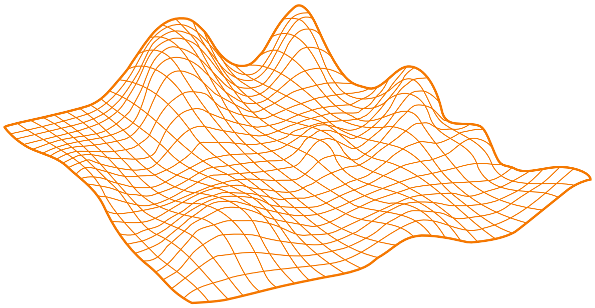
Analyzing NUMA Performance Based on Hardware Event Counters

Christoph Sterz



I would like to express my gratitude to my advisors Felix Eberhardt, Frank Feinbube, and Max Plauth for their comments, remarks, and dedication during this thesis. I further thank Professor Andreas Polze and the whole Operating Systems and Middleware Group for welcoming me to the research lab.

The work presented in this thesis would not have been possible without access to modern, top-notch hardware resources. Therefore, I thank the Hasso Plattner Institute and the Future SOC Initiative for granting me access to their high-end server systems.

Many fellow students and dear friends, Vincent Schwarzer, Wieland Hagen, Tom Herold, Toni Mattis, and Norman Rzepka, supported me during this work. I am grateful for all their valuable feedback I received for this thesis.

I express my sincere gratitude to Sven Köhler for letting me partake of his deep technical knowledge, and to Patrick Lühne for supporting me in improving this thesis with his zealous attention to detail.

Master's Thesis

Analyzing NUMA Performance Based on Hardware Event Counters

Christoph Sterz

July 22, 2016

Advisors

Prof. Dr. rer. nat. habil. Andreas Polze

Felix Eberhardt, M. Sc.

Frank Feinbube, M. Sc.

Max Plauth, M. Sc.

Operating Systems and Middleware Research Group

Abstract

Cost models play an important role when making design decisions for efficient software systems. These models can be embedded in operating systems and execution environments to optimize execution at run time. Adapting such models to new hardware architectures requires them to reflect more and more of the paradigms available to modern parallel computing systems. This thesis provides an overview of parallel cost models and describes that with the advent of *non-uniform memory access* architectures (NUMA), new models have been developed and the priorly existing ones need to be refined. Therefore, the existing NUMA models are analyzed, and a two-step strategy is proposed that incorporates low-level hardware counters as performance indicators. This thesis further focuses on these low-overhead hardware counters, which are available to all modern CPUs. For four major CPU vendors—ARM, AMD, Intel, and IBM—hardware counter specifics are presented and explained in detail. Four tools are developed, all accumulating and enriching specific counter information, to explore, measure, and visualize these low-overhead performance indicators. *EvSel* allows for measuring the whole plenitude of available counters. Two configurations of a program can be compared, and regressions from program parameters to performance indicators can be computed. *Phasenprüfer* attributes indicator measurement records to distinct ramp-up and calculation phases automatically. *Memhist* reveals the latency cost distribution of memory accesses—a major cost factor of recent programs. For analyzing NUMA systems, *Hydralisk* artificially strain individual interconnect links while simultaneously measuring their bandwidths. All mentioned tools are showcased and discussed alongside specific experiments in the realm of performance assessment. The tools can be obtained and contributed to in a GitHub repository.[1]

[1] https://github.com/chsterz/performance-tools.git

Zusammenfassung

Kostenmodelle sind von großer Relevanz für den Entwurf effizienter Softwaresysteme. Sie können als Teil von Betriebssystemen oder Ausführungsumgebungen durch Laufzeitentscheidungen einen optimalen Programmfluss bewirken. Dafür ist es notwendig, diese Modelle an die sich entwickelnde Hardware anzupassen, um die Mechanismen moderner, paralleler Rechnersysteme beschreiben zu können. Die Arbeit gibt einen Überblick über Kostenmodelle für parallele Systeme und motiviert die Notwendigkeit neuer Modelle für das Aufkommen moderner *nichtuniformer Speicherarchitekturen* (NUMA). Hierzu werden vorhandene NUMA-Modelle analysiert und Vorschläge für eine zweistufige Strategie erarbeitet, die auf den *performance counters* des Prozessors als Leistungsindikatoren basiert. Diese Arbeit legt ihren Fokus auf diese Zähler, welche in nahezu allen modernen Prozessoren zu finden sind. Details werden für vier der bedeutendsten Prozessorhersteller, ARM, AMD, Intel und IBM, zusammengefasst und erläutert. Die Arbeit stellt vier Programme vor, die die Hardware-Zählerinformationen zusammenführen und anreichern, um diese zu explorieren, messen und visualisieren. *EvSel* ermöglicht es, die Gesamtheit der Zähler zu erfassen, mit diesen Programme zu vergleichen und Regressionen für Parameterfolgen zu bestimmen. *Phasenprüfer* trennt automatisiert die Messinformationen von der Aufwärm- und der Ausführungsphase eines Programmes. *Memhist* erlaubt einen Einblick in die Verteilung von Latenzen für Speicherzugriffe, einem der wichtigsten Kostenfaktoren in modernen Computersystemen. Um NUMA-Systeme näher zu analysieren, wird *Hydralisk* als viertes Programm vorgestellt, das künstlich Belastungen auf den Verbindungen der NUMA-Knoten erstellt und gleichzeitig deren aktuelle Bandbreite bestimmt. Alle erstellten Programme und Messwerkzeuge werden architekturell und mit typischen Experimenten zur Leistungsfähigkeit von Computersystemen vorgestellt. Die Programme können über ein GitHub-Repository abgerufen und weiterentwickelt werden.[2]

[2] https://github.com/chsterz/performance-tools.git

Contents

1 Introduction

Modern computing systems need to employ parallelization to an increasing extent in order to perform efficiently on state-of-the-art computer architectures [125]. When optimizing for performance, parallel software can be designed with respect to multiple paradigms. On the one hand, *cost models* may be applied to theoretically estimate the performance characteristics of a program before actually executing it. Common performance issues can often be found with the help of cost models [44, 116]. However, such models are often complex and very theorical, making them inconvenient to apply in practice. On the other hand, performance may be analyzed with tools at runtime, which also highlight inefficiencies in specific parts of a software system. Commonly, multiple measurement tools and methods are combined to optimization strategies to trace back performance issues systematically [37, 97].

