

Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

Evaluating the Expressiveness of Hardware Performance Counters Tracked Alongside the Critical Path

Evaluation der Aussagekräftigkeit von Hardware-Performance-Countern gemessen entlang des kritischen Pfades

Masterarbeit

Moritz Brückner
Matrikelnummer: 412238

Aachen, den 6. Juli 2026

Erstgutachter: Prof. Dr. rer. nat. Matthias S. Müller (')

Zweitgutachter: Dr. rer. nat. Stefan Lankes (*)

Betreuer: Ben Thärigen, M.Sc. (')

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University

(*) Institute for Automation of Complex Power Systems, RWTH Aachen
University

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 6. Juli 2026

Kurzfassung

Die Betrachtung kritischer Pfade hat sich als nützlicher Bestandteil der Performanceanalyse paralleler Programme bewährt, da diese die für die Gesamtlaufzeit eines Programms verantwortlichen Programmaktivitäten beschreiben. Beim Einsatz von *On-The-Fly*-Analysetools für die Analyse kritischer Pfade ist das Nachhalten der Struktur der Pfade unpraktikabel, da die technischen Möglichkeiten dieser Tools von Natur aus begrenzt sind. Andere gängige Arten der Analyse paralleler Programme nutzen Hardware-Performance-Counter, deren Messwerte häufig zu Profilen zusammengefasst oder entlang von Threads oder Prozessen aufsummiert werden. Dies spiegelt jedoch alle Programmaktivitäten unabhängig ihres Einflusses auf die Performance wider. Messungen von Hardware-Performance-Countern entlang kritischer Pfade, wie kürzlich im *On-The-Fly Critical-Path-Tool* (OTF-CPT) implementiert, helfen bei beiden genannten Problemen: Sie enthalten nur solche Programmaktivitäten, die die Gesamtlaufzeit bestimmen, und können dabei helfen, den beobachteten kritischen Pfad selbst bei Unkenntnis seiner Struktur besser zu verstehen. Diese Arbeit präsentiert eine Methodik zur Performanceanalyse basierend auf der Ausgabe von OTF-CPT, welche mit fünf verschiedenen Anwendungen getestet wird, um die Aussagekräftigkeit der Messungen zu evaluieren. Die Methodik vergleicht den kritischen Pfad mit einzelnen Threads, um dessen Eigenschaften herauszustellen und dadurch mögliche Ursachen für seine Struktur zu untersuchen. Die durchgeführten Experimente zeigen, dass die Messungen entlang kritischer Pfade dazu genutzt werden können, Performancefaktoren zu bestimmen, die in Messungen einzelner Threads unsichtbar bleiben. Aufgrund der Verwendung von Hardware-Performance-Countern ist es möglich, bestimmte kausale Zusammenhänge zwischen der Hardwarenutzung und dem kritischen Pfad auszuschließen. Ein wesentliches Hindernis besteht darin, dass sich die Messergebnisse nicht auf einzelne Regionen im Quellcode eines Programms beziehen lassen, da die Struktur der gemessenen kritischen Pfade ungewiss bleibt. Dennoch zeigen die Ergebnisse, dass die Methodik dazu genutzt werden kann, Hypothesen aufzustellen, die als Grundlage für weitere Detailuntersuchungen dienen können. Über die Methodik hinaus werden in dieser Arbeit auch die nötigen Bedingungen und Konsequenzen einer Anpassung der Methodik auf andere Analysetools diskutiert. Obwohl eine direkte Anwendung basierend auf den von Analysetools wie Scalasca aktuell ausgegebenen Daten wahrscheinlich nicht funktionieren dürfte, ist sie nach einer entsprechenden Anpassung der Tools möglich.

Stichwörter: Hochleistungsrechnen, Analyse kritischer Pfade, Hardware-Performance-Counter, OTF-CPT

Abstract

Critical path analysis has proven to be a useful technique for the performance analysis of parallel programs, since it is concerned with the program activities responsible for the program's total execution time. However, if *on-the-fly* analysis tools are used for the analysis of critical paths, they usually cannot capture the structure of the critical paths because their capabilities are inherently limited. Other common approaches for the analysis of parallel programs make use of hardware performance counters, whose measurements are often aggregated as part of profiles or accumulated over time along threads or processes. However, this comprises *all* program activities independent of their impact on performance. Measurements of hardware performance counters along critical paths, as recently implemented in the *On-The-Fly Critical-Path-Tool* (OTF-CPT), mitigate both aforementioned problems: They only consider those program activities that determine the total execution time and may help to better understand the observed critical path even if its structure is unknown. This thesis introduces a performance analysis methodology based on output provided by OTF-CPT, which is applied to five different applications in order to evaluate the expressiveness of the tool's measurements. The methodology compares the critical path with individual threads in order to expose its characteristics, and thereby investigates potential causes for its structure. The conducted experiments show that the measurements along the critical path can be used to determine performance factors not visible in measurements of individual threads or processes. Due to the use of hardware performance counters, it is possible to rule out certain causal relationships between hardware usage and the critical path. However, because OTF-CPT does not capture the structure of critical paths, it is not possible to map the measurement results to individual regions in the program's source code, which constitutes a major obstacle. Nonetheless, the results show that the methodology can be used to create initial hypotheses as a basis for more in-depth analyses. This thesis also discusses the necessary conditions and implications of an adaptation of the methodology for other tools. While a direct adaptation is unlikely to work with current output capabilities of analysis tools such as Scalasca, modifying the tools themselves would allow for a successful application.

Keywords: High-Performance Computing, Critical Path Analysis, Hardware Performance Counters, OTF-CPT

Acknowledgements

Computations were performed with computing resources granted by RWTH Aachen University under project thes2117.

Contents

1. Introduction	1
2. Related Work	5
3. Background	9
3.1. Hardware Performance Counters	9
3.1.1. PAPI	10
3.2. Critical Paths	11
3.3. OTF-CPT	14
3.3.1. Dependent Metrics	17
3.4. Score-P & Scalasca	18
4. Analyzing Event Counts Along Critical Paths	21
4.1. Theoretical & Practical Considerations	21
4.2. Formal Definitions	23
4.3. Comparing Accumulated Metrics	24
4.4. Derived Metrics	25
4.5. Analysis Workflow	25
4.6. Adaptation for Other Tools	28
4.6.1. Score-P & Scalasca	29
5. Experimental Setup	33
5.1. Hardware and Software Environment	33
5.2. Measured Hardware Events & Dependent Metrics	34
5.3. Evaluated Applications	38
5.3.1. Random Histogram	39
5.3.2. Mergesort	41
5.3.3. Sparse Matrix-Vector Multiplication	41
5.3.4. DENISE Black Edition	42
5.3.5. NEST Simulator	44
6. Evaluation	47
6.1. Random Histogram	47
6.2. Mergesort	52
6.2.1. Outliers	53
6.2.2. G-CUE Scaling Behavior	55

6.3. Sparse Matrix-Vector Multiplication	58
6.3.1. Variant 1: Without <code>nowait</code>	58
6.3.2. Variant 2: With <code>nowait</code>	62
6.4. DENISE Black Edition	65
6.4.1. FTZ & DAZ Enabled	67
6.5. NEST Simulator	69
7. Conclusion	75
7.1. Future Work	77
A. SPMxV – Source Code	79
B. Additional Measurements	81
B.1. Random Histogram	81
B.1.1. <i>allBounded</i> = false, <i>noStageOffset</i> = false	81
B.1.2. <i>allBounded</i> = false, <i>noStageOffset</i> = true	82
B.1.3. <i>allBounded</i> = true, <i>noStageOffset</i> = false	83
B.2. Mergesort	85
B.3. Sparse Matrix-Vector Multiplication	87
B.3.1. Variant 1: Without <code>nowait</code>	87
B.3.2. Variant 2: With <code>nowait</code>	88
B.4. DENISE Black Edition	89
B.4.1. FTZ & DAZ Disabled	89
B.4.2. FTZ & DAZ Enabled	90
B.5. NEST Simulator	92
List of Figures	93
List of Tables	95
Bibliography	97

1. Introduction

Analyzing the performance of parallel programs is a challenging task. Modern software and hardware are complex, with a broad range of potential factors influencing performance. The complexity is amplified by a high degree of parallelism and long run times, which create a need for highly scalable analysis approaches that guide developers towards aspects worth optimizing.

Fundamentally, analyzing and optimizing application performance with respect to execution time always tries to make optimal use of the available hardware. This process takes place on different but interdependent levels of granularity for parallel applications: On a macroscopic and structural level, there is the relationship between the parts of the application which includes communication patterns and the distribution of the application's workload over time and space. On a lower, complementary level, analysis and performance optimization is concerned with the absolute duration of individual program activities, as compared to the macroscopic view that considers their interplay and thus has a strong focus on relative timings.

The analysis of *critical paths* emerged as a useful technique on the structural level: The critical path of a parallel application is the longest path of program activities that need to be executed in order; it thus determines the total execution time of the application. Optimizing the run time of program activities outside the critical path does not shorten the overall run time but only increases waiting time, hence any effort that aims to improve a program's run time should focus on the critical path.

An example for critical path analysis is the *On-The-Fly Critical-Path-Tool* (OTF-CPT) [1], which computes durations of critical path variants that assume certain program activities to take no time. It uses the durations to quantify structural inefficiencies of an evaluated program. OTF-CPT is an *on-the-fly* analysis tool, which, in comparison to *post-mortem* analysis, performs the analysis during execution of the evaluated program. This makes it a good choice for an initial performance investigation whose results can be used as a basis for more in-depth analyses.

OTF-CPT does not capture the structure of critical path variants as this would be impracticable for on-the-fly analysis. Internally, the tool only holds as much information as momentarily required to compute the durations of the critical path variants. While OTF-CPT's quantification of inefficiencies on the structural level is an important indicator of performance issues, it alone only provides limited insight into the reason(s) for the observed issues.

1. Introduction

On the lower level of performance analysis and optimization, hardware performance counters become particularly useful as a tool to understand how well a given region of code runs on a particular system. They can reveal various problems that may otherwise go unnoticed: branch mispredictions, cache misses, etc. Hardware event counts are typically measured for individual hotspots, aggregated in (function) profiles, or accumulated over time along individual threads or processes. Examples for this approach are `likwid-perfctr` [2], which measures values of hardware performance counters accumulated over individual threads of an application, and *Score-P* [3], which creates execution traces and profiles of parallel applications that optionally include measurements of hardware performance counters. While these measurements are relatively easy to perform, they reflect all activities within the application independent of whether they actually affect the program’s execution time.

A combination of critical path analysis with the analysis of hardware performance counters could mitigate both aforementioned problems: Measuring hardware performance counters along critical paths ensures that only program activities determining the total execution time are considered in the analysis, and the measurements may help to understand more about the observed critical paths including their causes. OTF-CPT recently got support for measuring hardware performance counters along critical paths [4] to extend the tool’s analysis capabilities. This gives rise to the question of what measurements of hardware performance counters accumulated along critical paths can tell about application performance, and in particular, whether they are more expressive than measurements accumulated along individual threads or processes, as it is common with traditional hardware performance measurement approaches.

This thesis proposes a methodology to analyze hardware performance counter measurements accumulated along critical paths. The methodology is applied to five different applications in order to evaluate whether the measurements are expressive and to compare them to measurements accumulated along individual threads. It is furthermore used to evaluate whether the measurements can be used to reason about what causes the critical path to become the critical path.

In addition to the evaluation of hardware performance counter measurements along critical paths, this thesis also discusses the necessary conditions and implications of an adaptation of the methodology to other tools.

Thesis Structure. The remaining parts of this thesis are structured as follows: Chapter 2 provides an overview of previous scientific work on the topics applied in this thesis. Chapter 3 introduces the concepts and analysis tools used throughout this thesis. Chapter 4 proposes a new methodology for analyzing measurements of hardware performance counters tracked along critical paths in comparison to measurements along individual threads. The end of this chapter (Section 4.6) deals with the applicability of the methodology, which is based on output provided by OTF-

CPT, to other tools. Chapter 5 addresses the implementation of the evaluation of the proposed methodology on the CLAIX-2023 HPC cluster. It also discusses the hardware events chosen for the evaluation and introduces the five evaluated applications, before presenting the analysis results in Chapter 6. The thesis concludes with a brief summary and a discussion of potential future research topics in Chapter 7.

2. Related Work

Critical path analysis. The notion of critical paths was originally developed in the context of project management [5] and later adapted for distributed computing by Yang and Miller [6]. Since then, the analysis of critical paths has been found to be a useful performance analysis technique due to its ability to locate the program activities responsible for a program’s execution time.

Some implementations expose the complete structure of the critical path [7, 8, 9] which allows for extremely fine-grained analyses but makes it difficult to quickly identify potential bottlenecks (see [10]). Other approaches report a profile of the activities found on the critical path with varying degrees of detailedness [6, 11, 12, 13]. In case of OTF-CPT, no structure is reported at all [1] since it is not explicitly tracked.

While the critical path is able to guide optimization efforts, it alone is not sufficient to show all bottlenecks such as load imbalance, which requires a consideration of individual processes or threads (see [10]). Moreover, application performance may depend on “effects outside the critical path” [14], which are considered by an analysis of wait states in combination with critical paths [10, 12, 15, 13] in the context of Scalasca [16].

Following an idea from Miller et al. [17], Alexander et al. coined the term *near-critical path* for execution paths whose duration is close to the duration of the critical path [18, 19]. Assuming no structural overlap between the critical path and the closest near-critical path, an optimization of the critical path can only improve the program’s execution time by the duration difference between the two paths. Hollingsworth and Miller call this difference *slack* [20, 21].

Miller et al. introduced the concept of *logical zeroing* [17] (the term was only later established in [20]) that can be used to perform a *what-if* analysis [22] by assuming certain program activities to take no time. Albeit done for other reasons than a what-if analysis, logical zeroing is indirectly used in OTF-CPT to compute different critical path variants of an application in order to calculate *fundamental performance factors* in the sense of Rosas et al. [23].

Constructing the critical path from measurements usually can be categorized into two approaches: On-the-fly analysis approaches like [11, 8, 1] compute the critical path during an execution of the program. Post-mortem approaches such as [6, 24, 12, 13, 25, 26] usually collect traces during program executions that are later used to reconstruct the critical path. The technique proposed by Schulz makes use of

2. Related Work

both approaches by tracking execution graphs local to each processor during runtime that are used to compute the critical path after program termination [7]. Saidi et al. developed a technique that analyzes critical paths of a *simulated* execution in a “full-system simulator” [27].

Hardware performance counters. The use of hardware performance counters is well established in performance analysis even outside scientific research. Many tools to collect and analyze hardware event counts have emerged over the years (e.g., [28, 2, 29, 30, 31, 3, 32¹, 1]).

Treibig et al. list common performance bottlenecks found in scientific applications and explain how they can be detected by hardware performance counter measurements [33]. Yasin proposed a method for top-down performance analysis for out-of-order CPUs based on hardware performance counters [34]. The method locates bottlenecks by quantifying the utilization of CPU pipeline slots and determining its causes. While the method is widely acknowledged and even used in *VTune* [30], the choice of measured hardware events is highly processor-specific, requiring deep knowledge about the used microarchitecture to fully adapt the methodology. Dutilleul et al. discuss some limitations of the top-down method and propose a sensitivity-based analysis that compares executions while “varying the usage or capacity of one or more resources” [35]. Yoo et al. developed a machine learning technique to diagnose known performance issues from hardware performance counter measurements [36]. Tarraf et al. presented an automated method that is able to quantify the noise-resiliency of hardware events [37].

Weaver et al. showed that hardware event counts are generally non-deterministic, e.g., due to being sensitive to external influences of the execution environment [38]. Zaparanuks et al. showed that different measurement patterns but also external factors such as code placement in memory have an enormous impact on measured cycle counts on the IA32 architecture [39].

Combined analysis of critical paths and hardware performance counters. In the context of hybrid MPI+CUDA applications, Schmitt et al. stated that “[t]he rating of optimization targets could benefit from the inclusion of hardware performance counters in the analysis” [9]. Shatalin et al. mapped hardware performance counters to critical paths in MPI programs [26], but their publication only considers very basic hardware metrics, lacks an explicit evaluation of the measurements and does not compare the results with measurements of program activities outside the critical path.

¹ Support for hardware performance counters measurements via PAPI [29] was only added later in <https://github.com/bsc-pm/dlb/commit/6faea8ab5b076f081419a05de3ceec91f35af56b> (accessed: Jun. 30, 2026)

Streichardt recently added support for the measurement of hardware performance counters along critical path variants in OTF-CPT [4]. The work also compares OTF-CPT measurements summed over all threads of an application with measurements using Score-P [3]. It concludes that cache accesses and misses are comparable whereas the amount of instructions and CPU clock cycles varies between the tools. However, the work does not mention whether the measurements stem from a single or several executions.

3. Background

This chapter provides a brief introduction to the concepts used throughout this thesis. It introduces hardware performance counters (Section 3.1), critical paths (Section 3.2), and the analysis tools used for the experiments conducted in the context of the presented work (Sections 3.3 and 3.4).

In the context of this thesis, parallel programs are assumed to adhere to the common two-level execution model in which every parallel program runs one or more processes, each consisting of one or more threads that execute the actual code. If a program follows a multiprocessing paradigm but does not use multithreading (e.g., by only using MPI for parallelization), a process is assumed to consist of a single thread. In appropriate places, the term “thread” may thus be used in place of the word “process” to allow for simpler explanations without explicitly dealing with different kinds of programming paradigms where it does not matter. Furthermore, it is assumed that every piece of work (computation or communication) happens as early as possible, i.e., that no *unnecessary* waiting time is inserted into the execution.

3.1. Hardware Performance Counters

Modern microprocessors commonly provide dedicated registers, called *hardware performance counters*, for counting occurrences of microprocessor-specific hardware events such as clock cycles, branch mispredictions, CPU cache requests, etc. These event counts help to get a detailed view of CPU behavior which may guide software performance analysis and provide explanations for observed performance issues.

A hardware performance counter may be *fixed*, in which case it is bound by the CPU manufacturer to one specific type of event only, or *programmable*, meaning that the event to count can be freely configured, for example by writing an identifier associated with the event into a certain CPU register [40]. Typically, the number of available programmable counters on a given CPU—and therefore, the number of event types that can be counted at the same time—is much lower than the number of provided events. If one wants to measure events belonging to more types than counters available, one either has to decide on a small selection of events before performing any measurements (and then potentially measure several program runs with different sets of events to obtain a more complete image), or make use of *multiplexing*, which samples counter values while quickly alternating between various sets of events within a single program execution.

3. Background

On the hardware level, performance counters are part of so-called *Performance Monitoring Units (PMUs)* [40, 41]. In modern processors, PMUs can usually be found on every CPU core but also outside of individual cores (a region called *uncore* by Intel [40]). This allows for counting core-specific hardware events but also events that originate from other parts of the circuitry outside the cores, such as shared caches, interconnects, and memory controllers. In addition to *core* and *uncore* events, Intel uses the term *offcore* [42] to describe events happening as a result of CPU core requests *leaving* the core towards the uncore [43].

While hardware counters allow for low-level insight into highly-specific CPU behavior, this very same property can pose a problem: Correctly interpreting event counts often requires deep technical knowledge about specific CPU microarchitectures, which may be difficult to obtain given the sparsity—or even non-existence, in some cases—of verified and publicly available information. Microprocessor manufacturers usually provide brief explanations of most of the countable events, but these might be difficult to understand without extensive prior knowledge on the subject (see [44]). Furthermore, some events involve intricacies that are only known for example due to experiments conducted by third parties, which may be difficult to verify. This inevitably leads to the conclusion that there may be other events whose quirks are simply not publicly known yet. Even if information from non-manufacturer sources is true, it may only be available for specific microarchitectures and therefore cannot be reliably assumed for the specific hardware at hand. Some CPUs even appear to have undocumented or “hidden” [45] events whose existence and semantics are not publicly specified at all [45, 46].

Additionally, event counts are not guaranteed to be deterministic or accurate; for instance, Weaver et al. [38] showed that counts for some x86-64 events are influenced by external factors such as hardware interrupts. Thus, measurements of small counter values usually should not be considered representative. Care also needs to be taken for long measurements as hardware performance registers can overflow. This condition can usually be caught by means of hardware interrupts or by reading the value of a status register flag [40, 47]. The latter requires that there is at most one overflow occurring between any two reads of the status register; otherwise, one or more overflows will be missed.

3.1.1. PAPI

The experiments conducted as part of this thesis make indirect use of the *Performance Application Programming Interface (PAPI)* [29], which is an open-source abstraction layer that allows to read values of hardware performance counters in a mostly platform-independent fashion. PAPI takes care of the interaction with raw CPU registers that is needed in order to program and read the performance counters, and it provides a mapping between many microprocessor-specific events and higher-level platform-independent metrics. Some of the predefined PAPI met-

rics map directly to one CPU-specific performance event whereas other so-called *derived metrics* result from a combination of multiple hardware performance events by means of a predefined formula.

By working with an abstraction on top of processor-specific hardware events, users can read from hardware performance counters without having to know all the underlying details, but more control about which specific events are read may be desired depending on the use case. However, the PAPI interface also accepts processor-specific event identifiers instead of its abstract higher-level metrics, therefore enabling easy counter measurements while still exposing specific hardware event values.

3.2. Critical Paths

Every execution of a parallel program can be understood as a set of program activities [6], which are single cohesive items of computation or communication. An activity is assumed to be uninterruptible from its abstract point of view; if it represents computation, it is therefore assumed to be executed on a single thread. The activities form edges on a directed acyclic graph (the *program activity graph* [6]) whose vertices correspond to the beginnings and ends of the program activities. The direction of each edge follows the natural temporal order of execution; it starts at the vertex of the beginning of its corresponding activity and ends in the vertex that represents the end of its activity. If the end of an activity marks the beginning of a new one (e.g., within each individual thread, all computation activities but the last are directly followed by another), the coinciding beginnings and ends form a single vertex in the program activity graph. Each edge in the program activity graph is associated with a weight that corresponds to the time it takes for the activity to execute¹.

The vertices of the program activity graph give rise to an implicit *wait-for* relation “ $\rightarrow$ ”: An activity a_2 *waits for* another activity a_1 (denoted by $a_1 \rightarrow a_2$) if the end of a_1 coincides with the beginning of a_2 ; in this case, the execution of a_2 cannot begin sooner than the end of a_1 ’s execution. This can be due to a direct dependency between two activities (e.g., due to communication between the activities), but it also occurs if two activities are executed in direct succession on a thread. The activities on any path through the program activity graph (i.e., $a_1 \rightarrow \dots \rightarrow a_n$ for n activities $a_1, \dots, a_n$) execute in sequence due to their wait-for constraints. Thus, the sum of the durations of the path’s activities determines its total execution duration. The *critical path*² of a parallel application is the path of activities in the program ac-

¹ Technically, the weight can be any other metric, but typically, it is some kind of duration.

² The notion of critical paths was originally developed by Kelley and Walker [5] in the context of project management to analyze and coordinate the scheduling of individual project activities. Yang and Miller [6] were the first to adapt this concept to the context of parallel computing.

3. Background

tivity graph with the longest overall duration³ [6]; it therefore determines the total execution time of the program. Consequently, it is only possible to decrease the total execution time by decreasing the time spent along the critical path, either (i) by decreasing the execution time of individual activities on the critical path, which in turn may or may not affect which activities make up the critical path, or (ii) by modifying the program in such a way that a different wait-for relation emerges. Because altering activities on the critical path may result in a new, differently structured critical path, a decrease in the duration of activities on the critical path may not actually shorten the critical path by the same amount [12, 13].

All wait-for dependencies of an activity must be done running before itself may start to execute. If some wait-for dependencies end sooner than others, this induces waiting time not filled by any activity. Because all activities start executing as soon as possible, no waiting time occurs on the critical path.

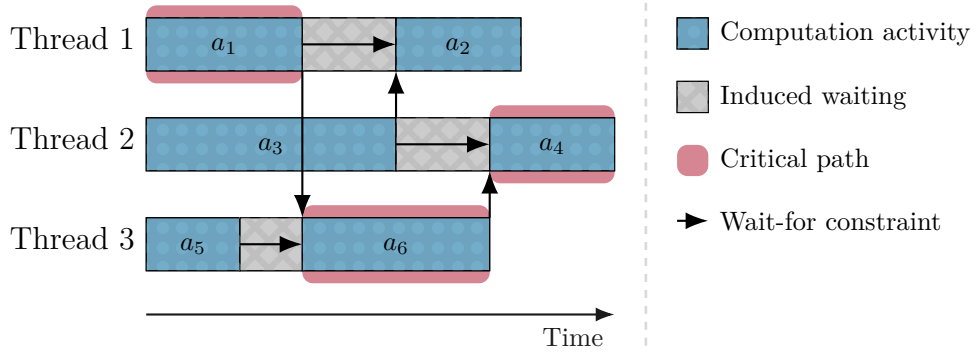


Figure 3.1.: Example execution sequence for three threads highlighting the critical path. For simplicity, communication activities are assumed to take no time (i.e., they have zero weights associated with their corresponding edges) and are not visualized.

Figure 3.1 illustrates the critical path resulting from an example program execution with three threads. Although activity a_3 takes the most time in the example execution, decreasing its duration does not reduce the program’s total run time: The only two activities that wait for a_3 are a_2 and a_4 , but a_2 already ends sooner than a_4 , and a_4 cannot start earlier because it also needs to wait for a_6 , which is not influenced by the change in duration of a_3 . Decreasing the duration of a_3 by a duration x would thus simply prolong the waiting time between a_3 and a_4 by x .

To apply the concept of a critical path to an actual program execution, one first needs to define the boundaries of activities. This strongly depends on how the program implements parallelization (e.g., by using MPI, OpenMP, or similar mechanisms), but typically, any transition between the program’s actual workload (its

³ In rare circumstances, two or even more paths may qualify as the longest path, in which case all are considered a critical path.

useful execution [1]) and code related to a parallel runtime is considered a boundary. Likewise, transitions between different code regions of parallel runtimes such as multiple different calls to MPI functions are considered boundaries as well. In the remainder of this thesis, the term *communication (activities)* will be used for the computational program activities that handle communication (i.e., code regions of parallel runtimes such as MPI or OpenMP), in contrast to *data transfers* that correspond to the actual exchange of data as part of that communication.

Only few publications specifically deal with the question of how to handle actual data transfers in the context of critical paths: Yang and Miller consider the correspondence between a send call and a matching receive call to be a unique edge in the program activity graph [6]. Likewise, Schulz defines an MPI-centric variant of a program activity graph whose edges include “message events” [7], for example between receive or send operations. Böhme [13] abstractly defines (but never explicitly refers to) *communication edges* between threads or processes synchronized by communication. At a later point, Böhme [13] describes how Scalasca (see Section 3.4) implements the detection of critical paths which appears to not explicitly consider data transfers as part of the program activity graph. Here, a communication activity that caused all other members of its communication to wait is considered to be part of the critical path for its entirety (see [13, Figure 5.4] for an example).

In the general case, it is only possible to determine the critical path by means of measurements due to the unpredictability of program executions. The duration of the measured and reconstructed critical path, which in the following will be called the *observed critical path*, approximates that of the *true critical path* that would be present during the very same program execution if there was no active measurement. However, the observed critical path does not necessarily preserve the structure of the true critical path: The measurement process usually introduces overhead as it interacts with the measured program, thereby influencing the program execution such that a different critical path could emerge. Additionally, the limited accuracy and precision of any measurement causes slight differences in reported activity durations, which in turn may affect the reconstruction of the critical path. The validity of the observed critical path depends on the existence of other execution paths with durations similar to the true critical path (*near-critical paths* [19]) and their properties. As long as the measurement impact is smaller than the difference in duration between the true critical path and any near-critical path, the observed critical path reflects the structure of the true critical path. Even if it is not guaranteed that the measurement preserves structure, the observed critical path is not necessarily rendered useless: If the measurement-related effects are mostly within the natural variation of the true critical path between application runs, it is possible to get a reasonably close view of the naturally occurring critical paths by repeating the measurement often enough.

3.3. OTF-CPT

The *On-The-Fly Critical-Path-Tool* (OTF-CPT) [1] is a performance analysis tool for applications using OpenMP and MPI. It uses critical path analysis to compute *fundamental performance factors* in the sense of Rosas et al. [23]. OTF-CPT does not determine the critical path for the original program activity graph (PAG) of an execution. Instead, it computes several critical paths, each for a variant of the PAG with zero weights for some activities, following a combination of two types of constraints. For the remainder of this thesis, a PAG is defined to be *restricted to* a set of activities if all activities *not* in the set have zero weights.

The first constraint type restricts the PAG to certain *kinds* of program activities (following [1]):

- The *CUE* (*critical path of useful execution*) only tracks useful execution and ignores any time spent in OpenMP and MPI routines. This is used for example to compute the *Transfer Efficiency* [1] by considering the CUE's duration as the execution duration if communication activities were instantaneous.
- The *COM* (*critical path of outside MPI time*) includes activities related to useful execution and OpenMP while ignoring time spent in MPI routines.
- Analogously to the COM, the *COO* (*critical path of outside OpenMP time*) includes activities related to useful execution and MPI while ignoring time spent in OpenMP routines.

Second, each of these three types of critical paths is computed for three different levels of the parallelization hierarchy:

- A *global* critical path is the critical path of a program activity graph that is not restricted to individual processes or threads.
- A *process-local* critical path is the critical path of a per-process PAG restricted to activities of the process it belongs to. Synchronization with other processes is thus ignored, i.e., an activity within the restricted process-local PAG does not need to wait for activities in other processes as they are considered to execute in zero time. For pure OpenMP applications, a process-local critical path is also a global critical path as there is only one process.
- Finally, a *thread-local* critical path is the critical path of a per-thread PAG restricted to activities of the single thread it belongs to. It completely ignores synchronization with all other threads. Since a single thread executes its activities linearly, this simply equals the entire execution of the thread, only with some edge weights set to zero according to the first constraint (CUE, COM, COO).

Following Protze et al. [1], this thesis denotes these different levels of critical paths using the prefix *G-* for a global critical path, *PL-* for a process-local critical path, and *TL-* for a thread-local critical path. For instance, the global critical path of useful

execution is denoted by the abbreviation *G-CUE*. These two types of constraints result in a total of nine kinds of critical paths, which in the following will be called *critical path variants*.

As a third, implicit constraint, the user can manually enable or disable OTF-CPT in their code to limit all program activity graphs to only span a certain region of the evaluated application instead of the entire execution time. This does not change the number of critical paths tracked by OTF-CPT since the tool either tracks all nine variants over the entire execution or over a selected region.

Note that altering the PAG leads to critical path variants whose activities are not guaranteed to lie on the critical path of the original unrestricted PAG. For instance, the activities on a G-CUE do not generally equal the useful execution activities on the critical path for the original PAG.

As discussed in Chapter 2, many tools that perform critical path analysis collect extensive traces to reconstruct the critical path after program termination. While this approach allows for a detailed analysis of the measured program execution, trace collection adds I/O-related overhead and individual traces can get extremely large for complex and long-running programs [13]. Contrary to these approaches, OTF-CPT performs its analysis *on-the-fly* while the application under evaluation is running. It traces the critical path variants of the evaluated application in real-time and keeps all data related to the analysis in memory. After the execution of the configured program region is completed, OTF-CPT performs a few final computations and outputs the results afterward. This approach is viable because OTF-CPT does not need to know the structure of the critical path for its analysis. To compute the fundamental performance factors, it only needs to consider the *durations* of the different critical paths [1], which greatly simplifies the implementation and reduces the amount of data that needs to be stored in order to trace the critical paths. Compared to execution traces which usually grow with the duration of the traced application, OTF-CPT’s memory requirements are independent of the execution duration.

