach can be categorized as a combination of *remote put* for small messages and another communication scheme, such as *local put* for medium to large messages. For the SCC, the maximum size of a small message, that a core can atomically write to, is equal to the size of a cache-line minus the size of a sent flag. Consequently, our optimization affects messages with a size of 32 B minus 1 bit to 4 B, dependent on the configuration.

As previously mentioned, the use of *tagged flags* represents an optimization especially in combination with probe flags. Here, the *remote put* communication scheme requires an additional synchronization step, with a negative impact on the latency of small messages.

3.4.3 Dynamic Buffer Allocation

RCCE family is based on the concept of a symmetric allocation scheme, which implements a symmetric memory model for the Local Memory Buffer. In this context, symmetric memory model means that all operations which allocate on-chip memory are executed as a collective operation. This implies a limitation to a static communication buffer, both in size and offset [vWMH11].

In this paragraph, we discuss an optimization that targets a higher point-to-point communication throughput by interleaving basic put and get operations to a communication buffer, for even smaller messages than the pipeline threshold of iRCCE (4 kB). The concept is a combination of pipelining memory copy operations and the coordination of concurrent accesses to the shared buffer. In order to realize this concept, we first introduce a dynamic allocation scheme which contradicts the symmetric memory model of RCCE. A dynamic allocation of a message passing buffer provides advantages regarding the use of on-chip memory, which represents a limited hardware resource. However, the dynamic handling of access to on-chip memory will introduce additional overhead, which prevents a general optimization for all message sizes.

The default communication protocol of RCCE is based on the combination of a static MPB allocation and mutual exclusion of shared buffer access. The start of a communication buffer and its size are fixed after initialization for all UEs. This simplifies the communication protocol, as concurrent access to the MPB is excluded and the location of a remote communication buffer can easily be derived.

Data transfer through blocking communication functions represents a communication scenario, where sender and receiver actively participate in the

communication progress. Assuming such a communication scenario, the new communication protocol is based on the organization of shared on-chip memory as a circular buffer. This represents an alternative dynamic allocation scheme, as the available on-chip shared memory is divided into slots, where each slot can represent the start or the end of communication buffers. Since circular buffers exist in different implementations, the chosen implementation that suits the given architecture controls concurrent read and write access by storing two pointer that mark the begin and the end of target buffer.

Figure 3.12 illustrates the allocation of on-chip memory with read and write pointers, which are located in on-chip memory. For the chosen implementation, an element is inserted at write position of the ring buffer and the element one slot after read position can be removed if the buffer is not empty. A disadvantage of this implementation is that only $n-1$ elements can be used for communication if the buffer has a size of n. If the element size is relatively small compared to the buffer size, this overhead is negligible.

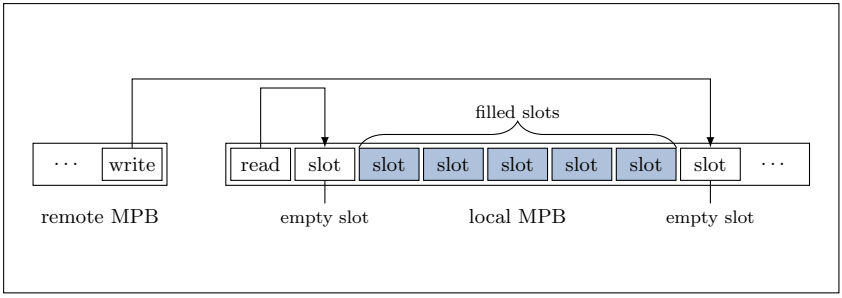


Figure 3.12: Dynamic on-chip memory allocation scheme for a ring buffer *(remote put)*

For a given architecture without atomic operations to on-chip memory, a major advantage of the chosen buffer implementation is the fact that a full buffer can be distinguished from an empty buffer by a simple pointer comparison. The size of these pointers defines the number of elements which can be managed. For example, a byte pointer can address 256 elements. With a slot size of 32 B which corresponds to the size of a cache-line, a communication buffer of 8 kB can be managed.

Latencies of reads and writes to shared memory have a direct influence on the communication performance of our new dynamic buffer allocation scheme.

Especially, low latencies of read and write operations to shared pointers are essential, because they control the buffer access. Consequently, we use the *bypass flag* option, as described in Section 2.3.4, to access shared pointers. Nevertheless, a tradeoff exists between inserting and removing large chunks of data and interleaving copy operations to on-chip memory.

If parameters are chosen appropriately, our new protocol will lead to promising results because of its flexibility. In the next paragraph, this assumption is verified with a comparison of estimated communication times to experimental results.

3.4.4 Results

Due to the predictability of single access latencies, a simple formula can be used to predict the throughput T for a specific message size x and a certain granularity n of access to the ring buffer. The term $t_\Delta(n)$ of Eq. (3.1) includes the protocol overhead and the transfer time for a single copy operation from private to shared memory or vice versa.

$$T(x,n) = \frac{x/n}{(x/n+1) \times t_\Delta(n)} \tag{3.1}$$

The protocol overhead consists of synchronization-and-coordination operations to control ring-buffer access. This overhead can be calculated by comparing a value of x to the black curve, where the message size is equal to the access granularity of the ring buffer. At this point, the new protocol is equivalent to the simple RCCE protocol.

Table 3.1: Transfer time related to message sizes for the Intel SCC without synchronization overhead

n [B]	64	128	256	512	1024
$t_\Delta - o$ [µs]	6.59	5.72	5.17	4.88	4.76

Table 3.1 holds the parameters for the Intel SCC, which have been used to predict the point-to-point communication throughput. The values can be calculated by the reciprocal value of the required time to transfer n byte en-bloc, from a cached private memory region to target on-chip memory region.

If the communication distance rises, only the parameters have to be adjusted to predict the communication throughput.

Once again, we use the ping-pong application as a benchmark for the new communication protocol.

Figure 3.13 compares the throughput of a ring buffer, as described in the last paragraph, that is used for a communication with different element sizes. For a comparison of the absolute results, the throughput is plotted over the message size.[11] For the measurements, single byte pointers are used to mark the begin and the end of available data regions, with a maximum size of 4 kB. We raised the chunk size of transferred data from 64 B to 1024 B and plotted the measured throughput as dots.

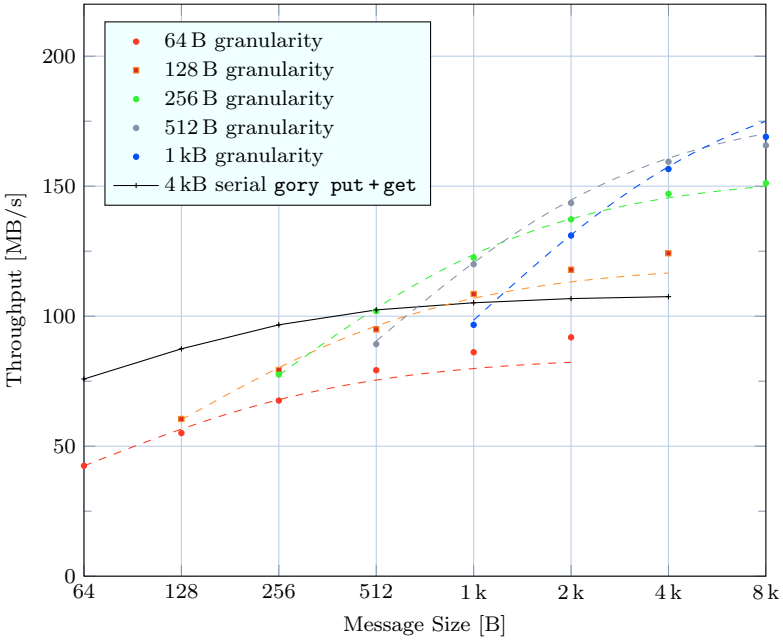


Figure 3.13: Estimated (dashed lines) vs. measured (dots) throughput for the ring buffer implementation with different granularities. Results are compared to serial data transfer (black line).

[11] core frequency: 533 MHz, MPB size: 4 kB

The black curve in Fig. 3.13 depicts the throughput of a two-copy operation data transfer without synchronization overhead. This curve represents an upper bound, because it leaves out protocol overhead and represents only time spend for effective communication. Consequently, a message of x B is copied en-bloc from private into shared and back to private memory. The described curve ends at 4 kB, because a maximum MPB size of 4 kB has been selected for the benchmark.

According to the described method, the communication throughput can be calculated by using the formula from Eq. (3.1). The calculated throughput is plotted in Fig. 3.13 as dashed lines. The resulting throughput of the experiment is plotted for characteristic message sizes as dots.

By comparing both results, it turned out that the predictability of the communication throughput is very accurate. Deviations can be explained by the algorithm which has been used to handle the transfer of data. An operation on cache-line granularity enables a more dynamic interaction, so that a rank does not have to wait until the complete slot is filled, to read out data, or until a slot is cleared, to write data.

It is shown, that a redesign of the basic message-passing-buffer concept reveals the full communication performance of a cluster-on-a-chip processor with software controlled on-chip memory. In order to realize a dynamic allocation scheme, optimization for flag based synchronization has been applied and alternative communication schemes for the Local Memory Buffer have been evaluated in this section.

3.5 Synchronization Constructs

From the objective of a runtime or communication library, synchronization constructs consist of shared data structures and cooperative operations, which can be used for the coordination of Units of Execution (UEs). UEs describe in this context an abstract entity of parallelly executed code on multiple cores, which can be easily mapped to threads or processes. In this section, we detail different barrier implementations, which are based on the hardware synchronization support of a many-core processor.

In 1990, Anderson motivates the fundamental need for hardware support of busy-wait mutual exclusion on shared memory multiprocessors [And90]. Consequently, he presents spin-lock alternatives to increase scalability, for instance by the use of a back-off policy. Previously, Lamport discovered that

pure software mutual exclusion is quite expensive for memory coupled clusters [Lam87]. Graunke and Thakkar proposed queuing-based locking algorithms for cache coherent systems instead of simple test and set locks [GT90]. Their experiments have shown that the strategy of spinning on different cachelines outperforms a centralized approach.

Following this fundamental work, low-latency synchronization methods, which make use of hardware synchronization support of the Intel SCC are described in this section. Specifically, efficient lock and barrier implementations are derived, which represent examples for synchronization constructs.

A barrier is a collective operation, which specifies a synchronization point for a certain number of UEs, so called groups. At least one point in time exists when all members of a group have reached the same point in parallel execution.

Critical sections are defined as a section of code, which can only be executed by a single process or thread at certain point in time. A technique to protect a critical section is mutual exclusion, which can be realized by the lock synchronization construct. A common implementation of a lock is a spin-lock, which represents a typical example for a central synchronization point and is widely used for shared memory systems. According to its name, the implementation is based on a busy waiting technique.

Parallel programs which follow the message passing paradigm rarely use barriers, since message passing implies an implicit synchronization. However, communication interfaces such as MPI (and RCCE) support a barrier function.

The described synchronization constructs have an important part in shared-memory programming, such as the fork-join programming model. Different levels of abstraction are available for multi-threading shared memory programming, either the programmer controls explicit creation of threads or choses a runtime assisted environment. POSIX is a standard which defines a portable interface for operating systems. Part of the related family of IEEE standards is an interface for parallel programming with threads in C programming language: `pthreads`[12]. Through an explicit start and end of parallel threads, a programming model such a fork-join can be easily implemented. A higher level of abstraction represents the de facto standard in shared memory programming: *OpenMP*.

[12]IEEE Std 1003.1c-1995

OpenMP introduces constructs and clauses to enable the creation of parallel shared-memory programs. Pragmas are used to give hints to the compiler, which uses a runtime to create a parallel application. Many of these constructs imply a barrier [CJvP07]. One example is the `parallel for` construct without `no wait` clause.

The design of scalable and low-latency synchronization algorithm for many-core systems is essential for achieving a high application performance. The implementation of a synchronization method has to take hardware characteristic of processor architectures into account. Especially, if central synchronization points exist, a negative effect on the scalability of algorithms can be resulting. Related research questions are, if fairness depends on the location of the remote memory location and does the characteristic of the on-chip interconnect influence latencies of memory requests.

3.5.1 Lock

For a classic distributed shared memory system, a classic spin-lock on hardware distributed memory is rather uncommon. This picture can change, assuming a low latency of distributed shared memory, that can be a result of target on-chip memory locations. If non-coherent memory coupled cores have access to hardware support for atomic operations such as *test-and-set*, the implementation of a spin-lock is straight forward. A drawback of this implementation is remote spinning and the possibility of high contention, because many cores on a single chip access a central synchronization point. For processor architectures with direct on-chip memory access, each read of a shared lock value is a direct access to a remote memory location.

Methods that increase scalability of locks exist, such as ticket locks, which can also guarantee a combination of FIFO ordering and busy waiting [MS91]. Additionally, if the meta information of a critical section is taken into account, such as the number of readers and writers of shared elements, read-copy update (RCU) is a method that can increase performance of locking significantly [McK+01].

Our approach is the use of a spin-lock to analyze the fairness of direct on-chip memory access for a processor which implements the network-on-chip paradigm. Therefore, a shared global counter is used in combination with a back-off function to emulate unified access times. If the interconnect provides predictable latencies without contention, we can use the *unified global counter* from Section 2.2 to record the total order of locks.

Listing 3.1: Spin-lock application

```
1  #include <time.h>
2  #define NUM_CORES 48
3  volatile unsigned long * synch_reg_addr[NUM_CORES];
4
5  acquire_lock(int id) {while(!Test_and_Set(synch_reg_addr[id]));}
6
7  release_lock(int id) {*synch_reg_addr[id] = 1;}
8
9  void main() {
10   unsigned int nsec=100;
11   for(int i=0; i<100;i++){
12     acquire_lock(0);
13     delay(nsec);
14     release_lock(0);
15   }
16 }
```

Our implementation does not target a fair distribution of critical sections, moreover the goal its to verify the derived cost model for contention and analyze the behavior of a central synchronization point for the Intel SCC. Listing 3.1 holds the source code of a micro-application, which has been used to study such a behavior.

The implementation of a spin-lock through test-and-set registers creates a central synchronization point. For the SCC, an atomic *test-and-set* operation has a specific address `synch_reg_addr` because it is implemented by a memory mapped register. Listing 3.1 shows a draft for the `acquire` and `release` functions, that are necessary to implement a lock.

The application generates a hot-spot, because a spin-lock is used to protect a critical section with a certain contention in a coarse-grained way. Our benchmark consists of continuous execution of critical sections with a constant length, which can be found very similar in the EPCC micro benchmarks [Bul99].

An example how critical sections could be chronologically ordered is illustrated in Fig. 3.14 on Page 83. A *unified global counter*, as described in Section 2.2.5, has been used to trace the total order of critical sections. As part

of a critical section, a shared counter is incremented, whereas the length of a critical section is independent of the communication distance to the counter.

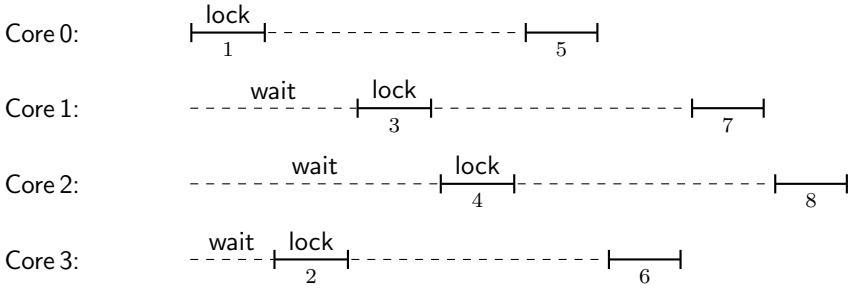


Figure 3.14: Example of using a lock to protect a critical section. The figure visualizes the chronological order, which is not previously defined for the given example.

For a central access pattern, target on-chip memory location can be located in a corner of the mesh interconnect as the worst case.[13] The scattered plot from Fig. 3.15 visualizes the measured order of critical sections. Each core has executed the critical section one hundred times, which leads to the execution of 4800 critical sections in total. Since each core has recorded the counter value for each access, the fairness of a busy-wait synchronization method with a central synchronization point can be classified.

Our experimental results verify the impact of the location to the fairness of a remote memory access. According to the described contention model for the Intel SCC in the previous chapter, a direct link between the location of a tile and the priority of accesses to a central synchronization point becomes visible.

Each core has a higher priority of access, which is located in a same row with the highly contended memory location. Related to the specific algorithm, cores with id 0 to 11, nearly pass around the lock during the first iterations. Not before this group of cores has finished continuously requesting the lock, another core, which is located on a row with a distance of 2, will get the lock with a higher probability.