With the advent of *non-uniform memory access* architectures (NUMA), theoretical cost models become increasingly complex because they need to account for the various topology characteristics of NUMA-based computer systems, for instance [52]. Likewise, recent developments in performance optimization strategies have to be adopted accordingly. To support such strategies, the set of performance analysis facilities needs to be enriched with powerful NUMA-specific tools and libraries.

This thesis addresses the need for new strategies and tools for performance assessment and optimization in NUMA systems. First, existing cost models for parallel computation are surveyed and categorized. NUMA models are discussed and motivated as another, distinct class of cost models. This thesis makes suggestions when analyzing performance on NUMA systems. Additionally, a new, two-step strategy for performance analysis is presented, which makes use of low-level hardware performance counters.

Second, these low-level hardware performance counters are explained in detail as they exist inside modern processors. The counters of four different hardware vendors are described: ARM, AMD, Intel, and IBM. Further, this thesis presents existing utilities for measuring performance on NUMA systems as related work.

Third, four novel measurement tools are developed leveraging the hardware facilities presented priorly. These tools are helpful for assessing performance with the proposed two-step strategy. *EvSel* measures the whole plenitude of available counters, compares program runs, and performs program parameter regressions. *Phasenprüfer* automatically attributes indicator measurement records to distinct ramp-up and calculation phases. *Memhist* reveals the latency cost distribution for memory accesses as a major cost factor of recent programs. For analyzing NUMA systems, a fourth tool, *Hydralisk*, is developed, which artificially loads individual interconnect links while simultaneously measuring their bandwidths. Figure 1.1 relates the four presented tools to the proposed two-step strategy.

The contribution of this thesis is organized in three chapters, which also summarize the accompanying work.

Chapter 2 introduces basic concepts and terminologies. This chapter then surveys existing cost models for parallel computation and, specifically, NUMA. However, drawbacks are identified when applying the theoretical models to NUMA applications. Thus, a novel strategy for optimizing performance in NUMA systems is motivated instead.

Chapter 3 contains the technical foundations and related work for developing performance optimization tools. First, the performance measurement utilities of CPUs are described. This chapter then details the performance counters of different hardware platforms. Finally, various existing performance optimization tools are described.

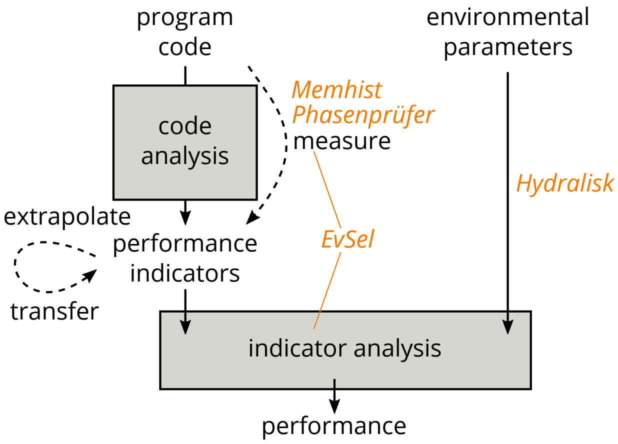


Figure 1.1: The four tools developed in this thesis (orange) incorporate hardware counters and help in adhering to the presented two-step strategy.

Chapter 4 introduces four novel performance analysis tools, which support the performance optimization strategy formulated earlier. The tools are built upon `perf` and use low-level hardware counters as an intermediate result. *EvSel* compares program runs and performs program parameter regressions. *Phasenprüfer* automatically identifies ramp-up and calculation phases of a tested program. *Memhist* reveals the latency cost distribution of memory accesses. *Hydralisk* artificially loads individual interconnect links while simultaneously measuring their bandwidths. This chapter explains development considerations of the tools, discusses their scope of applicability, and shows examplary usage scenarios.

Finally, Chapter 5 concludes and summarizes this work's contributions. This chapter provides prospects on the future of cost models and further outlines the work that is still necessary for creating more accurate cost models on heterogeneous systems in high-performance computing.

2 Cost Models and Proposals for Analyzing Performance

This chapter first introduces the theoretical foundations of this thesis and aims to define relevant terms. In addition, this chapter surveys and summarizes existing models for parallel computation. Then, existing cost models for NUMA systems are revisited and distinguished as a new class of cost models. However, drawbacks of applying such theoretical cost models to NUMA architectures are identified. This chapter then shortly discusses methods for static code analysis as a means to determine costs, which is found to be impracticable as well. Finally, instead of instantiating a NUMA model, a novel strategy for optimizing performance in NUMA systems is motivated, which is based on low-level hardware counters.

2.1 Cost Models

This section explains common key terms as they are to be understood in this chapter.

A *cost model* is the symbolic representation of a system's underlying mechanism, condensed as to explain one specific aspect of the system (see Figure 2.1) [120]. In the case of performance optimization, the specific aspect—or research target—to be understood are *costs*. While not changing the functional behavior in computation (that means, still fulfilling the specifications), reducing costs is desired.

For the sake of performance, there are two major types of models other than cost models (also called *machine models*, *execution models*, or *models of computation*) [109].