OTF-CPT maintains nine different clocks for each thread of the evaluated application, with each clock corresponding to one critical path variant tracked by the tool. Every clock is zero-initialized and can be started or stopped. If a clock gets started, the current wall-clock time is subtracted from its current value; if it gets stopped, the current wall-clock time is added instead. A stopped clock thus stores the cumulative duration between any two start and stop operations so far. Note that it is possible to compare several of the same clocks along different threads with each other if they are all started or all stopped: In both cases, a larger value indicates that the clock has been running for a longer time. When a clock’s thread passes the boundary between two program activities of which only one has a positive weight on the clock’s PAG, the clock is started or stopped accordingly. For example, a clock that only counts useful execution is stopped when an MPI or OpenMP routine is called, and resumed once the control flow returns from the routine.

3. Background

If two or more threads synchronize due to MPI or OpenMP communication, OTF-CPT updates the thread clocks based on the type of communication. For instance, for a matching pair of MPI send/receive operations, OTF-CPT sets the clocks of the receiving thread to the maximum of the clocks of both communication peers. A larger clock value corresponds to a longer execution path and therefore represents the execution path to the point of synchronization that did not incur waiting. In case of MPI barriers, all participating threads update their clocks to the maximum clock value among any communication peer. Since communication happens either purely within MPI-related activities or OpenMP-related activities, the exchanged clocks for a particular critical path variant are either all started or all stopped, so the maximum clock update always yields correct results.

At the end of the measurement, the critical path variants are represented by the corresponding clocks with the longest overall duration. Since OTF-CPT only tracks durations by default, it does not keep track of *how* it reached a local partial result. For the same reason, edges in a program activity graph that correspond to data transfers or similar synchronization can be safely ignored. Each synchronization happens concurrently while the corresponding MPI or OpenMP routines are executing; therefore, it does not matter when and for how long the synchronization took place.

OTF-CPT injects itself into the evaluated application by means of the PMPI [48, Section 15.2] and OMPT [49][50, Chapter 4] interfaces: Whenever the user code calls an MPI function (prefixed by `MPI_`), a wrapper implemented by OTF-CPT is called first which performs the necessary work for the tool and internally calls the underlying MPI implementation via a function prefixed with `PMPI_`. For OpenMP applications, OTF-CPT instead defines OMPT callbacks that are called by the OpenMP runtime when certain events occur.

Figure 3.2 illustrates OTF-CPT’s implementation for a matching pair of MPI send and receive operations. A call of an OTF-CPT MPI wrapper function denotes the boundary between useful execution and the beginning of MPI-related execution. Therefore, OTF-CPT stops the corresponding clocks for the CUE and COM variants upon entering the wrapper function. Then, the clock exchange is started; this is implemented by a separate pair of send and receive operations on a separate communicator in order to not interfere with user code [1]. Once the clock of the sender process is received, the clock of the receiver is updated according to the clock’s maximum. At the end, the stopped clocks are resumed and the control flow returns from the MPI wrapper function.

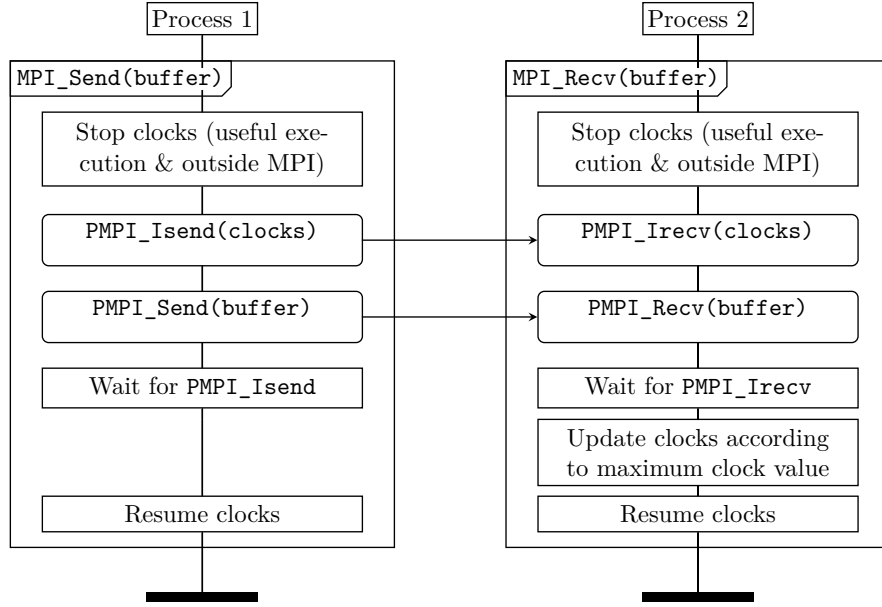


Figure 3.2.: OTF-CPT implementation for a matching pair of `MPI_Send()` and `MPI_Recv()` calls. Note that the exact steps used to exchange the clocks between the processes vary depending on the exact parameters of the MPI calls.

3.3.1. Dependent Metrics

OTF-CPT recently got support for PAPI-based measurements of hardware performance counters along all critical path variants [4]. The measurements of every selected hardware event are individually accumulated along individual critical path variants: For every critical path variant and hardware event, a single metric value is determined that describes how often that event occurred in the execution of the path. OTF-CPT calls the measured event counts *dependent metrics* since they are tracked *alongside* critical path variants which are purely determined by wall-clock times. Their accumulated values therefore *depend* on the critical paths from the perspective of the tool, even if both hardware events and critical paths technically reflect the application's execution on the hardware.

An example for is shown in Figure 3.3, which illustrates the G-CUE for an application with three threads. Every computational program activity is associated with a count of a fictional hardware event. While there is a path through the useful execution of the example application that amounts to $105,000 + 75,000 = 180,000$ occurrences of the event, the G-CUE only amounts to $40,000 + 30,000 + 15,000 = 85,000$ events because event counts are not taken into account for determining the critical path.

3. Background

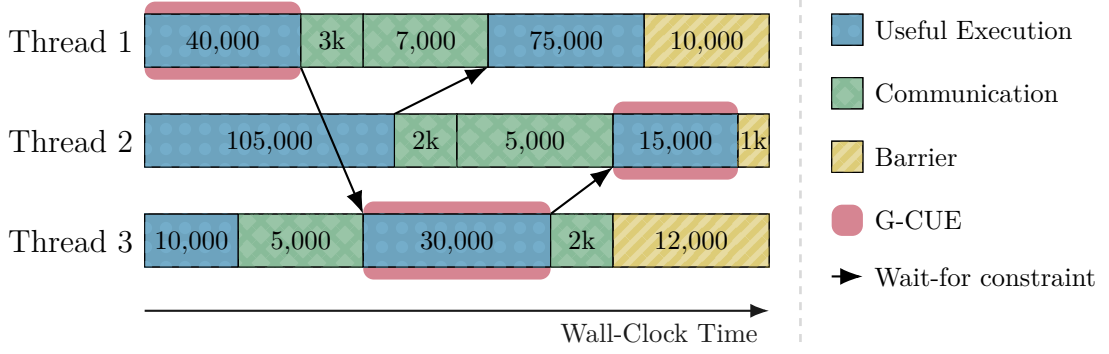


Figure 3.3.: Example sequence of program activities for three threads highlighting the G-CUE and a fictional hardware event. The sum of event counts along the G-CUE is smaller than the sum of event counts along other execution paths limited to useful execution, but only the wall-clock time determines the structure of the critical path. For simplicity, only communication-related wait-for dependencies are shown in the figure.

OTF-CPT implements dependent metrics by maintaining an array for the tracked hardware events along with each clock. Dependent metrics stored for a clock follow the same principle as the clock’s duration value: Each time a clock is started or stopped, OTF-CPT reads the dependent values, which in case of hardware performance counters⁴ causes a call to `PAPI_read()`. When a clock gets started, the current event count is subtracted from the clock’s value corresponding to that dependent metric. When a clock is stopped, the current event count is added to the clock’s event count value. The values of dependent metrics do not influence the structural outcome of a maximum clock update during synchronization between threads, as only the clocks’ time values are used in the comparison. The clock representing the longest path also sets the dependent metrics of all other synchronized clocks.

3.4. Score-P & Scalasca

Scalasca [16] is a performance analysis toolset for parallel programs. Its main concern is the performance of hybrid and non-hybrid applications using MPI, OpenMP, and POSIX threads [51]. The toolset is based on the *Score-P* performance measurement infrastructure [3], which can be used to instrument an application in order to collect execution traces and profiles. Based on the traces recorded by Score-P, Scalasca can reconstruct the critical path of an execution and compute the wall-clock duration of each instrumented function on the critical path. Score-P can also be configured to measure hardware performance counters using PAPI.

⁴ Technically, dependent metrics do not have to correspond to hardware events.

Scalasca reconstructs the critical path *post-mortem*: During an application run, Score-P performs its measurements and stores the results on disk. After the application terminates, Scalasca post-processes the collected data for analysis. Like OTF-CPT, Score-P can be configured to perform its data collection only for a certain instrumented code region.

Scalasca determines the critical path as follows (see [13]) In a first step, Scalasca replays the execution trace in order to detect induced waiting time outside the critical path as well as points of synchronization between threads. Waiting times are detected simply by comparing the start times of matching communication activities. Afterwards, Scalasca determines the end of the critical path which is trivially marked by the last ending program activity. Starting at this activity, Scalasca performs the replay in reverse along the thread on which the critical path ends, until a program activity with wait time is reached. Since this activity needed to wait on another activity, that other activity must be part of the critical path, hence the backward replay continues from there. This process is repeated until the start of the trace is reached.

4. Analyzing Event Counts Along Critical Paths

This chapter introduces a methodology for analyzing hardware event counts measured along variants of critical paths of a parallel application. The methodology makes use of comparisons between the G-CUE and individual TL-CUEs in order to understand the expressiveness of the G-CUE measurements in comparison to measurements of individual threads. It uses a hierarchical workflow for the comparisons that allows for creating hypotheses about which performance factors determine the critical path.

4.1. Theoretical & Practical Considerations

Ideally, the presented methodology should be able to analyze the global critical path through the complete program activity graph of evaluated executions. This critical path consists of all program activities that determine the execution's run time and are therefore guaranteed optimization targets. To shift the focus to a specific region such as a compute kernel, the critical path can be restricted to program activities within that region without losing its relevance.

Due to the use of hardware performance counters, we are most interested in the analysis of useful execution activities. The code of useful execution activities is typically written by the application developer or otherwise within their reach, so an analysis of useful execution activities can point the developer to issues they are able to fix themselves. Communication-related activities can equally affect performance, but their implementations are usually outside the developer's influence, leaving only little room for improvement apart from restructuring communication itself (i.e., choosing different communication patterns or primitives). The methodology should therefore be able to differentiate between useful execution and communication-related activities, as mixing both types of activities within a single metric would be of limited value due to preventing the developer from relating the analysis results to their own code.

Since OTF-CPT currently is the only available source of hardware performance counter measurements along observed critical paths, the methodology is fundamentally based on its output. A more generalized methodology might be possible (see Section 4.6) but currently could not be evaluated. Thus, the methodology is not

4. Analyzing Event Counts Along Critical Paths

able to consider the critical path of the complete program activity graph in the analysis; instead, only the nine different critical path variants provided by OTF-CPT (see Section 3.3) are available as input data.

For every metric and critical path variant tracked by OTF-CPT, a single accumulated measurement is reported. As a consequence, it is not possible to differentiate between activities within the critical path variants. Therefore, only the CUE variants (the critical paths of useful execution) are candidates for the primary analysis as any other variant would mix useful execution and communication-related activities¹. The main focus of the methodology lies on the G-CUE (which is used in place of the global critical path through the complete program activity graph, as motivated for the ideal case), and on individual TL-CUEs, which represent all useful execution activities within individual threads. This means that the methodology does not analyze the performance of the evaluated execution as a whole, but rather of the same execution if communication was instantaneous. While the G-CUE does not necessarily only consist of activities that would be found on the critical path of the entire execution, its use is what enables the methodology’s goal of locating reasons of the G-CUE’s structure, as the measurements are decoupled from communication-related factors.

The other critical path variants (COO and COM) could also be of interest for supportive comparisons in the analysis. However, in the clock-based on-the-fly algorithm used by OTF-CPT, COM- and COO-related clocks are not guaranteed to measure correct values for dependent metrics, rendering them unusable for analysis. The reason for this is found in the clock exchange between different threads or processes as implemented in OTF-CPT’s MAXUPDATE procedure and best explained with an example: If clocks are exchanged as part of an MPI send/receive operation, COO-related clocks are still running and their current value includes their subtracted start value. If, when a COO-related clock of both communicating processes is compared in OTF-CPT’s MAXUPDATE procedure, the sender process wins this comparison because its clock is running for a longer time, its clock overwrites the clock of the receiver process. This means that the clock that now transferred its values to a different thread may not necessarily be stopped on the same thread it was started on. For a purely time-based metric, this does not matter as long as time is reliably synchronized for all threads, but the values of dependent metrics are generally not the same on all threads at a given point in time. In that case, the current value of a dependent metric that is added when a clock is stopped may not be comparable to the subtracted metric value when the clock is started, resulting in invalid measurements with possibly even negative values. A fix for this issue

¹ Recall that different critical path variants may have different structures. Consequently, it is not possible to limit the metric of a critical path variant to some program activities by subtracting another variant’s metric from it. For example, subtracting CUE measurements from the measurements along the COO is not meaningful; it would not result in durations or event counts just for the MPI implementation. The only exception for this are thread-local critical paths, as they simply follow the structure of a single thread.

would be to stop running clocks immediately before data transfers and to resume them immediately after, such that a clock is always started and stopped within the same thread. However, this is infeasible, as this would require hooking into the implementations of parallel runtimes.

Two more limitations arise from the use of OTF-CPT: Since it is infeasible to explicitly track the structure of critical path variants in OTF-CPT, their structure remains unknown to the methodology. Furthermore, there is no available information about the temporal “distinctiveness” of the critical path: It is not known whether the critical path differs from a near-critical path by a few clock cycles only, or whether the closest candidate path differs by seconds, for example.

4.2. Formal Definitions

Given OTF-CPT measurement results for an execution of an application, the metrics measured along the G-CUE can be formalized as follows: In the context of a measured program execution, let $\mathbf{E}$ be the set of identifiers of all counted hardware events (see Table 5.1; for every application run, $\mathbf{E}$ corresponds to one of the two sets in the table). We also introduce a pseudo-event with the identifier `WallClockTime` that counts the elapsed wall-clock time which is also measured by OTF-CPT for its critical path variants. This neither constitutes a dependent metric nor an actual hardware event but can be mostly treated as such—from the methodology’s perspective, the only difference is that `WallClockTime` values are considered real numbers $\mathbb{R}_{\geq 0}$ instead of natural numbers $\mathbb{N}_0$. Thus, we redefine $\mathbf{E}$ to be the set of identifiers of all counted hardware events including `WallClockTime` to also allow for a duration-wise comparison of the G-CUE with activities outside of it. Given this definition, let $\text{GCUE} : \mathbf{E} \mapsto \mathbb{R}_{\geq 0}$ be the function² that maps a given event identifier to the event’s value along the global critical path of useful execution. If, for example, there are a million level two cache misses counted along the G-CUE’s activities which take two seconds to execute, $\text{GCUE}(\text{L2_RQSTS.ALL_DEMAND_MISS}) = 1,000,000$ and $\text{GCUE}(\text{WallClockTime}) = 2 \text{ s}$.

Furthermore, let $\mathbf{T}$ be the set of all individual threads in the program’s execution (note that every process consists of at least one thread). Let $\text{TLCUE} : (\mathbf{T} \times \mathbf{E}) \mapsto \mathbb{R}_{\geq 0}$ be the function that, given a thread and an event identifier, evaluates to the event’s value accumulated along the thread-local critical path of useful execution of the given thread, or, in other words, along all useful execution activities of that thread.

² The absence of a “-” in the function’s name is a deliberate choice to differentiate the function “GCUE” from the concept and mathematical structure “G-CUE”.

4.3. Comparing Accumulated Metrics

Analyzing metrics tracked along the G-CUE already allows for some insight in isolation, but the performance impact of individual values might be difficult to interpret. A comparison between the G-CUE and other sets of activities within the application's execution may help here as it can show what makes the G-CUE distinct from the rest of the program execution. Ideally, it would be possible to compare the G-CUE to *all* other execution paths in order to see where its measurements are situated within the distribution of execution paths, but tracking every path would not scale well with the duration and complexity of evaluated applications and is not implemented in OTF-CPT. However, there are two simpler but still useful comparisons that are possible with OTF-CPT's output: First, we want to compare the G-CUE with all individual TL-CUEs: If a metric tracked along the G-CUE is found to be inside the range of all TL-CUEs of the execution, the metric alone is not reflecting the reason for the structure of the G-CUE as there are execution paths with a lower or higher metric value than the G-CUE. This way, the methodology is able to rule out certain causes for the G-CUE's structure. If, on the contrary, a metric accumulated along the G-CUE is (far) outside the range of the TL-CUEs, this hints at (but does not prove) a relationship between the observed performance and hardware conditions reflected by the metric.

The second comparison is between the G-CUE and all useful execution activities outside of it. This can hint at optimization potentials: In some cases, the outside activities may approximate what another execution of the application could look like if the regions on the G-CUE did not exist. The comparison again makes use of thread-local critical paths of useful execution: For each measured metric $e \in \mathbf{E}$, summing up its values along all TL-CUEs results in the total metric value for all useful execution activities. Subtracting $\text{GCUE}(e)$ from it results in the metric's value for all useful execution activities *outside* the G-CUE. This value strongly depends on the amount of activities outside the G-CUE and consequently on the amount of threads in the program's execution. Therefore, it does not allow for a straightforward comparison with the G-CUE since every activity on the G-CUE is executed by exactly one thread. For an execution of n many threads, this leaves $n - 1$ threads outside the G-CUE at every point in time in the program activity graph that is restricted to useful execution. By dividing the metric value for activities outside the G-CUE by $n - 1$, we thus get the mean metric value per thread outside the G-CUE which is much better suited for a comparison. For a given event identifier $e \in \mathbf{E}$, this is defined as follows:

$$\text{avgOutsideGCUE}(e) := \frac{(\sum_{t \in \mathbf{T}} \text{TLCUE}(t, e)) - \text{GCUE}(e)}{|\mathbf{T}| - 1}, \text{ if } |\mathbf{T}| \geq 2$$

If $\mathbf{T} = 1$, the G-CUE simply corresponds to all useful execution within the program's execution, so no outside useful execution activities exist. In the following, the functions GCUE, TLCUE (with individual threads $t \in \mathbf{T}$ already applied to

form partially applied functions $\text{TLCUE}_t : \mathbf{E} \mapsto \mathbb{R}_{\geq 0}$), and avgOutsideGCUE are called *accumulators*.

4.4. Derived Metrics

Thus far, we were only concerned with the comparison of absolute event values accumulated along different parts of the execution. The analysis of such values makes sense for durations and some event counts, but in many cases, event counts depend on factors that are primarily reflected by other events. A simple example is the number of branch mispredictions (see Section 5.2): If two program activities execute the same loop and the same code per iteration with the same inputs and outputs, but the activities execute a different number of iterations, the activity with more iterations logically has an increased likelihood of branch mispredictions simply because it is performing more of the same work³. In order to compare event counts independent of the amount of an abstract notion of executed “work”, we want to relate some event counts to each other in the form of ratios. For our prior example, we would divide the number of branch mispredictions by the number of executed conditional branch instructions—the resulting value is adjusted for the amount of executed work and indicates *how well* that work is executed. Following common terminology, metrics defined by means of such ratios are henceforth called *derived metrics*. The specific derived metrics used in the work of this thesis are introduced in Section 5.2.

The accumulator functions are also defined for derived metrics: Given the formula of a derived metric and an accumulator, the accumulator’s value for that derived metric is obtained by substituting each metric in the formula by the result of its accumulation with the same accumulator. This is best understood by an example: Given a derived metric d informally defined as $d := \frac{e_1 + e_2}{e_3}$ for three event identifiers $e_1, e_2, e_3 \in \mathbf{E}$, the derived metrics’ value accumulated along the TL-CUE of a thread $t \in \mathbf{T}$ is calculated as

$$\text{TLCUE}(t, d) = \text{TLCUE}_t(d) = \frac{\text{TLCUE}_t(e_1) + \text{TLCUE}_t(e_2)}{\text{TLCUE}_t(e_3)}.$$

Note that it is only statistically meaningful to calculate the value of a derived metric if all metrics included in its formula are sourced from the same execution.

4.5. Analysis Workflow

Given the previous definitions, it is finally possible to introduce the analysis workflow of the methodology. The workflow follows a top-down approach that attempts

³ That is, of course, only if the loop’s code causes any branch mispredictions in the first place.

4. Analyzing Event Counts Along Critical Paths

to locate potential performance issues and reasons for the G-CUE’s structure by following a hierarchical chain of potential performance factors. A broad overview of the workflow is illustrated in Figure 4.1. While the choice of measured events generally depends on the hardware and the evaluated application, there are a few concrete analysis steps common to all evaluated applications that are presented in the following. Note that the conclusions made within the individual analysis steps do not claim totality, as several aspects may concurrently affect application performance.

The first analysis step is a comparison between the accumulated number of CPU clock cycles. If the duration of the G-CUE is found to be close to the range of individual TL-CUEs, an analysis of the application would likely be similarly useful if done without specifically considering critical paths. If, however, the G-CUE takes much more time than any particular TL-CUE, there must be serial chains of useful execution activities on multiple threads that leave other threads waiting for a significant amount of time. Therefore, a comparison of the G-CUE duration with

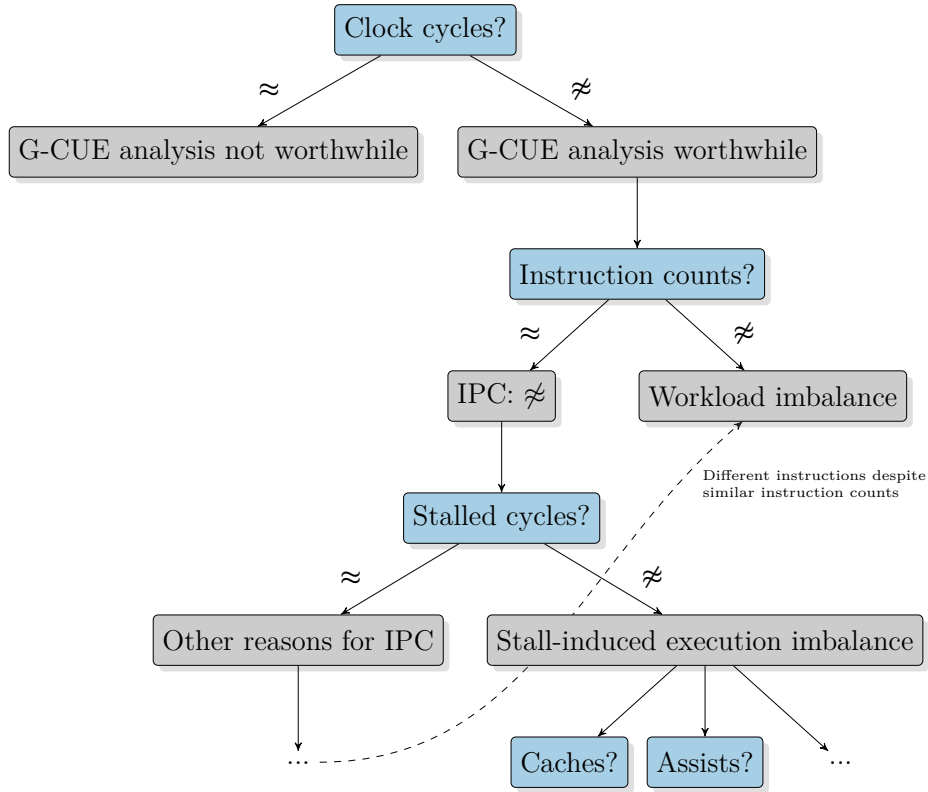


Figure 4.1.: The hierarchical workflow used in the methodology. Blue nodes represent analysis steps, gray nodes represent conclusions. The symbols “≈” and “≠” correspond to the comparison between the G-CUE and all TL-CUEs. “≈” denotes that the G-CUE’s values are similar to the range of TL-CUE values, “≠” denotes the opposite. Note that the decisions embodied by the tree are not exclusive and only show tendencies as multiple aspects may determine the G-CUE.

TL-CUE durations is useful to detect serialization-related performance issues on the structural performance level.

Note that a comparison between accumulated durations can be deceiving: Even if the G-CUE takes a similar amount of clock cycles than another thread, it could execute totally different code, potentially resulting in vastly different counts for various other events. In this situation, a naïve and wrong interpretation that is unaware of the executed code could come to the conclusion that the differing event counts must have no effect on performance in general. However, it is well possible that the event counts reflect significant performance factors that are simply masked by a difference in executed instructions.

Although the structure of the G-CUE and other execution paths is not known, it is possible to get a rough approximation of the executed code by means of instruction counts. This constitutes the second analysis step. While there usually is no practical way of separately counting *all* individual types of executed instructions, modern CPUs provide events to count the overall amount of executed instructions as well as certain groups of instructions such as memory-accessing instructions or floating-point instructions. These counts provide a rough estimate of the work executed from the perspective of the program’s source code and are thus another potential indicator of structural performance issues: If there is a significant difference between these counts along the G-CUE and for individual TL-CUEs or `avgOutsideGCUE`, and if the G-CUE takes significantly more time to execute than the other activities, this indicates an imbalance in the distribution of work along the program’s activities. If there is no such difference but there still is a difference in duration, it may either be attributed to a different number of instruction types that are not counted or to side effects of the executed instructions such as cache misses or branch mispredictions.

The combined effects of instruction counts and clock cycles are visible in the derived instructions per cycle (IPC) metric, which indicates the efficiency of the execution of instructions. Note that a comparison of the IPC metric between different program activities or execution paths is only meaningful if both sides of the comparison execute roughly⁴ the same instructions, as different instructions have different latencies. This condition cannot be guaranteed only given instruction counts, but a close similarity of instruction counts is a necessary precondition.

The number of stalled cycles is a further step in the hierarchical chain of potential performance reasons if the IPC significantly differs between the G-CUE and individual TL-CUEs. It thus constitutes the third step of analysis. If the difference in IPC is also visible in the number of stalled cycles, the IPC is affected by CPU pipeline stalls that delay the processor’s execution of instructions. If no such difference is visible in the amount of stalls, the reasons for IPC differences are found elsewhere.

⁴ To achieve perfect comparability, the exact same instructions need to be executed on both sides of the comparison. However, small differences in the executed instructions only cause small differences in the IPC metric, so a rough similarity is sufficient in our use case.

4. Analyzing Event Counts Along Critical Paths

An example for this are differences in executed instructions that are not visible in the measured instruction counts.

The subsequent analysis steps are highly explorative and application-specific. It is generally advisable to measure a broad range of event counts to ensure that different kinds of performance issues can be noticed, and depending on the evaluated application, specific fine-grained measurements might be necessary. As with the previous step, the focus lies on finding or eliminating potential reasons for performance issues along the G-CUE, combined with a comparison of the G-CUE's metrics to individual threads and outside activities in order to get a better understanding of the measured values. For the experiments conducted in this thesis, two sets of hardware performance events were counted in order to cover common performance factors such as cache misses or branch mispredictions. The specific events are explained in the following chapter.

4.6. Adaptation for Other Tools

In addition to developing and evaluating a methodology for the analysis of hardware performance counters tracked along critical paths, this thesis investigates whether the methodology, which is based on OTF-CPT, is also applicable to similar performance analysis tools. This section discusses the conditions of such an application as well as potential modifications to the methodology. Section 4.6.1 specifically investigates the applicability of the methodology in Scalasca.

A general answer to the question of applicability can rather trivially be inferred from the methodology's description in this chapter: A performance analysis tool must be capable to accumulate hardware performance counter measurements along the G-CUE and the TL-CUEs of an application. As long as this is possible, the methodology can be applied *as-is* on top of the tool's measurements. However, critical paths of useful execution are a concept of OTF-CPT and unlikely to be available in other tools. A generalization of the methodology to other kinds of critical paths, which is discussed in the following, could be more useful for other tools.

At this point, we need to recall that the G-CUE is a critical path through a program activity graph that is restricted to useful execution activities. It therefore resembles the critical path for a program execution where communication is instantaneous. Instead of reconstructing the G-CUE, it is more likely that a given tool reconstructs the critical path through the complete unrestricted program activity graph of an execution. This particular critical path, which resembles the critical path of the entire execution, will be abbreviated as *CP* in the following.

By construction, the CP also includes communication-related activities. This is desirable in many cases, but it might be disadvantageous for the analysis of hardware performance counters using the methodology: The methodology is unable to dif-

ferentiate between individual activities and therefore cannot differentiate between useful execution and communication. Not only does this limit the usefulness of the results, but communication-related code is usually part of parallel runtimes such as MPI or OpenMP—its performance is mostly outside the reach of parallel software developers.

This issue may be dealt with by limiting the activities on the CP to useful execution activities *after* the CP is reconstructed (given that the used performance analysis tool supports this feature). Note that this limited execution path, which is called CP_{UE} in the following and does not constitute a proper critical path, is not generally equal to the G-CUE: The G-CUE is a critical path through a program activity graph that is restricted to useful execution activities, whereas CP_{UE} is the result of limiting the critical path through a complete program activity graph to useful execution activities *after* the critical path has already been reconstructed. Unlike the G-CUE, CP_{UE} is affected by communication-related activities even though they are not part of its structure.

Independent of whether the CP is used as-is or limited to useful execution post-construction, its properties affect the interpretation of analysis results: As long as the structure of the observed critical path matches the structure of the true critical path (see Section 3.2), both the CP and CP_{UE} are guaranteed to only include program activities that actually determine the run time of the evaluated execution. This is not generally the case for the G-CUE because it only considers an execution isolated from communication. However, the proposed top-down approach to identify potential causes for the critical path would not work reliably for CP_{UE} , as its performance factors are not restricted to its activities.

It is also possible to only consider one of the G-CUE, the CP, or CP_{UE} for the methodology without comparing the metrics along this path to metrics outside of it (in the sense of individual TL-CUEs or `avgOutsideGCUE`). In this case, the usefulness and the expressiveness of the methodology suffers as the impact of individual metrics is more difficult to interpret from the G-CUE or other execution paths alone. In this scenario, an analysis concerned with identifying potential causes for the evaluated critical path would no longer be possible.

4.6.1. Score-P & Scalasca

The Scalasca toolset constitutes a potential candidate for the application of the methodology, as it is able to collect hardware event counts for program regions and to reconstruct the CP. The following discusses whether such an application is possible.

Scalasca makes use of Score-P to perform its measurements. Score-P creates execution traces in the *OTF2* format [52], containing counts of selected hardware events for each individual program activity. It also accumulates metrics such as

4. Analyzing Event Counts Along Critical Paths

event counts over time, as part of thread- or process-specific region profiles. Here, the term “region” typically corresponds to a function but can also be a user-defined region in the source code. For each metric and each thread or process in the profile, a single value is reported per region in an execution’s call graph: If every region `foo` takes three seconds to execute and `foo` is executed four times within the same context of the call graph, the reported profile value for the metric describing the region’s duration amounts to twelve seconds. Metrics can be accumulated in an exclusive way, in which case only the region itself, but no region executed within, is included in the metric, or in an inclusive way, in which case the metric recursively includes all metric values of regions executed within.

Accumulating hardware event counts along a TL-CUE or along all activities within a thread is possible from Score-P’s output, as all the necessary data is available in the execution traces and could also be collected from the profiles: Since a profile is thread-specific, the root region that encompasses all other regions can be used to obtain the total metric value over all regions (and therefore over all activities) within a thread. Obtaining TL-CUE values from profiles is more complicated: Here, individual inclusive or exclusive event counts throughout the call graph need to be added together in order to accumulate event counts for useful execution only. This is not straightforward but possible, given a definition of which regions are unrelated to useful execution and thus need to be excluded from the summation.

Unlike OTF-CPT, which does not track the structure of its critical path variants, Scalasca reconstructs the CP and is internally aware of its structure. However, Scalasca does not directly expose the CP’s detailed structure (i.e., individual activities) to the user. It only reports various time-related profile metrics related to critical paths [53, 12], following the same principles as event count metrics. One of these metrics is the *Critical path (sec)* metric, which summarizes how much time all instances of a region spent on the critical path [53, *Critical Path Profile*]. While this metric exposes the importance of a region with respect to the critical path, it and the other critical path metrics are not relatable to measured event counts: It is not clear how many events happened during the region instances *on the critical path* as compared to the region instances outside of it. As a consequence, the methodology currently cannot be used with Scalasca’s output only.