[13]Here the lower left tile on SCC's mesh with *core id:* 0 and 1 has been chosen, cf. Fig. 2.6

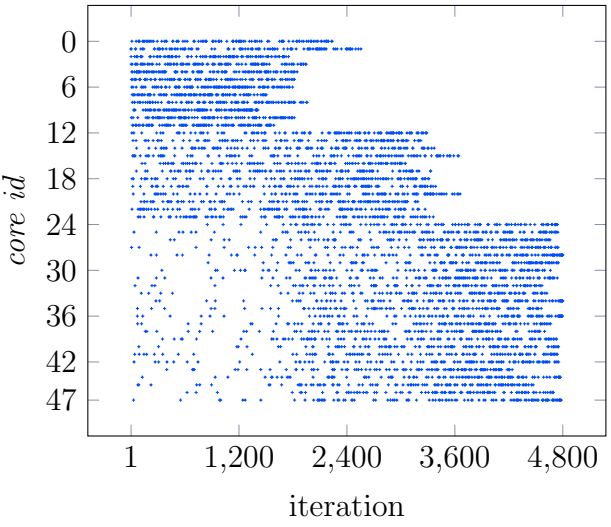


Figure 3.15: Iteration scattering of critical sections with 100 iterations per core [Reb+12c]

A classic strategy to avoid contention is the distribution of resources. In order to realize a distribution of on-chip synchronization resources, we compare a central spin-lock implementation to a tournament lock [GT90]. A tournament lock consists of multiple locks, which are accessed in a well defined order with the graphical representation of a tree. For the implementation of a tournament lock, multiple synchronization register are used to build an *n-ary* locking tree. The diagram in Fig. 3.16 compares measurements for a simple spin-lock to a tournament lock based on a binary tree with a depth of two.

Because three register are necessary to build a locking tree with a depth of 2, the maximum contention is reduced by a factor of 2. One lesson learned from the previous experiment is that cores are preferably assigned to locks sharing their y-coordinate.[14] As a first step, in the presented example each core contends for the assigned group lock and, as a second step, for the global lock to acquire the tournament lock. Accordingly, the locks are released in reverse order to release the tournament lock.

[14]This assignment clearly depends on the routing algorithm of the on-chip interconnect.

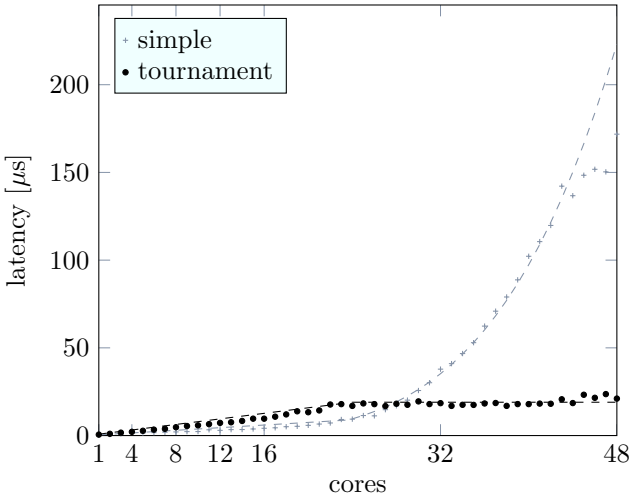


Figure 3.16: Scalability of spin-lock implementations [Reb+12c]

For a comparison, both spin-lock implementations have been used to protect a critical section with an increasing number of cores, which generates a high contention. Here the critical section is empty which generates a maximum contention, in contrast to the evaluation, where the total order of critical sections have been recorded. Thus, a core which has performed one thousand timed iterations, stops the timing but continues to request the lock to generate noise for the remaining cores. The plotted values from Fig. 3.16 are maximum times for an average of one million iterations to acquire and to release a lock.

The experimental results show a linear increasing runtime for the spin-lock application up to a core count of 24. This is an expected behavior, because of the linear characteristics of the synchronization algorithm. A further increase of the core count leads to a constant latency of 19 µs for the tournament lock, because the synchronization primitive limits the concurrency at this point, whereas the latency of a simple spin-lock grows quadratically.

The observed behavior of both synchronization primitives can be described by simple functions, which have been added to the diagram of Fig. 3.16 as dashed curves. The function $l_s(n)$ of Eq. (3.2) can be used to calculate the

latency of the simple spin-lock under high contention with n cores that tries to get the lock.

$$l_s(n) = \begin{cases} n \cdot 0.8\,\mu s & \text{if } n < 24 \\ 19\,\mu s & \text{if } 24 <= n \end{cases} \tag{3.2}$$

Equation (3.2) holds the function $l_t(n)$, which describes the latency of the tree lock under high contention with equal parameters.

$$l_t(n) = \begin{cases} n \cdot 0.4\,\mu s & \text{if } n < 24 \\ n^{4.5} \cdot 5\,ns & \text{if } 24 <= n \end{cases} \tag{3.3}$$

The locking path for the simple spin-lock is minimal and consists of two synchronization register accesses. With a depth of two, the locking path for the tournament lock consists of four synchronization register accesses, which results in a higher latency up to a core count of 28. Because of high-register latencies, contention of more than 42 cores have to be considered with caution. Due to the fact that cores can starve under high contention, exact values are of little importance.

3.5.2 Barrier

A barrier is a synchronization construct, which is commonly defined as a point, that can not be proceeded until all units of execution of a group specified by the construct have reached it. The access from a programmers perspective is either possible with explicit cooperative communication function calls or by implicit use of directives, that are used for instance to control a fork-join programming model.

Focussing on the implementation of a barrier, most algorithm can be logically separated into two phases. Thus, the single execution of a barrier, commonly termed as *sequence*, can be logically separated into a *gather* and a *release* phase for most algorithms. Whereas *gather* consists of collecting the information that each participant has reached the barrier. The information that all units have reached the barrier is distributed as *release*. Obviously, this separation is dependent on specific implementation and implementation exist, where both phases are merged, such as a butterfly barrier [Bro86]. Listing 3.2 shows a simplified template for the implementation of a barrier, which is used in this section to detail different algorithms. All specific methods are presented as extensions for this code snippet.

```
#include "RCCE_lib.h"
#include "iRCCE_atomic.h"
#include "iRCCE_barrier.h"
/* part of communicator for barrier specific information: */
typedef struct{
  int n;
  int id;
  /* implementation specific data */
  int master;
  /* ... */
} barrier_t;
void gather(barrier_t *);
void release(barrier_t *);

/* initialization encapsulated in RCCE_init() */
void iRCCE_barrier_init(barrier_t * b) {
  b=malloc(sizeof(barrier_t));
  b->n = RCCE_num_ues();
  b->id = RCCE_ue();
  b->master = 0;
  /* ... */
}

void iRCCE_barrier(barrier_t* b) {
  gather(b);
  release(b);
}
```

Listing 3.2: Code snippet of iRCCE barrier implementation

For example, we have implemented an SMP-like barrier based on a shared counter for the SCC. Another implementation of a barrier on the SCC is based on atomic operations to on-chip memory in combination with a linear lookup [Reb+11].

In contrast to these barrier variants, that use atomic operations for its effective implementation, the reference barrier of RCCE library only relies on atomic load and store operations to on-chip memory [MvW11]. Listing 3.3 gives an overview on the available synchronization support of the Intel SCC, which is supported by iRCCE *version 2.0*.

The main contribution of this section is the evaluation of classic synchronization algorithms for a many-core system without cache coherence. This includes the analysis of communication patterns to get a deeper understand-

```
/* ... */
/* ptr to local memory buffer */
extern type * RCCE_comm_buffer;
/* map on-chip memory twice (uncached) */
extern type * RCCE_flag_buffer;
// helper operation to calculate remote virtual flag address:
#define synch_buffer(lo,id) \
  (lo-RCCE_comm_buffer[RCCE_IAM]+RCCE_flag_buffer[id]);
// atomic test-and-set operation to on-chip location:
type atomic_test_and_set(type * ptr, int id);
 // atomic write in ptr (on-chip location) at id:
void atomic_store_n(type * ptr, int id, type val);
 // atomic read in ptr (on-chip location) at id:
type atomic_load_n(type * ptr, int id, type val);
// atomic add in ptr at id and return previous value:
type atomic_fetch_add(type * ptr, int id, type val);
/* ... */
```

Listing 3.3: part of `iRCCE_atomic.h`

ing of such an architecture. Specific attributes, such as the combination of software-controlled on-chip memory and a low latency on-chip interconnect, results in a predictability, which can be used for optimizations. Moreover, our experiments can be used to verify the communication model, which has been developed in this chapter.

In general, different attributes exist, that categorize the performance of a barrier implementation. Often micro-benchmarks are a good estimation, however a negative impact to the runtime of a specific application can not be excluded, especially if techniques such as remote spinning are used that can pollute a many-core interconnect. Arenstorf and Jordan discovered in 1989, that different situations have the demand for different implementations in order to achieve best performance [AJ89]. For a performance analysis, a comparison of the scalability and latency of selected barrier implementations is presented later in this section.

Table 3.2 compares different barrier implementations in terms of complexity in runtime and resources.

Table 3.2: Comparison of different Barrier implementations

	release	gather	hardware	spinning
linear	O(n)	O(n)	2 flags	*remote/local*
linear (fat)	O(n)	O(n)	n flags	*local*
k-ary tree	O(log n)	O(log n)	$(2 \times k)$ flags	*local*
centralized	O(n)	O(n)	2 shared counter	*remote*

Linear Barrier

The RCCE library holds a simple barrier that implements a *master-follower* approach. For this straight forward implementation, which is based on flags, UE with rank 0 is the master and all other ranks are followers. Listing 3.4 holds a simplified version of the barrier in C, which can be logically separated into a gather and a release phase. The complexity of this linear barrier implementation in terms of latency and on-chip memory consumption holds Table 3.2.

Two functions that abstract on-chip memory access are used to handle flag based synchronization, with `wait_flag` and `post_flag`. Both functions receive two parameters, first the location of target MPB and second the address of a local flag. Function `post_flag` sets a flag, for instance by writing a value that corresponds to set (e. g. `0x01`) to target on-chip memory location. Function `wait_flag` realizes a simple loop that continuously reads the given memory location until the value is non-zero.

In the first part of the barrier, the master checks *gather* flags in a round-robin order, which are located remote to the master and present for each rank. Next the master sets another remote flag at each other rank, the so called *release* flag, to release the followers by distributing the event that all cores have entered the barrier.

The elapsed times for the barrier implementation clearly show linear characteristics, which is a disadvantage in terms of scalability. Major advantage of the described method is a low on-chip memory footprint. Due to the fact that common message-passing based applications rarely use the barrier operation in a critical code path for synchronization purpose, the described implementation has been used as a reference for RCCE.

One optimization, which we have developed for the vSCC architecture, is changing the access pattern of the described linear barrier to local spinning.

Listing 3.4: Linear Barrier algorithm in C

```
1  /* -- Extends Listing 3.2: -- */
2  /* line 8: */
3    t_vcharp gather_flag;
4    t_vcharp release_flag;
5  /* line 16: */
6    b->gather_flag = RCCE_malloc(sizeof(char));
7    b->release_flag = RCCE_malloc(sizeof(char));
8  /* ------------------------- */
9  void wait_flag(t_vcharp l, int i) {while(atomic_load_n(l,i)!=0);}
10 void post_flag(t_vcharp l, int i) {atomic_store_n(l,i,1);}
11
12 void gather(barrier_t * b) {
13   if (b->id == b->master) { // master:
14       for(int i=0;i<(b->n);i++)
15         if(i != b->master) wait_flag(b->gather_flag,i);
16   } else { // follower:
17     post_flag(b->gather_flag, b->id);
18   }
19 }
20
21 void release(barrier_t * b) {
22   if(b->id == b->master) { // master:
23       for(int i=0;i<(b->n);i++)
24         if(i != master) post_flag(b->release,i);
25   } else { // follower:
26     wait_flag(b->release, b->id);
27   }
28 }
```

Listing 3.4: Linear Barrier algorithm in C

In detail, the gather phase of the reference implementation is problematic because it relies on reads to remote on-chip memory.

Especially for vSCC, a better option is a local placement of gather flags. This alternative allocation of gather flags changes the access pattern to local read but creates a higher on-chip memory footprint. We name this variant *fat-barrier*, because n flags are allocated instead of 2.

Chaotic Linear Barrier

Next, we discuss a new barrier algorithm that provides dynamic mapping of cores to on-chip synchronization resources. A requirement for the realization of dynamic mapping is a support of atomic *test-and-set* operation to on-chip memory. For the Intel SCC a general support of atomic operations to on-chip memory is missing. For our implementation, as described in Section 2.2.5, the emulation of selected atomic operations with synchronization register has been used.

Besides the implementation of a spin-lock, the presence of atomic *test-and-set* operations to on-chip memory locations enables a dynamic distribution of flags. We have investigated such a technique for the implementation of a chaotic linear barrier for the Intel SCC [Reb+11]. A fixed number of n *test-and-set* register is allocated[15] and initialized in an *unset* status.

This barrier algorithm can also be split into a gather and a release phase. First, each UE performs a linear search for a *free* spinning location, which means a shared variable in an *unset* status. In order to realize this, each UE performs a read operation on the synchronization register in a defined order, for instance according to the core ids from 0 to n. Between gather and release phase all UEs are spinning on a dedicated on-chip memory location except for the last incoming UE.

As a result of the read access, which is needed to perform an acquire operation, the value of target synchronization register changes its status to *set*.

The UE, which sees $n-1$ flags in *set* state, becomes the master and releases the first waiter, which is the UE spinning on the flag at first position. This UE releases the second waiter and so on, until all UEs have been released in linear way. Here the ordering is important, to avoid trespassing of UEs. Listing 3.5 shows the implementation of the chaotic linear barrier.

[15] n is equal to the size of a RCCE session

```
1  /* -- Extends Listing 3.2: -- */
2  /* line 8: */
3    t_vcharp flag;
4  /* line 16: */
5    b->flag = RCCE_malloc(sizeof(char));
6  /* ------------------------- */
7
8  // perform linear search for first available flag
9  void gather(barrier_t * b){
10   int step=0;
11   while(atomic_test_and_set(b->flag,step)) ++step;
12   b->master = step; /* mark position */
13 }
14
15 // release followers in a chain:
16 void release(barrier_t * b){
17   if(b->master == (b->n)-1){
18     atomic_store_n(b->flag,0,0);
19   } else {
20     while(atomic_load_n(b->flag,b->master) != 0);
21     atomic_store_n(b->flag,(b->master)+1,0);
22   }
23 }
```

Listing 3.5: Chaotic barrier algorithm in C

Central Barrier

For hardware distributed memory, the support of atomic *fetch-and-add* operations such as atomic increment can be used for the realization of a shared counter. A classic central barrier algorithm, which is based on two shared counter, has been proposed more than 20 years in the past by Lubachevsky [Lub90].

Implementation of this classic centralized barrier algorithm is possible, because the subsequently added hardware synchronization support of the Intel SCC research system emulates atomic increment operation to on-chip memory [Reb+12c]. Specific implementation uses two hardware counters, which are located off-chip. A control of these counters is realized with memory mapped registers, as described in Section 3.2.1.

According to Lubachevsky, to indicate its arrival, each UE increments target-shared-counter. Next, the last UE, which has incremented the counter

triggers a reset, while all other UEs are busy waiting for the reset. Target shared counter has to be exchanged according to Lubachevsky between even and odd sequences to avoid trespassing of UEs.

Off-chip synchronization resources, which are memory-mapped accessible have to be carefully used in combination with busy waiting, on a cluster-on-a-chip architecture without contention management. The access pattern of the release *phase* generates a request rate that cannot be served by the system interface FPGA, where the off-chip synchronization resources are located. According to the contention model of Chapter 2, a starvation of cores on the mesh with a lower access priority is resulting of the straight forward implementation of the Lubachevsky barrier. The consequence is a limited applicability of the emulation of atomic increment operations to on-chip memory location.

We propose a back-off, which represents a common method to relax the contention problem resulting of a central synchronization point [Reb+12c], as an adequate solution for the described issue [GT90; MS91].

First barrier implementation introduces an exponential back-off for the release cycle of the previously described central barrier implementation. This method significantly reduces the contention and already leads to promising results. Listing 3.6 holds source code for the resulting barrier implementation, which is based on a simple interface for atomic operations, such as specified by Listing 3.3. If hardware does not support the atomic increment operation, such as the SCC, this functionality has to be emulated. This emulation influences the back-off function `delay()` which can be suited for a given access pattern and specific latency, whereas the use of a quantified back-off leads to a significant reduction of contention.