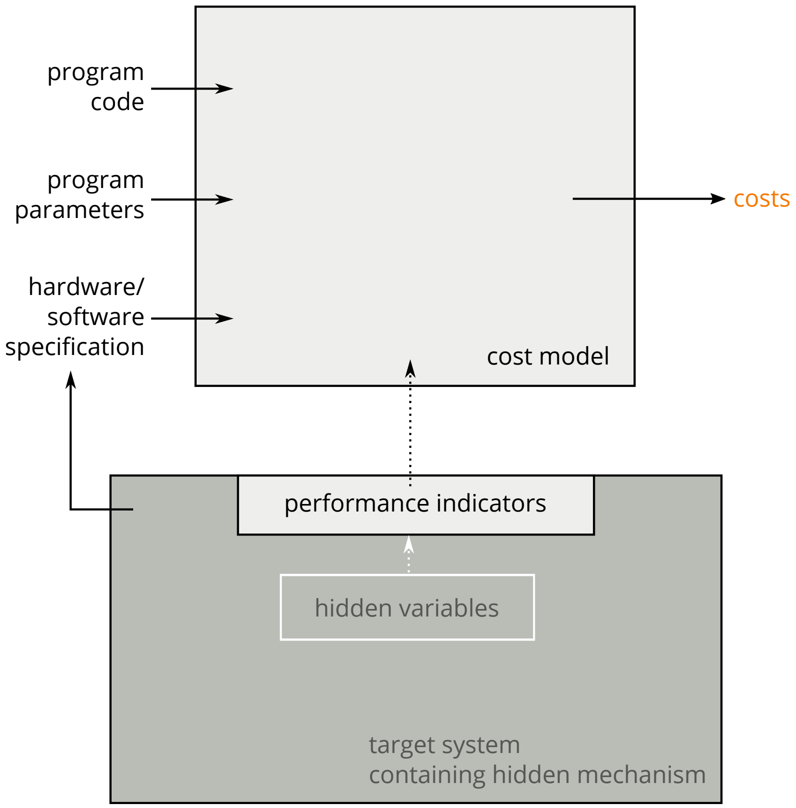


Figure 2.1: Cost model. A model describes a reduced aspect of the underlying system (shown in darker grey). For cost models, parameters such as the program, its configuration, and the target system's environment parameters are the input for retrieving costs. Systems often expose parts of their inner workings through performance indicators. Note that environmental parameters—in this case, software and hardware parameters—can either be embedded, that is, fixed for the model, or seen as additional parameters in more flexible cost models.

First, there are *programming models*, which support engineers in translating problems and concepts to program code. Second, *hardware* or *architectural models* describe the inner workings and underlying concepts of a machine [71, 85]. Exemplary use cases for each of the three model types are:

- Programmers optimize the parallel performance of a program using a cost model.
- Programmers look for programming languages and libraries that adhere to the right programming model and that fit their problem's domain.
- Programmers pick hardware for their data-intense algorithms according to the most suitable hardware architecture model, for instance, NUMA.

2.1.1 Performance Costs

Costs can be thought of as placeholders for anything that reduces the system's ability to fulfill the task from an operator's perspective. This may be time, energy consumption, or money savings. Costs in one domain of a program might have an impact on other domains and finally on the result, even though the execution still fulfills the program specifications [115]. For example, communication overhead inside a distributed system might influence the execution time. This, in turn, influences the energy consumption, which subsequently influences the money spent on electricity, cooling, and so on.

Though many other interpretations of costs (for instance, memory usage, response time, or throughput) might be considered, *execution time* is the commonly prevailing target for optimization [90]. In many cases, all other cost factors such as energy or cooling expenses are directly tied to the execution time [15, 119]. Even small time differences might result in absolute advantages (»all or nothing«) in the competition for customers or research grants. For the reasons mentioned above, this thesis considers execution time as costs only.

Note that the theoretical deduction of costs is only a heuristic evaluation of an actual system because of the simplifications made by the cost model [82].

2.1.2 Model Notation and Automated Evaluation

There exist many formulations of models, and these are interpretable by automatic systems to different extents. Cost models may be mere constructs of the mind, that is, the programmer's experience about what is efficient and what is not. These mental models can neither be gauged nor automatically applied by means of software engineering, though. In the literature, there are many models that are formulated in text form [116]. These already provide concrete guidance whether certain parameters should be higher or lower for better performance results. Such easy rules enable discrete decisions in automatic optimization facilities inside the software yet allow only for simplistic dependencies between input parameters and costs (»Higher is better« and so on).

2.1.3 Cost Functions

The most suitable models for automatic evaluation comprise algorithms or formulas that determine costs based on input parameters [85, 142]. This thesis refers to such mechanisms as *cost functions*. A cost function is an abstract, automatable expression of a system's underlying mechanism. While they are practical for automatic usage, these functions are hard to obtain or to generalize. Programmers might still prefer models in textual or graphical form or might only rely on prior experience.

Cost functions are mappings from a multi-dimensional space of input parameters to numeric cost values. The core task of descriptive models is to select a set of candidates from all parameters, which then form a subdimensional manifold (later called *cost landscape*). This also requires all input parameters to be numerically evaluateable.

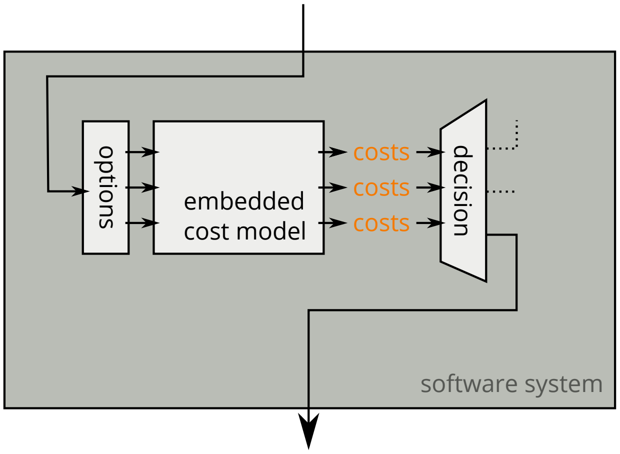


Figure 2.2: Embedded cost model. Often times, cost models are *embedded* in the software systems they are supposed to evaluate. At specific points in the program's execution flow, multiple options are generated and evaluated with respect to the embedded cost model. In the end, decisions that minimize costs are made accordingly.

These last two tasks—reducing dimensionality and numerically encoding complex input parameters such as the system's topology—are later discussed in Section 2.3.

2.1.4 The Necessity of Cost Models in Computation

Cost models are widely used in all areas of computing [83, 118]. Some models are able to compare two different execution paths of a program. With such models, software can make dynamic decisions at runtime for optimization purposes. In these cases, the cost models are *embedded* within the described system [67]. A generalized scheme of such embedded models is depicted in Figure 2.2. Embedded models can be found in most essential operating system algorithms and standard libraries for programming. By comparing modeled costs based on the requested amount of data, different resource allocation strategies might be employed, for example [136]. In the following, this thesis presents more examples for embedded cost models.