However, it is still possible to use the methodology on top of the output provided by Score-P, given a manual reconstruction of the G-CUE, CP, or CP_{UE} from the execution traces. The obtained critical path structure can then be used to accumulate event counts along the critical path from the data contained in the execution traces.

Alternatively, it is possible to implement the methodology directly within Scalasca, as it has access to all the necessary data and is already internally aware of the CP. For example, it would be possible to provide region profiles specific to the critical path for different kinds of metrics such as event counts. For every region in the execution’s call graph, the profile would specify the number of events that happened while the region was executed on the critical path. By reading the metrics associated with

the program's entry point, one can then get the accumulated event count for the entire critical path. In addition, Scalasca could also expose the complete detailed structure of the critical path in a separate output file. This could be used by other tools, for example for visualization in *Vampir* [54] similar to how it has already been done by Schmitt et al. [9].

Using Scalasca for the methodology would allow to map the analysis results back to individual regions in the source code of the evaluated application. This is not possible using OTF-CPT, as it does not track the structure of the critical path.

5. Experimental Setup

This chapter presents the practical implementation of the methodology introduced in the previous chapter. It contains an overview of the hardware and software environment used for the evaluation (Section 5.1) and explains the hardware counters measured for the evaluation as well as the derived metrics calculated from them (Section 5.2). Section 5.3 introduces the applications used for testing the methodology.

5.1. Hardware and Software Environment

The experiments conducted as part of this thesis were carried out on the RWTH Aachen Universities’ compute cluster *CLAIX-2023*. Each compute node within the cluster’s HPC segment is equipped with two Intel Xeon 8468 Sapphire Rapids CPUs. Each CPU has 48 cores with a base frequency of 2.1 GHz [55], resulting in a total of 96 CPU cores per compute node. All experiments ran on nodes with 256 GB of memory per complete node. Measurements were stored on HDDs on a Lustre-based file system with 500 GB/s read and write bandwidth [56].

To request computation resources from the cluster, the *Slurm* workload manager [57, 58] was used. Every experiment requested resources using Slurm’s `--exclusive` flag to get exclusive access to an entire compute node in order to prevent interference with other jobs running on the cluster. The experiments also requested to run with the maximum CPU frequency, using Slurm’s `--cpu-freq=high-high:performance` flag. The output of Slurm’s `sacct` command verifies that the flag is correctly set, but the experiments show that the request is not always respected by the cluster (see Section 6.2). Slurm’s documentation states that the flag is “treated as a request, not a requirement” [58, *sbatch* and *srun*]. `sacct` lists the average CPU frequency per Slurm job, but individual experiment configurations are executed as part of non-overlapping *job steps* for which the frequency is not reported on CLAIX-2023. Attempts to manually trace the CPU frequency did not succeed due to missing permissions on the cluster.

At the time of conducting the experiments, CLAIX-2023 ran on *Rocky Linux* 9.6 using kernel version 5.14.0. For all applications, the environment variable `OMP_PLACES` was set to `cores` and `OMP_PROC_BIND` was set to `close`. This ensures that every OpenMP thread is executed on its own core and that all n threads are assigned to the first n cores of the allocated hardware [50].

5.2. Measured Hardware Events & Dependent Metrics

During the measurements performed with OTF-CPT and Scalasca, various hardware performance events were counted. After each application run, the raw event counts were collected by a post-processing script in order to calculate the execution’s derived metrics.

The Intel Sapphire Rapids CPUs used by CLAIX-2023 expose four different fixed hardware performance counters for counting predefined hardware events, as well as eight programmable counters that can be assigned events from a comprehensive list of available performance events (see [42]). The measurements are limited to *core events* (see Section 3.1), as these are specific to individual cores and—given the OpenMP configuration mentioned in Section 5.1—individual threads. Consequently, every measurement along a critical path variant only includes event counts that actually happened as part of the corresponding activities, unlike *uncore events* that would also include events happening outside the examined activities. Note that even core events reflect hardware behavior that may depend on other threads and cores. For instance, the cache behavior with regard to a thread or process generally depends on the cache use of other threads and processes. *Offcore events* are also counted per-core and thus suitable for the critical path analysis, but their use was not permitted for the thesis project on CLAIX-2023.

Both OTF-CPT and Scalasca (which is also used in the experiments of this thesis; see Section 5.3) do not support multiplexed measurements of hardware performance counters. Therefore, the measurements were repeated with two sets of events shown in Table 5.1. The sets include events for counting clock cycles (including stalled cycles), various instruction counts, and more specific events related to caches, branch prediction behavior, and microcode assists. The listed event identifiers follow the naming found in [42].

The remainder of this section explains the measured events in more detail and introduces the derived metrics computed for the evaluated applications. Recall that derived metrics are only calculated for a given program execution if all metrics included in the derived metric’s formula are measured as part of the same execution.

Instruction counts. As motivated in Chapter 4, instruction counts approximate the amount of work performed by the CPU. They are analyzed individually but also used as part of many derived metrics presented in this chapter.

Intel Sapphire Rapids CPUs provide the hardware event `INST_RETIRED.ANY_P` to count the number of all *retired* instructions [42]. A retired instruction is an instruction that was fully executed and whose result is “committed to architectural state” [40] (i.e., it is visible to the executing program), as compared to instructions that are not executed or executed due to speculative execution following a mispre-

Set 1	Set 2
INST_RETIREDD.ANY_P ¹	INST_RETIREDD.ANY_P ¹
L1D.REPLACEMENT	ASSISTS.ANY
MEM_INST_RETIREDD.ANY	ASSISTS.PAGE_FAULT
L2_RQSTS.ALL_DEMAND_MISS	ASSISTS.FP
L2_RQSTS.ALL_DEMAND_REFERENCES	FP_ARITH_INST_RETIREDD.SCALAR
LONGEST_LAT_CACHE.MISS	FP_ARITH_INST_RETIREDD.VECTOR
LONGEST_LAT_CACHE.REFERENCE	BR_INST_RETIREDD.COND
CYCLE_ACTIVITY.STALLS_TOTAL	BR_MISP_RETIREDD.COND
CPU_CLK_UNHALTED.THREAD_P ¹	

Table 5.1.: The two sets of Intel hardware performance events counted during the experiments.

diction (see p. 36). There exists no core event on Intel Sapphire Rapids CPUs that counts all executed instructions independent of retirement.

In addition to the number of total retired instructions, several events are measured that are dedicated to specific types of retired instructions (see [42]):

- MEM_INST_RETIREDD.ANY counts the number of retired load and store instructions.
- FP_ARITH_INST_RETIREDD.SCALAR counts the number of scalar floating-point instructions that retired.
- FP_ARITH_INST_RETIREDD.VECTOR counts the number of vector floating-point instructions that retired.
- BR_INST_RETIREDD.COND counts the number of retired conditional branch instructions.

Instructions per cycle. Modern microprocessors are *superscalar*, meaning that they are able to fetch, decode, and execute multiple instructions in parallel [59]. The *IPC* metric (*Instructions Per Cycle*) measures the degree of this instruction level parallelism and reflects how fast the CPU is able to run through the incoming instruction stream.

This thesis computes the IPC as the ratio of retired CPU instructions per unhaltd clock cycle:

$$\frac{\text{INST_RETIREDD.ANY}}{\text{CPU_CLK_UNHALTED.THREAD}}$$

¹ It appears as if PAPI—or some other component on a lower level—mistakenly treats fixed hardware counters as programmable and vice versa on CLAIX-2023. For this reason, the programmable variants ending with “_P” are used for fixed events to avoid PAPI errors due to too many programmable counters being used.

Stalled cycles. Pipeline stalls delay the execution of instructions and are thus a potential reason for a low IPC. The percentage of stalled clock cycles (relative to all clock cycles) is computed as follows:

$$\frac{\text{CYCLE_ACTIVITY.STALLS_TOTAL}}{\text{CPU_CLK_UNHALTED.THREAD}}$$

Branch mispredictions. In processor architectures using instruction pipelining, different instructions are fetched, decoded, and executed in parallel. In case of conditional branch instructions, the condition’s value determines the next instructions, so the CPU needs to wait for the execution of the branch instruction before safely fetching the next instruction. This causes idling, so modern processors employ *speculative execution*: They try to predict the condition’s value and speculatively execute the next few instructions accordingly in order to ensure a fully saturated pipeline. Once the condition’s result is available and if the prediction is incorrect, some instructions were wrongly executed. Their result needs to be discarded, and the pipeline needs to be flushed and restarted with the correct instructions, which can incur a cost of tens of clock cycles [60, 61].

In this thesis, the number of conditional branch mispredictions is measured in relation to the amount of conditional branch instructions, as already motivated in Chapter 4:

$$\frac{\text{BR_MISP_RETIRED.COND}}{\text{BR_INST_RETIRED.COND}}$$

Note that an accumulated value of this metric (e.g., accumulated along the G-CUE) may be difficult to interpret as it does not consider individual branches in the application’s code. For example, a branch misprediction rate of 20% could mean that every single branch is correctly predicted in 80% of cases, which is usually good, but it could also mean that 20% of all branches cause a misprediction in all cases and the rest causes no misprediction at all. Only if some branches are more often mispredicted than correctly predicted, there is a high potential for branch-related optimizations.

Cache misses. Computer memory is organized in a hierarchy of fast caches and main memory, with the highest-level cache (level 1) being the fastest and main memory being the slowest to access. When a program accesses memory, the processor first tries to serve that memory request from the level 1 (L1) cache. If memory is not found within a cache, a *cache miss* occurs and the request is passed to the next level in the memory hierarchy. Due to the different speeds of cache levels and main memory and the additional passing and handling of requests between levels, cache misses can be a major influence for application performance. Therefore, the selected sets of measured hardware events contain events specifically related to cache misses on all three levels of caches found in Sapphire Rapids CPUs.

For each cache level, the amount of cache misses is divided by the amount of retired memory-accessing instructions:

$$\begin{aligned} & \frac{\text{L1D.REPLACEMENT}}{\text{MEM_INST_RETIRED.ANY}} && \text{(level 1, data)} \\ & \frac{\text{L2_RQSTS.ALL_DEMAND_MISS}}{\text{MEM_INST_RETIRED.ANY}} && \text{(level 2)} \\ & \frac{\text{LONGEST_LAT_CACHE.MISS}}{\text{MEM_INST_RETIRED.ANY}} && \text{(level 3)} \end{aligned}$$

The measurements for L1 caches only incorporate the L1 *data* cache (L1D), which is separate from the L1 *instruction* cache (L1I) that is not measured for the work of this thesis. Since there is no available event for measuring all L1 data cache misses on Intel Sapphire Rapids CPUs², the number of replaced cache lines is used instead as a rough approximation. Measurements for level 2 and 3 caches (L2 and L3, respectively) incorporate misses for both instructions and data accesses since the caches on these levels are *unified* and store both data and instructions.

The derived metrics shown above are relatively well-suited for comparisons between the G-CUE and other application parts as they divide by the amount of retired instructions. L2 and L3 cache miss rates that relate cache misses to cache accesses depend on the amount of cache misses on lower levels and are therefore not suited for use in the methodology.

In addition to the number of cache *misses*, the numbers of L2 and L3 cache *accesses* (L2_RQSTS.ALL_DEMAND_REFERENCES and LONGEST_LAT_CACHE.REFERENCE, respectively) were measured as well. These are used to compute the number of cache hits by subtracting the number of misses from the number of accesses. Note that these values on their own are also not useful for comparisons between the G-CUE and other applications, but by comparing hits on *all* levels on the memory hierarchy (including memory hits computed as the number of L3 misses), the distribution of memory accesses on all levels of the memory hierarchy can be better understood.

Microcode assists. Some modern microprocessors make use of special microcode routines to handle specific situations that are not fully implemented in the hardware’s regular execution path. Intel calls these routines *assists* [40]. Microcode assists can be extremely costly depending on the CPU and on the exact circumstances [40, 60, 62] and are therefore included in the events measured in this thesis.

² Intel Sapphire Rapids CPUs provide MEM_LOAD_RETIRED.L1_MISS which only counts L1D cache misses related to load instructions. A thorough discussion of hardware events related to L1D caches can be found in this GitHub Gist by Travis Downs: <https://gist.github.com/travisdowns/90a588deaaa1b93559fe2b8510f2a739> (accessed: Dec. 10, 2025).

5. Experimental Setup

Intel Sapphire Rapids CPUs provide four hardware events for assists, three of which are measured for the work of this thesis and divided by instruction counts for comparability:

$$\frac{\text{ASSISTS.ANY}}{\text{INST_RETIRED.ANY_P}} \quad (\text{all assists})$$

$$\frac{\text{ASSISTS.PAGE_FAULT}}{\text{INST_RETIRED.ANY_P}} \quad (\text{page faults})$$

$$\frac{\text{ASSISTS.FP}}{\text{FP_ARITH_INST_RETIRED.SCALAR} + \text{FP_ARITH_INST_RETIRED.VECTOR}} \quad (\text{floating-point operations})$$

There is a fourth, undocumented hardware event `ASSISTS.SSE_AVX_MIX` that presumably counts the amount of assists of events due to the mix of SSE and AVX vector instructions. It is not separately measured for this thesis as it is unlikely to be relevant in the tested applications and already measured as part of the `ASSISTS.ANY` event, which may additionally include assists that are not countable by a dedicated event.

5.3. Evaluated Applications

Five different applications were tested as part of this thesis. The selection comprises applications using OpenMP or MPI, as well as a hybrid application (NEST) that makes use of both. The *Random Histogram* application (Section 5.3.1) was specifically written to provoke a distinct G-CUE in order to verify the measurement results. Two algorithmic benchmarks (*Mergesort* [Section 5.3.2] and *SPMxV* [Section 5.3.3]) were chosen to evaluate specific aspects discussed in the corresponding sections. The applications *DENISE Black Edition* (Section 5.3.4) and *NEST* (Section 5.3.5) are used to evaluate performed measurements under real-world conditions.

Each application was executed without any measurement tools as a *baseline* for comparison, with OTF-CPT to measure dependent metrics, and with Scalasca to collect further data used as background knowledge in the evaluation. Runs with OTF-CPT and Scalasca were mutually exclusive, i.e., no application run used both tools at the same time. All program executions were repeated multiple times in order to observe variance and to get meaningful results.

All applications as well as OTF-CPT were compiled with Intel’s *oneAPI* C/C++ compiler (version 2024.2.0). At least two variants were compiled per tested application: one for the baseline case and OTF-CPT, and one compiled with the `scorep` compiler wrapper to build an instrumented binary suitable for the analysis with Scalasca.

For both the Random Histogram application and the algorithmic benchmarks, the measurements performed with OTF-CPT and Scalasca are limited to the application’s main workload (its *kernel*). This is done to ignore initialization and finalization phases of the tested applications in order to focus on the performance of the application kernels.

Performance issues often vary with the degree of parallelization (i.e., the amount of threads or processes). Apart from the Random Histogram application whose evaluation does not consider scaling behavior, the tested applications were executed with varying amounts of threads or processes as part of scaling experiments.

5.3.1. Random Histogram

In order to test the proposed methodology as well as the validity of the output of OTF-CPT, a purposefully ill-balanced application was written. It computes a histogram based on a large array of pseudorandom integers and is therefore called—for the lack of a better name—*Random Histogram*. The application uses MPI and consists of four processes, each computing a local histogram based on values received from all processes of the application.

The implementation is outlined in Algorithm 1: At the beginning of the program, every process generates a local source array of pseudorandom numbers. The numbers correspond to indices of histogram bins and are therefore confined to the number of bins of the local histograms. On one process (process 0), the numbers are instead limited to values between zero and seven (both inclusive) such that only the first eight bins of a corresponding histogram can contain non-zero values. Since the indices are stored as unsigned 64-bit integers in the program, this corresponds to the size of a single level 1 data cache line (64 bytes on Intel Sapphire Rapids CPUs).

The application kernel is split into four almost identical stages: In every stage, each process sends its local source array to one other process in the program (including itself). The selection of the destination process follows a cyclic topology with offsets increasing per stage: For a stage with index $s \in \{0, \dots, 3\}$ and a sender process with index $p \in \{0, \dots, 3\}$, the index of the destination process is determined by $(p + s) \bmod 4$. Once a process receives an array of histogram indices, it increases the bins of its local histogram according to the values it received. After all stages are executed, each local histogram is the same as it corresponds to the numbers sent by all processes.

The imbalance of the range of random data between processes causes only one process per stage to access histogram memory outside a single level 1 data cache line. This is illustrated in Figure 5.1 for the first stage of the application. In each stage of application, the process whose memory access is not limited to a single cache line is expected to cause many more cache misses than any other process and therefore take longer to execute the current program activity. Consequently, it is

5. Experimental Setup

Input localProcId: the index of the process executing the code
Input allBounded: if true, force all arrays to contain small values only
Input noStageOffset: if true, always send the local data to the next process

```

1: procedure RANDOMHISTOGRAM(localProcId, allBounded, noStageOffset)
2:   ▷ Initialization
3:   if localProcId = 0 ∧ ¬allBounded then
4:     | modulo ← NUM_BINS
5:   else
6:     | modulo ← 8
7:   source ← INITARRAYRANDOM(SRC_LEN, modulo)
8:   histogram ← INITARRAYZERO(NUM_BINS)
9:   BARRIER()
10:
11:   ▷ Compute kernel
12:   for stage in {0, ..., 3} do
13:     if noStageOffset then
14:       | dstProcId ← (localProcId + 1) mod 4
15:     else
16:       | dstProcId ← (localProcId + stage) mod 4
17:     SENDARRAYTO(dstProcId, source)
18:
19:     recv ← RECEIVEARRAY()
20:     for v in recv do
21:       | histogram[v] ← histogram[v] + 1
22:
23:   BARRIER()
24:

```

Annotations:
 ▷ L1 data cache line fits eight 64-bit values
 ▷ four processes in total

Algorithm 1: Pseudocode implementation of the random histogram application.

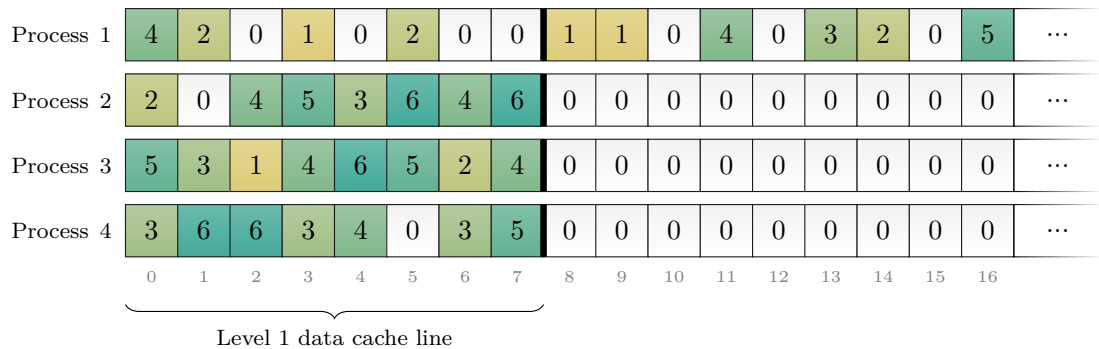


Figure 5.1.: Local histograms of all processes after the first stage of the Random Histogram application. For visualization purposes, the size of data sent between processes is heavily reduced.

expected that the G-CUE passes through all those slow program activities and shows a much higher number of cache misses throughout all levels of the cache hierarchy as compared to individual processes.

Algorithm 1 includes two input variables *allBounded* and *noStageOffset* which are used to configure the imbalance of the application. If *allBounded* is true, all source arrays are generated with random numbers in $\{0, \dots, 7\}$ such that all non-zero bins of every histogram are contained within a single L1 data cache line. If *noStageOffset* is true, the formula for determining each destination process is changed to $(p + 1) \bmod 4$, effectively causing only process 2 to perform all the cache-heavy work.

The application is compiled with the `-O3` optimization flag. The size of individual source arrays is configured as 128 MiB (2^{24} elements of type `uint64_t`) and the size of histograms is configured as 2 GiB (2^{28} bins of type `uint64_t`).

5.3.2. Mergesort

The *Mergesort* application is a simple multithreaded implementation of the merge-sort algorithm written in C++. It makes use of OpenMP tasking, which is why it was chosen to be evaluated as part of this thesis.

On (almost) every level of recursion in the algorithm, two OpenMP tasks are created, each delegated to sort one half of the unsorted array. This only happens as long as the array length is greater than a fixed predefined cutoff length; when the array has fewer elements than the cutoff length, the remaining work is executed in serial. In this case, no more tasks are created to avoid high overhead due to assigning very small amounts of work to the tasks. The merge operation on each recursive level follows the same principle: Both halves are merged in two parallel tasks starting at a certain cutoff length, below which a serial implementation is used. After each creation of two tasks, an OpenMP `taskwait` construct is inserted in the code to wait for the tasks on the current level of recursion.

The experiments conducted as part of this thesis use an input array of 300 million `int` elements (4 bytes each; about 1144 MiB in total) which is filled with random numbers in the interval $[0; 300,000,000]$. The application is executed with a varying numbers of OpenMP threads (1, 2, 6, 12, 24, 48, 96). The cutoff length for both sorting and merging is set to 8192 elements.

5.3.3. Sparse Matrix-Vector Multiplication

The *SPMxV* benchmark (short for SParse Matrix-Vector multiplication, with an “x” symbolizing the multiplication) is an OpenMP implementation of the matrix-vector multiplication

$$\vec{y} \leftarrow A\vec{x},$$

5. Experimental Setup

where A is a *sparse* matrix (i.e., it contains mostly zeroes). Zeroes in A do not affect the value of $\vec{y}$, so—given a sufficiently large matrix—it is crucial for performance to skip those values while performing the multiplication.

The implementation used in the context of this thesis is a slightly adapted version of the SPMxV code of the *Performance Optimization and Productivity Centre of Excellence in HPC* [63]. The code makes use of *Compressed Row Storage* (CRS), which stores the non-zero values for A in an array v such that all non-zero values are stored contiguously in memory in row-major order. The column index of every non-zero matrix value is stored in another array c as this is needed for multiplication with $\vec{x}$. A third array r contains the indices of v and c that start each row, which is needed to store the results in $\vec{y}$. The computation of a single row i of $\vec{y}$ then looks as follows:

$$\vec{y}(i) = \sum_{j=r(i)}^{r(i+1)-1} v(j) \cdot \vec{x}(c(j)) \quad (5.1)$$

To parallelize the matrix multiplication, SPMxV makes use of the fact that the individual rows of $\vec{y}$ can be computed independently of each other. It distributes an equal amount of rows among all threads of the application by means of a for loop in a parallel region using `#pragma omp for schedule(static)`.

Due to the nature of the computation, the SPMxV implementation theoretically achieves perfect CPU cache use for any access to v , c , r , and $\vec{y}$. However, the memory access pattern of $\vec{x}$ depends on the column indices of the non-zero values of A (as visible in Equation 5.1) and therefore causes cache misses depending on the structure of the matrix. In the worst case, every single read from $\vec{x}$ causes a cache miss.

The application is executed with a matrix of dimension $59,319 \times 59,319$ (3,518,743,761 values) with 853,747 non-zero values (ca. 0.024%). The structure of the matrix is shown in Figure 5.2. In order to measure a steady-state computation, the multiplication is repeated 100,000 times. The kernel’s source code is provided in Appendix A.

The application is compiled with the `-O3` compiler optimization flag. To observe performance behavior with varying numbers of threads, the application is executed with 1, 2, 8, 32, and 96 OpenMP threads.

5.3.4. DENISE Black Edition

DENISE Black Edition [64] (often shortened to *DENISE* in this thesis) is an open source software simulating the time-domain propagation of elastic waves through a two-dimensional medium. It is written in C and makes use of MPI for parallelization. The program works by partitioning the two-dimensional and discretized problem domain across MPI processes, each of which is responsible for a chunk of

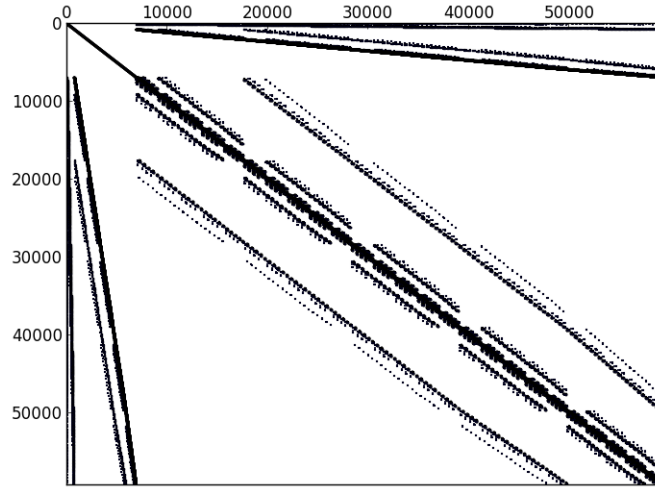


Figure 5.2.: Structure of the sparse matrix used for the SPMxV experiments. Black dots represent non-zero values.

the domain [65]. For the experiments, DENISE simulates a model of two *shots* (wave/energy sources) creating waves in blocks of concrete.

Prior research concluded that the performance of DENISE suffers from costly operations with *subnormal* floating-point numbers in some cases [66, 67, 68]. The *IEEE Standard for Floating-Point Arithmetic* (IEEE 754-2008; [69]) defines several floating-point formats, each including a representation for a standard range of floating-point values (*normal numbers*). To extend this range for the representation of tiny values with reduced precision, the standard defines a set of *subnormal numbers* for each format which consists of non-zero numbers whose magnitudes are smaller than the magnitude of the smallest normal number in the format [69]. Subnormal numbers differ from normal numbers in their encoding and as such need special handling in mathematical operations. In many CPUs, this is not implemented in hardware but instead relies on microcode assists (see Section 5.2) that are executed when needed and may take up to a few hundred clock cycles per operation depending on the CPU and on the exact circumstances [60, 62].

DENISE was specifically chosen to be evaluated as part of the thesis in order to analyze the critical path with respect to microcode assists, and to evaluate whether new insights can be gained from the analysis.

The code region measured in the conducted experiments is defined to enclose the entire workload of the application. It starts directly after the initial MPI barrier in line 104 of `src/denise.c` and ends directly before the call to `MPI_Comm_free` in line 148 of `src/denise.c`. The DENISE builds are based on commit `b07d81d` and were compiled with `mpicc` and `-O3`. The builds used in the initial experiments explicitly disable FTZ and DAZ modes (see Section 6.4.1) by passing `-no-ftz` to the compiler to ensure that subnormal floating-point numbers can occur in the evaluated

5. Experimental Setup

executions. Additional builds used in Section 6.4.1 pass `-ftz` instead to enable FTZ and DAZ modes. Minimal changes have been applied to the source code which most likely do not affect the measurements; this was done in order to fix compilation and to restore the `stdout` stream for the final output which inadvertently gets closed at many places throughout the source code.

The experiments are executed with 1×1 , 2×3 , 4×3 , 8×4 , and 8×8 MPI tasks.

5.3.5. NEST Simulator

The *NEST Simulator* [70] (*NEST* for short) simulates large neural network models in parallel using OpenMP and MPI. Neural networks in NEST consist of *nodes*, most of which are representing individual neurons, and *connections* between them to transmit *events* [71]. Some special-purpose nodes called *devices* do not represent neurons but are instead used for other tasks such as generating signals and injecting spikes into the network, or measuring network state [71].

NEST distributes all nodes to *virtual processes*, which correspond to the individual threads within MPI processes [71] (e.g., if the application is executed with eight threads per process and four MPI processes, it has 32 virtual processes). NEST discretizes the simulated time into even time steps (following [71]): In every time step, incoming node events are delivered to the nodes which then update their state. This is executed in parallel over all virtual processes. During the update of node states, nodes may create new events for other nodes which are exchanged after the parallel updates before continuing with the next simulation step.

The experiments in this thesis use NEST 3.9 [72] and execute NEST’s HPC benchmark example (`pynest/examples/hpc_benchmark.py`; also see [73]) which creates a balanced random neural network where each neuron follows the `iaf_psc_alpha` model [74]. The benchmark is written using NEST’s Python bindings *PyNEST* [75]; therefore, Scalasca measurements are initiated as shown in Listing 5.1.

```
1 python3 -m scorep --noinstrumenter --thread=omp:ompt --mpp=mpi "${NEST_DIR}/  
  nest-simulator/pynest/examples/hpc_benchmark.py"
```

Listing 5.1: Invocation of the Score-P Python module for NEST.

Table 5.2 shows the configuration of the NEST HPC benchmark as used in the experiments of this thesis. The benchmark is executed with a varying number of MPI tasks and an inversely proportional amount of OpenMP threads, so the total amount of virtual processes stays constant at 96. The two *simulated time* parameters were slightly altered to be multiples of 1.5 ms to avoid warnings of the NEST runtime.

Parameter	Values
Number of neurons	112,500 (90,000 excitatory, 22,500 inhibitory)
Simulated time (until equilibrium)	51 ms
Simulated time (actual simulation)	250.5 ms
Number of MPI tasks	1, 2, 4, 8, 12, 24, 48, 96
Number of OpenMP threads	96/<number of MPI tasks>

Table 5.2.: Configuration of the NEST HPC benchmark as used in the experiments of this thesis. Only parameters which are changed from the implementation's defaults are included in this table, all others are kept as-is.

6. Evaluation

This chapter presents the analysis results for the evaluated applications.

6.1. Random Histogram

The *Random Histogram* application was designed to provoke a large number of cache misses along the critical path. This was done in order to test whether the methodology is able to locate simple critical path imbalances.

In the application's kernel, the useful execution of each process is concerned with sorting the pseudorandom values it received from another process into its local histogram. Within each of the four kernel stages, one process receives an array whose values vastly exceed the L3 cache size. The useful execution activity of this process, which henceforth will be called an *unbounded activity*, needs to access a large range of memory in order to increase the counts in its histogram bins. It is therefore expected to cause a great amount of capacity-related cache misses throughout all levels of the cache hierarchy. All other processes within a stage receive values so small that they only access a tiny amount of memory that fits into a single L1 cache line. Their useful execution activities, which are called *bounded activities* in the following, are expected to not cause any capacity-related cache misses.

A greater number of cache misses in a program activity likely increases the number of stalled clock cycles, which in turn requires more cycles to execute the activity. Because of the stark difference in cache misses between unbounded and bounded activities, unbounded activities are thus expected to take much longer to execute. In turn, the G-CUE is expected to pass through all unbounded activities of the application. Individual processes only execute a single unbounded activity and three bounded ones, so in comparison to them, the G-CUE should show the following properties:

- The G-CUE is expected to exhibit higher cache rates through all levels of the cache hierarchy.
- The G-CUE is expected to exhibit a higher percentage of stalled cycles.
- Since bounded and unbounded activities execute the same code, the G-CUE is expected to have a lower IPC metric.
- The G-CUE is expected to need many more clock cycles.

6. Evaluation

In this specific case, the G-CUE is so distinct from other execution paths that its structure can be trivially reconstructed from the measurements of OTF-CPT and Score-P alone. For all executed runs of the application, the sum of the measured TL-CUE durations over every combination of all but one thread¹ is always strictly lower than the G-CUE duration:

$$\forall t' \in \mathbf{T}: \sum_{t \in \mathbf{T} \setminus \{t'\}} \text{TLCUE}(t, \text{WallClockTime}) < \text{GCUE}(\text{WallClockTime}).$$

Therefore, even a hypothetical G-CUE that *sequentially* goes through the entirety of all but one process would not reach the duration of the actually observed G-CUE. From this it follows that the G-CUE observed by OTF-CPT must pass through all processes in the given program runs, even if this does not prove that it actually passes through all unbounded activities.