Another method for the realization of a barrier implementation, which is based on a shared counter for the SCC, is the combination of the central *gather* phase from Listing 3.6 with a distributed *release* phase from Listing 3.4. Here, the linear method of the reference barrier implementation is used for the release cycle to avoid a high contention on the AIR. A better performance for a group size of more than 8 threads can be achieved, compared to the chaotic linear barrier implementation. A further advantage is that only one hardware synchronization register is allocated for this first AIR barrier implementation, but an additional flag is introduced.

```
1  /* -- Extends Listing 3.2: -- */
2  /* line 8: */
3    int * counter;
4    int phase; // helper variable toggles between even and odd phases
5  /* line 16: */
6    b->phase = 0;
7
8  void gather(barrier_t * b) {
9    if (atomic_fetch_add(b->counter,b->phase,1) == (b->n)-1) b->master = 1;
10 }
11
12 int release(barrier_t * b) {
13   if (master) atomic_store(b->counter, phase, 0);
14   else while(atomic_load(b->counter, phase)!=0){delay();}
15   b->phase = !(b->phase);
16 }
```

Listing 3.6: Atomic-increment barrier algorithm in C

3.5.3 Results

We use a micro benchmark to evaluate presented barrier implementations regarding their scalability. Figure 3.17 shows results for the Intel SCC, with average runtimes for each barrier variant for an increasing core size.

To enable a fair comparison of all barrier implementations, which are based on the special hardware synchronization support of the SCC, we use the reference barrier implementation of RCCE with a bypass-flag-optimization. As presented in Section 2.3.4, this optimization avoids a flush of the write-combining buffer for each write operation to a remote-flag value. A speedup of about 2x is resulting for the simple master-follower reference implementation, because the linear release phase which includes write access to remote on-chip memory is the dominant part of the overall runtime.

In detail, we can observe a stronger effect of a raising communication distance for the linear flag reference implementation compared to the chaotic linear barrier for an increasing core count. As more cores are added, the communication distance to the master raises inevitably on the 2D mesh. This effect strongly depends on the underlying communication topology. Our measurements show that both barrier implementations scale linearly with the number of cores. However, with a utilization of different access patterns and hardware-synchronization means, the linear slope could be strongly reduced.

An even better performance for a larger count of UEs can be achieved by using a tree-based communication pattern with a static mapping. In the diagram this implementation is named *tree flag*, whereas a classic 2-ary tree algorithm with 2 successors per node provides logarithmic scalability and outperforms the linear variant for more than 8 participating UEs.

The plotted orange curve from Fig. 3.17 shows that the runtime of a chaotic linear barrier is in the same order than the reference implementation. A major drawback of this algorithm is a remote spinning to on-chip memory. Additionally, the dynamic allocation scheme requires atomic *test-and-set* operations to on-chip memory.

The red curve from Fig. 3.17 shows the elapsed time for a barrier operation in average that targets a central counter. Here, delay has been introduced which is iteratively increased to realize busy-waiting with exponential back-off.

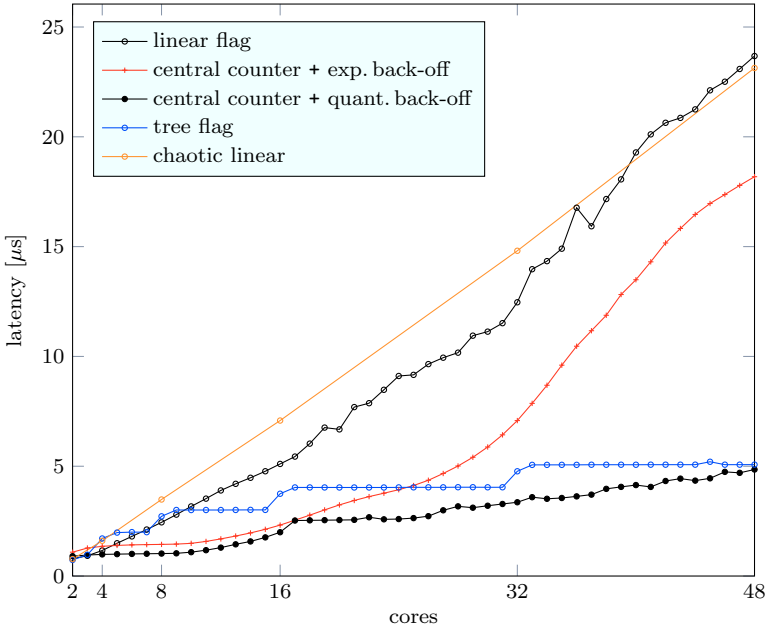


Figure 3.17: Scalability of selected barrier Implementations

Surprisingly, a significantly speedup can be achieved for the SCC platform with a barrier implementation according to Lubachevsky. The back-off barrier variants with a shared counter perform fast across the board, despite the fact that the counter is located off-chip, with a result that the latencies to the counter are 3 times higher than the access to a flag. Nevertheless, compared to the linear reference implementation, a speedup of up to 4.5 x can be achieved for the implementation of a central synchronization construct.

The results of our experiments from Section 3.5.1 have demonstrated, that a high contention is problematic for the SCC upon a certain core count (cf. Fig. 3.16). We detected, that remote spinning of more than 30 cores to a single off-chip synchronization register can lead to starvation of cores with a large distance to the physical location (cf. Fig. 2.6).

Consequently, the use of a back-off is essential for the barrier implementation based on a central counter and on remote-read operations. It enables a certain flexibility, so that a random number or the predictability of interconnect latencies can be taken into account for a back-off calculation. In order to explore these effects, we have implemented and compared two variants. The red curve from Fig. 3.17 represents an exponential back-off with a base of two and optimized upper and lower bounds for each number of cores. This means a maximum and minimum back-off is derived from the group size of the barrier. The black curve is resulting of a back-off function, which only takes the communications distance and degree of concurrency into account. Degree of concurrency means in this context, how many cores concurrently access the central synchronization point.

3.6 Conclusion

In the previous Chapter 2, we have presented a communication model for many-core processors with non-coherent memory coupled cores. We have derived a basic communication model for this kind of architecture. With different micro benchmarks, we have demonstrated the predictability of point-to-point communication and verified the communication model. In this context, we were able to analyze low-level communication for specific schemes of the many-core research vehicle. In detail, our communication model is useful to analyze different access patterns to software controlled on-chip memory. It can be used to explore limits of Intel's SCC many-core processor architecture that waives full chip cache coherence and includes architectural support for on-chip message passing.

Uncontrolled concurrent access can cause memory contention which degrades performance. The communication model has been extended to predict on-chip memory contention, dependent on the routing mechanism of an on-chip interconnect.

In this chapter, the development of a communication-and-synchronization interface for a cluster-on-a-chip processor has been described. It is shown, that effective communication can be realized by means of a light-weight communication layer. Therefore, two goals have been reached: optimization of the existing functionality and its extension, such as the analysis of alternative communication schemes and extension of communication protocols.

We specify a new interface for the Intel SCC as an extension of a basic communication environment. Another goal was to overcome limitations of an existing communication environment that prevents a non-blocking communication, which is a requirement for the use as an inter-kernel communication layer. Moreover, synchronization constructs as well as point-to-point communication has been optimized. We have shown that atomic operations are one important mean for the implementation of dynamic allocation schemes for on-chip memory and low-latency communication functions. In addition to that, we have successfully introduced relaxation techniques, such as a quantified back-off, to prevent starvation of cores resulting of central synchronization points.

4

"You should also be inspired to create your own hypervisor, using your own pets as logo." [Rus07]

System Software and Application

The role of system software for the many-core area is currently an open question, especially regarding operating system support for future processor architectures. In this context a large number of important challenges exist, such as the scalability of software design as well as energy efficient control for thousands of cores per chip, towards EXASCALE computing.

The focus of this chapter is integration of a small inter-kernel communication and synchronization layer to a bare metal framework and analysis of its overhead. In Chapter 3 of this dissertation, a spin lock was used to verify a contention model for a many-core architecture with on-chip networks.

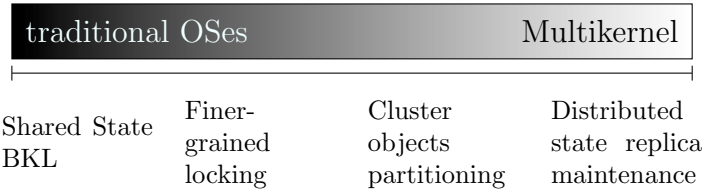


Figure 4.1: Locking granularity in operating systems

In general, locking is a method to protect concurrent access to shared resources, such as shared memory regions. An example are internal structures of

an operating system (OS) with concurrent access as a result of multithreading support.

Figure 4.1 illustrates the locking granularity of operating systems. In general, internal representation such as states and management structures are shared along parallel execution units to handle allocation of resources. Either implicit or explicit coordination from big kernel lock to replicated states is an option for the design of operating system kernel. For a uni-core processor with local data it would be sufficient to disable interrupts and thereby avoid a preemption within a critical section. If the number of cores grows, for instance as a result of many-core technology, a scalable design becomes more and more important.

Organization of this Chapter

The focus of this chapter is on system software support for the realization of applications that follow established programming concepts.

We have discussed programming models for an architecture such as the Intel SCC in previous work [Cla+13a]. Additionally, in previous publications we have presented *MetalSVM* and demonstrated its performance for the Intel SCC, which represents a many-core architecture with 48 in-order computing cores in the absence of cache coherence [Reb+12b]. First experiments with weaker memory consistency models are presented in our previous work for the SCC [Lan+12b]. To further analyze the scalability and effective communication in the absence of cache coherence, we have designed a virtual extension of the SCC's on-chip interconnect [Reb+12a]. This includes transparent access to the communication library and additional functionality with software emulation of system software extensions, such as a DMA controller and additional synchronization register. We name this architecture vSCC, because it consists of multiple SCC processors, which can be used as a cluster-on-a-chip architecture.

My contribution to the work which is presented in this chapter include:

- Development of the vSCC concept
- Contribution to a bare-metal framework for the Intel SCC [Reb+12b]
- Contribution to the implementation of a low-level communication and synchronization layer [Reb+11]

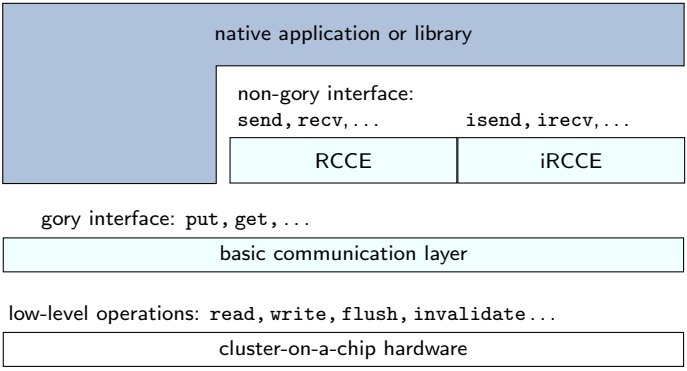


Figure 4.2: Focus of this chapter in relation to the investigated layered communication structure

Software extensions that emulate hardware support for communication and synchronization, for instance hide off-chip communication latencies and realize a working prototype with 240 cores based on the Intel SCC hardware [Reb+15].

This chapter is organized as follows. The next section summarizes related work on system software support, parallel programming and memory virtualization for many-core systems.

Section 4.2 presents the basic concept of MetalSVM. Section 4.3 describes the efficient implementation of a bare-metal hypervisor for the Intel SCC. Section 4.4 shows results for NPB BT and LU synthetic application benchmarks and Jacobi method kernel benchmark.

4.1 Related Work

Especially in the field of HPC, parallel applications do not run bare metal on a computer system. Widely used operating systems, such as Linux, are capable for multitasking and have been designed for different workloads, which includes user interaction as well as applications that target a high throughput. A certain jitter, also known as OS noise, can hardly be avoided for such a design. For highly scalable systems, studies have shown that the use of a

general purpose operating system can have negative impact to the overall system performance [Tsa+05]. Related research question is if special purpose operating systems are appropriate choice to target those issues.

MetalSVM is based on a distributed kernel approach, which is comparable to the *barrelfish-multikernel* approach [Bau+09]. However, the general design approach is different, so that in this related work a microkernel is described which targets a minimalistic functionality. We target the realization of a specialized monolithic kernel, because the virtualization of a shared memory machine is performance sensitive.

Helios is a research project that follows the concept of heterogeneous kernels, for coordination and execution of tasks [Nig+09]. Fensch and Cintra propose hardware support for remote cache access to maintain coherence of a tiled processor architecture [FC08]. Corey's concept is that applications control the sharing of data resources [Boy+08]. For the realization of such an approach, different abstractions are proposed that should be provided to applications.

Hardware-and-software solutions – that implement a low-level inter device communication for high performance interconnects – are related to our research on vSCC. A classic hardware approach to accelerate communication for a distributed memory system are special purpose processors. For instance, the Intel Paragon has followed such a concept in the 90's [Hoc94].

The reverse-acceleration approach targets the offload of communication from throughput optimized cores to latency optimized computing cores, instead of offloading compute intensive tasks to throughput optimized cores. Our research vehicle vSCC shares this basic idea with the reverse-acceleration approach [PLK09]. The reverse-acceleration approach has been successfully applied as a work-around for the Xeon Phi coprocessor.

For example, a hardware limitation of Xeon-based host systems prevents a good direct communication performance between two PCIe devices. The solution for MPI-based applications is a *proxy task* that is running on the host and acts as a message broker to the communication library [Pot+13]. This work follows a similar approach compared to our work regarding communication offload, where the general goal is to accelerate communication of many-core systems. However, our work differs, because we target a low-level communication path on data transfer layer and extensions to systems software instead of a modification of the communication library.

4.1.1 Programming Models

Basically, parallel programming models rely on language extensions or runtime libraries. From a system-software perspective, specific models can mainly be categorized if the address space is shared, partly shared or distributed. For example, MPI, OpenMP, and UPC fall into these categories, that are distributed memory, shared memory, and partitioned memory [UPC05].

A common implementation are threads and processes, which are both typically handled by the operating system with the main difference that they share an address space or not. Another distinction can be made if the communication and synchronization of parallel execution units is explicitly or implicitly expressed. Classic message passing based programming models provide explicit communication and implicit synchronization.

The concept of shared memory parallelization is implicit communication and explicit synchronization between threads. Related to our approach of integrating a shared virtual memory management to a bare metal hypervisor, different memory models are defined for parallel programming paradigms.

Abstraction of parallelism is an important task. A memory consistency model, also called memory model, specifies behavior of memory regarding read and write operations. Programming languages, that natively support the concept of thread-based shared-memory parallelism, for instance Java 5.0 or C++11 specify a memory model. The challenge for a memory model is that it should fit to the underlying hardware to avoid performance penalties. For instance, sequential consistency represents the strongest definition, which prevents optimization that shared-memory machines with multiple cores provide. Because sequential consistency assumes that a write operation to a shared memory location has to be observed by each core in the same order.

Relaxing the total order of read and write memory operations in a memory model leads to so called relaxed consistency models [AG96]. An example for a relaxed consistency model is lazy release consistency, which has been introduced by Keleher et al. [KCZ92].

The memory consistency model of OpenMP is an example of a relaxed consistency model. Since version 2.5, a concrete specification of the OpenMP memory model exists, instead of just describing case by case the behavior of memory in specific parallel regions [HS08]. For example, the flush directive, which can be used to control consistency in an explicit way, has lead to misunderstandings in the past, especially in combination with locks. Related issues become more and more present if the coherence is controlled in software.

OpenMP is a widely used interface and can be seen as the de-facto standard for Shared Memory Programming. Version 1.0 of OpenMP specifies a fork-join model for FORTRAN (1997) and C/C++ (1998), which especially suites loop parallelization. Parallelism of applications can be expressed through compiler directives, which are added by pragmas. A compiler with OpenMP support can use these pragmas to extend a program according to the specification. Main advantage of this expression of parallelism is the possibility of incremental parallelization, because the absence of some pragmas can still lead to correct parallel applications only with performance or scalability issues on specific hardware. Moreover, if pragmas are ignored by a compiler without support for OpenMP, it can create a serial version of the shared memory parallel application. In addition to that, the generation of a parallel application is based on a runtime which handles for instance the creation of threads and the housekeeping of a thread pool. Explicit synchronization can be expressed with OpenMP by constructs such as the `critical` or `barrier` construct and functions such as a `lock` routine. Implicit synchronization represents in OpenMP for example a parallel region without a no-wait clause, which implies a barrier.