A rather classical application of cost models are *database query planners* [50, 54]. In this example, multiple orders of joins, filters, and projections are analyzed and reordered for minimized I/O and CPU costs. Furthermore, more fundamental database components such as spatial indices are formed and analyzed using cost models [12].

Cost models can also be found in the realm of compilers and *just-in-time*-enabled (JIT) interpreters. The g++ compiler is able to express a stream of instructions in a cost-efficient way for different processor architectures, hereby reducing cycles and pipeline stalls as the cost target. By exploring the optimization flags, a search for the most effective program compilation can be performed [65]. Programmers are even able to track the cost considerations of g++ when analyzing costs for a verbose vectorizer [57]. Clang also makes use of a cost model to compare vectorized and non-vectorized compilations of the same code [128].

Modern JIT interpreters weigh the costs of analysis and compilation against speed gains for partly compiled code. RPython's tracing JIT does this implicitly. When surpassing a certain threshold of a counter attached to the backjump instruction, the repeatedly run code path is compiled to bytecode. Since this just-in-time compilation only works when adhering to previously analyzed assumptions, guards are inserted. Should one of these assumptions be violated, the guard triggers the interpreter either to execute the non-compiled code version or to compile the new path of execution, too [18]. In this way, cost optimizations in program runs are supported by the model's tuneability through thresholding parameters.

Frameworks for heterogeneous multiprocessing such as OpenCL, which are able to schedule work for different executors, are another realm of effective cost models [121].

The point in time at which cost models are applied and evaluated can differ. Papers exist that present cost models based on benchmarks run when installing the system [118] or on hardware lookups in a database (which the authors call *adaptive mapping*) [83].

Others employ execution time kernel profiling and device profiling to classify threads as memory-bound or CPU-bound and to schedule them accordingly [4].

2.2 Three Generations of Parallel Performance Models

With the development of computers towards parallel program execution, new models needed to be found [39]. This section portrays the development of parallel computation models in three historical eras as classified first by Zhang et al. [142]. Most computation models adapt and enrich few well-known models to address a certain characteristic that limited the applicability of that general model before. A rough timeline of the three historical eras of models in parallel computation can be seen in Figure 2.3. A few representative examples of each era are explained in more detail in the following.

The early, sequential models were fused representations of computation, programming, and hardware models in a single notation. Turing's theoretical model of a single-band machine was a description as well as prescription of band machines existing at that time. Nowadays, it serves primarily as a theoretical computation construct for theoretical computer scientists [131].

Von Neumann's *random-access machine* (RAM) [64] and the concept of random-access stored-program machines (later referred to as the *Von Neumann architecture* [133]) were both theoretical constructs and blueprints for most of the later computing hardware. Von Neumann's ideas are also found in the parallel models developed afterwards.

RAM can be seen as a *bridging model* from band machines to systems with arbitrarily-sized numbers of registers. Yet, a bridging model translating sequential concepts to their parallel counterparts cannot be found. In theoretical computer science, however, there are complexity classes comparing sequential and parallel executions of similar problems. *Nick's class* ($\mathscr{NC}$) describes problems that change complexity from polynomial to logarithmic when leveraging an arbitrary amount of parallel resources [9].

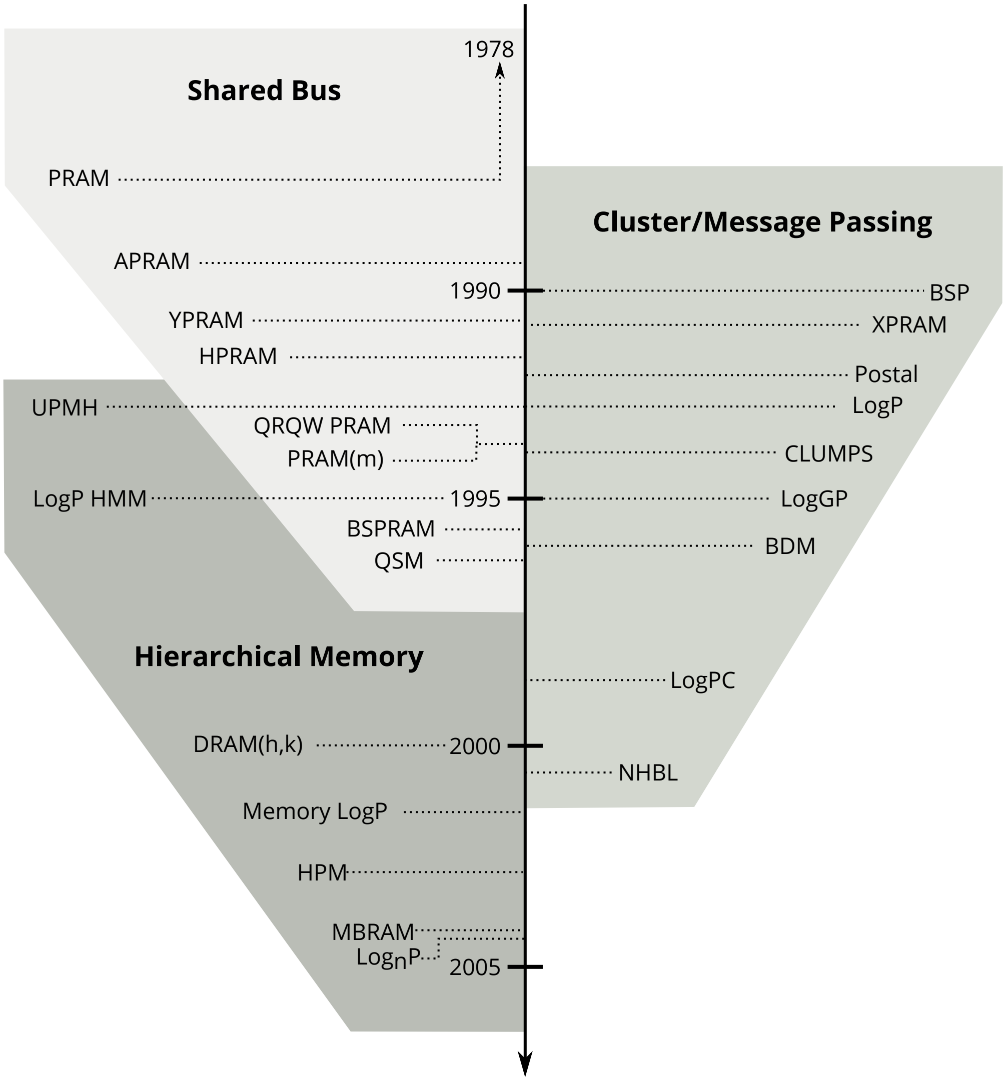


Figure 2.3: Historic models of parallel computation. The first era is characterized by a common bus among all processors. The second era addresses concerns of message passing in a clustered environment where memory is considered to be distributed. All third-era models contain a description of a memory hierarchy or the behavior thereof.