This, however, becomes clear when looking at a Score-P execution trace of the application kernel, which is illustrated in Figure 6.1 for a single run of the application. Every colored line corresponds to a cache-related performance metric derived from hardware performance counters. Within each program activity, the y-offset of each colored line segment represents the metric’s value for that program activity. This is a consequence of the fact that Score-P only reports counter values once at the boundary between two program activities: The displayed metric values are derived from differences of counter values between the two boundaries of an activity, so only a single value is known per activity. Activities with a gray background represent time spent within MPI routines; those with a white background represent useful execution (i.e., bounded and unbounded activities).

During any of the four stages of the application, the unbounded activity takes much more time to execute than all other bounded activities. While the unbounded activity is executing, all other processes spend most of their time waiting in a barrier (brighter shade of gray). Since the G-CUE is only able to transfer between processes during MPI communication, useful execution activities are generally uninterrupted on the critical path. Therefore, the critical path passes through the entirety of one bounded or unbounded activity within each stage. Since unbounded activities take much longer to execute, the G-CUE necessarily consists of the entirety of all four unbounded activities. While the execution trace only reflects the output of Score-P, the difference in execution time between bounded and unbounded activities is large enough to safely assume that the G-CUEs observed in OTF-CPT runs and the unobserved G-CUES in baseline runs are identical in structure.

Figure 6.2 shows five metrics measured within the application kernel: the amount of clock cycles, retired instructions (measured in both sets of performance counters and therefore available for $n = 16$ experiment runs), stalled clock cycles (abso-

¹ Recall that every MPI process in the Random Histogram application corresponds to exactly one thread, so the terms are used interchangeably here in order to apply to the definitions introduced in Chapter 4.

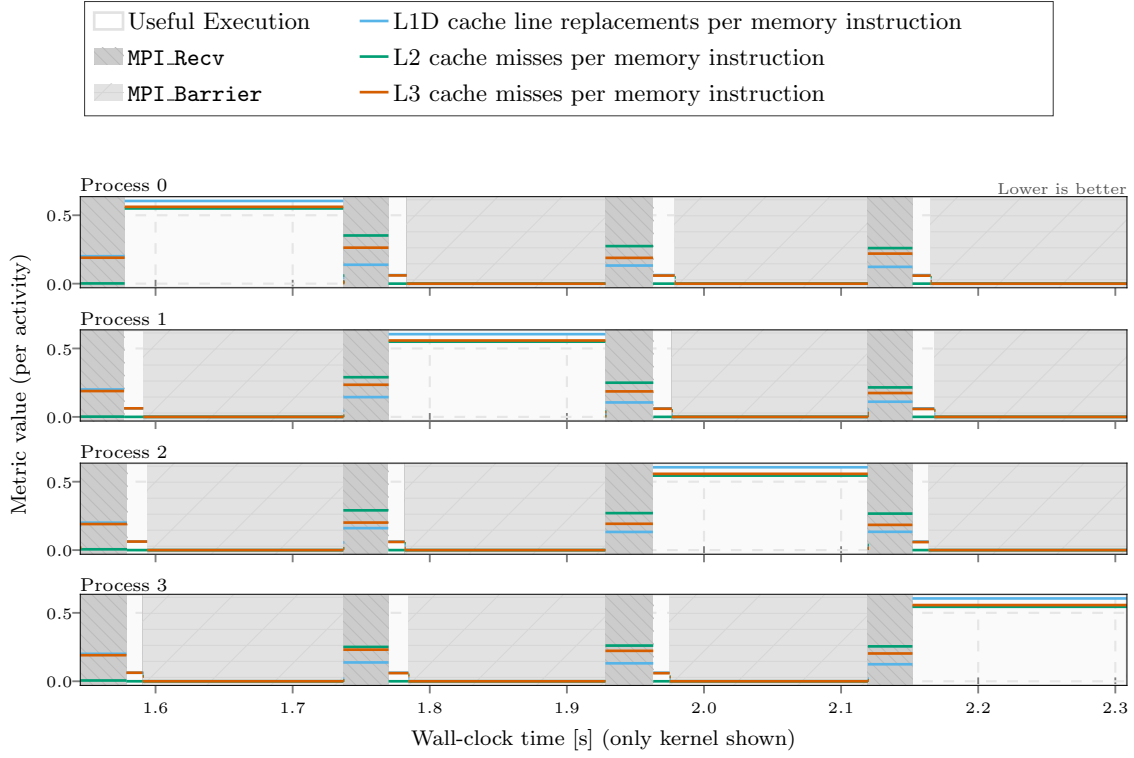


Figure 6.1.: Score-P execution trace of a single execution of the *Random Histogram* benchmark with four processes. MPI send operations are invisible in the trace as they are implemented by means of the non-blocking MPI_Isend operation that returns almost immediately.

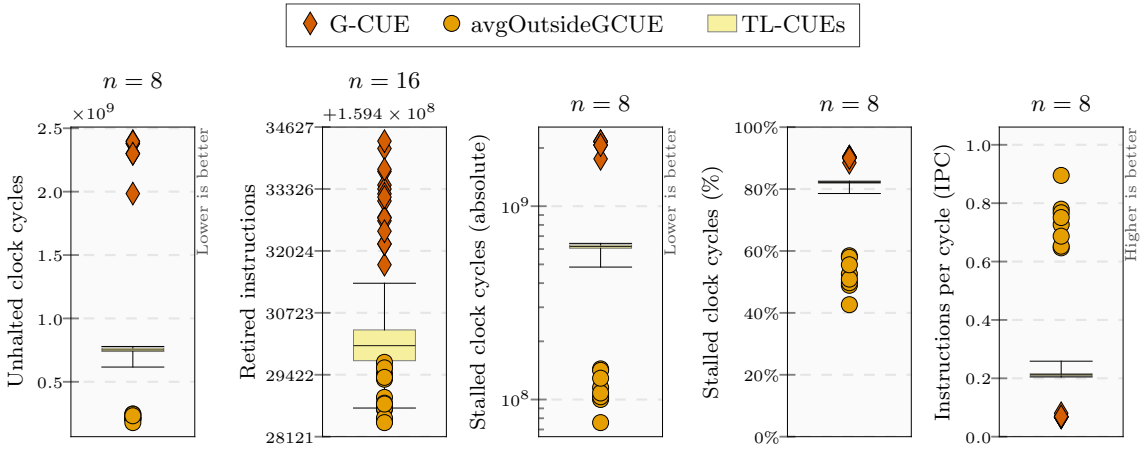


Figure 6.2.: Random Histogram: Unhalted clock cycles, retired instructions, stalled cycles, and the IPC, as measured by OTF-CPT for the kernel.

6. Evaluation

lute values and percentages), and the IPC metric. Red diamond markers correspond to values accumulated along the G-CUE and orange circles represent values of `avgOutsideGCUE` for each metric. In both cases, each marker corresponds to a single execution of the program. Yellow box plots include all TL-CUE values of all executions. Here and in all following plots in this thesis, the box plot whiskers represent minimum/maximum values and the center line corresponds to the median value. Due to the fact that the box plots include all executions, the figures do not compare a G-CUE with *only* the TL-CUEs of its execution, which would be impractical for figures illustrating many executions. Therefore, if a G-CUE is within the range of all TL-CUEs, it is not necessarily within the range of the TL-CUEs of its execution. This does not invalidate the analysis but requires more careful interpretation.

The figure clearly shows that some of the aforementioned expectations hold true: The G-CUE needs many more clock cycles than individual TL-CUEs (here: processes). This does not appear to be caused by the number of retired instructions, which only varies marginally between the G-CUE and individual TL-CUEs², but instead by the many stalled cycles that significantly reduce the throughput of executed instructions.

There are two more interesting observations to be made in Figure 6.2: First, the percentage of stalled clock cycles for individual TL-CUEs is close to the G-CUE values. This is due to the fact that unbounded activities take much more time to execute than bounded ones, causing the unbounded activity within a process to dominate its process’s clock cycles. The `avgOutsideGCUE` values, which deliberately only consist of useful execution activities outside the G-CUE (here: only bounded activities), are thus better suited for an evaluation of the G-CUE’s distinctiveness than a comparison with TL-CUEs. Second, despite the strong separation between the G-CUE and `avgOutsideGCUE` values, even the useful execution activities outside the G-CUE do not appear to be efficiently executed as evident from the high percentage of stalled cycles. This is not specifically considered in this analysis, however.

The amount of L1D cache line replacements and L2/L3 cache misses illustrated in Figure 6.3 also follow the expectations: On all levels of the cache hierarchy, there are many more cache misses per memory instruction along the G-CUE than along individual threads. This closely matches the metrics visible in the Score-P trace (Figure 6.1).

To further validate the computed metrics, a variant of the Random Histogram application was executed in which every process always sends its local data to the next process (`noStageOffset = true` in Algorithm 1). In this variant, every stage performs the exact same calculations, so all unbounded activities are part of a single process. It is expected that the G-CUE still passes through all unbounded activities and the values measured along the G-CUE should therefore remain the same as in the

² A more detailed overview of measured instruction counts can be found in Figure B.1.

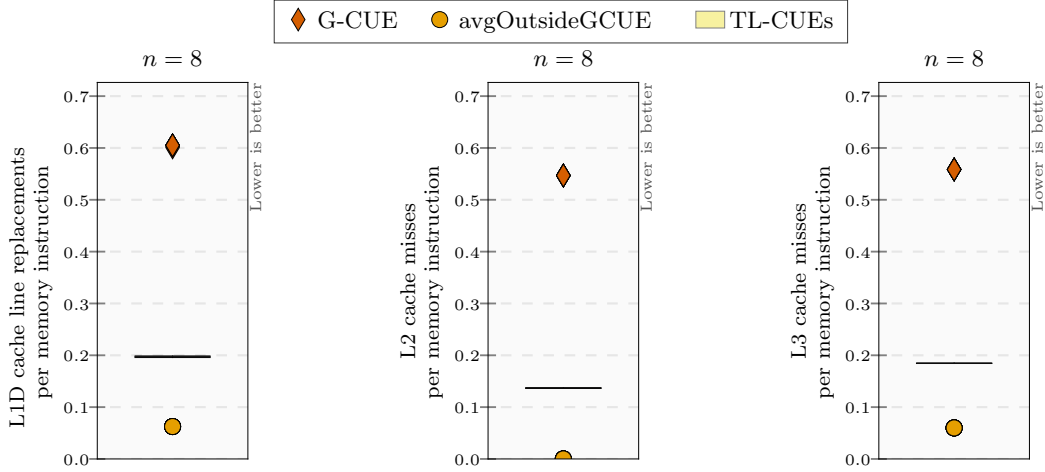


Figure 6.3.: Random Histogram: Cache-related metrics derived from OTF-CPT measurements of the kernel. The box plots corresponding to all TL-CUEs appear as a single line due to the measurements’ minimal variability.

original variant. However, the range of values for individual threads should expand: Now, one process (i.e., one thread) contains all the unbounded activities, so the G-CUE should equal the TL-CUE of that thread. Both of these expectations are confirmed in the measurements performed by OTF-CPT (see Section B.1.2).

Lastly, the Random Histogram application was modified to force all activities to become bounded activities (*allBounded* = true). Here, the G-CUE metrics become virtually identical to the metrics for any TL-CUE, although small differences are still visible in some measurements (see Section B.1.3 for details).

Overall, these results show that the proposed methodology is able to detect and locate performance factors affecting the critical path of useful execution. In this case, some traditional analysis methods would also be able to detect and sometimes even locate the (artificial) performance issue, but in some other approaches, the imbalance in the Random Histogram application would remain completely hidden. The *Serialization Efficiency* metric for parallel programs [23, 1], which reflects the maximum percentage of useful execution over any process, amounts to roughly 0.3 for the default configuration of the Random Histogram application (measured by OTF-CPT). This indicates that most of the time within a process is spent waiting, which in this case is almost entirely caused by the G-CUE. The *Serialization Efficiency* is therefore able to detect the issue, but it alone is not able to locate its reasons. Measuring hardware performance counters in a traditional way may or may not locate the imbalance in this application: If there is only one measured value per process, the imbalance remains hidden, since every process contains a single unbounded activity and three bounded ones. Even a simple function/region profile would not detect the imbalance if it does not differentiate individual function calls

(or region instances), since bounded and unbounded activities only differ in their accessed data and would be aggregated into the same profile entry. The imbalance can only be noticed from performance counter measurements alone if the measurement is more fine-grained and differentiates individual activities (e.g., as part of an execution trace). This can be done with Score-P, whose traces can be visualized in *Vampir* [54] similar to what is shown in Figure 6.1. With a visualization of these traces, it is trivial to find the cache-related imbalance of this simple application. The next sections in this thesis apply the methodology to algorithmic benchmarks and real-world applications in order to evaluate how well the methodology applies to non-fabricated and more complex cases.

6.2. Mergesort

The Mergesort application was chosen for the evaluation in order to investigate how the methodology applies to OpenMP tasking.

Figure 6.4 shows the durations of the kernel and the G-CUE within the kernel (left plot) as well as the resulting speedup with regard to one thread (right plot). Four out of twenty executions spent significantly more time than all other executions, with around ten seconds more for an execution with a single OpenMP thread. With an increase in threads, the absolute difference between the four outliers and the other runs quickly diminishes. The G-CUE durations closely mirror the total kernel durations even in case of the four outliers, indicating that the G-CUE is responsible for the majority of the run time.

The kernel scales well up until 24 OpenMP threads, after which the speedup slowly decreases along with a slight increase in variance. The G-CUE appears to scale better than the kernel in its entirety, suggesting that the overhead of OpenMP communication increases with the amount of OpenMP threads. This is also the likely explanation for the difference between the baseline’s and OTF-CPT’s kernel durations, as the latter executes extra code for OpenMP constructs in order to compute the critical path variants.

Due to OTF-CPT’s overhead, the reliability of its measurements is unclear. However, when executing the application with a smaller cutoff size of 2048, the overhead of OTF-CPT drastically increases whereas the counter measurements remain mostly similar. This could indicate that the overhead’s impact is negligible in this specific case.

The following analysis is split into two parts. First, it investigates the four outliers that took significantly longer to execute, in order to find their cause and to understand whether, and if so, how they affect the measurements. This is done by other means than the presented methodology, but necessary in order to understand the reliability of the results. The second part then applies the methodology to evaluate the scaling behavior on the upper end of the range of OpenMP thread numbers.

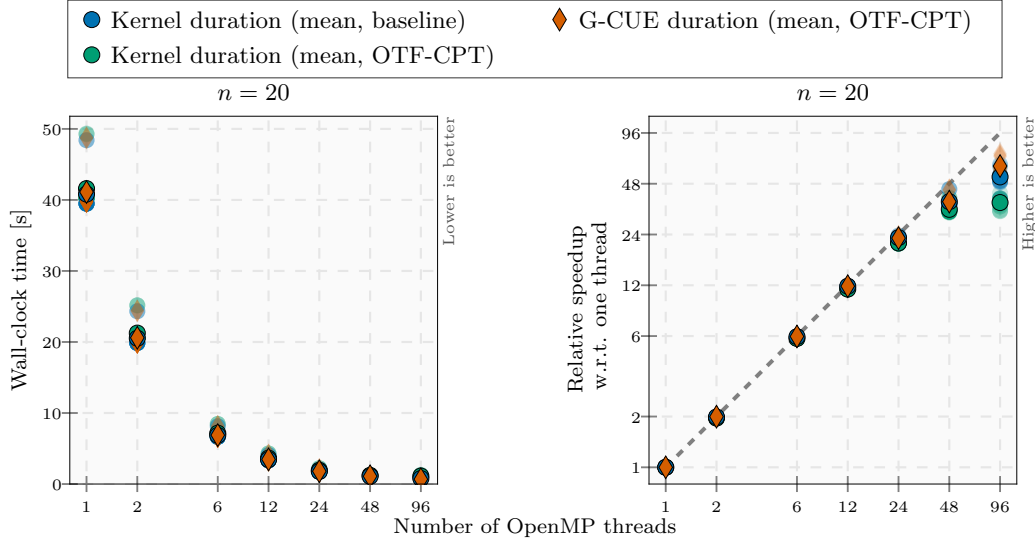


Figure 6.4.: Mergesort: Kernel and G-CUE durations (left) and the resulting speedup (right) for a varying number of OpenMP threads. Circle markers represent kernel durations for baseline (blue) and OTF-CPT (green) runs as reported by the application itself. Diamond markers represent durations of the kernels’ G-CUEs measured by OTF-CPT. The gray dashed diagonal corresponds to the slope of a linear speedup.

6.2.1. Outliers

Despite the outliers taking more time along the G-CUE, the two sets of measured event counts—and in particular, the number of unhalted clock cycles illustrated in Figure 6.5—do not show this difference. This leads to the hypothesis that the difference in G-CUE duration is caused by a change in CPU frequencies. A lower CPU frequency slows down the entire execution, potentially without (significantly) affecting the amount of unhalted clock cycles required to execute the given instructions. This could explain the decrease in duration *difference* with the increase in OpenMP threads: Given an almost linear speedup, a doubling of threads halves the execution time and therefore also halves the time attributed to a slower CPU clock.

As noted in Section 5.1, manual traces of the CPU frequency did not succeed on CLAIX-2023, but Slurm’s `sacct` command yields an estimate of the CPU frequencies by listing the average CPU frequency for a given job. The output of the `sacct` command for two executions of the Mergesort benchmark is shown in Listing 6.1. Slurm job 1137323 corresponds to an execution that takes 40.17 seconds to execute the kernel with a single OpenMP thread; for this job, Slurm reports an average CPU frequency of 3.63 GHz. Job 1137325 takes 49.26 seconds for the same workload on a single thread and has an average CPU frequency of just 2.84 GHz. The average frequency of the first job is around 27.8% higher than the frequency of the second job, roughly matching the 22.6% increase in kernel duration from 40.17 s to 49.26 s.

6. Evaluation

1	JobID	AllocCPUS	AveCPUFreq
2	-----	-----	-----
3	1137323	96	
4	1137323.bat+	96	3.63M
5	1137323.ext+	96	0
6	1137323.0	96	0
7	1137323.1	48	0
8	1137323.2	24	0
9	1137323.3	12	0
10	1137323.4	6	0
11	1137323.5	2	0
12	1137323.6	1	0
13	1137325	96	
14	1137325.bat+	96	2.84M
15	1137325.ext+	96	0
16	1137325.0	96	0
17	1137325.1	48	0
18	1137325.2	24	0
19	1137325.3	12	0
20	1137325.4	6	0
21	1137325.5	2	0
22	1137325.6	1	0

Listing 6.1: Output of `sacct --format JobID,AllocCPUS,AveCPUFreq` for two invocations of the Mergesort application. Slurm job **1137323** corresponds to an execution that takes 40.17 seconds to execute the kernel with a single OpenMP thread; job **1137325** takes 49.26 seconds for that configuration. AveCPUFreq is given in kHz [58, *sacct*].

`sacct` does not provide CPU frequency measurements for individual job steps (lines beginning with `<JobID>.<digit>` in Listing 6.1) and their kernels. To better understand the impact of the difference in frequency with regard to the application kernel, additional measurements of clock-related events not included in Table 5.1 were performed. Unlike `CPU_CLK_UNHALTED.THREAD`, which depends on the CPU's clock frequency, the `CPU_CLK_UNHALTED.REF_TSC` event counts the number of *reference cycles* that increases on a fixed time step and is therefore independent of any change in frequency [42]. For the same two executions as shown in Listing 6.1, the numbers of reference cycles along the G-CUE amounts to 101,655,559,992 (slow job) and 83,012,994,540 (fast job). The slow job takes around 22.5% more reference cycles along the G-CUE than the fast job, which almost exactly matches the difference in G-CUE durations of those jobs (22.6%).

It is therefore safe to say that the observed difference in kernel durations is indeed attributable to a difference in CPU frequency, although the reason for the different frequencies remains unknown. Since the two sets of event counts measured for the 20 executions analyzed in the remaining evaluation do not show this difference, the outliers can be safely included in the analysis. A plot of the absolute kernel durations and the resulting speedup without the outliers for comparison is found in Figure B.8.

The same outlier patterns as observed for Mergesort were present for SPMxV (Section 6.3) and DENISE Black Edition (Section 6.4). To avoid redundancy, these outliers are not discussed further in the respective sections.

6.2.2. G-CUE Scaling Behavior

Figure 6.5 shows the amount of retired instructions and unhalted clock cycles as well as the resulting IPC metric. Both clock cycles and instructions show a notable increase in the range of TL-CUE values with an increase in OpenMP threads, with a slightly stronger pattern visible in the number of retired instructions. A similar increase is found in the G-CUE lying above the thread with the highest event counts, whose offset to any individual thread increases with an increase in threads. In turn, both metrics do not scale linearly along the G-CUE. Executions with very few threads show almost no difference between individual TL-CUEs, the G-CUE, and avgOutsideGCUE.

Throughout most thread configurations of the application, the IPC is higher along the G-CUE than along individual threads. The number of stalled clock cycles (Figure 6.6), which is independent of executed instructions and therefore more meaningful for a comparison of different execution paths with different instruction counts, shows the same pattern (here, lower values are better). Both the IPC and the number of stalled cycles suggest that the G-CUE executes its workload more efficiently than individual threads. Consequently, the non-linear scaling of clock cycles likely does not result from an inefficient execution but rather from a different size of the workload (i.e., a different amount of instructions).

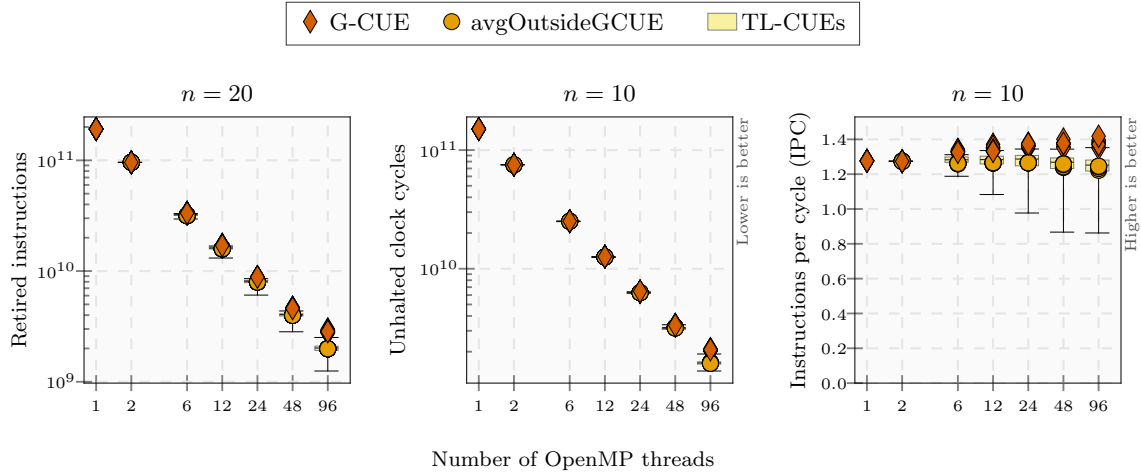


Figure 6.5.: Mergesort: Retired instructions and unhalted clock cycles as measured by OTF-CPT for the kernel, as well as the resulting instructions per cycle (IPC).

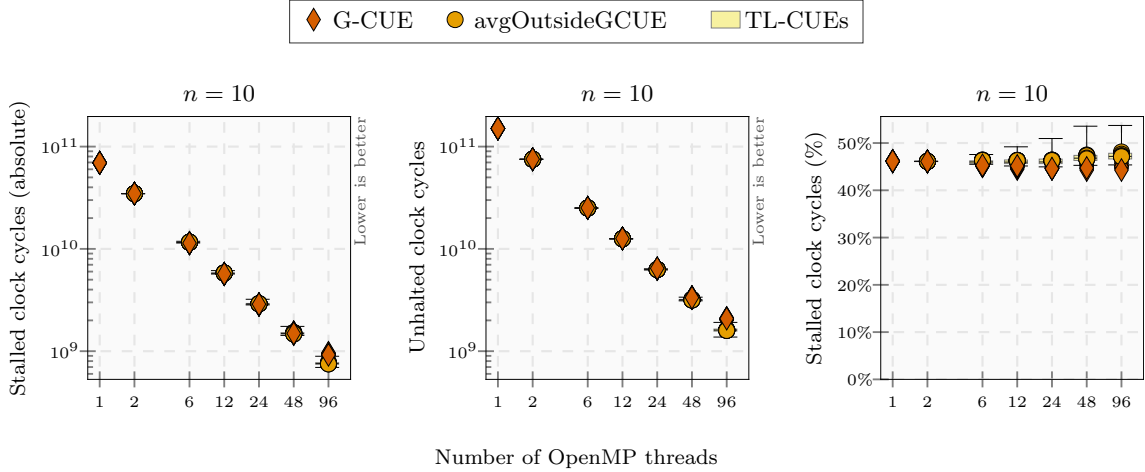


Figure 6.6.: Mergesort: Stalled clock cycles and unhalted clock cycles as measured by OTF-CPT, and their ratio.

Because the G-CUE’s structure and the instructions alongside of it are not known, the observed instruction count pattern cannot be directly attributed to individual sections or even lines of code. However, the Mergesort source code limits the amount of potential reasons due to its small size (a few hundred lines) and gives rise to a hypothesis which is discussed in the following.

Both the parallel and serial merge operations use binary search to find a single value within one of the merging halves. Since the merge operation is performed on many different pairs of array halves, the amount of loop iterations in the binary search and thereby the amount of executed instructions varies from merge to merge. The Mergesort application is executed with a fixed input size and fixed array values based on a hard-coded seed for pseudorandom number generation, so the number of OpenMP tasks and the number of required iterations per task is fixed as well. This means that an increase in the number of threads entails a decrease in the number of tasks scheduled to each thread. As a consequence, fluctuations in the amount of loop iterations within tasks become more visible as the number of threads increases, as each task has a larger share in the instruction count accumulated over a thread’s useful execution. This is likely amplified due to the use of OpenMP `taskwait` constructs in the source code, which introduce scheduling constraints and waiting time. If this hypothesis is true, the G-CUE might pass through iteration-heavy tasks and thus execute more instructions than any individual thread.

While there is no straightforward way to verify this hypothesis, there are at least two necessary conditions that must be met if the hypothesis is true. First, since the hypothesis assumes a difference in the *distribution* of instructions rather than a difference in the total number of instructions, the total number of instructions over all TL-CUEs should be similar regardless of the amount of OpenMP threads. This is true in the given measurements.

Second, the widened range of instruction counts should be mirrored by all kinds of instructions found in the merge operation’s binary search. Figure 6.7 shows a breakdown of the number of retired instructions into individual types of instructions. The leftmost plot shows the total amount of retired instructions whereas all other plots represent event counts for individual groups of instructions that count towards the overall instruction count. The difference in retired instructions can be seen in the number of memory-related instructions and retired conditional branch instructions as well. The Mergesort kernel does not use floating-point arithmetic, but a small yet non-zero number of scalar floating-point instructions was measured for the kernel. This is likely a side effect of using OTF-CPT, since some operations internal to the tool are executed while OTF-CPT still tracks useful execution. Therefore, floating-point instructions are ignored in this analysis. Unfortunately, only few types of instructions can be measured on Sapphire Rapids CPUs; most notably, no hardware event exists that counts the number of retired instructions related to integer arithmetic. Consequently, the available measurements cannot fully prove the second necessary condition for the hypothesis. A final conclusion on the cause of the observed scaling behavior is therefore not possible.

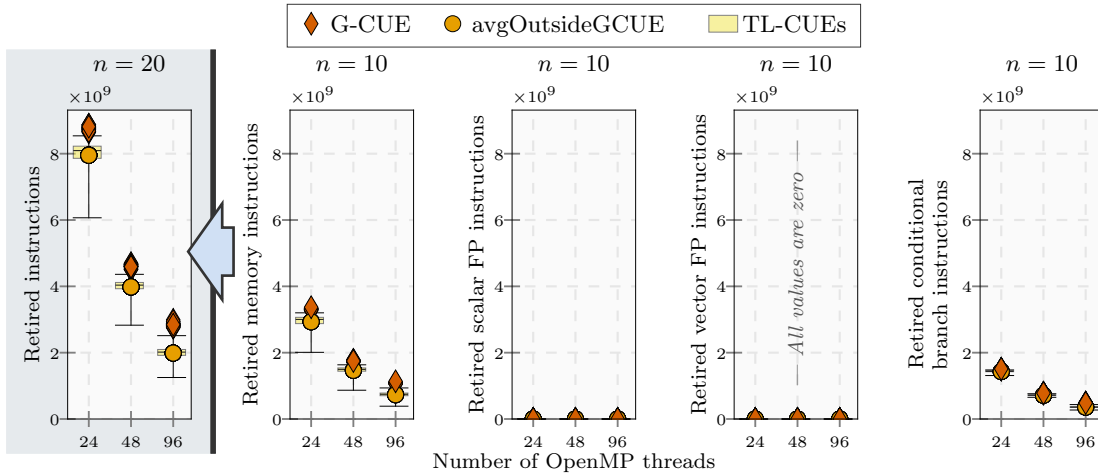


Figure 6.7.: Mergesort: Instruction counts for various types of instructions as measured by OTF-CPT for the kernel. Only executions with 24, 48, and 96 threads are shown. The number of retired instructions on the left includes all other measured instruction counts as well as other instruction types that cannot be counted separately.

6.3. Sparse Matrix-Vector Multiplication

This section presents the evaluation of the SPMxV benchmark that makes use of OpenMP. Two variants of the benchmark were measured and evaluated, whose kernel's source code is provided in Appendix A:

1. The original version of the application does not scale well, which makes it a good candidate for analysis. However, this variant causes high overhead for measurements with OTF-CPT, raising concerns about the reliability of the measurements.
2. To fix the overhead, the `nowait` clause [50, Section 2.11.4] was added to the worksharing for loop whose iterations each compute a single row of the output vector. This reduced the measurement overhead to the point where it is no longer a concern, but the scaling was also fixed in the process, making it potentially less interesting for analysis.

Since the evaluation found noteworthy results for both variants and a potential relationship between the `nowait` clause and measurements along the G-CUE, the following discusses both variants.

6.3.1. Variant 1: Without `nowait`

Variant 1 of the SPMxV application closely matches the original source code [63] apart from minor changes (e.g., to instrument the code). Within a parallel OpenMP region, the application executes 100,000 repetitions of the matrix-vector multiplication. The multiplication is parallelized by means of a worksharing for loop [50, Section 2.11.4] with static scheduling (i.e., each thread executes roughly the same amount of loop iterations) in a parallel region. Every iteration in the loop computes a single value in the output vector.

Figure 6.8 shows the durations of the kernel and its G-CUE (left plot) as well as the resulting speedup of all measured durations (right plot). Up until 24 threads, the baseline kernel scales optimally, with some configurations (6–24 threads) even superlinearly. For higher numbers of threads, the baseline appears to suffer from serious performance issues. Throughout all configurations, the G-CUE closely mirrors the baseline kernel durations, suggesting that it is mostly responsible for the overall performance. Only in case of 48 and 96 threads, it slightly detaches from the kernel durations.