OpenMP has been clearly designed for shared memory machines, also called SMP systems. Version 4.0 of OpenMP has been released in 2013 and introduces accelerator support. Here, copy clauses are for instance used to transfer data between distributed memory regions. Therefore, dependencies are expressed to decide which data has to be copied from source to target and vice versa. This creates potential to program machines consisting of multiple coherency domains [OMP13].

In order to create the possibility of executing parallel programs on hardware without shared memory, data dependencies in OpenMP have to be expressed explicitly for a target region as part of a device construct. Main target of these new constructs is the support of accelerator and coprocessor hardware, however the offloading concept is not limited to this kind of hardware. A device is an abstract unit that could also represent nodes of a symmetric cluster or another coherency domain on a many-core processor without full-chip cache coherence.

4.1.2 Virtualization

In general, virtualization is a technique, that describes the use of a software that behaves like another hardware or software component. Regarding com-

puter systems, virtualization is present in multiple areas, such as computer hardware, networks or operating systems.

The configuration of a virtualized operating system consists of a host and a guest system. The typical task of a hypervisor is the management of multiple guests that share hardware resources by running on a single host. In such a scenario, the interaction between the components is that each guest, as a separate operating system instance, runs on top of a hypervisor. Dependent on its classification, a type-1 hypervisor can run directly on the hardware, which is also called bare-metal. A hypervisor that is classified as type-2 runs on top of an operating system [Gol73].

Another distinction, that is commonly made, is how the guest operating system executes privileged operations. The technique that a guest operating system is aware of being executed on top of a hypervisor, is known as paravirtualization [WSG02]. Instead of directly executing a *systemcall*, the guest traps a *hypercall* to execute privileged operations. This is a contrast to full virtualization, which enables the execution of unmodified operating systems, without such a support.

In this work, the virtualization of a shared memory system for an x86-based many-core system within a paravirtualized environment is discussed. The goal is a support of legacy shared-memory programs on hardware without cache coherence in combination with message-passing applications which can run close to bare-metal.

A common way for communication of systems with hardware distributed memory is message passing. However, many applications show a strong benefit using shared-memory programming model. Shared Virtual Memory (SVM) is a classic concept to enable the shared memory programming model on DSM systems. A cluster, that appears as an SMP system, is commonly called SVM system.

Many implementations are realized as additional libraries or programming language extensions. In this case, only parts of the program data will be shared and a strict disjunction between private and shared memory is required. Intel's Cluster OpenMP is a typical example of an SVM system. First experiences with Intel Cluster OpenMP have been presented by Terboven et al. [Ter+08]. It turned out that, the disjunction between private and shared memory has side effects on traditional programming languages like *C/C++*. For instance, if a data structure is located in the shared virtual memory, the programmer has to guarantee that all pointers within this data

structure refer also to the shared memory. This restriction of the environment prevents execution of existing OpenMP applications.

An important attribute of an SVM system is, that only the memory is virtualized. Consequently, the access to other resources of distributed systems, such as the file system, requires additional support. The integration of an SVM system into a distributed operating system, in order to offer the view of a unique SMP machine, increases the usability.

Since IVY [LH89], a lot of work has been done on SVM systems. TreadMarks [Kel+94] is an important SVM system, that Intel's Cluster OpenMP was based on. However, those systems are commonly based on traditional message-passing oriented networks or use a RDMA (Remote Data Memory Access) engine to access remote memory locations. Setup costs to program RDMA engines are high and increase the overhead of using an SVM system.

The Scalable Coherent Interface (SCI) belongs to the memory-mapped networks and offers a transparent read and write access to remote memory [SCI93]. This represents a similar basic communication concept to the Intel SCC architecture. In contrast to a cluster-on-a-chip architecture, that waives full chip cache coherence, the SCI standard defines a cache coherency protocol. This feature of SCI has never been realized by commercial hardware because of the connection of the processor to the SCI network. Due to the fact that PCI devices were used as network cards such a feature, which needs processor internal information, such as the status of a cache-line, could not be supported. The similar communication concept of SCI and SCC is based on several processing units that are able to communicate transparently over shared memory regions without the support of cache coherence. Both share the attribute that the amount of low-latency remote memory is limited. For SCI, this low-latency memory is located on the network device. As a result, the ratio of local to remote memory latencies differs significantly between the two systems, because of the on-chip network and the on-chip remotely addressable memory integration of the Intel SCC. To decrease this ratio, with the SCC, the x86 processor architecture has been adapted to the new communication concept.

Several projects have realized an SVM system on top of an SCI cluster, however with limited usability due to the implementation at user level. *NOA* [MP98] used SCI as fast message-passing interconnect and did not exploit the capabilities of a transparent remote-read-and-write memory access.

Another approach is the integration of an SVM system into virtual machines, for an easy application of common operating systems and development

environments without changes. An example for a hypervisor-based SVM system is vNUMA, that has been implemented for Intel Itanium processor architecture [CH09]. One founder of vNUMA argues in: *"Many-core Chips – A Case for Virtual Shared Memory"*, to extend this concept for Many-Core Chips, based on the experiences with vNUMA. The basic idea is to provide a flexible and scalable platform for shared memory programming with an additional software layer. The proposed virtual shared memory system targets an absence of contention, for example applications without lots of communication [Gam+99].

Regarding x86-based compute clusters, ScaleMP[1] has developed the so-called vSMP architecture, which enables cluster-wide coherent memory sharing. This architecture implements a virtualization layer underneath the OS that handles distributed memory accesses via InfiniBand based communication. Since both approaches implement an SVM system as a virtualization layer besides hardware and operating system, they share similarities to the *MetalSVM* approach.

The main difference between existing approaches and *MetalSVM* is their limitation to established interconnect fabrics.

4.2 Concept of MetalSVM

MetalSVM is designed to support the new communication concept of the SCC, which includes distinguished capabilities of transparent read and write access to its global on-die and off-die distributed shared memory.

The project started in 2010 at RWTH Aachen University[2] and targets the implementation of a virtualization layer to a bare-metal hypervisor. The main goal of this new hypervisor is software-controlled coherence for future many-core systems. As a result, common operating systems – that combine SMP and para-virtualization support – will be able to run on non-coherent memory coupled cores in a transparent way. Figure 4.3 illustrates the basic concept of *MetalSVM*.

In contrast to the mentioned SVM libraries, our SVM system targets the integration to a small hypervisor, so that an operating system such as Linux is able to run as a virtual machine on top of *MetalSVM*. Following this approach, the effort in development and maintainability is cheaper compared to a new

[1] `http://www.scalemp.com`

[2] initially funded by Intel Labs Braunschweig

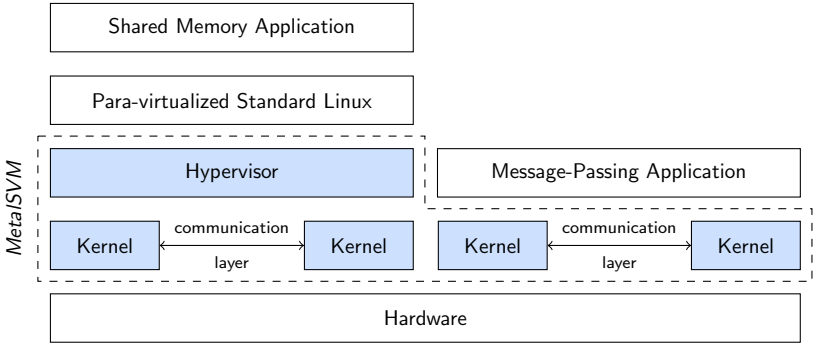


Figure 4.3: Basic Concept of MetalSVM

distributed operating system because of its unique view to the hardware in contrast to all components of an operating system, such as the file system and process management.

The SVM system of MetalSVM locates shared pages or write notices in hardware-distributed shared-memory. As a result, each core can transparently access shared data which suits a one-sided communication system. Memory consistency can be controlled with a flexibility, so that changes can be communicated by small messages, which are sent by our inter-kernel communication layer iRCCE.

Example of those changes is control of consistency on page granularity. Regarding strong memory model each page can exactly have a single owner. For a memory access that targets a page frame by a core which is not its owner, an access violation occurs. A message is send to the owner, which has to flush local cashes to recover memory consistency and delete its write and read permissions. Next, the page access notification can be send to the requesting core, which can set read and write permissions and retry the memory access.

4.2.1 Motivation of MetalSVM

The focus in this section is the efficient implementation of a minimalistic operating system kernel as a bare-metal hypervisor for a many-core processor. Source, functionality, and the operation of our kernel, as well as the interaction

with a communication layer is described in the following. Furthermore, the boot procedure of the SCC is described in detail from reset to the starting point of our light-weight operating system kernel. This procedure is performed by a bare-metal framework, which is part of the MetalSVM project. Finally, the performance of a paravirtualized Linux guest on the SCC hardware is evaluated and results are shown for context switch latencies for Linux and MetalSVM hosts.

The main difference to our approach is that vSMP and vNUMA explicitly use message-passing between the cluster nodes to transfer the content of the page frames, whereas our SVM system can cope with direct access to these page frames. In fact, we want to exploit the SVM system with SCC's distinguishing capabilities of transparent read-and-write access to the global off-die shared memory. This feature will help to overcome a drawback of other hypervisor-based approaches regarding fine granular operations. An evaluation of ScaleMP's vSMP – with synthetic kernel benchmarks as well as with real-world applications – has shown that vSMP architecture can stand the test if its distinct NUMA characteristic is taken into account [Sch+10].

The evaluation of vSMP has shown that the expensive synchronization is a major drawback for this kind of architecture. We believe that especially this drawback will not occur in the context of our solution for the SCC, because *MetalSVM* provides better options to realize scalable synchronization primitives.

4.2.2 Integration of iRCCE into MetalSVM

RCCE library has been designed as a light-weight communication library for the SCC architecture. This implies the limitation to blocking communication, which disqualifies its use as inter-kernel communication layer. For instance circular dependencies, a common source of deadlocks for blocking communication, can not be excluded for such a communication scenario.

We have developed an extension – with iRCCE – that provides non-blocking [Cla+13b] and asynchronous communication [Lan+12a]. An obvious way for the realization of non-blocking communication functions are additional threads, which handle communication in the background. As a result, the application thread can immediately return from the communication function. Although this approach seems to be quite convenient, it is not applicable in *bare-metal* environments where a program runs without any operating system support, including support for multithreading.

Another approach could be that the application triggers the communication progress itself. Regarding the design of *MetalSVM*, each kernel, as a bare metal application, can handle this task. For our implementation, a non-blocking communication function returns a so called *request handle*, which can be used to trigger its progress with `push`, `test` or `wait` functions.

In the context of *MetalSVM*, the communication progress is triggered at the occurrence of interrupts, exceptions or system calls. The *MetalSVM* kernel checks at these points, if a message is pending. In order to realize that, all messages start with a header, which specifies its type and payload. The definition of a message type is important, because this layer will be used to send messages between the instances of the SVM system.

The maximum delay between sending and receiving a message header is as large as a time slice, because at least after one time slice an interrupt will trigger, which checks for incoming messages. *MetalSVM* allows the sender to trigger a remote interrupt on the side of the receiver in order to reduce the delay. This creates additional overhead, because the calculation on a remote core would become unnecessarily interrupted. Nevertheless, for high priority messages, we provide this option.

For communication between coherency domains, besides iRCCE, *MetalSVM* supports a TCP/IP stack, which is briefly described in the following paragraph on device drivers.

4.3 Efficient implementation of a bare-metal Hypervisor

The integration of an SVM management system influences the design of a hypervisor kernel. In this section, we detail the implementation of such a kernel including interrupt handler, device driver, file system, and the hypervisor.

As an application example the implementation of a light-weight operating system kernel for the Intel SCC is discussed. For portability reasons, the design of this kernel follows a classic concept, that divides the implementation into hardware dependent and independent parts. As a result, different hardware architectures can be supported with less effort.

An existing interface from the Linux kernel was our choice for integration to *MetalSVM* [Lan10]. If the interaction between host and guest is based on a de-facto standard interface, a main advantage is that major changes to the

Linux kernel code can be avoided. *Paravirt-ops* represents a well-established interface to run Linux as a para-virtualized guest. This interface is part of the standard Linux kernel and used, beside KVM [Kiv+07] and Xen [Bar+03], for the realization of our approach.

Instead of integrating one of these complex hypervisors to *MetalSVM*, *lguest* is a small hypervisor which provides all required features for the realization of the *MetalSVM* project [Rus07]. Moreover it does not rely on hardware virtualization support such as Intel VT-x, which does not exist for the SCC.

4.3.1 Bare metal framework

The main difference between Intel's SCC research processor and other x86 processors is a missing BIOS or equivalent firmware interface support. Moreover, the SCC experimental platform has no stand-alone memory initialization. The only possibility to boot an operating system or a bare-metal application on the processor cores of the SCC is preloading their memory content into a bootable state of its private memory regions. As described in Section 2.2, the general system initialization works with a standard PC, which has a direct access to memory and the configuration register of the SCC.

MetalSVM is *Multiboot*[3] compliant. As a result, a standard boot loader such as GRUB can boot *MetalSVM* on commodity x86 hardware. The boot procedure of the SCC research platform is different and thus briefly described in the remainder of this paragraph. In other words, the main task of our bare-metal framework is to get the SCC experimental processor into a *Multiboot* compliant state.

First, the reset pins of selected SCC cores are pulled which stops its execution. Next, Lookup Table (LUT) are initialized and a separate memory image is loaded to each memory controller of the SCC. As the SCC does not provide a boot loader, our framework provides minimal assembler code which is located at a hardwired address that the instruction pointer of each core holds after reset. This code initializes the stack pointer and installs a rudimentary Global Descriptor Table (GDT), which is a data structure that x86 based architectures use to handle memory regions with different characteristics, so called segments [Intel07]. The presence of this information is a requirement, so that the setup routine can switch the processor from real to protected mode and subsequently to 32 bit mode. As a final step, our routine

[3] `http://www.gnu.org/software/grub/manual/multiboot/`

jumps to the starting point of a bare metal application or respectively the *MetalSVM* kernel.

MetalSVM uses ELF, which is the standard binary file format for Unix systems which was not supported by SCC's software stack `sccKit`. We used GNU utility `objcopy` to generate a loadable, raw binary kernel file without symbols and relocation information.

Composed to a single image with the provided `sccKit` tools are: The startup routine (from real to protected mode) as previously described and information, which are generally provided by the bootloader and the kernel itself.

As a next step, a configuration file for the LUTs and one object file per memory controller of the SCC platform can be created. Specific object files can be loaded into the off-die memory of the SCC before releasing the reset pins of the SCC cores.

Device Drivers

With *QEMU*[4], a generic and open source machine emulator and virtualizer has been used for the development and for testing purposes. Through the integration of a driver for the Realtek RTL8139 network chip, which is also supported by *QEMU* as an emulated device, standard kernel components could be tested in a simple way.

The communication between the SCC cores running *MetalSVM* is not limited to the iRCCE library and its mailbox extension. With the integration of *lwIP*, a light-weight TCP/IP library, the flexibility is increased [Dun01]. Consequently, BSD sockets are made available to user space applications to establish communication between the SCC cores and the MCPC. In previous work, we have shown the performance gain of the resulting network layer [Lan+12a].

Beside other devices, the network capabilities of *MetalSVM* will be forwarded from the guest operating system to the hypervisor through *virtio*. Rusty Russell proposed virtio to create an efficient and well-maintained framework for IO-virtualization of virtual devices commonly used by different hypervisors [Rus08]. In our scenario, for instance the network capabilities of *MetalSVM* are used as a backend by just forwarding the requests of the Linux guest operating system to the hypervisor.

[4] `http://www.qemu.org/`

Interrupt Management

The SCC platform includes 48 cores that are based on P54C. As a second-generation Pentium core, the P54C was the first processor which integrates a local Advanced Programmable Interrupt Controller (APIC) on-chip. A local APIC can be used to trigger the scheduler periodically by programming the local timer interrupt. *MetalSVM* provides a simple priority-based round-robin scheduler or can be configured in tick-less mode.