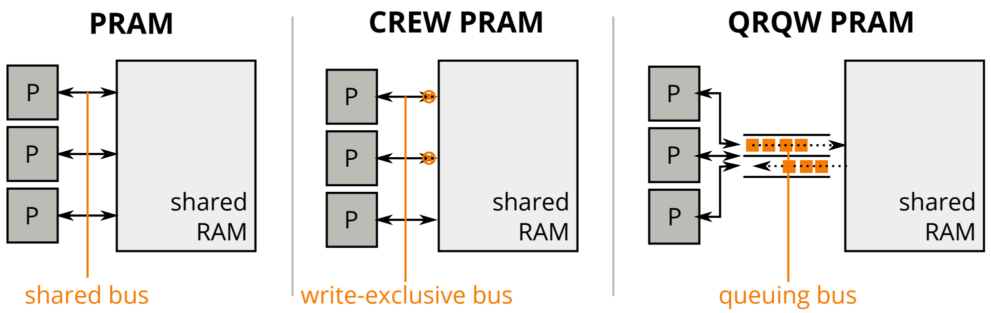


Figure 2.4: Selected manifestations of PRAM models. The processors (shown in darker grey) all access the data over a single bus. On the left, the original PRAM model (representing CRCW PRAM) is shown. In CREW PRAM (middle), writing to RAM is modeled to be exclusive to ensure data correctness when multiple processors access it. On the right (QRQW PRAM), congestion and bus limitations are modeled with a queue.

2.2.1 Shared-Bus Era

The first era of parallel computation models describes the dominating shared-bus systems of that time. Multiple concurrent RAM processors execute unit-cost instructions on commonly shared memory in lockstep fashion (all parallel execution steps happen synchronously) [100]. This quite accurately describes PRAM, the most popular model of this era (see Figure 2.4) [25].

The PRAM model was later enhanced by describing its memory read (R) and write (W) properties. The *concurrent read/concurrent write* (CRCW) PRAM model, for instance, allows all processors to simultaneously access a certain memory cell [74]. The additional letters ›E‹ for *exclusive* access behavior, ›O‹ for owning behavior, and ›Q‹ for *queued* behavior can be found, which characterize the PRAM access [51].

For asynchronous execution, APRAM [28], *asynchronous PRAM* [56], and XPRAM [93] remove the lockstep property and introduce synchronization steps with zero costs. *Queued shared memory* (QSM) as well as the queuing property of memory accesses (Q) allow to simulate congestion of the shared bus [101]. Hierarchical behavior is achieved in the variants YPRAM [93] and HPRAM [63], where partitioning instructions allow for simulating multiple PRAM subunits inside the machine. These subcomputing units communicate among each other with increased costs, described by inefficiency factors.

There exist extensions considering memory access latencies (LPRAM) and bandwidth information (BPRAM) [3]. These can be applied when the application is especially bound by one of those properties. Many PRAM models can be simulated to assess costs for the innermost and costly parts of algorithms.

PRAM can be found in the later eras as well. BSPRAM, for example, fuses bulk synchronous behavior with the refinements of concurrent memory accesses [130]. Still, PRAM with its singular memory space is what programmers might consider first because of its simplicity and applicability on current hardware.

2.2.2 Cluster Era

The second era of parallel computation is characterized by distributed memory and moderately slow interconnects (such as ethernet networks), both representing the cluster supercomputers apparent at that time. Together with the idea of clusters, *message passing* became part of the models. Notable representatives are *bulk synchronous parallel* (BSP) [132], LogP [44], and their variations (see Figure 2.5).

BSP can still be considered the most suitable model for analyzing work distribution problems of parallel processors. In BSP, a concurrent section is executed by multiple processors. The processors then wait at a global barrier to resynchronize for communication, which is carried out in a point-to-point fashion. Lastly, a global synchronization barrier is passed by all threads to begin a new round of concurrent computation. These three steps form a so-called *superstep* of computation. Performance hereby depends on the slowest processor in terms of execution and the communication phases.

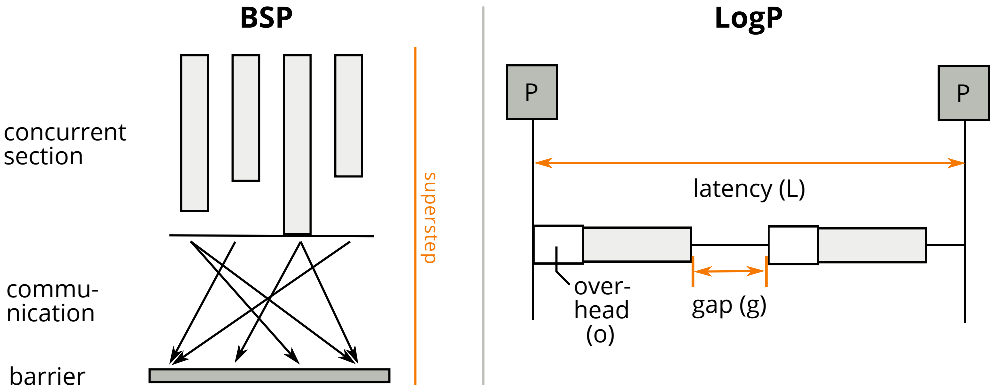


Figure 2.5: BSP and LogP. For BSP (left), a concurrent section of computation is followed by communication among the processing units. Each process then stops and waits at a common barrier for resynchronization. All substeps combine to a so-called *superstep*. LogP (right) describes properties of messages among processors, here marked in darker grey. A fixed latency for sending messages is assumed in this model. Additionally, the costs of creating the message (overhead), and the channel-specific gap between messages describe the whole communication.