The kernel durations for OTF-CPT runs scale much worse, indicating severe overhead due to the use of OTF-CPT. This appears to be a consequence of implicit barriers (e.g., found at the end of worksharing for loops [50, Section 2.11.4]): Measurements using Scalasca (not shown in the plots) exhibit a strong increase in time spent within OpenMP, and specifically within implicit barriers, for an increase in the number of threads. This directly correlates with the *number* of implicit barriers,

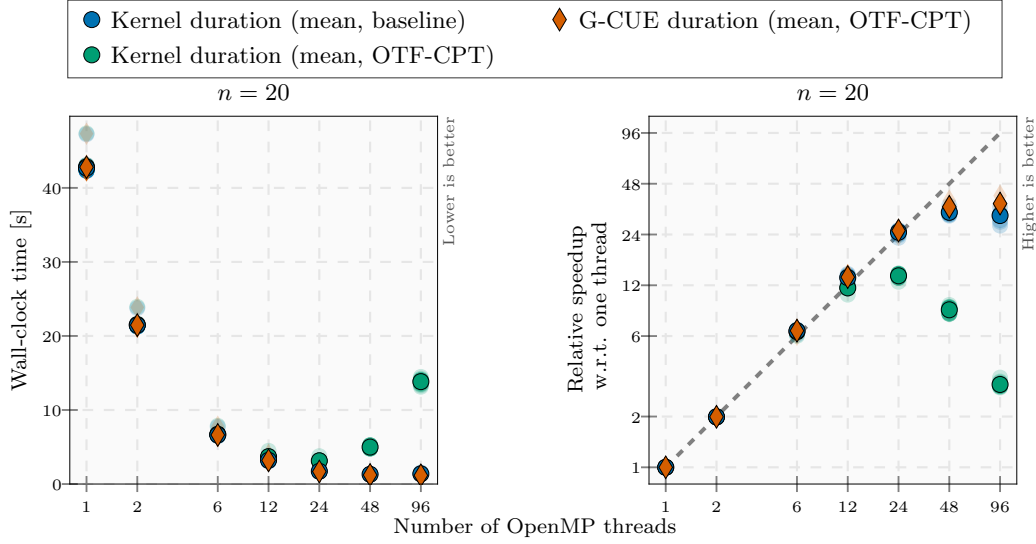


Figure 6.8.: SPMxV: Kernel and G-CUE durations (left) and the resulting speedup (right) for a varying number of OpenMP threads. Circle markers represent kernel run times for baseline (blue) and OTF-CPT (green) runs as reported by the application itself. Diamond markers represent the kernel’s G-CUE durations as measured by OTF-CPT. The gray dashed diagonal corresponds to the slope of a linear speedup.

which Scalasca reports to be equal to (i) the amount of executed worksharing for loops, of which there are 100,000 per thread, plus (ii) the amount of parallel regions, of which there is one per thread:

$$\text{\#implicit barriers} = \text{\#threads} \cdot (100,000 + 1).$$

Every single instance of a barrier triggers a callback function in OTF-CPT, which incurs a tiny—but substantial, in total—overhead. Consequently, the reliability of OTF-CPT’s measurements is unclear. However, because the workload is distributed by means of static scheduling (i.e., the distribution is independent of run-time characteristics), it is well possible that the overhead can be mostly ignored for measurements related to useful execution. The fix for this overhead is discussed in Section 6.3.2.

Measurements of the number of retired instructions and clock cycles as well as the resulting IPC (illustrated in Figure 6.9) show that the G-CUE scaling behavior is a result of taking more clock cycles to execute. This is not mirrored in the amount of retired instructions which scales linearly, indicating that the increase in clock cycles along the G-CUE is due to the execution of the given instructions as visible in the IPC metric. The IPC metric also reveals the reason for the superlinear speedup in case of 6–24 threads, which is due to a decrease in performance for configurations with one and two threads (recall that speedup is computed with regard to a single thread). Independent of the amount of threads, the G-CUE’s IPC value is on the low

6. Evaluation

end of the range of individual threads and in case of 48 and 96 threads even below that. The reason for the visible pattern in IPC appears to be found in the amount of stalled cycles (Figure 6.10), which (inversely) mirrors the IPC measurements. The increase in stalls at the lower and higher ends of the numbers of threads was found to likely be a consequence of two different causes, which are discussed in the following.

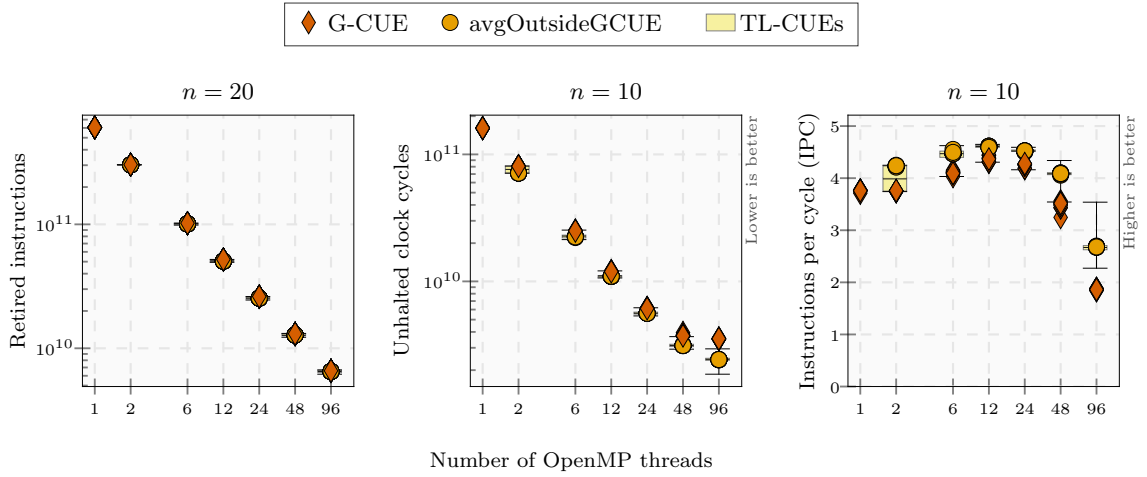


Figure 6.9.: SPMxV: Retired instructions and unhalted clock cycles as measured by OTF-CPT for the kernel, as well as the resulting instructions per cycle (IPC).

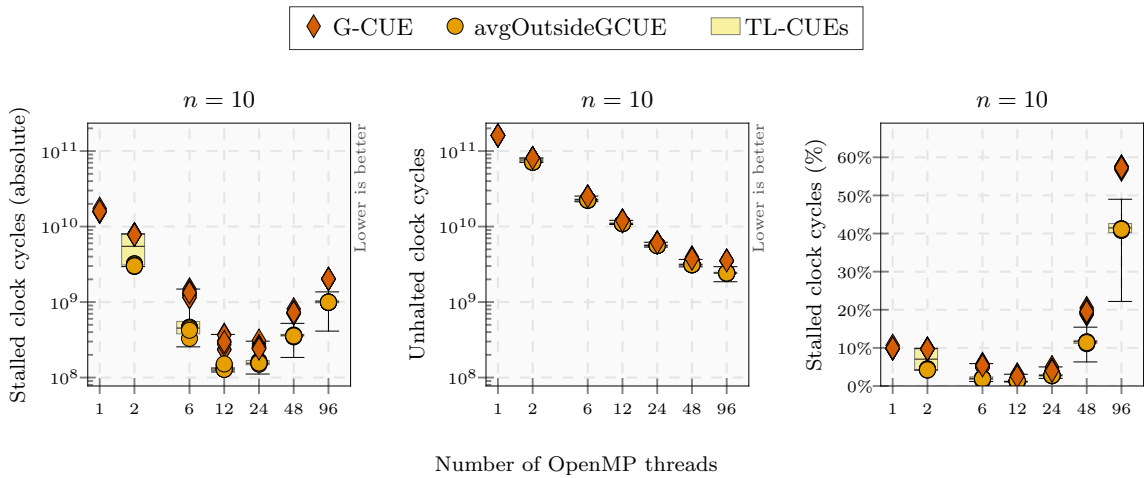


Figure 6.10.: SPMxV: Stalled clock cycles and unhalted clock cycles as measured by OTF-CPT for the kernel, and their ratio.

The reason for the increase in stalled cycles in case of one, two, and six threads is potentially found in the L2 cache behavior shown in Figure 6.11. Despite the variance, a clear trend is visible in the figure, which shows a much higher amount of L2 cache misses in case of one or two threads than in case of 48 or 96 threads, for

example. The G-CUE closely follows the TL-CUE with the highest cache miss rate in its execution (not well visible in the plots), suggesting that it closely follows the loop iterations of a single thread. In the experiments, the first thread within an execution caused the most cache misses, which coincides with the first rows of the matrix having the most non-zero values (see Figure 5.2). The potential cause for the increase in cache misses with few threads is found in the fact that the matrix multiplication repeats 100,000 times: With fewer threads, a single thread accesses more matrix rows within its computation. At some point, the values within the rows assigned to each thread do no longer fully fit into the L2 cache, so subsequent repetitions of the multiplication do not find the first assigned rows in the L2 cache. The source of the variance in the TL-CUE and G-CUE measurements remains unclear; the potential cause might lie in the shape of the non-zero values found in the matrix and in the execution environment, since the L2 cache is shared between processes running on the node.

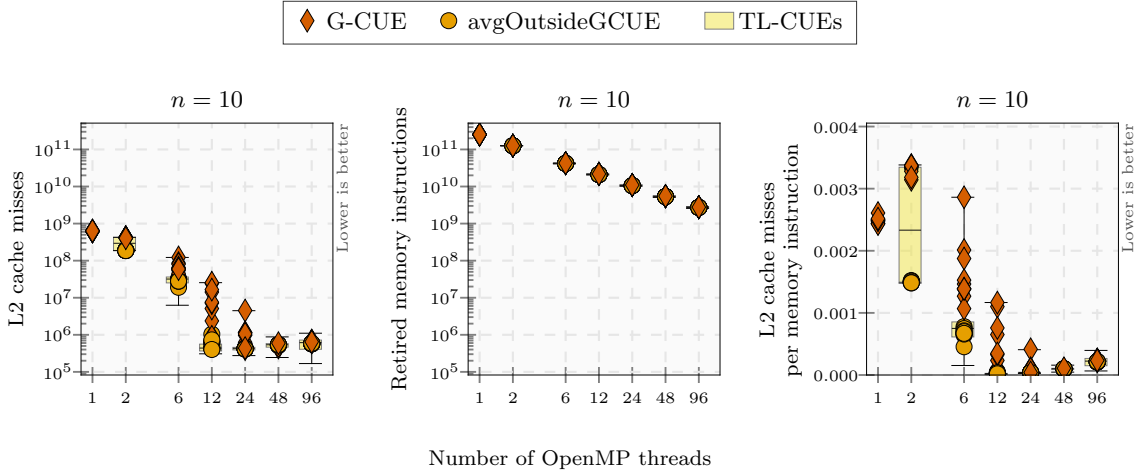


Figure 6.11.: SPMxV: L2 cache misses and retired memory instructions as measured by OTF-CPT for the kernel, and their ratio.

The bad scaling behavior for higher thread numbers is potentially caused by a large amount of page-fault-related microcode assists. Figure 6.12 shows the amount of page fault assists in relation to the number of retired instructions. Here, a strong increase in page fault assists per retired instruction can be observed with an increase in threads. Especially in case of 96 threads, the G-CUE's values lie far outside the values of individual TL-CUEs, suggesting that the amount of page faults may be a reason for the decrease in performance along the G-CUE. By coincidence, a relationship between the amount of page fault assists and the use of the `nowait` clause was discovered. This is discussed in more detail in the next section.

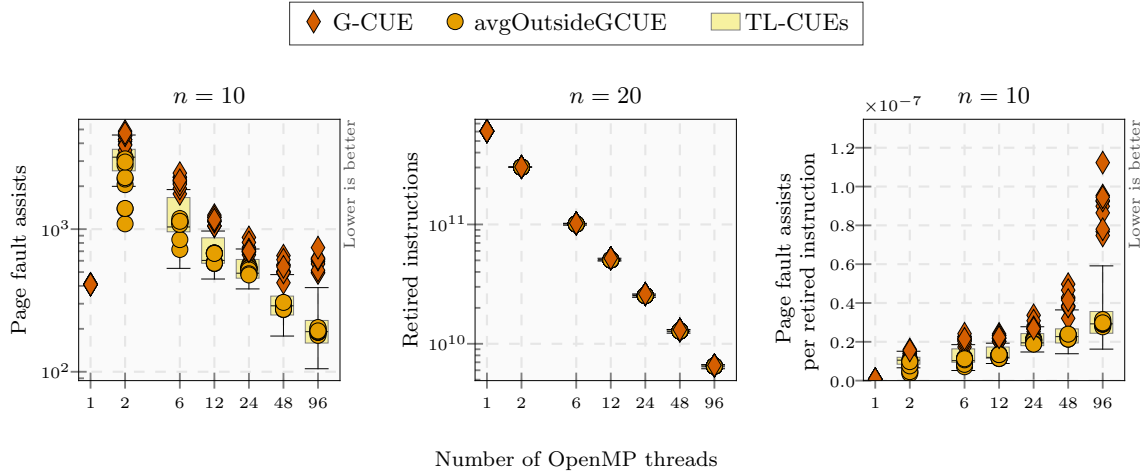


Figure 6.12.: SPMxV: Page fault assists and retired instructions as measured by OTF-CPT for the kernel, and their ratio.

6.3.2. Variant 2: With `nowait`

In order to fix the measurement overhead observed for the first variant of the SPMxV application, the `nowait` clause was added to the application’s worksharing for loop that distributes rows of the input matrix to individual threads. By adding this clause, no implicit barrier is found at the end of the worksharing for loop region.

Due to the fix, variant 2 does not suffer from the barrier-induced measurement overhead. This is visible in Figure 6.13 which illustrates absolute durations and their resulting speedup for the kernel region as well as the G-CUE’s duration within the kernel region. The fix also affects the scaling behavior of the baseline case and the G-CUE, which are both close to optimal in variant 2. Therefore, the use of `nowait` also impacts useful execution despite targeting communication-related performance. The following evaluation only discusses results related to `nowait`; the findings from variant 1 with regard to cache usage remain largely unchanged (see Figure B.15).

Both the IPC metric (Figure B.14) and the amount of stalled cycles (Figure 6.14) show a substantial improvement in case of 48 and 96 OpenMP threads. This correlates with the number of page-fault-related microcode assists illustrated in Figure 6.15, which are now significantly reduced along the G-CUE and along all but one TL-CUEs. The number of assists along the TL-CUE with the largest numbers of assists in an execution remains similar: In case of 96 threads, each thread in variant 1 has around 100–300 page fault microcode assists; in variant 2, only a single thread (often thread 2 or 3) has around 300 assists whereas all other threads have less than ten or even zero page fault assists.

Another change caused by the addition of `nowait` is the L1D cache behavior shown in Figure 6.16 for variant 2. The amount of L1D cache line replacements per in-

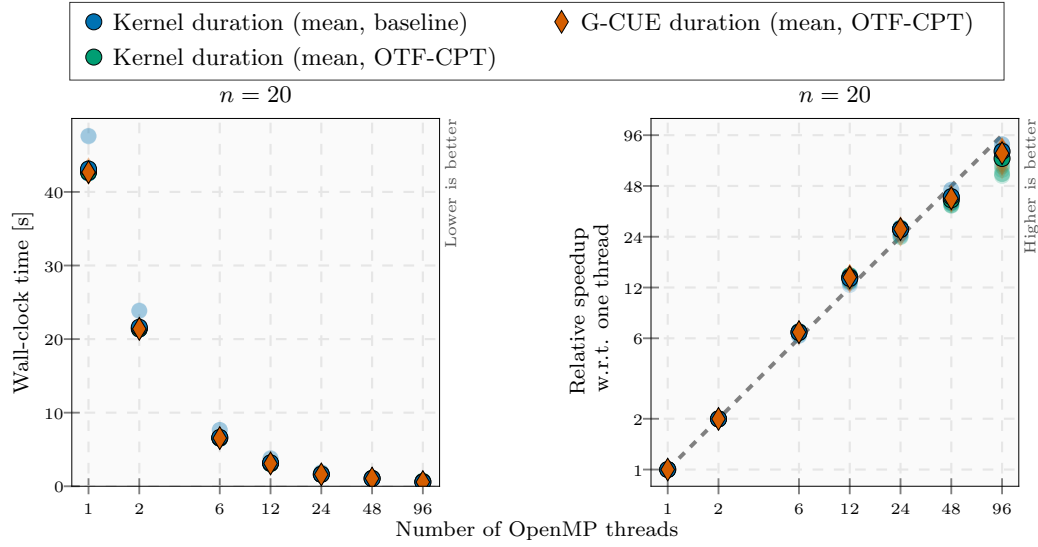


Figure 6.13.: SPMxV (nowait): Kernel and G-CUE durations (left) and the resulting speedup (right) for a varying number of OpenMP threads. Circle markers represent kernel run times for baseline (blue) and OTF-CPT (green) runs as reported by the application itself. Diamond markers represent the kernel's G-CUE durations as measured by OTF-CPT. The gray dashed diagonal corresponds to the slope of a linear speedup.

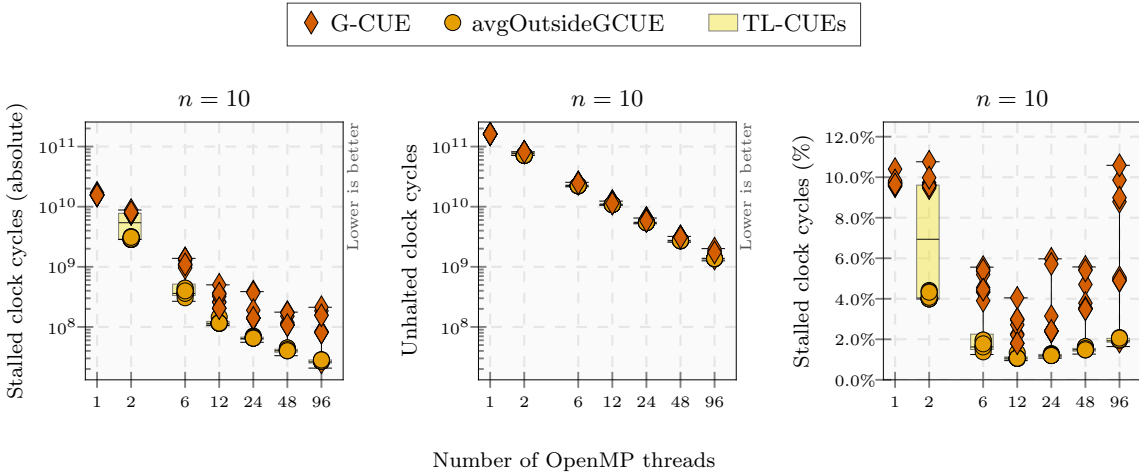


Figure 6.14.: SPMxV (nowait): Stalled clock cycles and unhalted clock cycles as measured by OTF-CPT for the kernel, and their ratio.

6. Evaluation

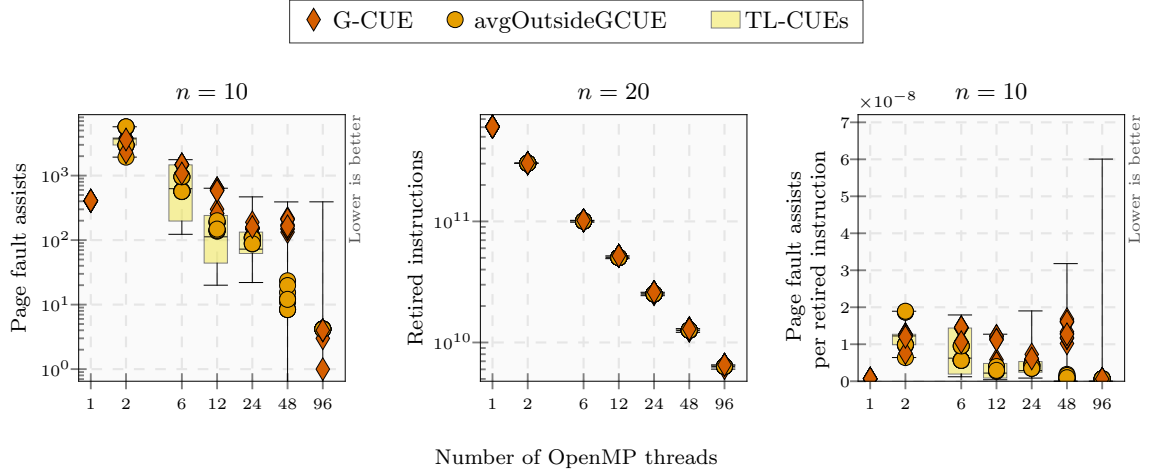


Figure 6.15.: SPMxV (nowait): Page fault assists and retired instructions as measured by OTF-CPT for the kernel, and their ratio.

struction along individual TL-CUEs is the same in both variants (see Figure B.12 for variant 1). However, the G-CUE's values, which in variant 1 only approximated the maximum TL-CUE up until 24 threads, now almost all approximate the maximum TL-CUE even with 48 or 96 threads. This could indicate a shift in what limits the G-CUE in case of 48 and 96 threads. This pattern, however, is not always found in all executions: In most of the ten experiment runs ($n = 10$), a single execution with more than 24 threads (the exact number of threads appears to follow no specific pattern) measured G-CUE values that were close to the median TL-CUE. The reason for this is not known, but the pattern might indicate that the G-CUE changes between several execution paths with completely different causes for their durations.

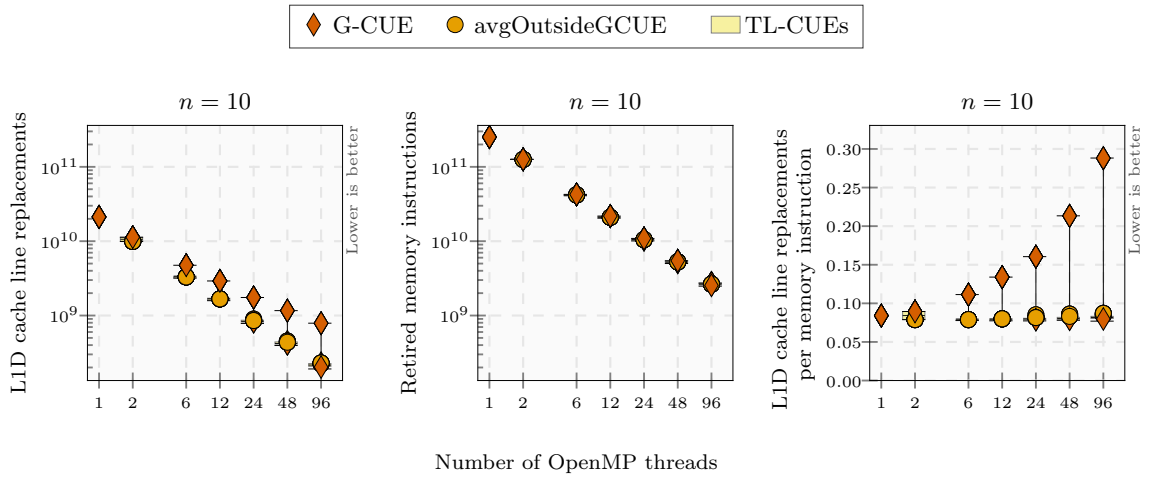


Figure 6.16.: SPMxV (nowait): L1D cache line replacements and retired memory instructions as measured by OTF-CPT for the kernel, and their ratio.

6.4. DENISE Black Edition

As discussed in Section 5.3.4, prior research concluded that the performance of DENISE Black Edition can suffer due to a high amount of subnormal floating-point values. While there might be other performance issues as well, the evaluation of DENISE in this thesis specifically focuses on subnormal floating-point values with regard to the critical path.

Figure 6.17 shows the durations of DENISE executions and their G-CUEs (left plot), as well as the resulting relative speedup of all measured durations (right plot). In all cases, the G-CUE closely matches the total duration, indicating that it is largely responsible for the observed performance. The application does not scale well: Even with only six processes, the application runs notably slower than expected in this strong scaling setting.

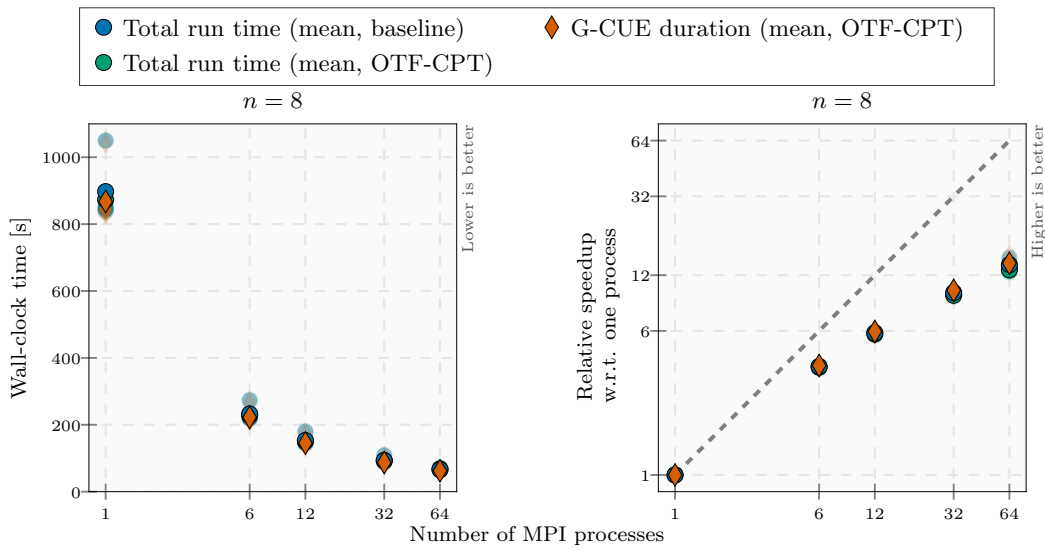


Figure 6.17.: DENISE: Total execution times and G-CUE durations (left), and the resulting speedup (right) for a varying number of MPI processes. Circle markers represent total application run times for baseline (blue) and OTF-CPT (green) runs. Orange diamond markers represent G-CUE durations as measured by OTF-CPT. The gray dashed diagonal corresponds to the slope of a linear speedup.

Figure 6.18 illustrates the number of retired instructions, unhalting clock cycles, and the resulting IPC metric. It clearly shows that the scaling behavior is due to the G-CUE, which executes significantly slower than all individual TL-CUEs. This difference in clock cycles cannot be attributed to a different amount of instructions as shown in the left plot of the figure (also see Figure B.17 for more details). The IPC metric (right plot of Figure 6.18) is much lower along the G-CUE than along individual threads, indicating that the difference in clock cycles is instead due to a difference in how the given instructions are executed.

6. Evaluation

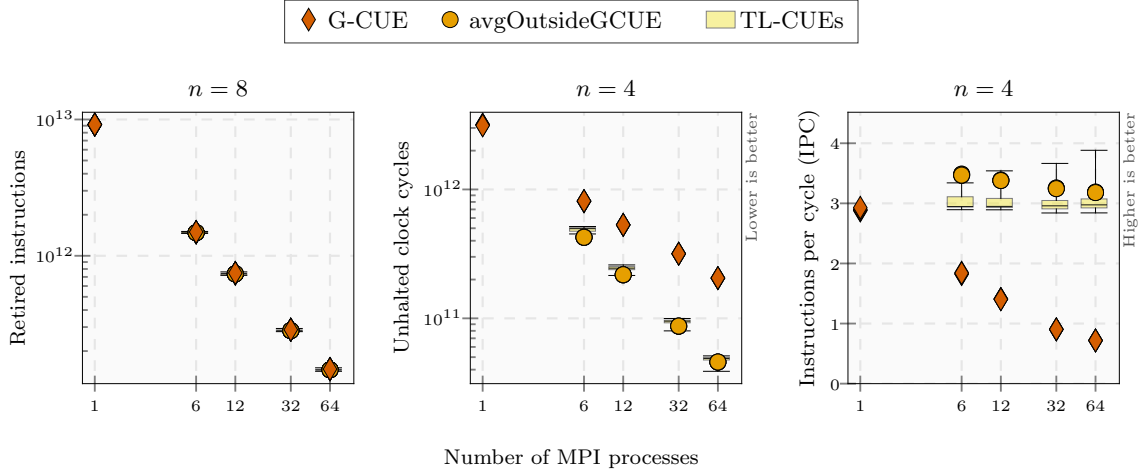


Figure 6.18.: DENISE: Retired instructions and unhalted clock cycles as measured by OTF-CPT, as well as the resulting instructions per cycle (IPC).

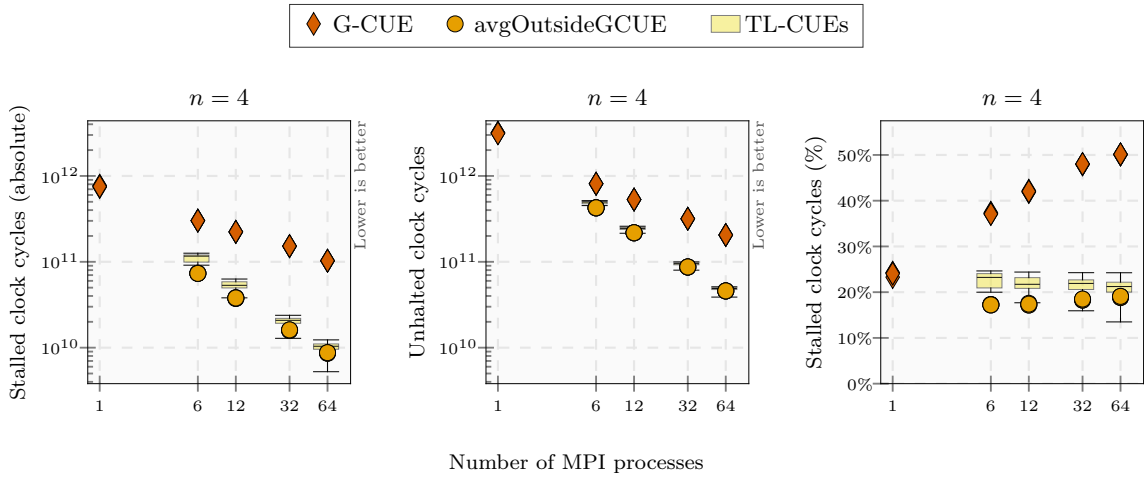


Figure 6.19.: DENISE: Stalled clock cycles and unhalted clock cycles as measured by OTF-CPT, and their ratio.

Figure 6.19 shows the number of stalled clock cycles in absolute numbers (left plot) and as a percentage of all unhalted cycles (right plot), as well as the number of unhalted clock cycles (middle plot) used to calculate the percentage. The extreme separation between the G-CUE and individual threads is also evident in the percentage of stalled clock cycles, which thus appears to be the main reason for the observed IPC. Along the G-CUE and most TL-CUEs, the percentage of stalled clock cycles is constantly above 20%, with the G-CUE increasing up to ca. 50% with 64 MPI processes. This indicates a severe performance issue, aligning with the observed scaling behavior.

Measurements of floating-point assists as shown in Figure 6.20 reveal the potential reason for the extreme amount of stalled cycles especially along the G-CUE: Both

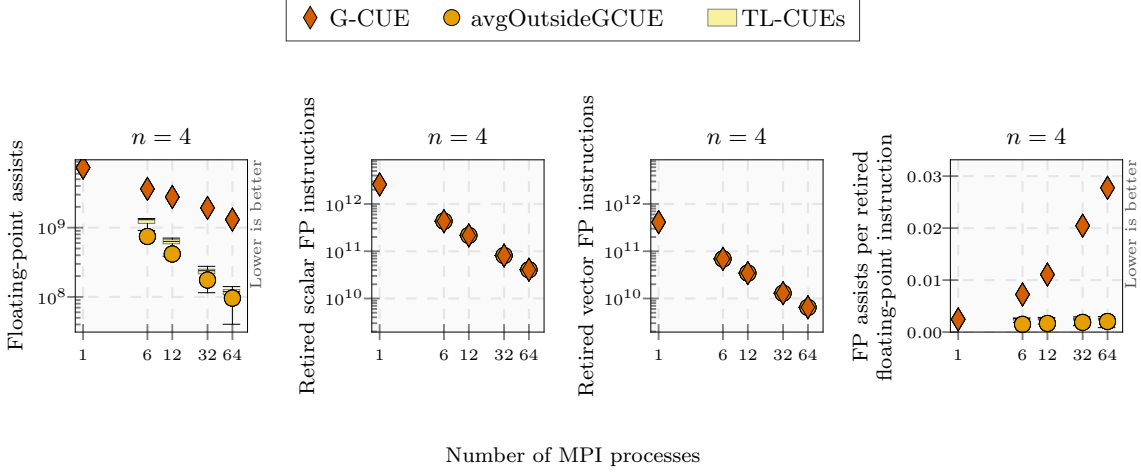


Figure 6.20.: DENISE: Floating-point assists and floating-point instructions as measured by OTF-CPT, and their ratio.

the absolute amounts of floating-point assists and the derived ratio of floating-point assists per retired floating-point instruction feature the separation already noted for the previously discussed metrics. Along individual TL-CUEs, less than 0.004 floating-point assists occurred per floating-point instruction, whereas along the G-CUE, the amount rapidly increases to slightly less than 0.03 with an increase in MPI processes.