Interrupts are important for the SCC, because the architecture does not use the traditional way to integrate I/O devices (*IO-APIC*) or to send inter-processor interrupts (IPIs). Besides the timer interrupt, the local APIC provides two programmable local interrupts (*LINT0* and *LINT1*). Therefore, a core configuration register exists for each core of the SCC, which is mapped to the address space of all cores. As a result, core x can trigger interrupts on core y. By using this mechanism to trigger interrupts, the receiving core has no information on the origin of the interrupt.

Since `sccKit` version *1.4.0*, the System Interface of the SCC includes a Global Interrupt Controller (GIC), which provides a more flexible way to handle interrupts [SCC11]. If interrupts are triggered by the GIC, the receiving core is able to determine the origin of this interrupt, which was not possible in the previous configuration. *MetalSVM* uses the GIC especially for inter-core communication with a mailbox system [Lan+12b]. Here, the interrupt source is important, because otherwise all mailboxes have to be checked if an interrupt occurs.

File system

MetalSVM has an elementary *inode* file system with the intention to use a volatile ramdisk which can be manipulated at runtime. As the SCC system does not provide large amount of non-volatile storage, a file system is limited in use.

Moreover, the integration of a C library extends the usage of *MetalSVM*, such as a support of bare-metal message-passing applications. We have chosen *newlib*[5] which is a C library designed for embedded systems, because it meets the requirements of streamline intel architecture based cores, such as the SCC cores. This C library represents a restricted implementation, which is optimized in terms of memory footprint and performance. Related attributes

[5] `http://sourceware.org/newlib/`

that a single many-core processor core can share with embedded processor architectures are: in-order execution, a moderate frequency and relatively small caches.

Scheduler

The presented hypervisor has different requirements than the scheduling methods, that modern operating systems implement. We intend to handle only a couple of tasks with a predictable ratio, interaction, and static priorities. Examples of those tasks are: *daemon tasks* that realize communication in the background and *monitoring tasks* that count occurring page faults and trace changes of ownership.

A simple but fast algorithm has been applied to manage tasks. The scheduler keeps an array with as many items as priority steps exist. Per priority there is one linked list of tasks waiting for execution. Between timeslices the scheduler appends the previous task to the end of its priority list and selects the head of the current processed priority level list for execution.

The small set of implemented priorities in *MetalSVM* arises potential for optimizations. One optimization is already implemented in the networking layer. Network packet traffic is handled by a special kernel task with flexible priority. This way it is possible to balance between high network throughput and overall system latency.

4.3.2 Many-core Virtualization

A first prototype which implements an SVM system to a bare metal hypervisor has been previously published [Lan+12a]. Additionally, we have published further optimizations of the prototype and first experiments with relaxed consistency models [Lan+12b]. In the subsequent paragraph overhead of our bare-metal framework is evaluated and in the remainder of this chapter the scalability of its light-weight communication layer is analyzed.

4.3.3 Hypervisor Performance

For the realization of a transparent Shared Virtual Memory environment within a para-virtualized environment, the hypervisor has a major impact on the overall system performance. If the hypervisor introduces a significant

overhead, this can degrade performance of the guest operating system especially regarding memory management, context switch and process handling.

Measurements of three representative latencies identify a reduction of lguest's virtualization overhead in combination with *MetalSVM*. The context switch from guest to host execution is performed for each *hypercall* and at the majority of interrupts. Moreover, page faults of a guest application can involve up to three guest-host roundtrips.

Table 4.1: Benchmark results for the Intel SCC platform (Linux *2.6.38.3*)

Benchmark	Hypervisor		Ratio $\frac{MSVM}{Linux}$
	Linux	*MetalSVM*	
Host-guest context switch	2 042	2 113	103 %
Page fault	918 679	867 676	94 %
`getpid()`	191	191	100 %
`fork()`	3 216 767	3 101 387	96 %
`vfork()`	220 317	236 207	107 %
`pthread_create()`	16 256 988	10 883 839	67 %

Values in processor ticks

To quantify the overhead of our environment, we present latencies of selected *system calls*, such as `getpid`, `fork`, `vfork`, and `pthread_create`, in Table 4.1. Because of its low net execution time and due to the fact that it does not involve a host-guest switch, `getpid` indicates the overhead of a system call. To measure the elapsed time, for the creation of a task and the copy operation of a whole page directory of the original task, we used `fork` and `vfork`. For the execution time of `pthread_create` a significant difference between Linux and MetalSVM can be observed. This effect can be explained by the coarse granularity of the current timer implementation of *MetalSVM*.

Context Switch Latency

In this section, the advantages of a bare-metal framework for *MetalSVM* are discussed. To quantify the overhead of our implementation for the Intel SCC,

we compare the context-switch latencies of MetalSVM to sccLinux. In addition to that, we compare the *lguest* implementation of *MetalSVM* and Linux *2.6.38.3*. For the benchmark results which are presented in this section, a single instance of the host operating system is running on a single core of the SCC platform[6].

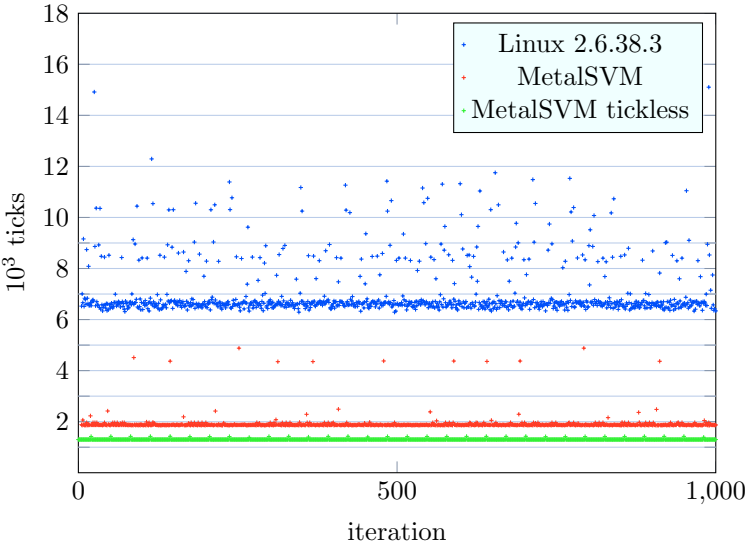


Figure 4.4: Scheduling overhead of MetalSVM [Reb+12b]

To measure the context switch latency, two tasks are running in such an environment with a high priority. Each task periodically reads the time stamp counter in a loop and records the difference between subsequent counter values. *Gaps* in a specific range indicate the latency of a context switch. Specifically the gap has to be lower than a time-slice and the interrupt handler. The method is similar to the classic hourglass benchmark [Reg02].

Figure 4.4 shows benchmark results for context-switch latencies of *Linux* and *MetalSVM*. Specific values of Linux have a minimum around 6400 pro-

[6]core/mesh/memory frequency: 533 MHz/800 MHz/800 MHz

cessor cycles, whereas the distribution of latencies has a certain noise which has no clear signature and changes from time to time.

In contrast to the latencies of *MetalSVM*, which have a high predictability with a minimal context switch latency of around 2100 processor cycles. In terms of predictability, the diagram shows a second level of around 5000 ticks for the context switch latencies of *MetalSVM* with a regular pattern. This effect can be explained by a background task of the network driver, which becomes periodically active.

The results show that our self-developed operating-system kernel achieves a significant reduction of overhead. This is an advantage, if the execution of system software is in the critical path, such as the access of a parallel application to a memory region with software-controlled coherence. However, the accumulated overhead reduction, which is a result of the lower context switch latencies, is negligible and can not motivate the bare metal execution of applications in general, because its disadvantages outweight. Nevertheless, the high predictability of our light-weight execution environment is a main advantage regarding the scalability of architectures with many tightly-coupled processor cores.

Moreover, our framework has a simple code-base which is easy to understand, to modify and also to validate.

4.4 Application Examples

In this section, floating-point-intensive benchmark results are discussed to explore attributes of Intel SCC's new communication concept. First, experimental results of a common solver, that represents a class of so called stencil codes, is analyzed. Second, a set of synthetic applications is used to demonstrate the effect of our optimizations to the RCCE family.

4.4.1 Jacobi

The Jacobi method is a prototype for stencil-based iterative methods in numerical analysis and simulation [HW11]. For instance, it can be used to simulate the distribution of temperature or pressure by solving the discretized diffusion equation on a rectangular lattice subject to Dirichlet boundary conditions. The C code in Listing 4.1 shows a simplified sequential version of the Jacobi method. The implementation consists of two loops that iterate over all

```
1 /* ... */
2 for(int j=1;j<N-1;j++){
3   for(int i=1,i<N-1;i++){
4     v[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]);
5   }
6 }
7 memcpy(u,v,N*N*sizeof(double));
8 /* ... */
```

Listing 4.1: Code snippet of a five point stencil

columns and rows of two dimensional arrays, `u` and `v` in the given example. Figure 4.5 illustrates the data access of the algorithm, when a new value is written to a data point of the array `v`, after reading four surrounding data points from the array `u`. Between two iterations, the values of both arrays must be replaced to avoid a mix of old and new values for the stencil calculation. As a first sequential optimization, the data transfer from the naïve version can be replaced by just exchanging the indices.

We compare two parallel programming models, shared memory and message passing with domain decomposition for the SCC which can be supported by *MetalSVM*.

The two dimensional space is statically divided into disjunct parts whereas access to neighboring cells, in this case stripes, is read only. Main difference between both version is the exchange of data at the boundaries and the coordination of parallel execution.

For the message passing version, only data elements at the boundaries, so called ghost cells, have to be exchanged. RCCE provides send and receive functions to handle explicit data transfer. The restriction of RCCE to blocking communication is appropriate for the given communication scenario.

For the shared-memory version, a valid solution for the coordination of threads is adding a barrier between iterations. This eliminates data dependencies between two iterations, because an overlapping of iterations is avoided, which could be a result of parallel execution without synchronization.

Figure 4.6 on Page 120 shows the results for the iterative Jacobi solver, which is discussed in this section with a problem size of 1024×512 elements of double precision floating point values.

The comparison of both versions of this simple application example illustrates the importance of the memory abstraction for a many-core system

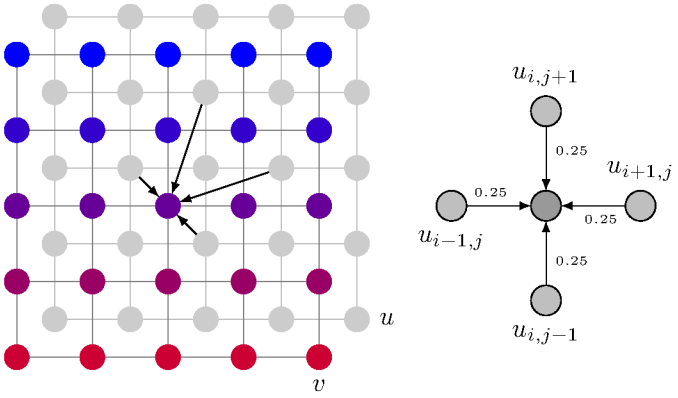


Figure 4.5: Stencil example: Heat Distribution Problem [Lan+12b]

without hardware support for cache coherency. For a small core count up to 16 cores, the shared memory version has a lower runtime, which means a higher application performance in terms of floating point operations, with a constant offset.

For a larger core-count with up to 48 cores, the local data, which means stripes of the matrices, fit into the Level 2 Cache of the SCC for the message passing variant. As a result, the runtime decreases significantly resulting in a super-linear scalability for the chosen application. Consequently, the message-passing variant outperforms the shared memory variant upon a certain core count. Adding more and more cores to the parallel application changes the ratio of local and remote access to the shared memory region, which means data from previous and current iteration, thereby that explicit data exchange of shared data and working on local data becomes more attractive.

This effect underlines the advantage of software controlled coherency, because system software can be adapted also during runtime to a given application. In addition to that, legacy shared-memory applications can be supported by many-core hardware without full-chip cache coherence.

The task of the SVM system is to manage the ownership of shared memory regions. In strong consistency mode, read and write access are performed exclusively to each page. This is contrary to lazy-release consistency, where replicated states can exist of each page as long as a synchronization point is

reached. For the investigated implementation of the Jacobi method, we used a strict alignment of matrix rows on page granularity (4 kB).

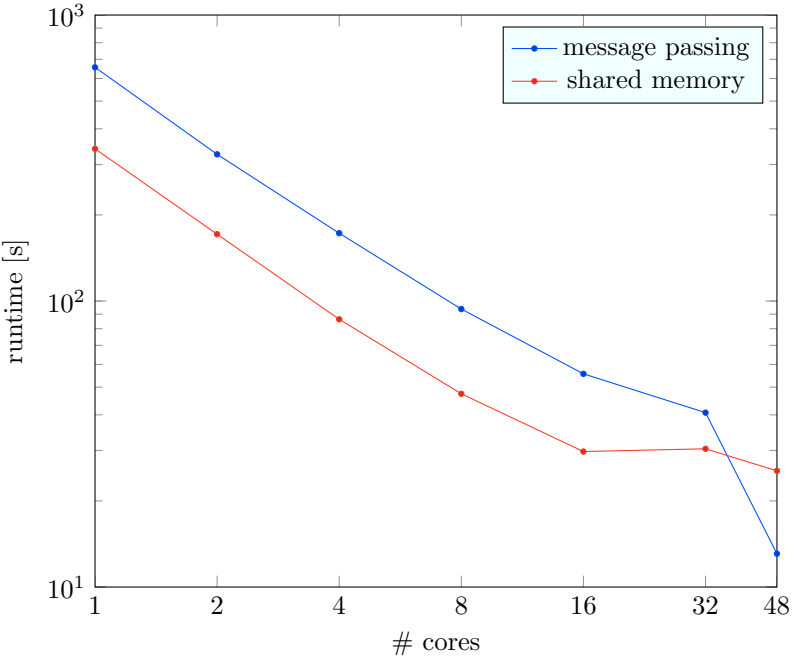


Figure 4.6: Jacobi strong scaling results

Experiments with strong and relaxed consistency models have been presented in the related work. These results are promising and demonstrate the low overhead of our SVM system [Lan+12b], but are beyond the scope of this dissertation. Due to the underlying memory access pattern, for the results which are presented here, minor differences are resulting from different consistency models.

The benchmark results of our bare-metal framework are limited to a core count of 48, which is the maximum core count of the SCC. Higher scalability of these tests to vSCC give no further insights because of its implementation details. The communication task, which is part of vSCC, is integrated to the host driver. This implementation is highly flexible, but the latency

is only predictable within certain bounds [RW14]. It introduces additional sources of OS jitter and as a result our bare-metal framework can not show its advantages.

In the next paragraph, results for a common synthetic application benchmark are discussed. We will use a combination of iRCCE and sccLinux for these measurements.

4.4.2 NPB

Mattson et al. [Mat+10] have ported the MPI versions of BT and LU pseudo programs from NAS Parallel Benchmark (NPB) suite [Bai+91] to native RCCE. Both applications mimic computation and communication of computational fluid dynamics applications. We have used these floating point intensive applications to benchmark our vSCC prototype and optimizations of iRCCE.

Figure 4.7 shows absolute performance-results of BT and LU in class C problem size ($162 \times 162 \times 162$). This problem size is suitable for the vSCC with 240 cores, as each core has a peak performance of 533 MFLOP/s.

Here, strong scaling behavior is analyzed which means that a fixed problem size is computed by an increasing number of cores. Despite the fact that absolute performance counts for the SCC are of minor relevance, the peak performance of our system with 225 cores is about 120 GFLOP/s in total. Moreover, the scalability of the new communication concept is important for different communication patterns, which are evaluated in the following. Due to the applied work distribution method and communication pattern, the application can only handle a number of processes, which is a square number. Consequently, 225 represents the maximum configuration for BT.

Similar restrictions apply to LU, where the number of cores has to be equal to a power of two, which results in a maximum of 128 processes. Peak performance for vSCC with 128 cores is about 68 GFLOP/s in total.