Conversely to costs, the loss of parallelization potential can be determined by summing up the waiting time for synchronization and communication.

A common practice to overcome these losses is *work stealing*, which can either start directly at the beginning of the parallel section or when the first processor goes idle [16].

LogP can be seen as the asynchronous counterpart of BSP [44]. Four parameters describe computation among processors: latency L, overhead o, minimum gap between messages g, and the number of processors P. In their work, the authors of LogP show how algorithms like *prefix sum*, *fast Fourier transform* (FFT), and *LU decomposition* can be modeled and improved with the help of LogP.

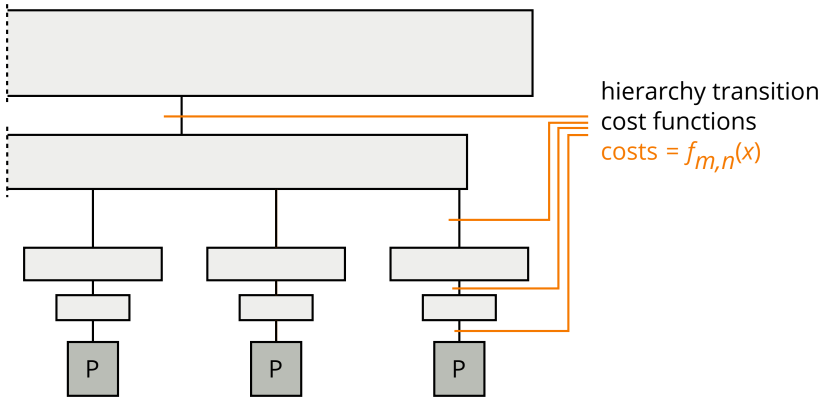


Figure 2.6: Hierarchical models. In the third generation of models in parallel computation, memory accesses across the increasingly complicated levels of caching hierarchies became important. Cost functions $f_{m,n}(x)$ that describe hierarchical transition costs from level m to level n and depend on the memory location x of the data can be found in most of these models.

2.2.3 Hierarchical Era

With an ever-increasing speed gap between instruction execution and operand fetching, the role of caching in computer systems increased. Alongside this development, models that embrace memory hierarchies evolved [6].

The models RAM(h, k) [142] and UPMH [5] both assume multiple caching hierarchies that have access costs $c = f(x)$ depending on the accessed memory location x. Because the ratio between cache hitting and cache missing cannot be formulated in simple parameters, complex functions are needed to describe them. These formulas depend on the temporal as well as the spatial locality of the memory accesses.

There are LogP representations of caching hierarchies, for instance, *Memory LogP* [24], where caching is modeled by message passing between the hierarchical cache layers. However, neither access patterns nor cache affinity are considered with Memory LogP.

2.3 NUMA Cost Models

Having presented key terms and the history of models of parallel computation, this section outlines cost models specific to non-uniform memory access architectures. Then, this section motivates how these models form a new class of cost models.

2.3.1 NUMA Architectures

The development of non-uniform memory architectures can be seen as an answer to the ever-increasing need for computational power. While the gap between performance of CPUs and their associated memory modules increases [46], even more cores are added to processors, which poses an even higher demand on memory bandwidths. NUMA architectures can be seen as a response to this demand. By associating memory via multiple, distributed memory controllers to local CPUs, more bandwidth can be achieved through parallelized local accesses.

Reaching all memory regions from the individual processors of the system is possible with the help of *interconnects*, which form a network among the processors. This architecture results in at least two classes of accesses: *local*, fast accesses through the CPUs' dedicated memory controllers and slower, *remote* accesses through the interconnect network. Because of the differences in access latencies, the processor experiences different memory regions with different access behaviors. This breaks the assumptions of the RAM model (where memory accesses had *unit costs* for all processors alike) and explains the »non-uniform« in the architecture's name. An exemplary, resulting NUMA topology can be seen in Figure 2.7. Note how this topology cannot be seen as a hierarchical tree structure but rather as a graph of executors and memory locations.

To allow programmers to manage data in a unified and contiguous fashion, cache coherency protocols are adapted to this new architecture. While cache coherency was maintained inside processors before (managing consistency among multiple layers of

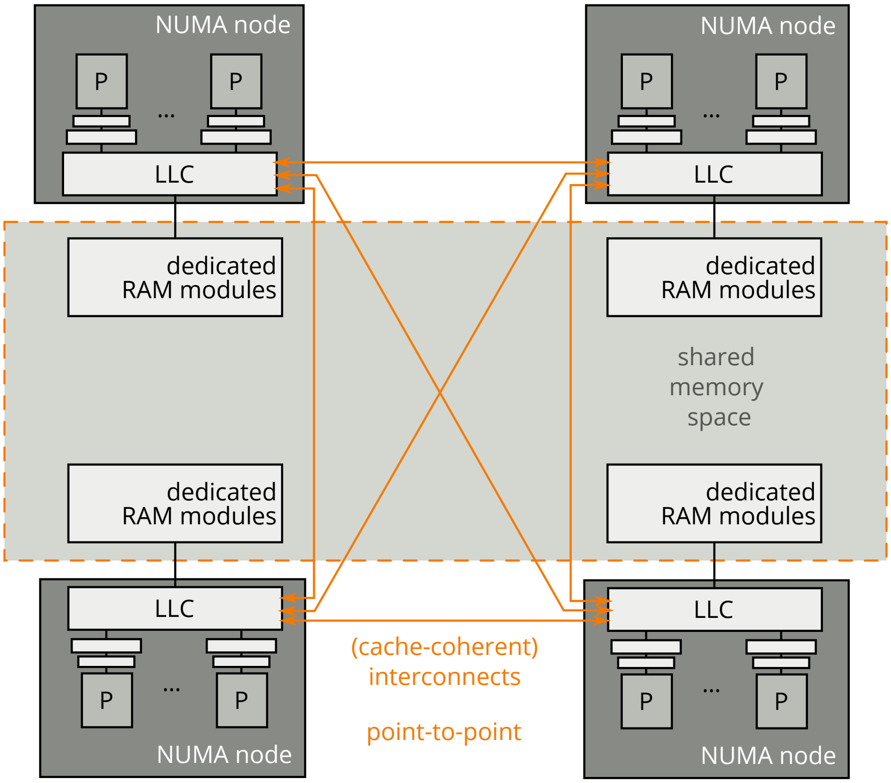


Figure 2.7: CC-NUMA. Multiple processors form so-called *NUMA nodes* comprising individual, associated caching hierarchies. The *last-level caches* (LLC) of these nodes are connected with each other to enable the data transfer and to uphold cache coherency for the complete system. Caused by cache coherency and the system-wide access to data, main memory appears as being contiguous.

caches), with NUMA, it becomes the responsibility of the overall computing system to ensure data consistency during program execution. Popular mechanisms for cache coherency are directory-based and snooping-based protocols [13].