While the low values of the derived metric may not look as if the microcode assists matter, they only represent a frequency but not the *impact*. The impact of floating-point assists varies per microarchitecture. There seems to be no publicly available data for the *Golden Cove* microarchitecture used by Intel Sapphire Rapids CPUs, but it has been shown experimentally that a single subnormal floating-point operation may cost between 120 and 300 cycles on other x86-64 microarchitectures [60, 62]. According to Fog [60], newer Intel microarchitectures among the tested ones (*Skylake*, *Ice Lake*, and *Tiger Lake*) have lower penalties for operations with subnormal floating-point values than older ones and even incur no penalty at all when adding a normal and a subnormal number. Some of the newer AMD microarchitectures (*AMD Zen 1–5*) only incur a penalty of a few cycles for subnormal floating-point operations [60]. Bjørndalen and Anshus showed that the execution time of a program can strongly correlate with the amount of denormalized floating-point results [76].

6.4.1. FTZ & DAZ Enabled

To evaluate the impact of the observed microcode assists, the prior results are compared to measurements of DENISE runs for which the DENISE binary was compiled with the `-ftz` Intel compiler flag, compared to `-no-ftz` that was enabled before.

6. Evaluation

This flag instructs the compiler to insert code that enables the FTZ and DAZ flags in the MXCSR *control and status* register [77, 40]:

- The FTZ flag enables the *Flush-To-Zero* mode, which, unless subnormal floating-point results should be handled by an explicitly configured exception handler, “returns a zero result with the sign of the true result” [40] for an operation that would otherwise generate a subnormal floating-point result.
- The DAZ flag enables the *Denormals-Are-Zeros* mode, which “converts all denormal source operands to a zero with the sign of the original operand before performing any computations on them” [40]. This ensures that even subnormal floating-point values that are either hard-coded in the binary or read from data written by other threads with other MXCSR flags do not cause microcode assists.

The `-ftz` compiler flag therefore completely eliminates any subnormal floating-point operations in the compiled binary. Neither one of the FTZ and DAZ modes is compatible with the *IEEE Standard for Floating-Point Arithmetic* (IEEE-754; [69]) [40] since they affect the result of floating-point computations. DENISE reported all simulations performed for this thesis to be stable even with `-ftz`. However, it is not clear whether the results of the simulation are still correct or sufficiently precise for their intended use.

In DENISE runs with `-ftz`, the reported number of floating-point microcode assists is zero (see Figure B.19), confirming that the flags were correctly set. The elimination of floating-point microcode assists seems to significantly affect the number of stalled cycles, which is now less than 10% in all measurements (Figure 6.21). The separation between the G-CUE and individual threads does not exist anymore, even though there are still more stalled cycles along the G-CUE than for activities outside of it. The IPC, which is shown in Figure B.20, shows the same improvements.

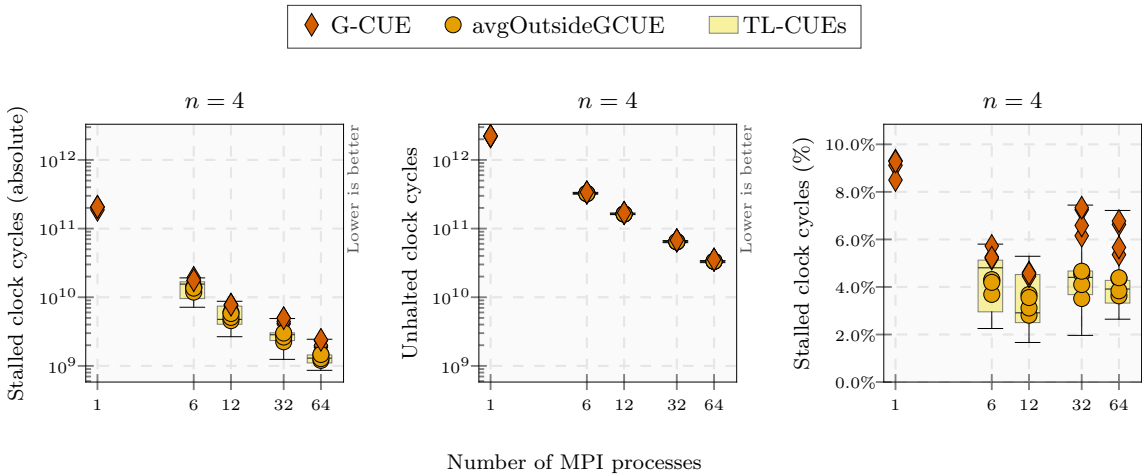


Figure 6.21.: DENISE (FTZ+DAZ): Stalled clock cycles and unhalted clock cycles as measured by OTF-CPT, and their ratio.

Figure 6.22 illustrates the absolute execution durations for the `-ftz` build and the resulting speedup, clearly showing that both absolute performance and the speedup have significantly improved. For example, runs using a single MPI process execute around 200 s faster if no subnormal floating-point values occur. In the `-ftz` variant, the G-CUE scales slightly better than the entire application, indicating that some of the remaining scalability issues are attributed to communication-related factors.

In conclusion, the DENISE experiments reproduce the findings of the prior research on DENISE’s performance with respect to subnormal floating-point values. While an analysis of per-thread measurements can also find the underlying issue, there is a clear separation between the G-CUE and individual threads which likely explains the severity of the performance issue. It is plausible that the G-CUE of the `-no-ftz` variant is due to the number of floating-point assists in the program’s activities.

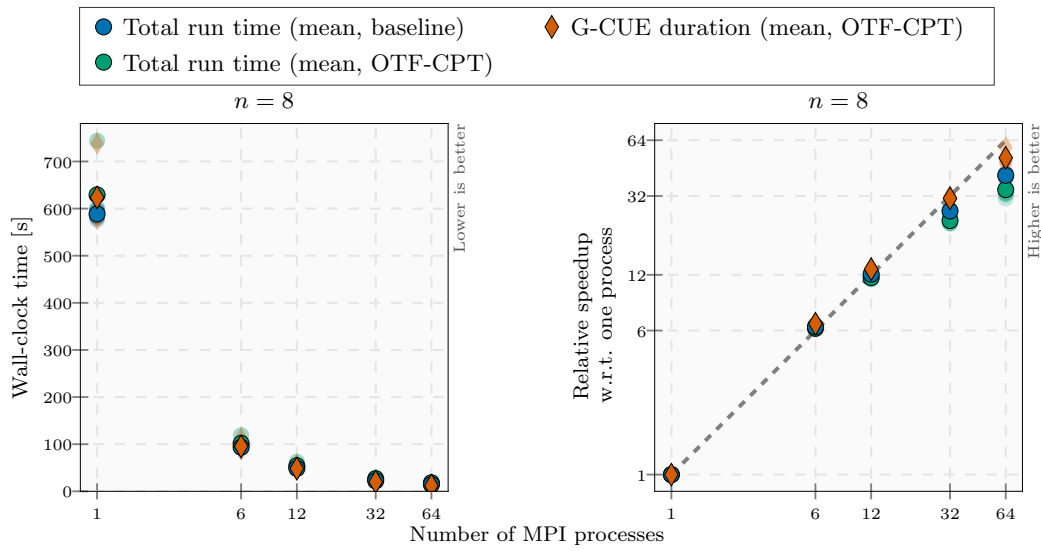


Figure 6.22.: DENISE (FTZ+DAZ): Total execution times and G-CUE durations (left), and the resulting speedup (right) for a varying number of MPI processes. Circle markers represent total application run times for baseline (blue) and OTF-CPT (green) runs. Orange diamond markers represent G-CUE durations as measured by OTF-CPT. The gray dashed diagonal corresponds to the slope of a linear speedup.

6.5. NEST Simulator

In contrast to the other evaluated applications, the NEST simulator experiments do not vary the overall degree of parallelism. Instead, the *distribution* of parallel resources (i.e., OpenMP threads per process and MPI processes) is varied such that the total amount of threads in the application remains constant at 96 (i.e., a full compute node with every CPU core executing a single thread). The overall workload

6. Evaluation

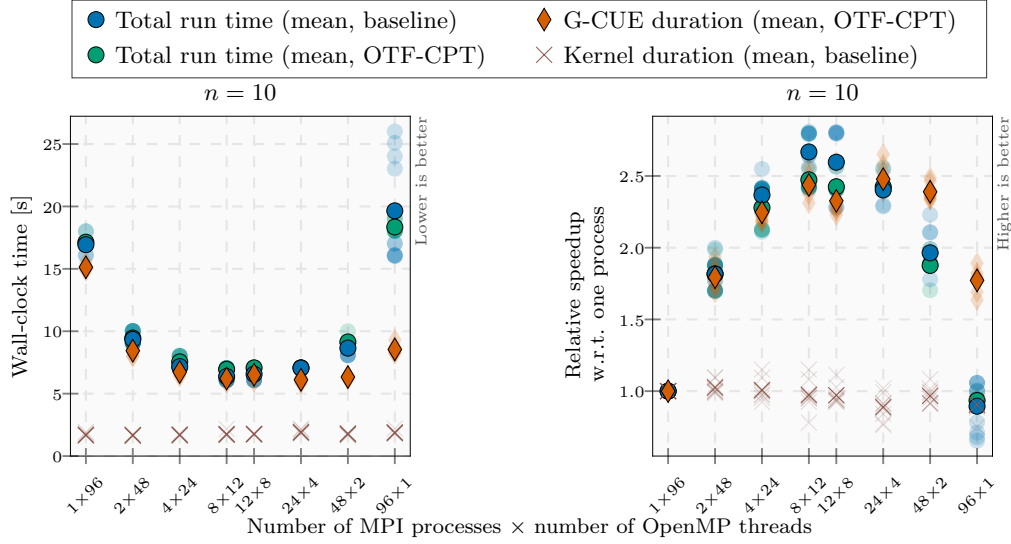


Figure 6.23.: NEST Simulator: Various execution durations (left) in the NEST HPC benchmark and the resulting speedup (right) for a varying distribution of MPI processes and OpenMP threads. Circle markers represent total application run times for baseline (blue) and OTF-CPT (green) runs. Diamond markers represent G-CUE durations as measured by OTF-CPT. X-shaped markers correspond to the kernel durations as reported by the application itself during baseline runs.

as defined by the number of neurons and the simulated time (see Chapter 5) also remains fixed.

As shown in Figure 6.23 (blue and green circle markers), both ends of the range of process/thread configurations execute much slower than more balanced configurations. This is mirrored in the G-CUE durations (orange diamonds) up until 12 or 24 processes, suggesting that the G-CUE has a large impact on performance for those configurations. In case of 48 and especially 96 processes, however, the G-CUE duration detaches from the total run time: Both total and G-CUE durations show an increase in execution time (the G-CUE only in case of 96 processes), but the application’s run time appears to be mainly driven by factors other than the G-CUE. Consequently, the G-CUE-centric methodology is at best able to provide an incomplete explanation of the observed performance those cases. Other analysis approaches are required to fully understand the application performance. Figure 6.23 also shows a high variance in durations, especially in the total run time of the baseline executions.

While there are no event counts or critical path durations measured for the application kernel, NEST itself reports the duration of its simulation kernel. This is illustrated by means of x-shaped markers in Figure 6.23. The figure shows that the kernel does not suffer from the observed scaling issues and only represents a small

part of the total run time. This suggests that the observed performance issues are caused by code outside the simulation. While performance optimization in a scientific context commonly targets kernel times, the measurements show that it might be worthwhile to also consider other execution phases of an application.

Figure 6.24 shows the number of retired instructions and the number of unhalted clock cycles during the execution of NEST, as well as the resulting IPC metric. The amount of clock cycles (center plot) along the G-CUE and along the thread with the most clock cycles strongly increases with the number of OpenMP threads. The same trend is visible in the amount of retired instructions (left plot), suggesting that it could be the reason for the increase in clock cycles and therefore the reason for the loss in performance in case of few MPI processes. Only in case of 1×96 processes/threads, the G-CUE measurements clearly outnumber every individual thread. Thus, a thread-centric analysis that does not specifically consider the G-CUE would be able to find similar results. All but one thread in every execution are found close to the mean instruction and clock cycle counts; the raw TL-CUE data collected by OTF-CPT reveals that only thread 0 (the OpenMP primary thread [50]) exhibits the extreme increase in clock cycles and instruction counts given more OpenMP threads. Given the huge difference in instruction counts and therefore in executed instructions, a comparison of the IPC metric between different execution paths of the program is likely not meaningful (see Section 4.5). Both instruction and clock cycle counts clearly show variance between different executions.

To investigate potential reasons for the increase in overall instruction counts when increasing the number of OpenMP threads, a breakdown of individual groups of instructions is illustrated in Figure 6.25. The leftmost plot shows the total amount of retired instructions whereas all other plots represent event counts for individual

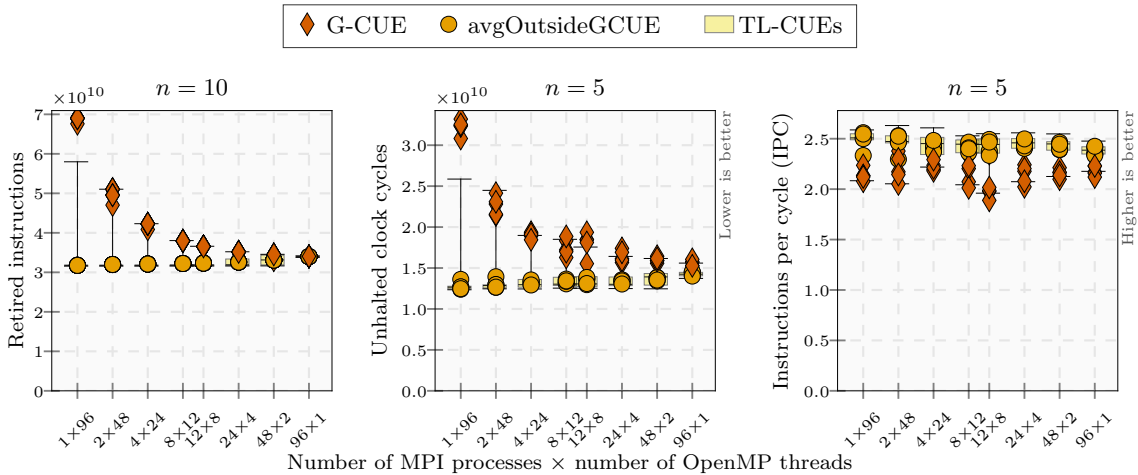


Figure 6.24.: NEST Simulator: Retired instructions and unhalted clock cycles as measured by OTF-CPT, as well as the resulting instructions per cycle (right).

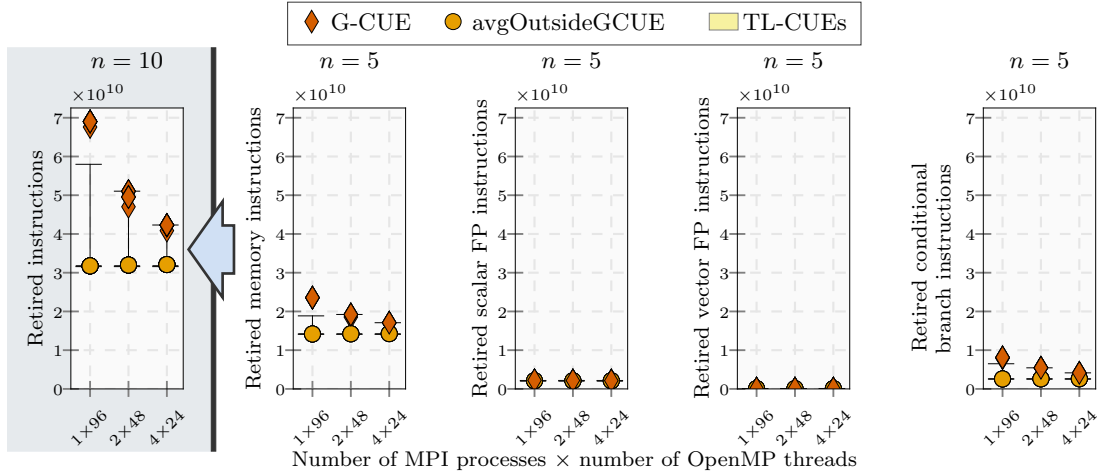


Figure 6.25.: NEST Simulator: Instruction counts for various types of instructions as measured by OTF-CPT. Only executions with one, two, and four processes are shown. The number of retired instructions on the left includes all other measured instruction counts as well as other instruction types that cannot be counted separately.

groups of instructions that count towards the overall instruction count. The figure clearly shows that the increase in total retired instructions along the G-CUE and the master thread cannot be solely attributed to one type of instruction; instead, several groups of instructions, including those that are not measured by the events represented by the four rightmost plots, are responsible for the overall increase. While there might be more fine-grained hardware performance events available on other processors that can help to further analyze the number of retired instructions, this is the point where the methodology reaches a limit: Due to the unknown structure of the G-CUE, it is not possible to reliably attribute the difference in instructions to certain regions in the source code.

A similar increase with the number of OpenMP threads can be found in the absolute amount of stalled cycles shown in the leftmost plot in Figure 6.26. However, this trend is no longer visible in the percentage of stalled cycles: Here, the G-CUE performs worse than the activities outside of it and worse than the majority of individual threads, but this pattern is visible to varying degrees over all process/thread configurations. This suggests that the increase in clock cycles with more OpenMP threads is indeed attributable to the number of instructions rather than the specific execution of those instructions. The amount of stalled cycles also show a huge variance between different executions. This might be (partly) attributed to variance found in the number of L2 cache misses, which are shown in Figure B.22, and the number of microcode assists related to page faults, which are shown in Figure B.23.

So far, the analysis has been primarily concerned with configurations using fewer MPI processes. As already noted, the G-CUE also clearly suffers from a loss of

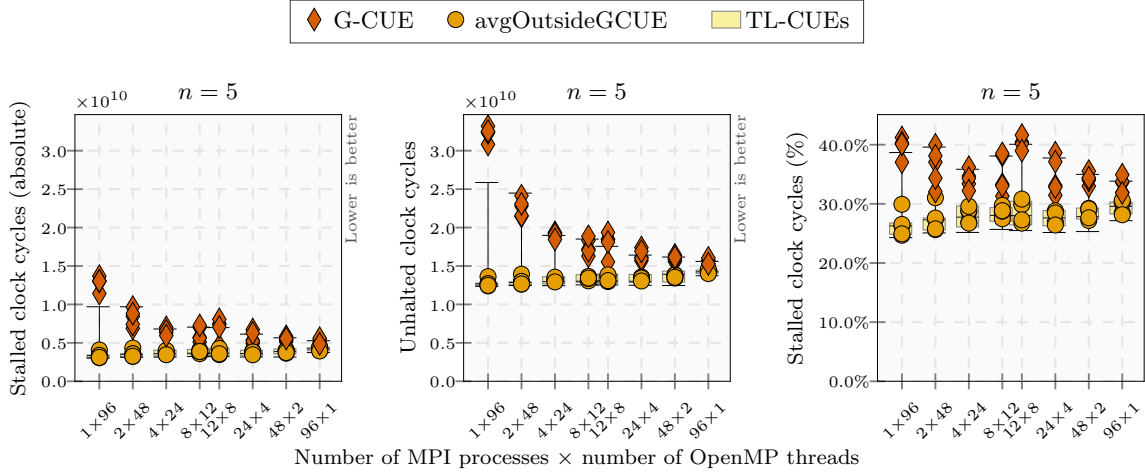


Figure 6.26.: NEST Simulator: Stalled clock cycles and unhalted clock cycles as measured by OTF-CPT, and their ratio.

performance in the 96×1 configuration as evident from its wall-clock duration. This cannot be explained with the hardware performance events listed in Table 5.1, and even separate measurements of the `CPU_CLK_UNHALTED.REF_TSC` hardware event, which represents CPU time independent of the CPU’s frequency, do not show this pattern.

While it is possible that the measurements are incorrect or inaccurate, the most likely explanation is found in the comparison of wall-clock times with event counts: Since hardware performance counters are registers, their values are bound to individual processes. Only hardware events that occur while a process is actively executed are counted in the context of the process. The same does not hold true for the wall-clock time, which obviously continues to increase even when a process is inactive. This gives rise to the hypothesis that there are more context switches along the G-CUE in case of 96 processes, affecting its total wall-clock duration.

In order to test this hypothesis, the amount of context switches was additionally measured via `/usr/bin/time` for a single execution of the NEST simulator while OTF-CPT was active. The measurements differentiate between *voluntary* context switches that occur, for example, due to a process being momentarily out of work [78], and *involuntary* context switches that occur when a process’s time slice expires [78]. The measured execution had over eleven times more voluntary context switches (`%w`) in the 96×1 configuration (2,172,541 in total) than in the 1×96 configuration (192,389 in total) and over 15 times more involuntary context switches (`%c`; 553,787 vs. 35,490). The amount of voluntary context switches was similar for all processes within the 96×1 configuration, but individual processes had up to six times more involuntary context switches than most others. While the measurements of the context switches cannot be mapped to the G-CUE, they indicate that the increase

6. Evaluation

in G-CUE duration for 96 MPI processes is indeed due to an increase in context switches and time in which the processes are not actively executed.

Interpreting the results of the 96×1 configuration is not straightforward as the reasons for the context switches and their impact on the G-CUE are not known. Given the slower G-CUE durations, the amount of voluntary context switches could hint at optimization opportunities that also include useful execution activities. More thorough investigations are needed to fully understand the underlying issues.

7. Conclusion

This thesis presented a methodology to analyze parallel application performance by means of hardware performance counters tracked along critical paths. At its core, the methodology consists of a comparison between metrics tracked along the global critical path of useful execution (G-CUE) and along individual thread-local critical paths of useful execution (TL-CUEs), as well as mean metric values outside the G-CUE (avgOutsideGCUE). The comparison with individual threads helps to understand whether a metric measured along the G-CUE potentially dictates the structure of the G-CUE, whereas the comparison with avgOutsideGCUE metrics gives a rough estimate of how the application might behave if the program activities on the G-CUE did not exist. Metrics are analyzed within a hierarchical top-down approach that follows a chain of indirect to direct potential causes in order to understand relationships between the hardware and the observed critical paths.

The methodology was tested with five different applications in order to evaluate the thesis's research questions. In all five cases, it was successfully used to find potential performance issues, some of which are not visible when only metrics accumulated along individual threads are analyzed. The Random Histogram application, which was deliberately written to produce a distinct G-CUE with a verifiable structure, revealed that the G-CUE may show performance issues more clearly than all TL-CUEs. For DENISE Black Edition, the G-CUE measurements could explain large parts of the application's scaling behavior whereas measurements along individual threads were unable to.

Depending on the availability of specific hardware performance events on given hardware, the methodology is able to identify potential reasons for the duration of a given G-CUE and can therefore identify factors that potentially limit useful execution performance. The evaluations of the Mergesort implementation and NEST revealed that their performance might be limited due to an uneven distribution of instructions along OpenMP tasks (Mergesort) and threads (NEST). In case of DENISE Black Edition, the G-CUE was found to be significantly affected by the amount of floating-point related microcode assists due to subnormal floating-point values.

For some of the evaluated applications, their configurations were specifically chosen such that the overhead of OTF-CPT, which was used as the measurement tool, was reasonably low. OTF-CPT suffers from large overhead with lots of fine-grained communication over many communication peers such as extensive uses of barriers. This is a natural consequence of performing an on-the-fly analysis, as every commu-

7. Conclusion

nication needs additional immediate handling causing an amplification of its impact on performance. An example for the potential overhead impact is SPMxV where a change to the OpenMP code outside useful execution activities also influenced the G-CUE. While it is possible that this is solely attributed to the impact of changing the program’s communication, it cannot be ruled out that some observed changes may be a result of the measurement overhead itself. Since many measured metrics remained similar between experiments with a varying degree of overhead (e.g., in case of Mergesort), it is possible that its impact on measurements may be negligible in some cases.

Since only critical paths of useful execution are evaluated, the methodology is unable to identify some performance issues related to communication between threads or processes, as evident in the evaluation of the NEST Simulator. For this reason, the methodology should be used alongside other performance analysis tools to get a complete view on application performance. The methodology is well-suited to create initial hypotheses which can then be analyzed in more depth by post-mortem tools.

A major limitation of the methodology is its inability to directly map the measurement results to individual regions in the source code of a tested application. This is a consequence of using metrics accumulated along execution paths without explicitly tracking their structure, which in turn is a result of using measurements based on on-the-fly analysis for which tracking the critical path’s structure is impracticable. The inability to map results to individual regions poses a major obstacle for analysis and in particular the resulting optimization. Due to this limitation, the evaluation in this thesis could only construct hypotheses in many situations—a full assessment of the methodology’s expressiveness is only possible with knowledge about the critical path’s structure. Performance analysis tools that keep track of the critical path’s structure (such as Scalasca) might assist in an analysis, but since they usually do not measure critical paths of useful execution, their supporting use for the methodology may be severely limited.

Two further limitations arise from the measurements’ accumulative nature: First, it is not generally possible to infer whether the observed critical path passes through the same program activities for multiple executions of the application. Significantly different measurements between runs may indicate variance in the critical path’s structure, but the reverse is not generally true. Second, a tracked critical path may consist of several distinct phases with greatly differing characteristics which cannot be detected by the measurements.

Since the methodology is heavily based on concepts of OTF-CPT, this thesis also discussed the necessary conditions and implications of an adaptation of the methodology for other tools. It comes to the conclusion that, while a direct adaptation is unlikely to work only given the current output capabilities of analysis tools such as Scalasca, an adaptation is possible in principle, given some modifications to the tools themselves.

7.1. Future Work

This thesis only implemented a proof-of-concept of the presented methodology, but further evaluation is required to fully understand the methodology’s strengths and shortcomings. This includes the impact of measurement overhead and measurement precision, which has not been fully evaluated in this thesis. To enable reliable analyses using the methodology’s implementation, further research on this aspect is required.

In order to better understand the reliability of measurements and to find a lower bound on optimization potential, it might be helpful to extend the methodology with a “minimum slack” metric (following the notion of *slack* introduced in [20] and [21]). This new metric would correspond to the duration difference between a measured critical path and the closest candidate path. Measuring this metric should be relatively straightforward to implement in on-the-fly analysis tools such as OTF-CPT: Whenever two or more paths merge into a single vertex in the program activity graph (e.g., at every MAXUPDATE in OTF-CPT), the difference in duration between the longest path and the next-longest path is compared with the current minimum slack value. If the minimum slack metric has a greater value or is not yet initialized, the newly computed difference replaces the metric value since it is the new minimum difference. From the perspective of the measurement tool, the final metric value then reflects the *closest decision* for what is considered a part of the critical path. The metric acts as an indicator for the temporal distinctiveness of the critical path and can be used to detect the existence of near-critical paths. It provides insight as to how much the overall performance can be guaranteed to improve by optimizing the critical path, before a different path might emerge as a new critical path. Moreover, a minimum slack metric much larger than the measurement overhead would imply that the overhead likely does not affect the structures of the observed critical paths.

In order to account for the limitation that the methodology cannot directly point at certain problematic regions in the source code of an application, it might be desirable to combine its results with measurements that track the share of selected regions along the critical path or to make use of a what-if analysis by means of logical zeroing [17, 22, 20] to estimate the impact of individual regions on the critical path. Both approaches are currently not implemented in OTF-CPT but should be feasible to implement.

The methodology focused on G-CUEs and TL-CUEs, but OTF-CPT also tracks *process-local critical paths of useful executions* (PL-CUEs) whose analysis might give additional insight. If a fix is found to enable reliable measurements of hardware counters along critical paths of outside OpenMP time (*COO* variants) and critical paths of outside MPI time (*COM* variants), the analysis might benefit from an inclusion of these variants.

A. SPMxV – Source Code

Listing A.1 provides the source code of the SPMxV kernel as used for the experiments presented in this thesis. It is a slightly adapted version of the SPMxV implementation of the Performance Optimisation and Productivity Centre of Excellence in HPC [63]. The `BENCHMARK_NOWAIT` define is used to toggle between the two variants presented in Section 6.3. `MEASUREMENT_START` and `MEASUREMENT_STOP` are macros to start and stop the measurements performed with OTF-CPT and Score-P.

```
1 MEASUREMENT_START("kernel_region");
2 t1 = omp_get_wtime();
3
4 #pragma omp parallel num_threads(tOptions->iNumThreads)
5 {
6     // run kernel iNumRepetitions times
7     for (int rep = 0; rep < iNumRepetitions; rep++)
8     {
9         // yAx-kernel
10        #ifdef BENCHMARK_NOWAIT
11            #pragma omp for schedule(static) nowait
12        #else
13            #pragma omp for schedule(static)
14        #endif
15        for (int i = 0; i < tInput->stNumRows; i++)
16        {
17            double sum = 0.0;
18            int rowbeg = Arow[i];
19            int rowend = Arow[i+1];
20            for (int nz = rowbeg; nz < rowend; nz++)
21            {
22                sum += Aval[nz] * x[ Acol[nz] ];
23            }
24            y[i] = sum;
25        }
26    }
27 }
28
29 t2 = omp_get_wtime(); // t2 - t1 = total kernel duration
30 MEASUREMENT_STOP();
```

Listing A.1: SPMxV kernel source code.

B. Additional Measurements

This appendix lists some additional measurements that were not dealt with in detail within the main chapters of this thesis.

B.1. Random Histogram

B.1.1. *allBounded* = false, *noStageOffset* = false

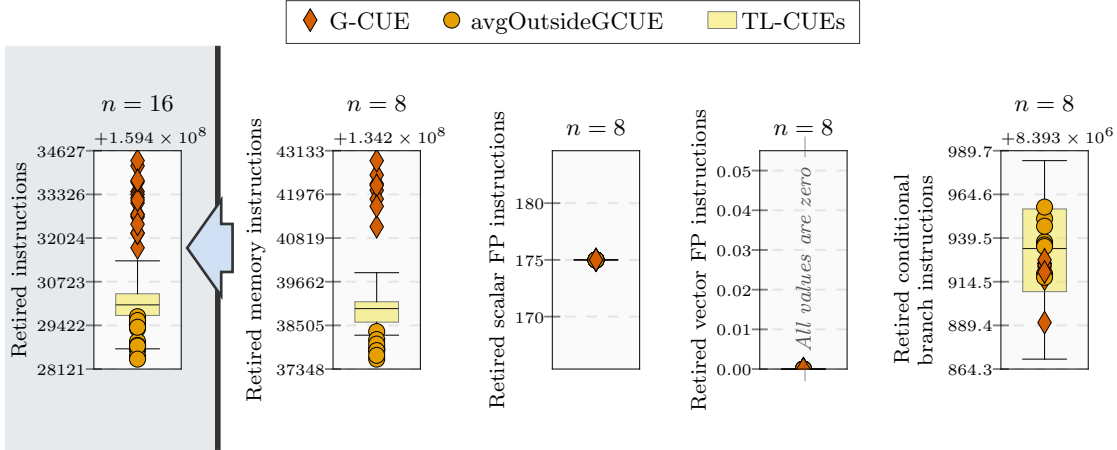


Figure B.1.: Random Histogram: Instruction counts for various types of instructions as measured by OTF-CPT. The number of retired instructions on the left includes all other measured instruction counts as well as other instruction types that cannot be counted separately.