Two synthetic applications, also called real world kernel, have been investigated because of their different communication pattern. Figure 4.8 holds BT and LU results for the Intel SCC research system[7] for different problem sizes. The real world LU kernel uses many small messages for communication, in contrast to BT which uses few large messages for communication. As the

[7] core/mesh/memory frequency: 533 MHz/800 MHz/800 MHz

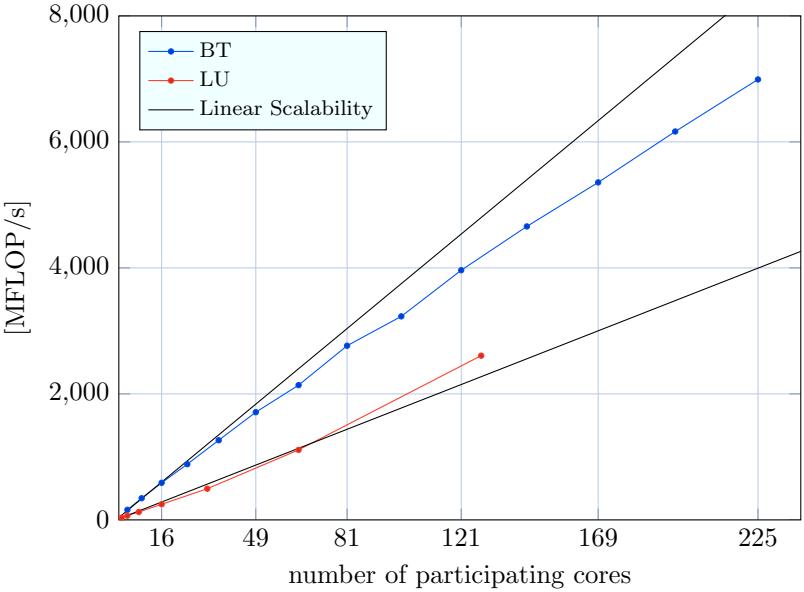
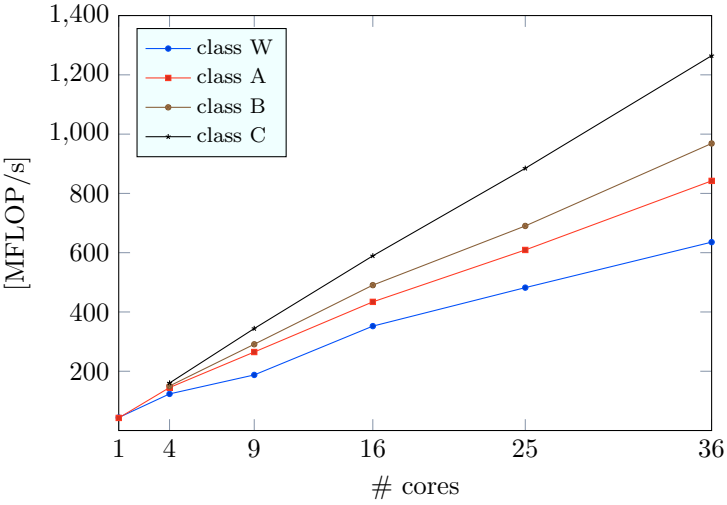


Figure 4.7: NPB BT and LU strong scaling performance results

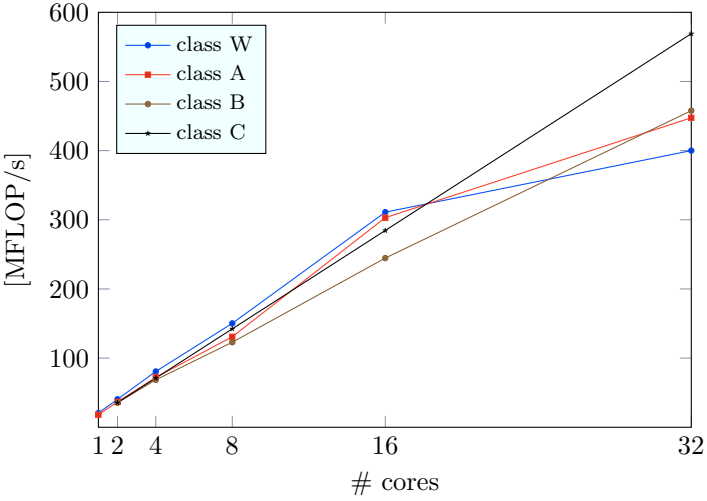
overall amount of communication grows $O(n^2)$ and the computation grows $O(n^3)$, we see an increasing performance for an increasing problem size.

All in all, an excellent scalability can be achieved with our optimizations, that additionally hide a high latency communication path in an effective way. These results demonstrate a connection between communication overhead and system performance for the investigated floating point intensive parallel programs with different communication patterns.

The NPB BT application uses a communication pattern which is based on locality. Message-passing applications need an appropriate placement of processes to nodes of a cluster to achieve a good performance, especially for heterogenous networks. In other words, applications should prefer connections with high throughput for communication. This basic assumption is also true for a cluster-on-a-chip processor such as the Intel SCC, where processes are mapped to cores.



(a) BT



(b) LU

Figure 4.8: NPB BT and LU for different problem sizes

Figure 4.9 visualizes the total communication amount exemplarily for a scenario with 64 cores of BT, to further detail the advantages of the vSCC prototype. Every filled square of the diagram indicates a communication between two ranks (x is sender and y receiver), whereas dark means high and light means low communication traffic. We had to instrument the RCCE reference implementation to record the amount of transferred data. This instrumentation was necessary because support for performance analysis tools is not intended. For inter-device communication such a functionality could be integrated to the host driver.

However, the obtained information gives further details to the communication pattern of the parallel application. We see a major part in communication of neighboring ranks.

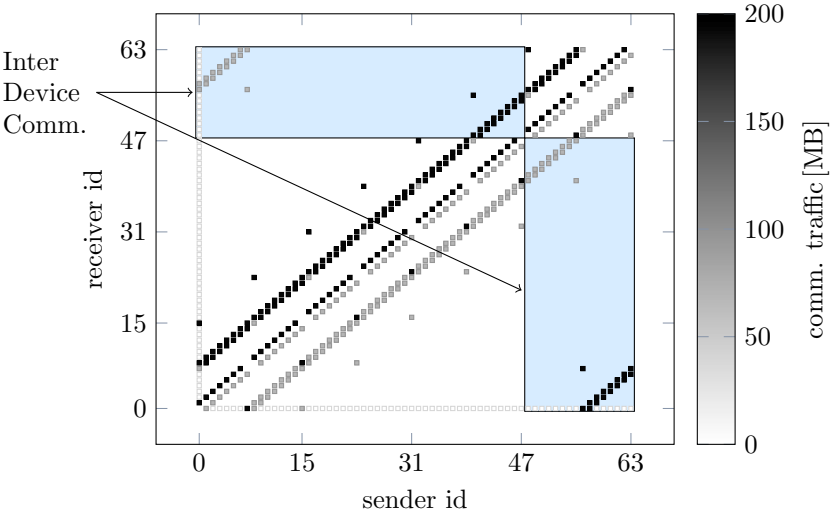


Figure 4.9: NPB BT (class C) communication traffic of 64 cores [Reb+15]

4.5 Conclusion

In the previous chapter of this work, we discussed alternative communication schemes for direct on-chip memory access and developed new communication protocols for this kind of many-core systems. Promising micro benchmark results on the Intel SCC were a result of our optimizations, which underlines the potential of predictable data movement for network-on-chip based systems. In order to estimate communication performance, effects to on-chip synchronization and contention to the new communication concept have been modeled in the previous chapter. This analysis creates foundation for analytical methods to derive the limits of a cluster-on-a-chip architecture, so that the communication model can be directly verified by using experimental hardware.

In this chapter, we present the MetalSVM project, which targets a flexible support of different memory models through the virtualization of a many-core memory system in the absence of hardware cache coherence.

In this context of evaluating system-software support for future many-core systems, the efficient implementation of a bare-metal hypervisor for a x86-based many-core architecture is described in detail. We compare message passing and shared memory version of a stencil code, to quantify the overhead of our approach and demonstrate the scalability. Results show that applications without sharing data and those with controlled contention can create great benefit for such an architecture.

Moreover, we were able to identify important attributes of a system with configurable cache coherency and hardware support for on-chip memory and propose extensions. Our extensions have been implemented in system software and lead to a full working prototype of a many-core processor with 240 tightly coupled cores. This prototype consists of a multi-chip package of SCCs, whereas its proprietary on-chip protocol is routed through the host system of the many-core devices. A requirement of this method was the connection of multiple SCCs to a single host.

Application as well as micro benchmark results have demonstrated the quality of our realization and verified the scalability of the new communication extensions. For the virtual extension, we are able to recover 25 % of effective on-chip communication throughput, despite the fact that the latency for both communication paths differs by 100 x for each on-chip packet. Moreover, our new research environment can be used as a guideline for similar future projects.

5

Conclusion

Many-core technology will be a prominent aspect of future architectures aimed to meet the rising demand for computational performance of scientific applications. Accordingly, the reduction of architectural overhead is important to design efficient and scalable microprocessors with thousands of cores per chip. Scaling current micro processor design will result in a growing chip complexity. Given the current state of the art, it is doubtful whether established concepts can address this challenge. Communication will become even more important, especially for a processor with thousands of cores within a given power budget per chip. Thus to achieve efficient communication, unconventional methods such as waiving memory abstraction should be taken into account.

This outlook represents a fundamental change for Intel Architecture, which has become prevalent in the field of high-performance computing. On the one hand, the shared-memory paradigm is well-established for programming generations of multi-core processors that commonly implement hardware-coherent caches. On the other hand, basic techniques for communication and synchronization, which also imply runtime systems and programming libraries, are based on this hardware support.

As a result of architectural changes, the implementation of new communication concepts raises a key challenge for system software in the many-core era. In this context, jitter has been identified as a source of negative influence on communication performance that can further harm scalability. As a consequence, predictability is an important attribute to develop for soft-

ware optimizations such as new communication concepts and sophisticated communication protocols for future processors.

The goal of this dissertation is to explore efficient communication for many-core systems in the absence of full-chip cache coherence. In order to fulfill this task, a new communication concept has been analyzed in detail for x86-based cores.

In the second chapter, hardware requirements are defined that are needed for a general implementation of new communication concepts. The basic structure of this work is bottom-up concerning many-core hardware, software abstraction and applications. The focus of the third chapter is a small communication layer for a cluster-on-a-chip architecture that includes communication protocols and synchronization aspects. The fourth chapter covers system software support for these architectures and presents application results that demonstrate the quality of our implementation.
The key contributions of this work can be summarized as follows:

- We have developed communication models for different communication patterns for software controlled on-chip memory of network-on-chip many-core architectures. This development includes the analysis and optimization of synchronization primitives based on centralized algorithms, distributed algorithms, and a combination of both. In addition, point-to-point-communication patterns as well as collective-communication patterns have been analyzed and optimized.

- We have designed and implemented a low-level message-passing interface that supports different communication schemes for hardware-distributed low-latency memory and sophisticated communication protocols. These have been optimized by the use of our communication models.

- We have evaluated extensions for x86-based many-core architectures to accelerate hardware support for message passing and have realized concepts for hiding memory latency. This enables further scalability of a cluster-on-a-chip, which has been verified by the realization of a virtual extension for network-on-chip based processors.

In the next section, the methods that have been used to achieve these goals are summarized. The subsequent section concludes the results section.

5.1 Methods

The SCC experimental architecture is the first example of an Intel microprocessor that provides shared but non-coherent on-chip memory. It represents a hybrid many-core architecture, according to a common classification of parallel programming paradigms in terms of pure shared-memory or message-passing support. Consequently, both models can be enabled. This fact makes it possible to explore different consistency models for data replication and sophisticated communication protocols.

To achieve the lowest overhead for a detailed analysis of different communication schemes and a high predictability of software controlled communication, we have developed a bare metal framework for the Intel SCC. Regarding network-on-chip-based architectures, memory contention and network congestion can limit scalability. Memory location becomes essential if access is controlled in an explicit way. Classic optimization techniques used to overcome this issue are discussed. For instance, a back off with feedback is successfully used to realize central synchronization points that outperform classic synchronization algorithms on hardware-distributed shared memory. This detailed analysis shows that a careful design of synchronization methods and their access pattern is essential to effective communication.

In Chapter 4, system-software support and the results of synthetic application benchmarks are discussed. This support includes extensions to new communication concepts of explicit and on-chip data movement. Specifically, the communication-offload approach is implemented to create a tightly coupled multi-chip many-core processor, In this way, the flexibility of the SCC research system is used to virtually expand the on-chip interconnect.

The realization of these extensions creates with vSCC an innovative research environment for network-on-chip-based many-core processors. One major achievement of vSCC is a virtual extension of the on-chip interconnect, which results in a full working prototype with 120 tiles and 240 cores. Our setup demonstrates the scalability of Intel SCC's cluster-on-a-chip architecture. Moreover, we have presented the design and implementation of a virtual direct memory access (vDMA) controller, which enables a reevaluation of the SCC's communication concept. This additional support has been integrated to the low-level communication library iRCCE.

Explicit access to hardware-distributed on-chip memory enables a certain flexibility. In Chapter 3, the design and implementation of a low-latency communication-and-synchronization layer for future many-core systems is pre-

sented. Mapping a message-passing interface to a one-sided communication system represents a classic approach. As a result, a two-sided communication protocol uses on-chip memory as an internal buffer. An important task regarding such a protocol is to qualify dynamic allocation schemes for on-chip memory locations. Here, the new communication concept provides the flexibility of remote-put, remote-get or all-local communication. This includes the analysis of different communication schemes, which become an option because of the low-latency remote-memory access.

The realization of fast and hardware-based low-level synchronization primitives is discussed in the second part of this chapter. Synchronization algorithms based on atomic operations to shared-memory locations are compared to classic flag-based variants. As a result of modeling communication, we have analytically derived optimizations for implementations of barrier-and-lock constructs for the Intel SCC. Required parameters can be calculated with low-level timing analysis, which has also been used to get a better understanding of the new communication concepts.

In Chapter 2, basic attributes of a tiled x86-based many-core architecture are summarized and related to this work. Moreover, the realization of a hardware abstraction is discussed and a communication model is developed for software-controlled on-chip memory.

5.2 Results

In summary, implementation of the SCC research vehicle has been of great value for many-core software research. Besides the simulation and emulation of future processor architectures, new insights for low-level software engineering are resulting from such experimental hardware.

One important attribute of extensions is flexible mapping of core addresses to system addresses for the research chip, which is based on Intel Architecture. The combination with a low-memory abstraction results in explicit on-chip memory access and enables analysis of different communication and synchronization schemes. Our extensions lead to a full working prototype with 240 tightly connected cores based on x86. Consequently, existing low level communication libraries can be used without changes to the interface to realize a virtual on-chip communication between tiles that are not located on a single chip. Only minor modifications to the hardware abstraction layer – such

as mapping the on-chip memory of additional tiles – were needed to address more than 48 cores.

In addition to that, with iRCCE we have proposed communication-and-synchronization extensions to a basic communication interface. Rather than being limited to blocking communication, our first set of extensions enables non-blocking communication without the need for additional operating-system support. Further extensions enable parallel applications to use hierarchy-aware communication patterns and to achieve best performance – especially with respect to collective communication – by exposing locality to the programmer. These extensions are part of iRCCE with vSCC support.

A third set of extensions covers the synchronization support of a light-weight communication environment. Hardware-supported synchronization methods have been analyzed in detail to further increase the functionality of iRCCE. The scalability of synchronization methods based on hardware support, such as atomic operation and software controlled on-chip memory, can be significantly increased.

Our installation uncovers scalability issues of a static and of a symmetric allocation scheme for software-controlled on-chip memory. Consequently, we develop alternatives and analytically derived optimal patterns for explicit on-chip memory access. Our study shows how synchronization primitives can be exemplarily realized for the SCC by means of hardware features such as atomic operations. In this context, barrier algorithms for shared memory have been ported and optimized for the synchronization support of the SCC.

Busy-wait synchronization methods lead to promising results if the characteristics of the underlying communication interconnect is taken into account. An evaluation of the hardware synchronization support of a cluster-on-a-chip architecture is presented in the second chapter of this work. This evaluation verifies issues common to busy-wait synchronization techniques on future many-core architectures regarding fairness and scalability. It has been shown that our approach is advantageous especially in highly contended situations.

The experiments presented here show massive improvements through common optimizations such as an exponential back-off or the use of multiple hardware synchronization primitives. Scalability of a single synchronization point can be increased by structuring the physical resources in a hierarchical way. Obviously, this is a trade-off between an optimal scalability, and allocation of synchronization registers as resources are scarce. If the implementation of mechanisms is carefully chosen, the use of hardware support for synchronization constructs, such as a barrier, leads to promising results. Especially

for highly-contended locks, an exponential back-off algorithm can lead to an excellent scalability compared to a straight forward implementation.

This study addresses the amount of contention control that is needed for NoC many-core architectures that run specific applications. Moreover, we develop concepts to explore the limits for different communication patterns on this kind of architecture.