Non-uniform memory architectures can be seen as the answer to high memory access costs compared to the costs of computations on the data itself [127]. Because of this, many models in the research field focus on memory accesses as their main cost factors.

Note that when optimizing costs in such highly parallelized hardware and software environments, one has to first solve shortcomings in the single-thread behavior. Before it is worth consulting NUMA models for cost analyses and performance optimizations, programmers should care about more fundamental concerns such as single-thread performance, efficient locking, and communication mechanisms in addition to an optimal work distribution among processors [37].

2.3.2 Representative NUMA Cost Models

This section presents an overview of related work in the field of cost models specifically designed for NUMA architectures. Often times, these models bear memory access costs as their key factors [84, 21, 53]. This thesis summarizes these approaches and finally proposes properties a future model for the NUMA era has to fulfill.

Braithwaite et al. describe a machine-based model that is built upon prior measurements on the hardware, which determine bandwidth and latencies of the NUMA interconnects [19]. In their exemplary study, the authors identify equivalence classes among cores. With these classes at hand, the authors are able to describe interconnect topologies numerically. This information, fused with measured bandwidth and latencies, is then used to evaluate the model on exemplary benchmark workloads (Graph500, Linpack, NPB-IS). Depending on the memory cost affinity of each workload, the authors validate the applicability of their cost model.

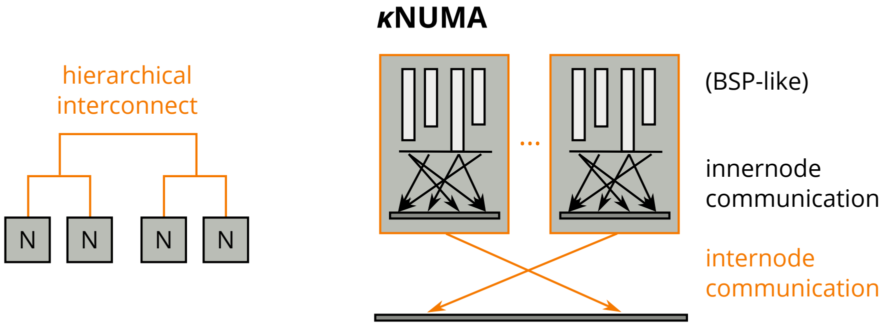


Figure 2.8: κNUMA. Schmollinger et al. model nested BSP behavior via tree-interconnected nodes (shown in dark grey) with depth κ [111]. While an internal BSP behavior can be found inside the nodes, an additional, similar behavior is modeled for internode communication.

Schmollinger and Kaufman propose a model named κNUMA, which is aimed at clusters and SMP machines [111]. The model builds on top of the concept of communication in BSP, extending it through submachine functionality (see Figure 2.8). κNUMA can be thought of as a κ-deep tree hierarchy of processors. Although this approach seems to adhere to hierarchical models, their proposed model also supports a ring topology. On a rather theoretical level, the authors build a cost function that integrates subprocessor communication costs into global superstep costs. While they outline that BSP is not directly transferrable to parallel SMP machines, the authors' model works well for problems with global notifications similar to MPI's `gather` and `scatter` functionality via the tree or ring structures [117].

In his work, Forsell proposes a model for converting workloads with low thread-level parallelism from NUMA to a PRAM counterpart [52]. The paper puts emphasis on hiding latencies by grouping processors to act on a single state, resulting in a PRAM NUMA model. Because no actual cost function yet a rather conceptual machine model is presented in this paper, the work can only be considered a theoretical contribution.

The work of Zhang and Qin models NUMA interconnects as switching networks [140]. With the analogy to resistors in a network of similar topology, the authors predict access times for the matrix multiplication example. Although their paper is a rather early contribution, Zhang and Qin are able to classify and isolate problems such as congestion both in the model and in the evaluation on their early NUMA machines.

Further work putting emphasis on memory access costs can also be seen to be closely related to this thesis, since NUMA models aim to tackle memory-bound problems. In their work, Ma et al. develop a »memory access model for highly-threaded many-core architectures« [84]. They describe how fast context switching enables memory hiding in modern multi-threaded environments. Their model TMM (*threaded many-core model*) is validated against four shortest-path algorithms, where it is able to predict performance well. While the authors only use two levels of hierarchy, which is not always be the case in NUMA systems, the extension to complex networks comprising multiple hierarchies is possible and discussed in the authors' outlook.

Byna et al. present formulas and an evaluation of memory accesses [21]. With a handful of parameters describing the topology or memory hierarchy and additional parameters characterizing data accesses, they are able to foresee the costs of memory accesses for the widely used matrix transposition algorithm.

In other works, parameters of memory access cost models are studied. Wu et al. examine the influence of memory locality for the example of compression algorithms with their memory profiler LEAP [139].

In conclusion, all presented works pursue the formulation of models in the context of NUMA. Some of the NUMA cost models are extensions to legacy models such as BSP and RAM. The majority of related work focuses on evaluating their models on examples, thus verifying their applicability.

2.3.3 NUMA Cost Models as a New Class of Cost Models

This section discusses why NUMA cost models should be seen as a new class of models instead of considering them part of the third-generation (hierarchical) model category.

It is of high importance to point out that NUMA architectures do not adhere to existing hierarchical models, which were presented in Section 2.2. NUMA is not just an addition of a »virtual« caching hierarchy of local memory accesses within the generally more costly global memory space. If there were such a virtual caching behavior, it would surely be an argument for the applicability of hierarchical models as presented in the Chapter 2. However, this assumption misses important properties of the memory in NUMA systems. While caches can be seen as copies of values that originally reside in main memory, the memory regions in NUMA are not copies. They are singular (the only) instances of information distributed in partitions and connected via a graph of interconnects [62]. Moving or copying this data is not implicitly done by hardware during calculations but comes at a cost.