B.1.2. *allBounded* = false, *noStageOffset* = true

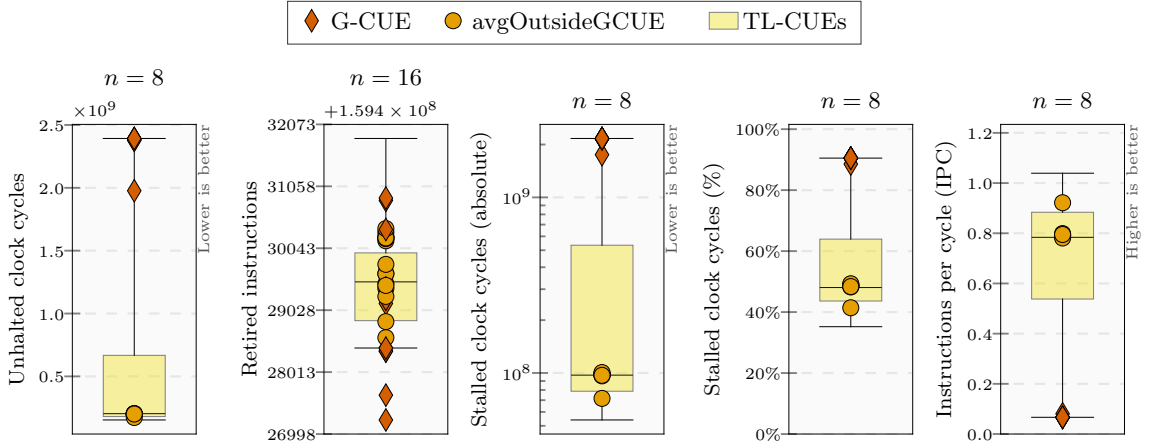


Figure B.2.: Random Histogram (*noStageOffset*): Unhalted clock cycles, retired instructions, stalled cycles, and the IPC, as measured by OTF-CPT for the kernel.

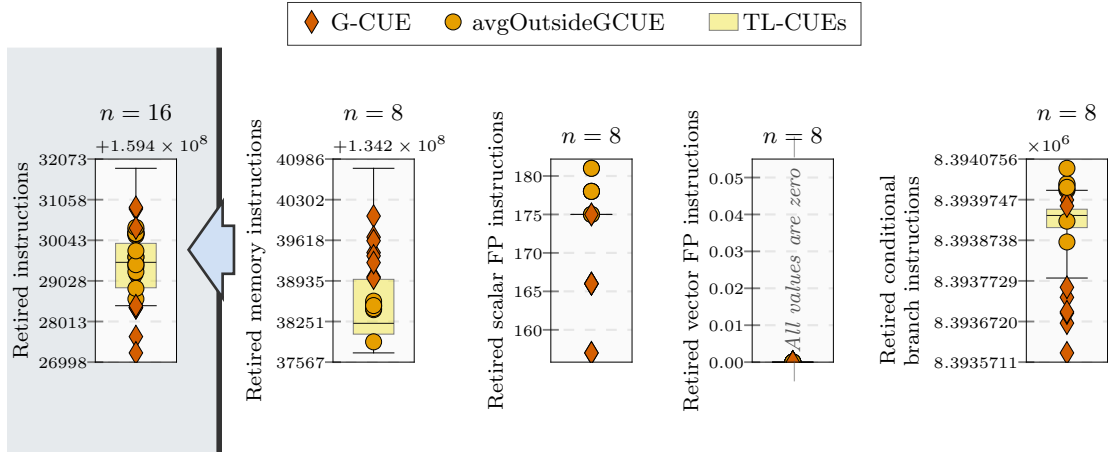


Figure B.3.: Random Histogram (*noStageOffset*): Instruction counts for various types of instructions as measured by OTF-CPT. The number of retired instructions on the left includes all other measured instruction counts as well as other instruction types that cannot be counted separately.

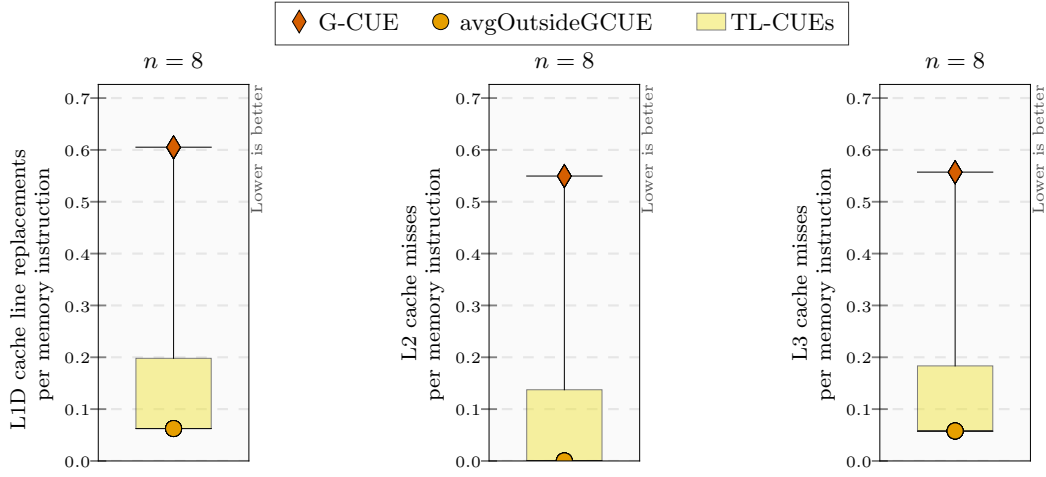


Figure B.4.: Random Histogram (*noStageOffset*): Cache-related metrics derived from OTF-CPT measurements of the kernel.

B.1.3. *allBounded* = true, *noStageOffset* = false

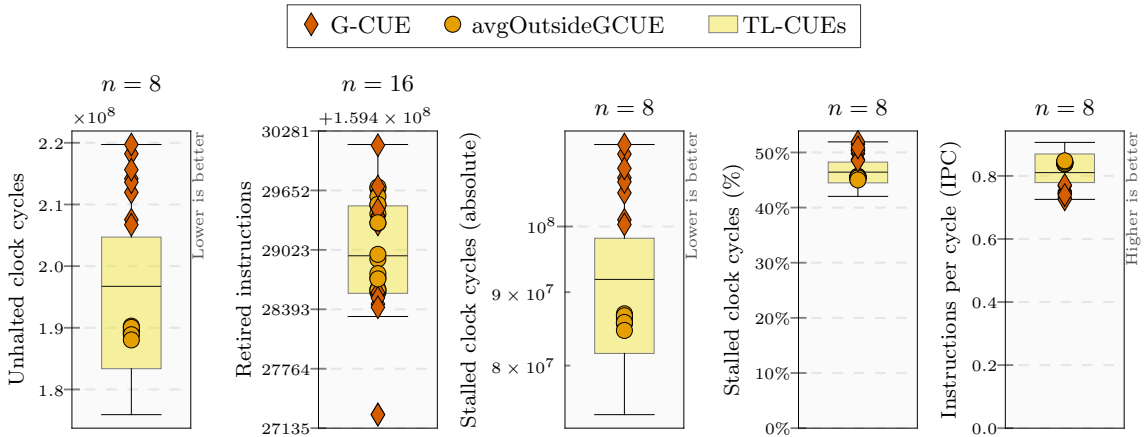


Figure B.5.: Random Histogram (*allBounded*): Unhalted clock cycles, retired instructions, stalled cycles, and the IPC, as measured by OTF-CPT for the kernel.

B. Additional Measurements

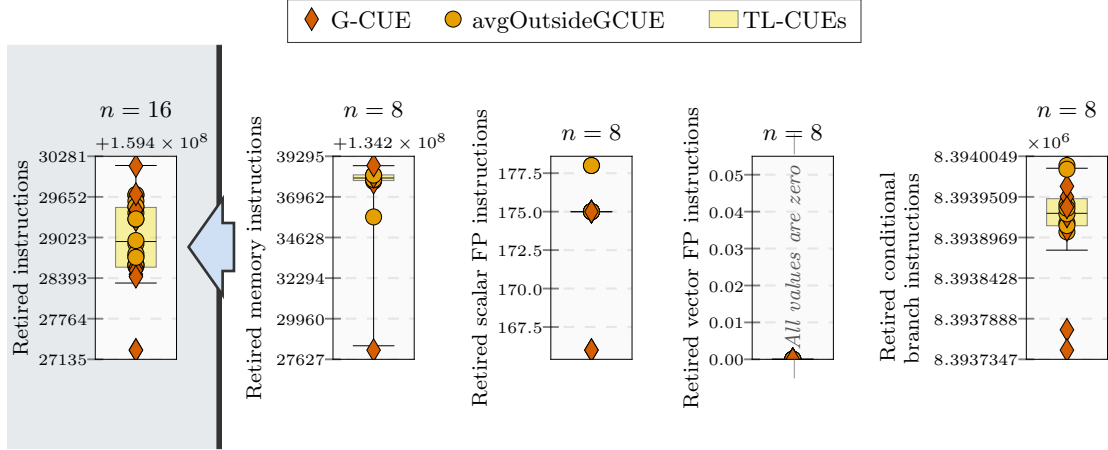


Figure B.6.: Random Histogram (*allBounded*): Instruction counts for various types of instructions as measured by OTF-CPT. The number of retired instructions on the left includes all other measured instruction counts as well as other instruction types that cannot be counted separately.

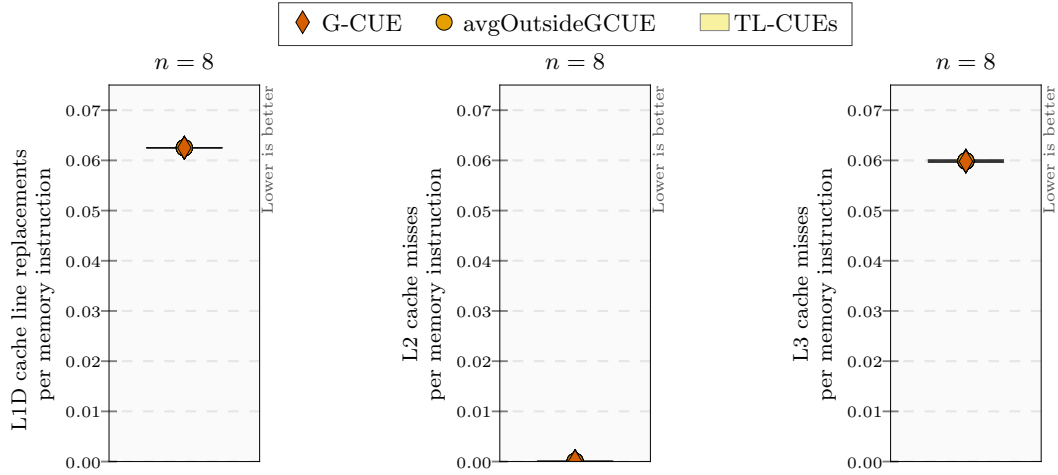


Figure B.7.: Random Histogram (*allBounded*): Cache-related metrics derived from OTF-CPT measurements of the kernel. The box plots corresponding to all TL-CUEs appear as a single line due to the measurements' minimal variability.

B.2. Mergesort

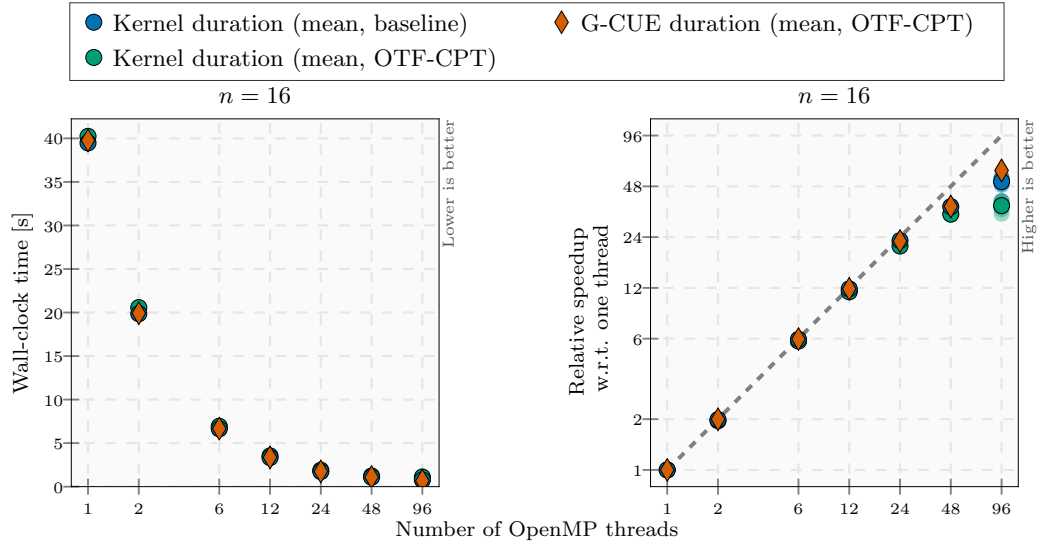


Figure B.8.: Figure 6.4 with the four outliers removed.

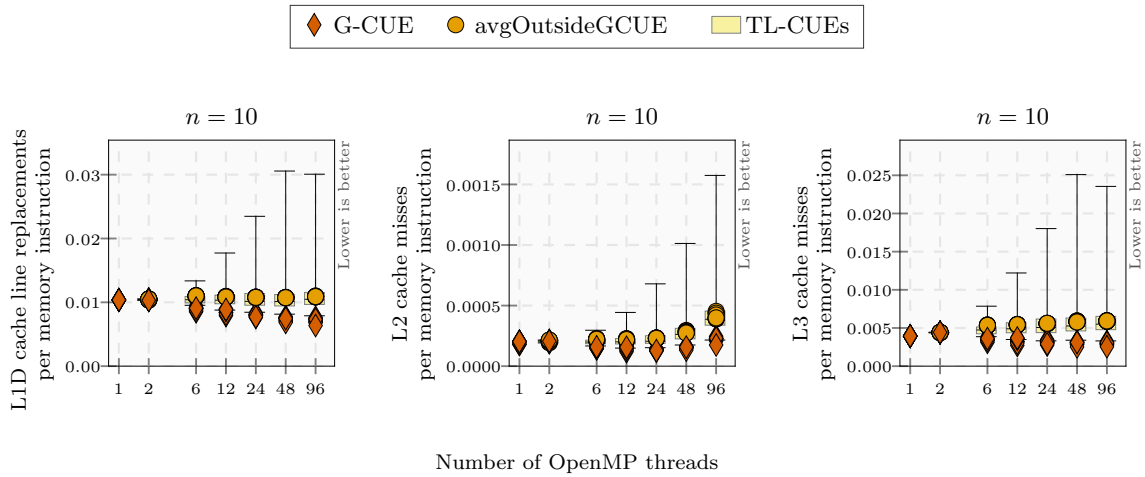


Figure B.9.: Mergesort: Cache-related metrics derived from OTF-CPT measurements of the kernel.

B. Additional Measurements

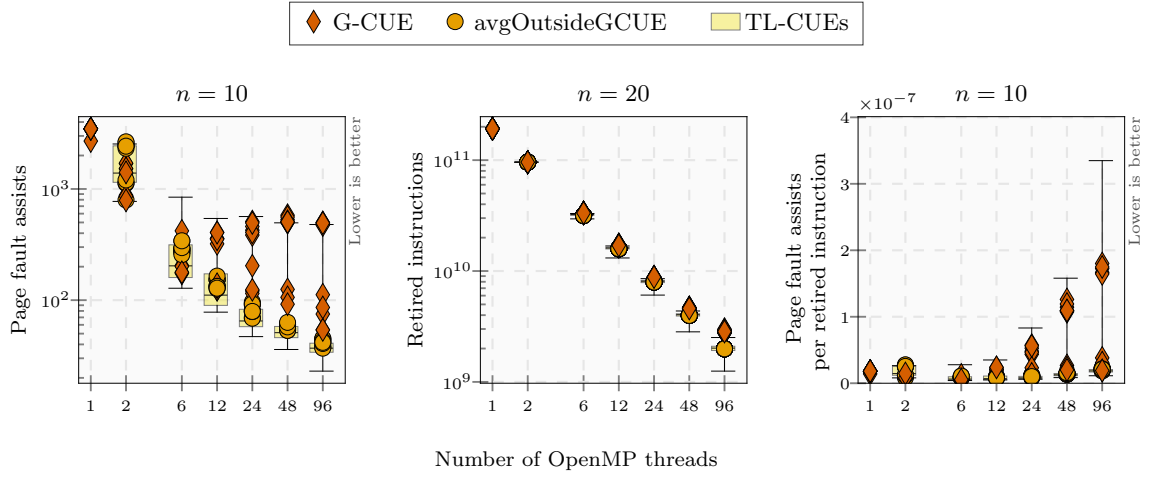


Figure B.10.: Mergesort: Page fault assists and retired instructions as measured by OTF-CPT for the kernel, and their ratio.

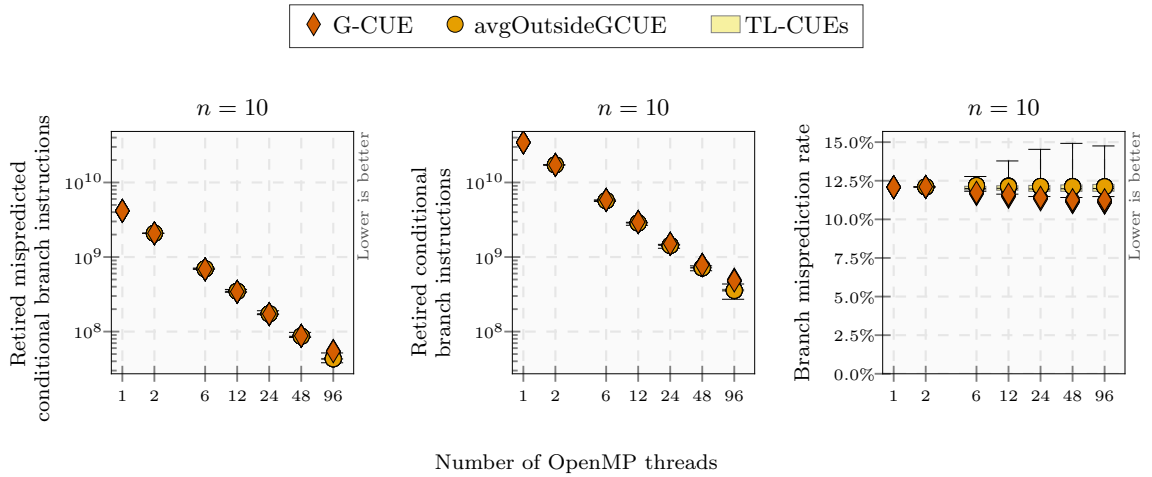


Figure B.11.: Mergesort: Mispredicted branch instructions and all retired branch instructions as measured by OTF-CPT for the kernel, and their ratio.

B.3. Sparse Matrix-Vector Multiplication

B.3.1. Variant 1: Without `nowait`

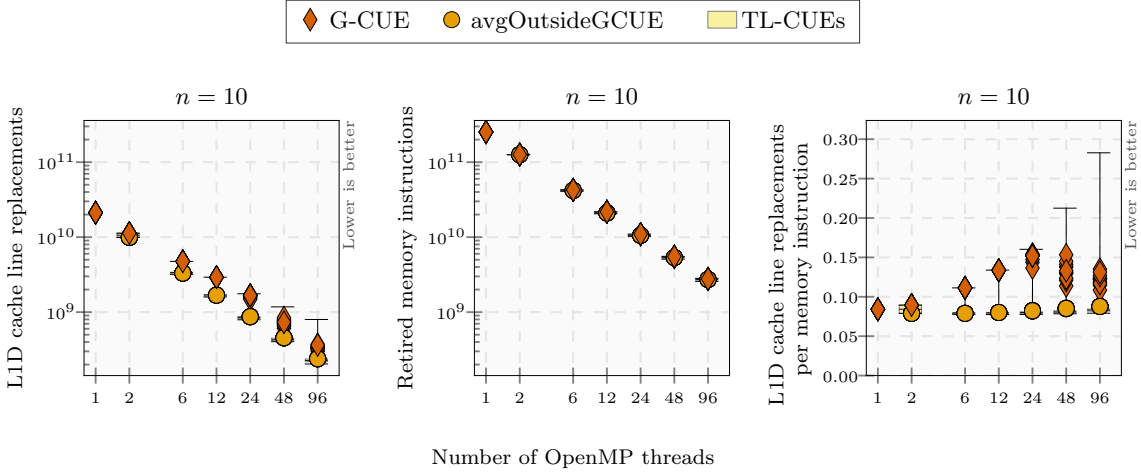


Figure B.12.: SPMxV: L1D cache line replacements and retired memory instructions as measured by OTF-CPT for the kernel, and their ratio.

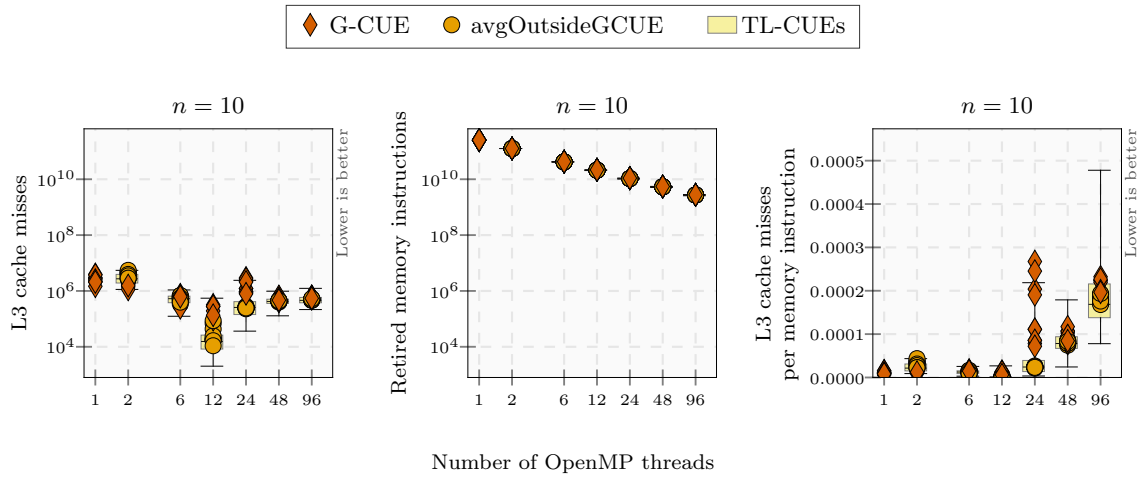


Figure B.13.: SPMxV: L3 cache misses and retired memory instructions as measured by OTF-CPT for the kernel, and their ratio.

B.3.2. Variant 2: With `nowait`

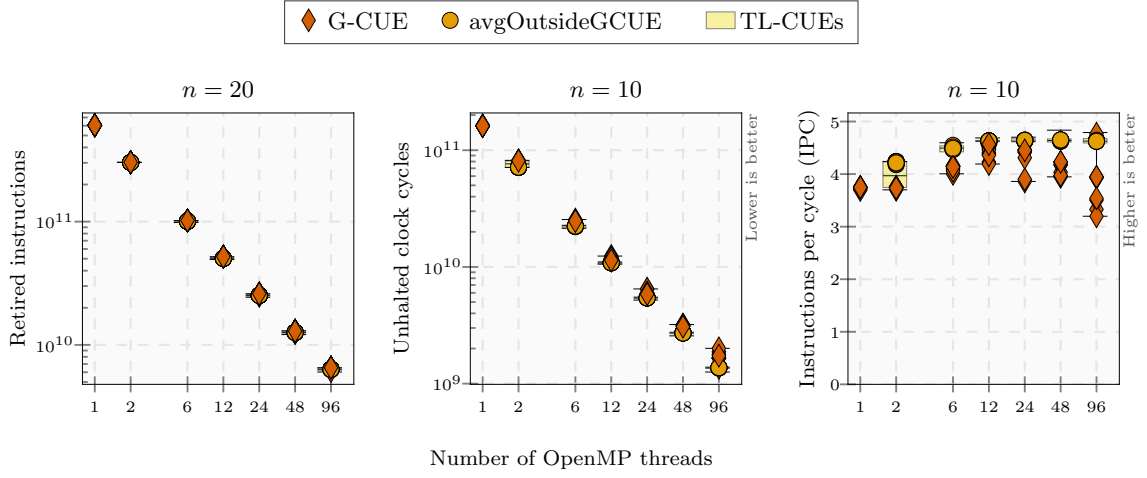


Figure B.14.: SPMxV (`nowait`): Retired instructions and unhalted clock cycles as measured by OTF-CPT for the kernel, as well as the resulting instructions per cycle (IPC).

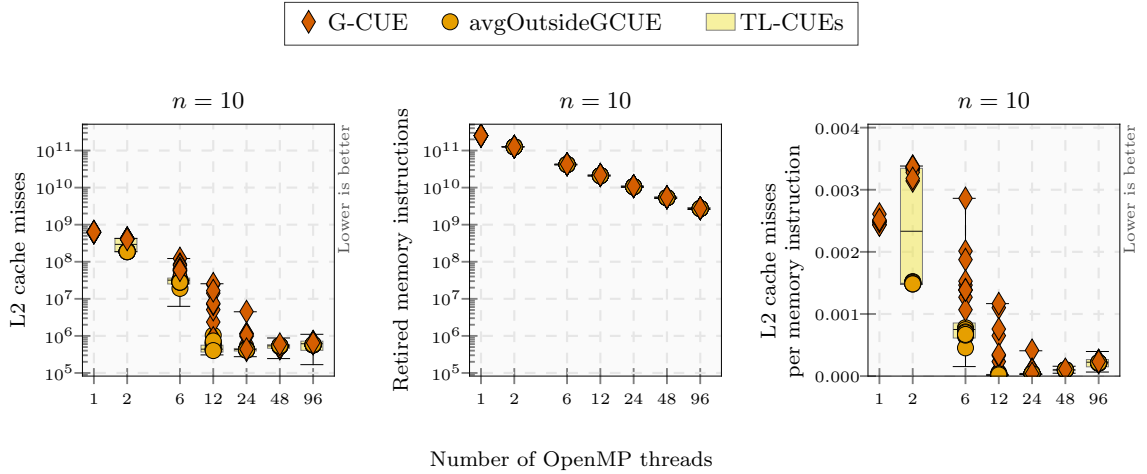


Figure B.15.: SPMxV (`nowait`): L2 cache misses and retired memory instructions as measured by OTF-CPT for the kernel, and their ratio.

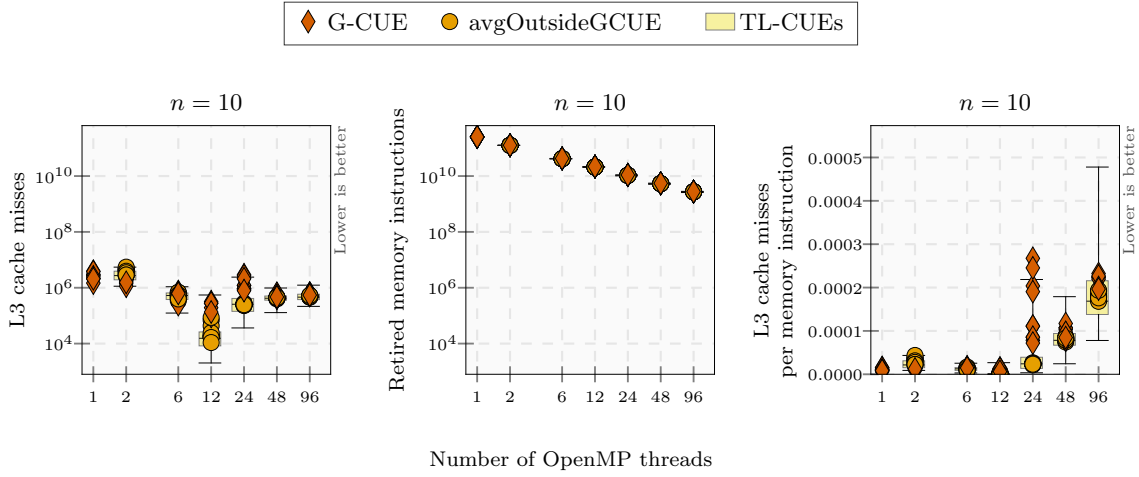


Figure B.16.: SPMxV (nowait): L3 cache misses and retired memory instructions as measured by OTF-CPT for the kernel, and their ratio.

B.4. DENISE Black Edition

B.4.1. FTZ & DAZ Disabled

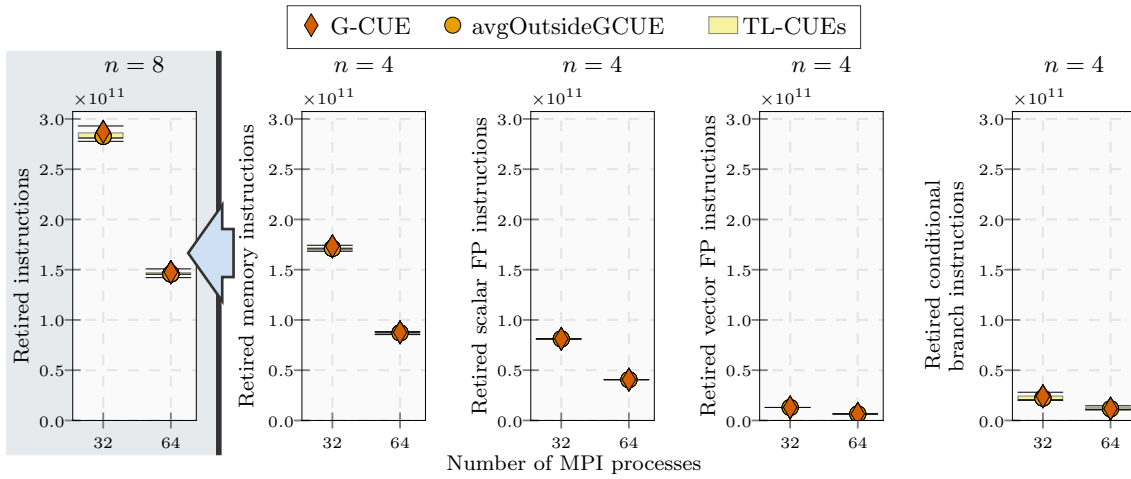


Figure B.17.: DENISE (FTZ & DAZ disabled): Instruction counts for various types of instructions as measured by OTF-CPT. Only executions with 32 and 64 processes are shown. The number of retired instructions on the left includes all other measured instruction counts as well as other instruction types that cannot be counted separately.

B. Additional Measurements

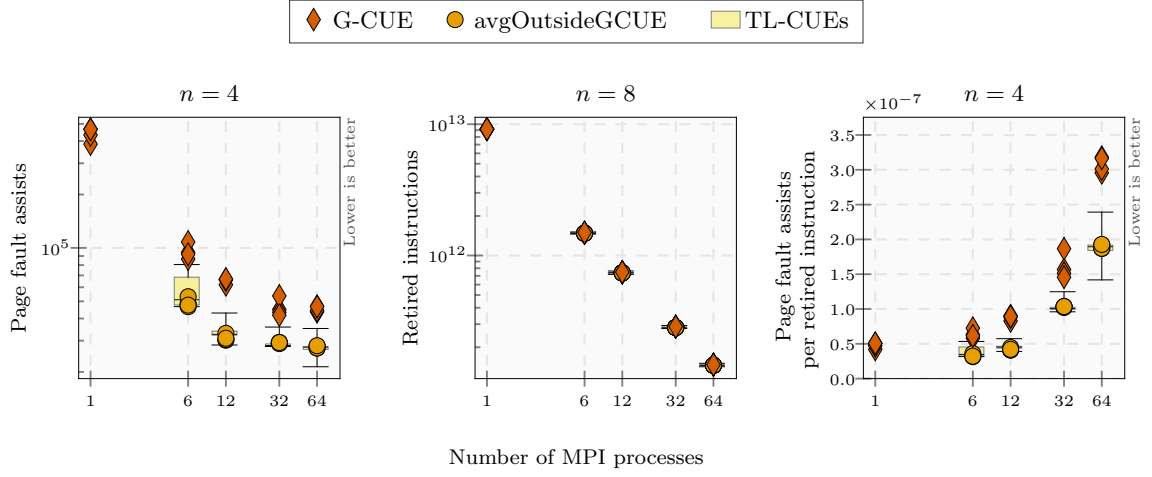


Figure B.18.: DENISE (FTZ & DAZ disabled): Page fault assists and retired instructions as measured by OTF-CPT, and their ratio.