Our many-core communication environment follows a layered approach. As a result, specific components such as the hardware-abstraction layer have to be re-implemented to support future architectures. An example of a future many-core architecture, which shares characteristic attributes, is Intel's Knights Landing (KNL) with 72 cores and a 2D mesh. A main difference between KNL and the experimental SCC processor is the basic core architecture. Another difference is that KNL implements full-chip cache coherence but provides stacked memory, called MCDRAM, with high bandwidth and low latency [Mor]. Access to this kind of memory will be configurable, such that it can be controlled by software. A configuration which may be changed at reboot of the system, comparable to the cluster-on-die (CoD) technology, is available for processors with more than 10 cores based on Haswell microarchitecture. In fact, the cluster-on-die feature is used to term multiple NUMA domains on a single processor die, in contrast to a cluster-on-a-chip that includes a loss of full-chip cache-coherence. Nevertheless, it is an example of the importance of memory access to increasing chip complexity.

In this dissertation, communication models have been developed for abstract many-core architectures that follow the NoC paradigm. As a result, communication methods which are based on these models will be applicable for future many-core architectures. To enable a reuse of methods, we specify a small set of parameters that can be determined by simple micro-benchmarks and can thus be easily adjusted.

List of Abbreviations

AIR Atomic Increment Register 30
API application programming interface 53
APIC Advanced Programmable Interrupt Controller 113
CFD computational fluid dynamics 121
CISC Complex Instruction Set Computer 12
CoC cluster-on-a-chip 5
CoD cluster-on-die 132
COTS components off-the-shelf 56
CPU central processing unit 15
DMA direct memory access 51
DSM distributed shared memory 11
FIFO First In, First Out 41
FPGA field-programmable gate array 22
FSB front-side bus 19
GDT Global Descriptor Table 111
GIC Global Interrupt Controller 113
HPC high-performance computing 50
IPI inter-processor interrupt 113
KNL Knights Landing 20
L1 Level 1 Cache 22
L2 Level 2 Cache 23
LLC Last Level Cache 17
LMB Local Memory Buffer 23

lprg local-put, remote-get 71
LUT Lookup Table 24
MARC Many-core Applications Research Community 20
MC memory controller 21
MCPC Management Console PC 21
MIC Many Integrated Core 19
MIU Mesh Interconnect Unit 22
MMU Memory Management Unit 26
MPB Message Passing Buffer 23
MPBT Message Passing Buffer Tagged 27
MPI Message Passing Interface 52
NoC network-on-chip 12
NPB NAS Parallel Benchmark 121
NUMA non-uniform memory access 17
OS operating system 100
PCIe Peripheral Component Interconnect Express 21
QPI QuickPath Interconnect 19
RCCE Rock Creek Communication Environment 48
RISC Reduced Instruction Set Computer 12
rplg remote-put, local-get 72
SCC Single-chip Cloud Computer 20
SCI Scalable Coherent Interface 106
SCIF Symmetric Communications InterFace 51
SIF System Interface 22
SMP symmetric multiprocessor 17
SPMD Single Program Multiple Data 52
SVM Shared Virtual Memory 4
TLB Translation Look aside Buffer 26
TSR Test and Set Register 30

UE Unit of Execution 53
vDMA virtual direct memory access 129
vSCC virtual Single-chip Cloud Computer 100

References

[AG96] S. V. Adve and K. Gharachorloo. “Shared memory consistency models: a tutorial”. In: *Computer* 29.12 (1996), pp. 66–76.

[AJ89] N. S. Arenstorf and H. F. Jordan. “Comparing barrier algorithms”. In: *Parallel Computing* 12.2 (1989), pp. 157–170. DOI: 10.1016/0167-8191(89)90050-1.

[And90] T. E. Anderson. “The performance of spin lock alternatives for shared-memory multiprocessors”. In: *IEEE Transactions on Parallel and Distributed Systems* 1.1 (Jan. 1990), pp. 6–16. DOI: 10.1109/71.80120.

[Bai+91] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. “The NAS Parallel Benchmarks”. In: *Intl. Journal of Supercomputer Applications* 5.3 (1991), pp. 66–73.

[Ban+02] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. “Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems”. In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*. CODES ’02. Estes Park, Colorado, 2002, pp. 73–78. ISBN: 1-58113-542-4. DOI: 10.1145/774789.774805.

[Bar+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. “Xen and the art of virtualization”. In: *SIGOPS Operating Systems Review* 37.5 (2003), pp. 164–177.

[Bau+09] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. “The Multikernel: A New OS Architecture for Scalable Multicore Systems”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP ’09. Big Sky, Montana,

USA, 2009, pp. 29–44. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629579.

[Bel+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect". In: *IEEE International Solid-State Circuits Conference, ISSCC 2008. Digest of Technical Papers*. Feb. 2008, pp. 88–598. DOI: 10.1109/ISSCC.2008.4523070.

[Bon02] D. Bonachea. *GASNet Specification*. University of California at Berkeley. 2002.

[Bor07] S. Borkar. "Thousand Core Chips: A Technology Perspective". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC'07. San Diego, California, 2007, pp. 746–749. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278667.

[Boy+08] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. "Corey: An Operating System for Many Cores". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California, 2008, pp. 43–57. URL: http://dl.acm.org/citation.cfm?id=1855741.1855745.

[Bro86] E. D. Brooks. "The butterfly barrier". In: *International Journal of Parallel Programming* 15.4 (1986), pp. 295–307. DOI: 10.1007/BF01407877.

[BS93] W. J. Bolosky and M. L. Scott. "False Sharing and Its Effect on Shared Memory Performance". In: *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems*. Vol. 4. Sedms'93. San Diego, California, 1993. URL: http://dl.acm.org/citation.cfm?id=1295480.1295483.

[Bul99] J. M. Bull. "Measuring Synchronisation and Scheduling Overheads in OpenMP". In: *Proceedings of the 1st European Workshop on OpenMP*. Lund, Sweden, Oct. 1999, pp. 99–105.

[CH09] M. Chapman and G. Heiser. "vNUMA: A Virtual Shared-memory Multiprocessor". In: *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*. USENIX'09. San Diego, California, 2009, pp. 349–362. URL: http://dl.acm.org/citation.cfm?id=1855807.1855809.

[Cha+05] D. Chandra, F. Guo, S. Kim, and Y. Solihin. "Predicting inter-thread cache contention on a chip multi-processor architecture". In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. HPCA-11. Feb. 2005, pp. 340–351. DOI: 10.1109/HPCA.2005.27.

[Cha10] E. Chan. *RCCE_comm: a Collective Library for the Intel Single-chip Cloud Computer*. Sept. 2010. URL: https://communities.intel.com/servlet/JiveServlet/previewBody/5663-102-1-8763/RCCE_comm_02.pdf (visited on 01/20/2015).

[CJvP07] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming. Portable Shared Memory Parallel Programming*. Vol. 10. The MIT Press, 2007. ISBN: 978-0-26-253302-7.

[Cla+11] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. "Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor". In: *Proceedings of the 2011 International Conference on High Performance Computing and Simulation*. HPCS. Istanbul, Turkey, July 2011, pp. 525–532. DOI: 10.1109/HPCSim.2011.5999870.

[Cla+12] C. Clauss, S. Pickartz, S. Lankes, and T. Bemmerl. "Hierarchy-Aware Message-Passing in the Upcoming Many-Core Era". In: *Grid Computing – Technology and Applications, Widespread Coverage and New Horizons*. 2012, pp. 151–178. ISBN: 978-953-51-0604-3. URL: http://www.intechopen.com/articles/show/title/hierarchy-aware-message-passing-in-the-upcoming-many-core-era.

[Cla+13a] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. "New system software for parallel programming models on the Intel SCC many-core processor". In: *Concurrency and Computation: Practice and Experience* (2013). DOI: 10.1002/cpe.3033.

[Cla+13b] C. Clauss, S. Lankes, P. Reble, J. Galowicz, S. Pickartz, and T. Bemmerl. *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer*. Users Guide and API Manual. Version 2.0. RWTH Aachen University, Mar. 2013. URL: https://communities.intel.com/docs/DOC-6003 (visited on 01/20/2015).

[CRK11] I. A. Comprés Ureña, M. Riepen, and M. Konow. "RCKMPI – Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC)". In: *Recent Advances in the Message Passing Interface*. Vol. 6960. Lecture Notes in Computer Science. 2011, pp. 208–217. ISBN: 978-3-642-24448-3. DOI: 10.1007/978-3-642-24449-0_24.

[CS06] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors". In: *SIGARCH Comput. Archit. News* 34.2 (May 2006), pp. 264–276. DOI: 10.1145/1150019.1136509.

[Cul+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. "LogP: Towards a Realistic Model of Parallel Computation". In: *SIGPLAN Not.* 28.7 (July 1993), pp. 1–12. DOI: 10.1145/173284.155333.

[Dal92] W. Dally. "Virtual-channel flow control". In: *IEEE Transactions on Parallel and Distributed Systems* 3.2 (Mar. 1992), pp. 194–205. DOI: 10.1109/71.127260.

[Day+14] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh. "SCORPIO: A 36-Core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC with In-Network Ordering". In: *Proceedings of the 41st International Symposium on Computer Architecture, ISCA 2014*. Minneapolis, MN, USA, July 2014. DOI: 978-1-4799-4394-4/14.

[Don+11] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A.

Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Müller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. "The International Exascale Software Project roadmap". In: *International Journal of High Performance Computing Applications* 25.1 (2011), pp. 3–60. DOI: 10.1177/1094342010391989.

[Dre07] U. Drepper. *What every programmer should know about memory*. Nov. 2007. URL: http://lwn.net/Articles/250967/ (visited on 01/20/2015).

[DSB99] M. Dormanns, K. Scholtyssik, and T. Bemmerl. "A Shared Memory Programming Interface for SCI Clusters". In: *SCI: Scalable Coherent Interface*. Vol. 1734. Lecture Notes in Computer Science. 1999, pp. 281–290. ISBN: 978-3-540-66696-7. DOI: 10.1007/10704208_22.

[DT04] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004. ISBN: 0-12-200751-4.

[Dun01] A. Dunkels. *Design and Implementation of the lwIP TCP/IP Stack*. Swedish Institute of Computer Science, 2001.

[Esm+11] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. "Dark silicon and the end of multicore scaling". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA. June 2011, pp. 365–376.

[Esm+13] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. "Power Challenges May End the Multicore Era". In: *Commun. ACM* 56.2 (Feb. 2013), pp. 93–102. DOI: 10.1145/2408776.2408797.

[FAV97] M. I. Frank, A. Agarwal, and M. K. Vernon. "LoPC: Modeling Contention in Parallel Algorithms". In: *SIGPLAN Not.* 32.7 (June 1997), pp. 276–287. DOI: 10.1145/263767.263803.

[FC08] C. Fensch and M. Cintra. “An OS-based alternative to full hardware coherence on tiled CMPs”. In: *Proceedings of 14th International Symposium on High Performance Computer Architecture*. HPCA ’08. Feb. 2008, pp. 355–366. DOI: 10.1109/HPCA.2008.4658652.

[Gam+99] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. “Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System”. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI ’99. New Orleans, Louisiana, USA, 1999, pp. 87–100. ISBN: 1-880446-39-1. URL: http://dl.acm.org/citation.cfm?id=296806.296814.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. Cambridge, MA, USA: MIT Press, 1994. ISBN: 0-262-57104-8.

[Gol73] R. P. Goldberg. “Architectural Principles for Virtual Computer Systems”. PhD thesis. Harvard University, 1973.

[gop12] *Interview with James Reinders*. Sept. 2012. URL: http://goparallel.sourceforge.net/ask-james-reinders-multicore-vs-manycore/ (visited on 01/20/2015).

[Gri+11] M. Gries, U. Hoffmann, M. Konow, and M. Riepen. “SCC: A Flexible Architecture for Many-Core Platform Research”. In: *Computing in Science & Engineering* 13.6 (Nov. 2011), pp. 79–83. DOI: 10.1109/MCSE.2011.109.

[GT90] G. Graunke and S. Thakkar. “Synchronization algorithms for shared-memory multiprocessors”. In: *Computer* 23.6 (June 1990), pp. 60–69. DOI: 10.1109/2.55501.

[HM08] M. Hill and M. Marty. “Amdahl’s Law in the Multicore Era”. In: *Computer* 41.7 (July 2008), pp. 33–38. DOI: 10.1109/MC.2008.209.

[Hoc94] R. W. Hockney. “The Communication Challenge for MPP: Intel Paragon and Meiko CS-2”. In: *Parallel Computing* 20.3 (Mar. 1994), pp. 389–398. DOI: 10.1016/S0167-8191(06)80021-9.

[Hoe+05] T. Hoefler, L. Cerquetti, T. Mehlan, F. Mietke, and W. Rehm. "A Practical Approach to the Rating of Barrier Algorithms Using the LogP Model and Open MPI". In: *41st International Conference on Parallel Processing Workshops*. 2005, pp. 562–569. ISBN: 0-7695-2381-1. DOI: 10.1109/ICPPW.2005.14.

[Hof05] H. Hofstee. "Power efficient processor architecture and the cell processor". In: *11th International Symposium on High-Performance Computer Architecture*. HPCA-11. Feb. 2005, pp. 258–262. DOI: 10.1109/HPCA.2005.26.

[How+11] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. van der Wijngaart. "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling". In: *IEEE Journal of Solid-State Circuits* 46.1 (Jan. 2011), pp. 173–183. DOI: 10.1109/JSSC.2010.2079450.

[HP12] J. L. Hennessy and D. A. Patterson. *Computer Architecture*. 5th ed. A quantitative approach. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[HS08] J. P. Hoeflinger and B. R. D. Supinski. "The OpenMP memory model". In: *OpenMP Shared Memory Parallel Programming*. Lecture Notes in Computer Science. Eugene, OR, USA, 2008, pp. 167–177. ISBN: 978-3-540-68554-8. DOI: 10.1007/978-3-540-68555-5_14.

[HW11] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. New York, NY, USA: CRC Press, 2011. ISBN: 978-1-4398-1192-4.

[Intel06] *From a Few Cores to Many: A Tera-scale Computing Research Overview*. White Paper. Intel Corporation, 2006. URL: http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-tera-scale-research-paper.pdf (visited on 01/20/2015).

[Intel07] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. Aug. 2007.

[Intel95] *Pentium Processor Family Developer's Manual*. Architecture and Programming Manual. Intel Corporation. 1995. ISBN: 1-55512-195-0.

[Juc+04] G. Juckeland, S. Börner, M. Kluge, S. Kölling, W. E. Nagel, S. Pflüger, H. Röding, S. Seidl, T. William, and R. Wloch. "BenchIT - Performance measurement and comparison for scientific applications". In: *Advances in Parallel Computing* 13 (2004), pp. 501–508.

[KBV00] T. Kielmann, H. Bal, and K. Verstoep. "Fast Measurement of LogP Parameters for Message Passing Platforms". In: *Parallel and Distributed Processing*. Vol. 1800. Lecture Notes in Computer Science. 2000, pp. 1176–1183. ISBN: 978-3-540-67442-9. DOI: 10.1007/3-540-45591-4_162.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. "Lazy Release Consistency for Software Distributed Shared Memory". In: *SIGARCH Comput. Archit. News* 20.2 (Apr. 1992), pp. 13–21. DOI: 10.1145/146628.139676.

[Kel+94] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems". In: *Proceedings of the USENIX Winter 1994 Technical Conference*. WTEC'94. San Francisco, California, 1994.

[Kiv+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. "KVM: the Linux virtual machine monitor". In: *Proceedings of the Linux Symposium*. Vol. 1. 2007, pp. 225–230.

[KRS88] C. P. Kruskal, L. Rudolph, and M. Snir. "Efficient Synchronization of Multiprocessors with Shared Memory". In: *ACM Trans. Program. Lang. Syst.* 10.4 (Oct. 1988), pp. 579–601. DOI: 10.1145/48022.48024.

[KS02] G. B. Kandiraju and A. Sivasubramaniam. "Going the Distance for TLB Prefetching: An Application-driven Study". In: *SIGARCH Comput. Archit. News* 30.2 (May 2002), pp. 195–206. DOI: 10.1145/545214.545237.

[KSM05] B.-J. Kwak, N.-O. Song, and M. Miller. "Performance analysis of exponential backoff". In: *IEEE/ACM Transactions on Networking* 13.2 (Apr. 2005), pp. 343–355. DOI: 10.1109/TNET.2005.845533.

[Kum+11] R. Kumar, T. G. Mattson, G. Pokam, and R. van der Wijngaart. "The Case for Message Passing on Many-Core Chips". In: *Multiprocessor System-on-Chip*. 2011, pp. 115–123. ISBN: 978-1-4419-6459-5. DOI: 10.1007/978-1-4419-6460-1_5.

[Lam87] L. Lamport. "A Fast Mutual Exclusion Algorithm". In: *ACM Trans. Comput. Syst.* 5.1 (Jan. 1987), pp. 1–11. DOI: 10.1145/7351.7352.

[Lan+12a] S. Lankes, P. Reble, C. Clauss, and O. Sinnen. "The Path to MetalSVM: Shared Virtual Memory for the SCC". In: *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*. 55. Potsdam, Germany, Feb. 2012, pp. 7–14. ISBN: 978-3-86956-169-1.

[Lan+12b] S. Lankes, P. Reble, O. Sinnen, and C. Clauss. "Revisiting Shared Virtual Memory Systems for Non-coherent Memory-coupled Cores". In: *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM'12. New Orleans, Louisiana, 2012, pp. 45–54. ISBN: 978-1-4503-1211-0. DOI: 10.1145/2141702.2141708.

[Lan10] S. Lankes. *First Experiences with SCC and a Comparison with Established Architectures*. Invited talk at the 1st MARC Symposium. Braunschweig, Germany, Nov. 2010.

[LH89] K. Li and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems". In: *ACM Trans. Comput. Syst.* 7.4 (Nov. 1989), pp. 321–359. DOI: 10.1145/75104.75105.

[Lu+07] S.-L. L. Lu, P. Yiannacouras, R. Kassa, M. Konow, and T. Suh. "An FPGA-based Pentium in a complete desktop system". In: *Proceedings of the 2007 ACM/SIGDA 15th international symposium*. New York, New York, USA, 2007, pp. 53–59.

[Lub90] B. D. Lubachevsky. "Synchronization barrier and related tools for shared memory parallel programming". In: *International Journal of Parallel Programming* 19.3 (1990), pp. 225–250. DOI: 10.1007/BF01407956.

[MARC] Intel Corporation. *Many-core Applications Research Community (MARC)*. URL: http://communities.intel.com/community/marc (visited on 01/20/2015).

[Mat+10] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. "The 48-core SCC Processor: The Programmer's View". In: *Proceedings of the 2010 ACM/IEEE Conference on Supercomputing*. SC'10. New Orleans, LA, USA, Nov. 2010.

[Mat11] T. Mattson. *The Future of Many Core Computing: A tale of two processors*. Invited talk at TUM. Munich, Germany: Intel Labs, June 2011.

[MC11] *Multicore Communications API (MCAPI) Specification*. The Multicore Association. Mar. 2011.

[McC95] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

[McK+01] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. "Read-copy update". In: *AUUG Conference Proceedings*. Sydney, Australia, Sept. 2001, pp. 175–184. ISBN: 0-9577532-2-5.

[MF01] C. Moritz and M. Frank. "LoGPG: Modeling network contention in message-passing programs". In: *IEEE Transactions on Parallel and Distributed Systems* 12.4 (Apr. 2001), pp. 404–415. DOI: 10.1109/71.920589.

[MHS12] M. M. K. Martin, M. D. Hill, and D. J. Sorin. "Why On-chip Cache Coherence is Here to Stay". In: *Commun. ACM* 55.7 (July 2012), pp. 78–89. DOI: 10.1145/2209249.2209269.

[Moo65] G. E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38 (Apr. 1965).

[Mor] T. P. Morgan. *More Knights Landing Xeon Phi Secrets Unveiled*. The Platform. URL: http://www.theplatform.net/2015/03/25/more-knights-landing-xeon-phi-secrets-unveiled/ (visited on 05/04/2015).

[MP98] D. Mentré and T. Priol. "NOA: A Shared Virtual Memory over a SCI cluster". In: *Proceedings of SCI-Europe'98, a conference stream of EMMSEC'98*. 1998, pp. 43–50.

[MPI94] Message Passing Interface Forum. "MPI: A Message Passing Interface Standard". In: *International Journal of Supercomputer Applications* 8.3/4 (1994), pp. 159–416.

[MS91] J. M. Mellor-Crummey and M. L. Scott. "Algorithms for Scalable Synchronization on Shared-memory Multiprocessors". In: *ACM Trans. Comput. Syst.* 9.1 (Feb. 1991), pp. 21–65. DOI: 10.1145/103727.103729.

[MTL78] R. McGill, J. W. Tukey, and W. A. Larsen. "Variations of box plots". In: *The American Statistician* 32.1 (1978), pp. 12–16.

[MvW11] T. Mattson and R. van der Wijngaart. *RCCE: a Small Library for Many-Core Communication*. Software Version 2.0. Intel Corporation. Jan. 2011. URL: https://communities.intel.com/docs/DOC-5628 (visited on 01/20/2015).

[Nie+06] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. "High performance remote memory access communication: The ARMCI approach". In: *International Journal of High Performance Computing Applications* 20.2 (2006), pp. 233–253.

[Nig+09] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. "Helios: Heterogeneous Multiprocessing with Satellite Kernels". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA, 2009, pp. 221–234. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629597.

[NM93] L. Ni and P. McKinley. "A survey of wormhole routing techniques in direct networks". In: *Computer* 26.2 (Feb. 1993), pp. 62–76. DOI: 10.1109/2.191995.

[NO97] B. Nayfeh and K. Olukotun. "A single-chip multiprocessor". In: *Computer* 30.9 (Sept. 1997), pp. 79–85. DOI: 10.1109/2.612253.

[OMP13] *OpenMP API*. Specification. Version 4.0. OpenMP Architecture Review Board, July 2013.

[OSS09] S. Owens, S. Sarkar, and P. Sewell. "A Better x86 Memory Model: x86-TSO". In: *Theorem Proving in Higher Order Logics*. Vol. 5674. Lecture Notes in Computer Science. 2009, pp. 391–407. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_27.

[Owe+07] J. D. Owens, W. J. Dally, R. Ho, D. N. J. Jayasimha, S. W. Keckler, and L.-S. Peh. "Research Challenges for On-Chip Interconnection Networks". In: *IEEE Micro* 27.5 (2007), pp. 96–108. DOI: 10.1109/MM.2007.4378787.

[PLK09] S. Pakin, M. Lang, and D. K. Kerbyson. "The reverse-acceleration model for programming petascale hybrid systems". In: *IBM Journal of Research and Development* 53.5 (Sept. 2009), pp. 721–735. DOI: 10.1147/JRD.2009.5429074.

[Poo+11] S. W. Poole, O. Hernandez, J. A. Kuehn, G. M. Shipman, A. Curtis, and K. Feind. "OpenSHMEM - Toward a Unified RMA Model". In: *Encyclopedia of Parallel Computing*. 2011, pp. 1379–1391. ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4_490.

[Pot+13] S. Potluri, A. Venkatesh, D. Bureddy, K. Kandalla, and D. Panda. "Efficient Intra-node Communication on Intel-MIC Clusters". In: *Proceedings of 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid. May 2013, pp. 128–135. DOI: 10.1109/CCGrid.2013.86.

[QPI09] Intel Corporation. *An Introduction to the Intel QuickPath Interconnect*. Jan. 2009. URL: http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html (visited on 01/20/2015).

[RCL13] P. Reble, C. Clauss, and S. Lankes. "One-sided Communication and Synchronization for Non-coherent Memory-coupled Cores". In: *Proceedings of the 2013 International Conference on High Performance Computing and Simulation*. HPCS. July 2013, pp. 390–397. DOI: 10.1109/HPCSim.2013.6641445.

[Reb+11] P. Reble, S. Lankes, C. Clauss, and T. Bemmerl. "A Fast Inter-Kernel Communication and Synchronization Layer for MetalSVM". In: *Proceedings of the 3rd MARC Symposium*. Ettlingen, Germany, July 2011, pp. 19–23. ISBN: 978-3-86644-717-2. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937.

[Reb+12a] P. Reble, C. Clauss, M. Riepen, S. Lankes, and T. Bemmerl. "Connecting the Cloud: Transparent and Flexible Communication for a Cluster of Intel SCCs". In: *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*. Aachen, Germany, Dec. 2012, pp. 13–19. ISBN: 978-3-00-039545-1. URL: http://darwin.bth.rwth-aachen.de/opus3/volltexte/2012/4383/.

[Reb+12b] P. Reble, J. Galowicz, S. Lankes, and T. Bemmerl. "Efficient Implementation of the bare-metal hypervisor MetalSVM for the SCC". In: *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*. Toulouse, France, July 2012, pp. 59–65. ISBN: 978-2-7257-0016-8. URL: http://hal.archives-ouvertes.fr/hal-00719037.

[Reb+12c] P. Reble, S. Lankes, F. Zeitz, and T. Bemmerl. "Evaluation of Hardware Synchronization Support of the SCC Many-Core Processor". In: *4th USENIX Workshop on Hot Topics in Parallelism*. HOTPAR'12. poster paper. Berkeley, CA, USA, June 2012. URL: https://www.usenix.org/system/files/conference/hotpar12/hotpar12-final9.pdf.

[Reb+15] P. Reble, S. Lankes, F. Fischer, and M. S. Müller. "Effective Communication for a System of Cluster-on-a-Chip Processors". In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM'15. San Francisco, California, 2015. ISBN: 978-1-4503-3404-4. DOI: 10.1145/2712386.2712393.

[Reg02] J. Regehr. "Inferring Scheduling Behavior with Hourglass". In: *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA, June 2002, pp. 143–156. ISBN: 1-880446-01-4. URL: http://dl.acm.org/citation.cfm?id=647056.715933.

[RH13] S. Ramos and T. Hoefler. "Modeling Communication in Cache-coherent SMP Systems: A Case-study with Xeon Phi". In: *Proceedings of the 22rd International Symposium on High-performance Parallel and Distributed Computing*. HPDC '13. New York, New York, USA, 2013, pp. 97–108. ISBN: 978-1-4503-1910-2. DOI: 10.1145/2462902.2462916.

[Rot11] R. Rotta. "On Efficient Message Passing on the Intel SCC". In: *Proceedings of the 3rd MARC Symposium*. Ettlingen, Germany, July 2011, pp. 53–58. ISBN: 978-3-86644-717-2.

[Rus07] R. Russell. "lguest: Implementing the little Linux hypervisor". In: *Proceedings of the Linux Symposium (OLS'07)*. Ottawa, Canada, 2007, pp. 173–177. URL: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-173-178.pdf.

[Rus08] R. Russell. "Virtio: Towards a De-facto Standard for Virtual I/O Devices". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 95–103. DOI: 10.1145/1400097.1400108.

[RW14] P. Reble and G. Wassen. "Towards Predictability of Operating System Supported Communication for PCIe Based Clusters". In: *Euro-Par 2013: Parallel Processing Workshops*. Vol. 8374. Lecture Notes in Computer Science. 2014, pp. 833–842. ISBN: 978-3-642-54419-4. DOI: 10.1007/978-3-642-54420-0_81.

[SBdS08] M. Schulz, G. Bronevetsky, and B. R. de Supinski. "On the Performance of Transparent MPI Piggyback Messages". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Vol. 5205. Lecture Notes in Computer Science. 2008, pp. 194–201. ISBN: 978-3-540-87474-4. DOI: 10.1007/978-3-540-87475-1_28.

[SCC10] *SCC External Architecture Specification (EAS)*. Revision 1.1. Intel Corporation. Nov. 2010. URL: http://communities.intel.com/docs/DOC-5852 (visited on 01/20/2015).

[SCC11] *The sccKit 1.4.x User's Guide*. Revision 1.1. Intel Labs. Nov. 2011. URL: https://communities.intel.com/servlet/JiveServlet/previewBody/6241-102-3-22263/sccKit14x_UsersGuide_Parts1-7.pdf (visited on 11/20/2014).

[SCC12] *The SCC Programmer's Guide*. Revision 1.0. Intel Corporation. Jan. 2012. URL: https://communities.intel.com/servlet/JiveServlet/previewBody/5684-102-8-22523/SCCProgrammersGuide.pdf (visited on 01/20/2015).

[Sch+10] D. Schmidl, C. Terboven, A. Wolf, D. an Mey, and C. Bischof. "How to Scale Nested OpenMP Applications on the ScaleMP vSMP Architecture". In: *Proceedings of the IEEE International Conference on Cluster Computing*. CLUSTER. Sept. 2010, pp. 29–37. DOI: 10.1109/CLUSTER.2010.38.

[SCI93] *Standard for Scalable Coherent Interface (SCI)*. 1596. IEEE, 1993. DOI: 10.1109/IEEESTD.1993.120366.

[SCIF14] *Symmetric Communications Interface (SCIF) for Intel Xeon Phi Product Family*. Users Guide. Intel Corporation, Feb. 2014. URL: http://registrationcenter.intel.com/irc_nas/5079/scif_userguide.pdf (visited on 01/20/2015).

[SDM11] J. Shalf, S. Dosanjh, and J. Morrison. "Exascale Computing Technology Challenges". In: *High Performance Computing for Computational Science - VECPAR 2010*. Vol. 6449. Lecture Notes in Computer Science. 2011, pp. 1–25. ISBN: 978-3-642-19327-9. DOI: 10.1007/978-3-642-19328-6_1.

[Sei+08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. "Larrabee: A Many-core x86 Architecture for Visual Computing". In: *ACM Trans. Graph.* 27.3 (Aug. 2008), 18:1–18:15. DOI: 10.1145/1360612.1360617.

[SS08] K. Strandberg and S. Schoenberg. *Synchronization in a Many-Core World*. White Paper. Intel Corporation, 2008. URL: https://software.intel.com/en-us/articles/synchronization-in-a-many-core-world (visited on 01/20/2015).

[ST98] A.-H. Smai and L.-E. Thorelli. "Global reactive congestion control in multicomputer networks". In: *Proceedings of the 5th International Conference on High Performance Computing*. HIPC'98. Dec. 1998, pp. 179–186. DOI: 10.1109/HIPC.1998.737987.

[Star99] *Star Wars: Episode I – The Phantom Menace*. VHS. Lucas Arts, 1999.

[Sut05] H. Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's Journal* 30.3 (2005), pp. 202–210.

[SY05] M. Snir and J. Yu. *On the Theory of Spatial and Temporal Locality*. Tech. rep. UIUCDCS-R-2005-2611. University of Illinois at Urbana-Champaign, July 2005.

[Tan07] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2007. ISBN: 978-0-13-600663-3.

[Ter+08] C. Terboven, D. an Mey, D. Schmidl, and M. Wagner. "First Experiences with Intel Cluster OpenMP". In: *OpenMP in a New Era of Parallelism*. Vol. 5004. Lecture Notes in Computer Science. 2008, pp. 48–59.

[TLH94] J. Torrellas, M. Lam, and J. L. Hennessy. "False sharing and spatial locality in multiprocessor caches". In: *IEEE Transactions on Computers* 43.6 (June 1994), pp. 651–663. DOI: 10.1109/12.286299.

[Tsa+05] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. "System Noise, OS Clock Ticks, and Fine-grained Parallel Applications". In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. Cambridge, Massachusetts, 2005, pp. 303–312. ISBN: 1-59593-167-8. DOI: 10.1145/1088149.1088190.

[TvS13] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Always learning. Pearson Education, Limited, 2013. ISBN: 978-1292025520.

[UPC05] *UPC Language Specifications*. Version 1.2. UPC Consortium. Lawrence Berkeley National Lab, LBNL-59208, 2005.

[Vaj11] A. Vajda. *Programming Many-Core Chips*. Springer, June 2011. ISBN: 978-1-4419-9739-5.

[Van+08] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS". In: *IEEE Journal of Solid-State Circuits* 43.1 (Jan. 2008), pp. 29–41. DOI: 10.1109/JSSC.2007.910957.

[vWMH11] R. F. van der Wijngaart, T. G. Mattson, and W. Haas. "Lightweight Communications on Intel's Single-chip Cloud Computer Processor". In: *SIGOPS Oper. Syst. Rev.* 45.1 (Feb. 2011), pp. 73–83. DOI: 10.1145/1945023.1945033.

[WM95] W. A. Wulf and S. A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. DOI: 10.1145/216585.216588.

[WSG02] A. Whitaker, M. Shaw, and S. D. Gribble. "Denali: Lightweight virtual machines for distributed and networked applications". In: *Proceedings of the USENIX Annual Technical Conference*. 2002.

[WWP09] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (Apr. 2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

[YTL87] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors". In: *IEEE Transactions on Computers* C-36.4 (Apr. 1987), pp. 388–395. DOI: 10.1109/TC.1987.1676921.