B.4.2. FTZ & DAZ Enabled

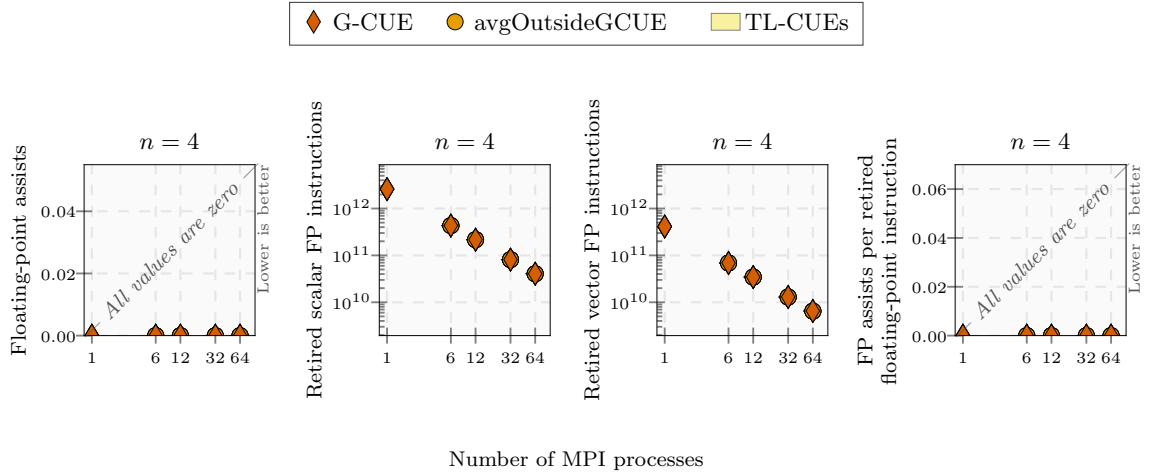


Figure B.19.: DENISE (FTZ+DAZ): Floating-point assists and floating-point instructions as measured by OTF-CPT, and their ratio.

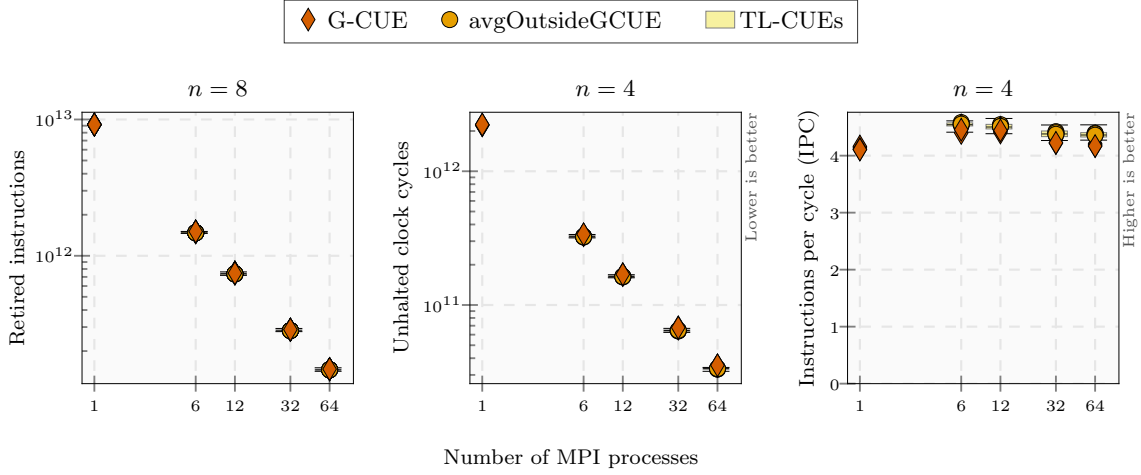


Figure B.20.: DENISE (FTZ+DAZ): Retired instructions and unhalted clock cycles as measured by OTF-CPT, as well as the resulting instructions per cycle (IPC).

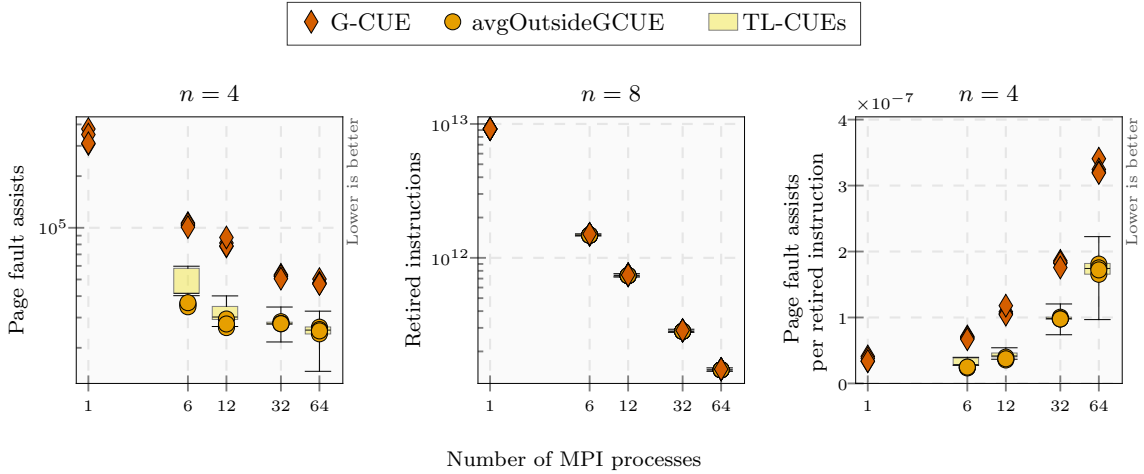


Figure B.21.: DENISE (FTZ+DAZ): Page fault assists and retired instructions as measured by OTF-CPT, and their ratio.

B.5. NEST Simulator

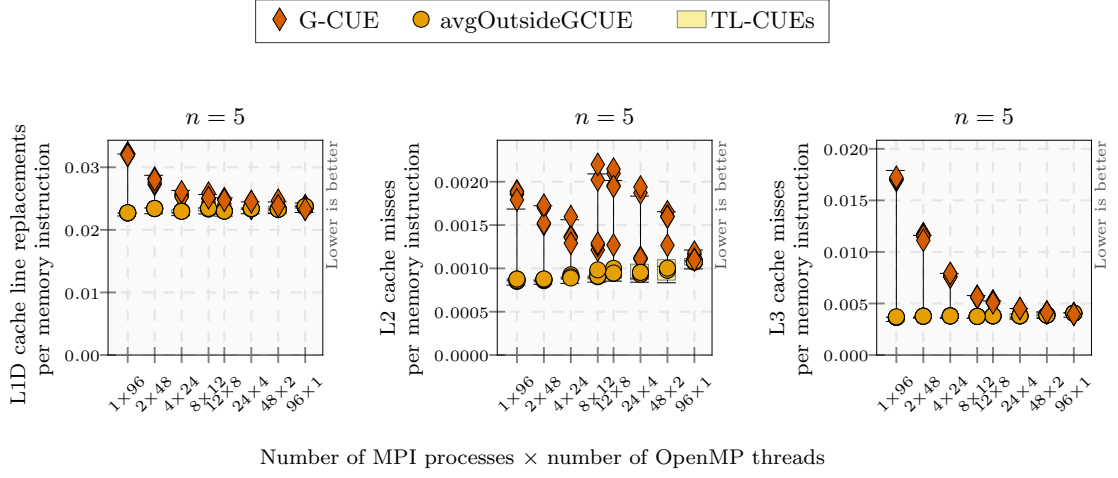


Figure B.22.: NEST Simulator: Cache-related metrics derived from OTF-CPT measurements.

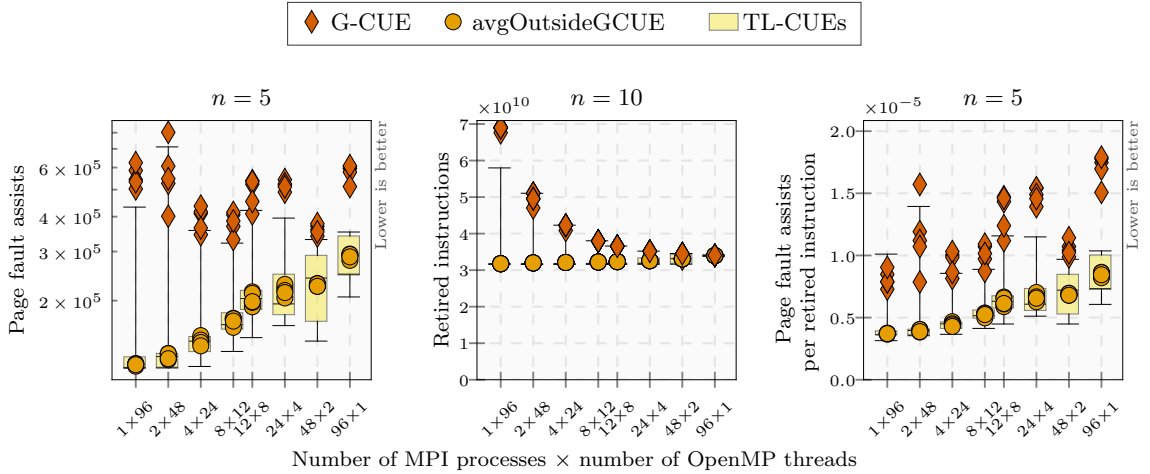


Figure B.23.: NEST Simulator: Page fault assists and retired instructions as measured by OTF-CPT, and their ratio.

List of Figures

3.1.	Example execution sequence for three threads highlighting the critical path.	12
3.2.	OTF-CPT implementation for a matching pair of <code>MPI_Send()</code> and <code>MPI_Recv()</code> calls.	17
3.3.	Example sequence of program activities for three threads highlighting the G-CUE and a fictional hardware event.	18
4.1.	The hierarchical workflow used in the methodology.	26
5.1.	Local histograms of all processes after the first stage of the Random Histogram application.	40
5.2.	Structure of the sparse matrix used for the SPMxV experiments. . .	43
6.1.	Score-P execution trace of a single execution of the <i>Random Histogram</i> benchmark with four processes.	49
6.2.	Random Histogram: Unhalted clock cycles, retired instructions, stalled cycles, and the IPC.	49
6.3.	Random Histogram: Cache-related metrics.	51
6.4.	Mergesort: Kernel and G-CUE durations as well as the resulting speedup.	53
6.5.	Mergesort: Retired instructions, unhalted clock cycles, and the resulting IPC.	55
6.6.	Mergesort: Stalled clock cycles, unhalted clock cycles, and their ratio. . .	56
6.7.	Mergesort: Instruction counts for various types of instructions. . . .	57
6.8.	SPMxV: Kernel and G-CUE durations as well as the resulting speedup for a varying number of OpenMP threads.	59
6.9.	SPMxV: Retired instructions, unhalted clock cycles, and the resulting IPC.	60
6.10.	SPMxV: Stalled clock cycles, unhalted clock cycles, and their ratio. .	60
6.11.	SPMxV: L2 cache misses, retired memory instructions, and their ratio.	61
6.12.	SPMxV: Page fault assists, retired instructions, and their ratio. . . .	62
6.13.	SPMxV (<code>nowait</code>): Kernel and G-CUE durations as well as the resulting speedup for a varying number of OpenMP threads.	63
6.14.	SPMxV (<code>nowait</code>): Stalled clock cycles, unhalted clock cycles, and their ratio.	63

List of Figures

6.15. SPMxV (nowait): Page fault assists, retired instructions, and their ratio.	64
6.16. SPMxV (nowait): L1D cache line replacements, retired memory instructions, and their ratio.	64
6.17. DENISE Black Edition (FTZ & DAZ disabled): Total execution times, G-CUE durations, and the resulting speedup.	65
6.18. DENISE Black Edition (FTZ & DAZ disabled): Retired instructions, unhalting clock cycles, and the resulting IPC.	66
6.19. DENISE Black Edition (FTZ & DAZ disabled): Stalled clock cycles, unhalting clock cycles, and their ratio.	66
6.20. DENISE Black Edition (FTZ & DAZ disabled): Floating-point assists, floating-point instructions, and their ratio.	67
6.21. DENISE Black Edition (FTZ+DAZ): Stalled clock cycles, unhalting clock cycles, and their ratio.	68
6.22. DENISE Black Edition (FTZ+DAZ): Total execution times, G-CUE durations, and the resulting speedup.	69
6.23. NEST Simulator: Various execution durations and the resulting speedup for a varying distribution of MPI processes and OpenMP threads. . .	70
6.24. NEST Simulator: Retired instructions, unhalting clock cycles, and the resulting IPC.	71
6.25. NEST Simulator: Instruction counts for various types of instructions. . .	72
6.26. NEST Simulator: Stalled clock cycles, unhalting clock cycles, and their ratio.	73

List of Tables

5.1. The two sets of Intel hardware performance events counted during the experiments.	35
5.2. Configuration of the NEST HPC benchmark as used in the experiments of this thesis.	45

Bibliography

- [1] J. Protze, F. Orland, K. Haldar, T. Koritzius, and C. Terboven, “On-the-Fly Calculation of Model Factors for Multi-paradigm Applications”, in *Euro-Par 2022: Parallel Processing*, ser. Lecture Notes in Computer Science, J. Cano and P. Trinder, Eds., vol. 13440, Cham: Springer International Publishing, 2022, pp. 69–84. DOI: 10.1007/978-3-031-12597-3_5
- [2] J. Treibig, G. Hager, and G. Wellein, “LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments”, in *2010 39th International Conference on Parallel Processing Workshops*, San Diego, CA, USA: IEEE, 2010, pp. 207–216. DOI: 10.1109/ICPPW.2010.38
- [3] A. Knüpfer et al., “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir”, in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds., Berlin, Heidelberg: Springer, 2012, pp. 79–91. DOI: 10.1007/978-3-642-31476-6_7
- [4] L. Streichardt, “Dependent Metrics for On-the-Fly Critical Path Analysis”, Bachelor Thesis, RWTH Aachen University, Aachen, Germany, 2026. DOI: <https://doi.org/10.18154/RWTH-2026-04073>
- [5] J. E. Kelley and M. R. Walker, “Critical-Path Planning and Scheduling”, in *Proceedings of the Eastern Joint Computer Conference*, vol. 16, Boston, Massachusetts: ACM Press, 1959, pp. 160–173. DOI: 10.1145/1460299.1460318
- [6] C.-Q. Yang and B. Miller, “Critical path analysis for the execution of parallel and distributed programs”, in *Proceedings: The 8th International Conference on Distributed Computing Systems*, San Jose, CA, USA: IEEE Comput. Soc. Press, 1988, pp. 366–373. DOI: 10.1109/DCS.1988.12538
- [7] M. Schulz, “Extracting Critical Path Graphs from MPI Applications”, in *2005 IEEE International Conference on Cluster Computing*, Burlington, MA, USA: IEEE, 2005, pp. 1–10. DOI: 10.1109/CLUSTER.2005.347035
- [8] I. Dooley and L. V. Kale, “Detecting and Using Critical Paths at Runtime in Message Driven Parallel Programs”, in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Atlanta, GA, USA: IEEE, 2010. DOI: 10.1109/IPDPSW.2010.5470844

- [9] F. Schmitt, R. Dietrich, and G. Juckeland, “Scalable critical-path analysis and optimization guidance for hybrid MPI-CUDA applications”, *The International Journal of High Performance Computing Applications*, vol. 31, no. 6, pp. 485–498, 2017. DOI: 10.1177/1094342016661865
- [10] D. Böhme, F. Wolf, and M. Geimer, “Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications”, in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, Shanghai, China: IEEE, 2012, pp. 2538–2541. DOI: 10.1109/IPDPSW.2012.321
- [11] J. K. Hollingsworth, “An Online Computation of Critical Path Profiling”, in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools - SPDT '96*, Philadelphia, Pennsylvania, United States: ACM Press, 1996, pp. 11–20. DOI: 10.1145/238020.238024
- [12] D. Böhme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer, “Scalable Critical-Path Based Performance Analysis”, in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, Shanghai, China: IEEE, 2012, pp. 1330–1340. DOI: 10.1109/IPDPS.2012.120
- [13] D. Böhme, *Characterizing Load and Communication Imbalance in Parallel Applications* (Schriften des Forschungszentrums Jülich / IAS Series 23). Jülich: Forschungszentrum Jülich, Zentralbibliothek, 2014, ISBN: 978-3-89336-940-9.
- [14] D. Bohme, M. Geimer, F. Wolf, and L. Arnold, “Identifying the root causes of wait states in large-scale parallel applications”, in *2010 39th International Conference on Parallel Processing*, San Diego, CA, USA: IEEE, 2010, pp. 90–100. DOI: 10.1109/ICPP.2010.18
- [15] M.-A. Hermanns, M. Miklosch, D. Böhme, and F. Wolf, “Understanding the Formation of Wait States in Applications with One-sided Communication”, in *Proceedings of the 20th European MPI Users’ Group Meeting*, Madrid, Spain, 2013, pp. 73–78. DOI: 10.1145/2488551.2488569
- [16] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, “The Scalasca performance toolset architecture”, *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010. DOI: 10.1002/cpe.1556
- [17] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski, “IPS-2: The Second Generation of a Parallel Program Measurement System”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 206–217, 1990. DOI: 10.1109/71.80132
- [18] C. Alexander, D. Reese, and J. Harden, “Near-Critical Path Analysis of Program Activity Graphs”, in *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Durham, NC, USA: IEEE, 1994, pp. 308–317. DOI: 10.1109/MASCOT.1994.284406

- [19] C. A. Alexander, A. B. Lambert, D. S. Reese, J. C. Harden, and R. B. Brightwell, “Near-Critical Path Analysis: A Tool for Parallel Program Optimization”, in *Proceedings of the First Southern Symposium on Computing*, The University of Southern Mississippi, 1998.
- [20] J. K. Hollingsworth and B. P. Miller, “Parallel Program Performance Metrics: A Comparison and Validation”, in *Supercomputing '92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Minneapolis, MN, USA: IEEE Comput. Soc. Press, 1992, pp. 4–13. DOI: 10.1109/SUPERC.1992.236715
- [21] B. P. Miller and J. K. Hollingsworth, “Slack: A New Performance Metric for Parallel Programs”, University of Wisconsin-Madison, Madison, WI, USA, Tech. Rep. TR1260, 1994.
- [22] J. K. Hollingsworth, R. B. Irvin, and B. P. Miller, “The integration of application and system based metrics in a parallel program performance tool”, in *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA: Association for Computing Machinery, 1991, pp. 189–200. DOI: 10.1145/109625.109645
- [23] C. Rosas, J. Giménez, and J. Labarta, “Scalability prediction for fundamental performance factors”, *Supercomputing Frontiers and Innovations*, vol. 1, no. 2, pp. 4–19, 2014. DOI: 10.14529/jsfi140201 Accessed: Mar. 4, 2026.
- [24] B. J. Overeinder and P. M. A. Sloot, “Parallel performance evaluation through critical path analysis”, in *High-Performance Computing and Networking*, ser. Lecture Notes in Computer Science, B. Hertzberger and G. Serazzi, Eds., vol. 919, Berlin, Heidelberg: Springer, 1995, pp. 634–639. DOI: <https://doi.org/10.1007/BFb0046692>
- [25] C. Herold, O. Krzikalla, and A. Knüpfer, “Optimizing One-Sided Communication of Parallel Applications using Critical Path Methods”, in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Orlando, FL, USA: IEEE, 2017, pp. 567–576. DOI: 10.1109/IPDPSW.2017.64
- [26] A. Shatalin, V. Slobodskoy, and M. Fatin, “Root Causing MPI Workloads Imbalance Issues via Scalable MPI Critical Path Analysis”, in *Supercomputing*, ser. Lecture Notes in Computer Science, V. Voevodin, S. Sobolev, M. Yakobovskiy, and R. Shagaliyev, Eds., vol. 13708, Cham: Springer International Publishing, 2022, pp. 501–521. DOI: 10.1007/978-3-031-22941-1_37
- [27] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge, “Full-System Critical Path Analysis”, in *ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, TX, USA: IEEE, 2008, pp. 63–74. DOI: 10.1109/ISPASS.2008.4510739
- [28] *Perf(1) - Linux manual page*, 2024. Accessed: Jun. 29, 2026. [Online]. Available: <https://www.man7.org/linux/man-pages/man1/perf.1.html>

- [29] H. Jagode, A. Danalis, G. Congiu, D. Barry, A. Castaldo, and J. Dongarra, “Advancements of PAPI for the exascale generation”, *The International Journal of High Performance Computing Applications*, vol. 39, no. 2, pp. 251–268, 2025. DOI: 10.1177/10943420241303884
- [30] Intel Corporation, *Intel® VTune™ Profiler*. Accessed: Jun. 30, 2026. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>
- [31] L. Adhianto et al., “HPCTOOLKIT: Tools for performance analysis of optimized parallel programs”, *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010. DOI: 10.1002/cpe.1553
- [32] V. Lopez, G. Ramirez Miranda, and M. Garcia-Gasulla, “TALP: A Lightweight Tool to Unveil Parallel Efficiency of Large-scale Executions”, in *Proceedings of the 2021 on Performance Engineering, Modelling, Analysis, and Visualization Strategy*, Virtual Event, Sweden: Association for Computing Machinery, 2021, pp. 3–10. DOI: 10.1145/3452412.3462753
- [33] J. Treibig, G. Hager, and G. Wellein, “Performance Patterns and Hardware Metrics on Modern Multicore Processors: Best Practices for Performance Engineering”, in *Euro-Par 2012: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, I. Caragiannis et al., Eds., vol. 7640, Berlin, Heidelberg: Springer, 2013, pp. 451–460. DOI: 10.1007/978-3-642-36949-0_50
- [34] A. Yasin, “A Top-Down Method for Performance Analysis and Counters Architecture”, in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, USA: IEEE, 2014, pp. 35–44. DOI: 10.1109/ISPASS.2014.6844459
- [35] A. Dutilleul et al., *Performance Debugging through Microarchitectural Sensitivity and Causality Analysis*, arXiv:2412.13207 [cs.DC], 2024. DOI: 10.48550/arXiv.2412.13207
- [36] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. H. Campbell, “ADP: Automated diagnosis of performance Pathologies Using Hardware Events”, *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, pp. 283–294, 2012. DOI: 10.1145/2318857.225479
- [37] A. Tarraf, A. GeiB, L. Fuchs, and F. Wolf, “Fantastic Hardware Counters and How to Find Them: Automating the Detection of Noise-Resilient Performance Counters in HPC”, in *Proceedings of the SC ’25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, St. Louis, MO, USA: Association for Computing Machinery, 2025, pp. 1587–1600. DOI: 10.1145/3731599.3767517

- [38] V. M. Weaver, D. Terpstra, and S. Moore, “Non-determinism and overcount on modern hardware performance counter implementations”, in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, USA: IEEE, 2013, pp. 215–224. DOI: 10.1109/ISPASS.2013.6557172
- [39] D. Zaparanuks, M. Jovic, and M. Hauswirth, “Accuracy of Performance Counter Measurements”, in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, Boston, MA, USA: IEEE, 2009, pp. 23–32. DOI: 10.1109/ISPASS.2009.4919635
- [40] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, 2025. Accessed: Dec. 8, 2025. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- [41] Arm Limited, *Arm Cortex-R82 Processor Technical Reference Manual*, 2025. Accessed: Apr. 13, 2026. [Online]. Available: <https://developer.arm.com/documentation/102670/0301>
- [42] *PerfMon Events*. Accessed: Dec. 8, 2025. [Online]. Available: <https://perfmon-events.intel.com/>
- [43] V. M. Weaver, *System-wide Performance Counter Measurements: Offcore, Uncore, and Northbridge Performance Events in Modern Processors*, UMaine VMW Group Tech Report: UMAINE-VMW-TR-UNCORE-OFFCORE-2015-10, 2017. Accessed: Feb. 9, 2026. [Online]. Available: https://web.eece.maine.edu/~vweaver/projects/perf_events/uncore/offcore_uncore.pdf
- [44] A. Danalis, H. Jagode, Hanumantharayappa, S. Ragate, and J. Dongarra, “Counter Inspection Toolkit: Making Sense Out of Hardware Performance Events”, in *Tools for High Performance Computing 2017*, C. Niethammer, M. M. Resch, W. E. Nagel, H. Brunst, and H. Mix, Eds., Cham: Springer International Publishing, 2019, pp. 17–37. DOI: 10.1007/978-3-030-11987-4_2
- [45] Y. Yang, P. Qiu, C. Wang, Y. Jin, D. Wang, and G. Qu, *Exploration and Exploitation of Hidden PMU Events*, arXiv:2304.12072 [cs], 2023. DOI: 10.48550/arXiv.2304.12072
- [46] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, “Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker”, in *Proceedings of the 31st USENIX Security Symposium*, Boston, Massachusetts, USA: USENIX Association, 2022, pp. 699–716. Accessed: Feb. 8, 2026. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-zirui>
- [47] Arm Limited, *Arm® Architecture Reference Manual for A-profile architecture*, 2026. Accessed: Apr. 13, 2026. [Online]. Available: <https://developer.arm.com/documentation/ddi0487/mb>

- [48] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard – Version 4.0*, 2021. Accessed: Apr. 27, 2026. [Online]. Available: <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>
- [49] A. E. Eichenberger et al., “OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis”, in *OpenMP in the Era of Low Power Devices and Accelerators*, ser. Lecture Notes in Computer Science, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds., vol. 8122, Berlin, Heidelberg: Springer, 2013, pp. 171–185. DOI: 10.1007/978-3-642-40698-0_13
- [50] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, 2020. Accessed: Apr. 27, 2026. [Online]. Available: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf>
- [51] The Scalasca Development Team, *Scalasca User Guide*, 2025. Accessed: Jan. 7, 2026. [Online]. Available: <https://apps.fz-juelich.de/scalasca/releases/scalasca/2.6/docs/UserGuide.pdf>
- [52] D. Eschweiler et al., *Open Trace Format Version 2 (OTF2)*, 2025. DOI: 10.5281/zenodo.15100643
- [53] Forschungszentrum Jülich GmbH, Jülich Supercomputing Centre, *Performance properties*. Accessed: Apr. 20, 2026. [Online]. Available: https://apps.fz-juelich.de/scalasca/releases/scalasca/2.6/help/scalasca_patterns-2.6.html
- [54] A. Knüpfer et al., “The Vampir Performance Analysis Tool-Set”, in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmeler, B. Krammer, and A. Schulz, Eds., Stuttgart: Springer, 2008, pp. 139–155. DOI: 10.1007/978-3-540-68564-7_9
- [55] IT Center RWTH Aachen, *CLAIX-2023 (RWTH High Performance Computing (Linux)) - IT Center Help*. Accessed: Dec. 17, 2025. [Online]. Available: <https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/fbd107191cf14c4b8307f44f545cf68a/>
- [56] IT Center RWTH Aachen, *Storage Systems (RWTH High Performance Computing (Linux)) - IT Center Help*. Accessed: Jun. 3, 2026. [Online]. Available: <https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/be5fa2da96094e2e84f40c9778f88e2b/>
- [57] A. B. Yoo, M. A. Jette, and M. Grondona, “SLURM: Simple Linux Utility for Resource Management”, in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., vol. 2862, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2003, pp. 44–60. DOI: 10.1007/10968987_3
- [58] SchedMD LLC, *Slurm Workload Manager - Documentation*. Accessed: Jun. 15, 2026. [Online]. Available: <https://slurm.schedmd.com/>

- [59] J. E. Smith and G. S. Sohi, “The Microarchitecture of Superscalar Processors”, *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, 1995. DOI: 10.1109/5.476078
- [60] A. Fog, *The microarchitecture of Intel, AMD, and VIA CPUs*, 2025. Accessed: Dec. 17, 2025. [Online]. Available: <https://agner.org/optimize/microarchitecture.pdf>
- [61] S. Eyerhan, J. E. Smith, and L. Eeckhout, “Characterizing the Branch Misprediction Penalty”, in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, TX, USA: IEEE, 2006, pp. 48–58. DOI: 10.1109/ISPASS.2006.1620789
- [62] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein, *Short Note on Costs of Floating Point Operations on current x86-64 Architectures: Denormals, Overflow, Underflow, and Division by Zero*, arXiv:1506.03997 [cs], 2015. DOI: 10.48550/arXiv.1506.03997
- [63] Performance Optimisation and Productivity Centre of Excellence in HPC, *Sparse Matrix Vector Multiplication*. Accessed: Mar. 5, 2026. [Online]. Available: https://co-design.pop-coe.eu/versions/spmxv/epi-spmxv_main.html
- [64] D. Köhn, “Time Domain 2D Elastic Full Waveform Tomography”, Ph.D. dissertation, Christian-Albrechts-Universität zu Kiel, Kiel, 2011. Accessed: Dec. 17, 2025. [Online]. Available: https://macau.uni-kiel.de/receive/diss_mods_00006786
- [65] D. Köhn, D. De Nil, and A. Kurzmann, *DENISE User Manual*, 2021. Accessed: Dec. 17, 2025. [Online]. Available: https://danielkoehnsite.wordpress.com/wp-content/uploads/2021/06/manual_denise.pdf
- [66] Performance Optimisation and Productivity Centre of Excellence in HPC, *DENISE Black Edition subnormals kernel*. Accessed: Mar. 26, 2026. [Online]. Available: <https://co-design.pop-coe.eu/programs/subnormals/index.html>
- [67] Performance Optimisation and Productivity Centre of Excellence in HPC, *Subnormal number handling performance evaluation*. Accessed: Mar. 26, 2026. [Online]. Available: https://co-design.pop-coe.eu/experiments/subnormals/subnormals_main.html
- [68] Performance Optimisation and Productivity Centre of Excellence in HPC, *Avoiding subnormal floating-point calculations*. Accessed: Mar. 26, 2026. [Online]. Available: <https://co-design.pop-coe.eu/best-practices/avoid-subnormals.html>
- [69] “IEEE Standard for Floating-Point Arithmetic”, *IEEE Std 754-2008*, pp. 1–70, 2008. DOI: 10.1109/IEEESTD.2008.4610935

- [70] M.-O. Gewaltig and M. Diesmann, “NEST (NEural Simulation Tool)”, *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007. DOI: 10.4249/scholarpedia.1430
- [71] H. E. Plesser, J. M. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig, “Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers”, in *Euro-Par 2007 Parallel Processing*, ser. Lecture Notes in Computer Science, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds., vol. 4641, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2007, pp. 672–681. DOI: 10.1007/978-3-540-74466-5_71
- [72] D. Terhorst et al., *NEST 3.9*, Zenodo, 2025. DOI: 10.5281/zenodo.17036827
- [73] *Random balanced network HPC benchmark — NEST Simulator Documentation*. Accessed: Feb. 10, 2026. [Online]. Available: https://nest-simulator.readthedocs.io/en/v3.9/auto_examples/hpc_benchmark.html
- [74] *Iaf_psc_alpha – Leaky integrate-and-fire model with alpha-shaped input currents — NEST Simulator Documentation*. Accessed: Mar. 31, 2026. [Online]. Available: https://nest-simulator.readthedocs.io/en/stable/models/iaf_psc_alpha.html
- [75] J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig, “PyNEST: A convenient interface to the NEST simulator”, *Frontiers in Neuroinformatics*, vol. 2, 2009. DOI: 10.3389/neuro.11.012.2008
- [76] J. M. Bjørndalen and O. J. Anshus, “Trusting Floating Point Benchmarks – Are Your Benchmarks Really Data Independent?”, in *Applied Parallel Computing. State of the Art in Scientific Computing*, B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, Eds., vol. 4699, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2007, pp. 178–188. DOI: 10.1007/978-3-540-75755-9_23
- [77] Intel Corporation, *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*, 2024. Accessed: Dec. 2, 2025. [Online]. Available: <https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-2/overview.html>
- [78] *Time(1) - Linux manual page*, 2026. Accessed: Jun. 22, 2026. [Online]. Available: <https://www.man7.org/linux/man-pages/man1/time.1.html>