For the hierarchical era, there were only few possibilities for processors and operating systems to influence the placement of data in the caches directly.[1] However, such placements can be achieved with memory in NUMA regions. In NUMA systems, the facilities of operating systems and libraries can directly steer memory allocations and movements across the system. The API provided through `libnuma` enables programmers to explicitly place data, for instance [76].

Earlier, when using caching hierarchies, programmers had to rely on their experience and their knowledge about the hardware architecture when formulating code to keep data inside the caches. Only when stated implicitly in the code, memory was local with respect to execution time and memory space. In contrast, NUMA models might assume certain data placement policies (for example, interleaving, local, and first-touch, which

[1] In x86 assembly, there exist instructions for cache eviction, prefetching, and cache bypassing. However, these are just suggestions to the CPU.

are offered by the Linux kernel) for their cost computation [17]. NUMA models could either rely on optimal data distribution or determine the costs of inefficiencies therein.

While cache hit probability depends both on temporal and spatial locality, NUMA does not benefit from temporally close memory accesses. For caching, only the hierarchy level is important (all data residing in L2 caches behaves similarly), yet in NUMA, the exact location of data in the topology comes to play when determining costs.

Furthermore, memory accesses can be seen as truly concurrent in the case of NUMA. Continuous accesses were bound by a single memory controller's bandwidth in the hierarchical, 3rd-era models before. NUMA architectures, however, consider both possibilities: multiple simultaneous, local accesses—and therefore truly concurrent reads and writes—or remote accesses culminating in one or more interconnects, which results in congestion and link saturation.

In the future, memory and processing might be distributed heterogeneously. Then, especially the costs and loss savings of moving data for either faster memory accesses or accelerated computation become crucial to consider. With the prospect of future heterogeneous architectures scaling up to core-to-cloud systems where worldwide, highly specialized execution can be envisioned, these data movement and scheduling considerations become imminent [2]. For all the aforementioned reasons, this thesis distinguishes NUMA cost models from the prior hierarchical, 3rd-era cost models.

2.4 Proposals and Limitations for the Development of NUMA Cost Models

In this section, multiple ideas are proposed to be considered when continuing the development of NUMA cost models. This section discusses why it is impractical to deduce costs with a function purely obtained from prior measurements. In the end, a summary is given on the practical applicability of cost models for NUMA performance analyses.

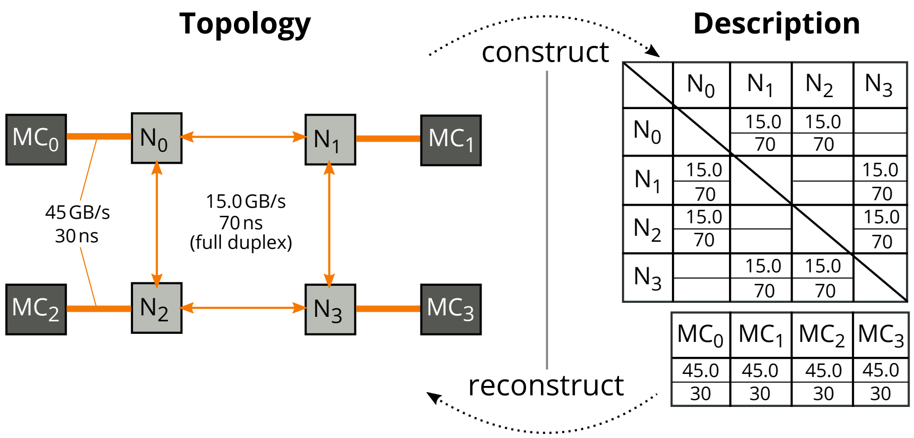
2.4.1 Topology Description

Existing systems mentioned earlier modeled the interconnects through either abstract behavior or a hierarchical memory layout. However, for determining precise costs, it is relevant to numerically capture the exact topology of the NUMA interconnects, as argued in Section 2.3.3. This section makes a proposal for such a numerical description while pointing that the existing SLIT tables are not suitable in all cases.

Topologies describing the hardware are abstract representations of the means of communication that exist between the processors. In both of the historical eras of clusters as well as hierarchical models, certain communication parameters were used as well. For the cluster era and its characteristic LogP and BSP message passing approaches, behavioral aspects of the network between processors (or cluster nodes) were sufficient for describing the whole topology. Latencies and bandwidths together with properties such as the gap between messages formed the parameters of those approaches. In the hierarchical memory era, even parameters such as cache sizes and cache line widths could be found in the models.

In both of these eras, congestion and the exact link topology were not considered, though. For clusters, the network (often ethernet) was so slow that it was latency-bound in almost all cases. For the hierarchical age, the tree structures never exposed complex contention, since all links were used independently or contended about a singular path to memory. The spatial aspect of such topologies was not considered.

For a NUMA model, both latencies and bandwidths should be described in terms of single variables anymore [141]. Instead, adjacency matrices need to describe connectedness as well as topological information as they would for a graph problem. The original topology in the form of a graph can be reconstructed through adjacency matrices without loss of information. It is important to admit that there is no trivial mathematical method that can express the contention of links. Bandwidths, both on the intercon-



	N$_0$	N$_1$	N$_2$	N$_3$
N$_0$		15.0 70	15.0 70	
N$_1$	15.0 70			15.0 70
N$_2$	15.0 70			15.0 70
N$_3$		15.0 70	15.0 70	

MC$_0$	MC$_1$	MC$_2$	MC$_3$
45.0 30	45.0 30	45.0 30	45.0 30

Figure 2.9: Topology description. In this example, a multi-hop network is modeled. Data flow in a system can be represented using adjacency matrices. As long as it is possible to reconstruct the existing topology, no information is lost. In contrast to SLIT tables, topological information is preserved with this method.

nects and the dedicated memory controllers, should be modeled as contingents that are reduced upon usage. This is sketched in Figure 2.9.

Although current systems reveal latency information (only) via the ACPI *system locality information table* (SLIT) [79], this method is suboptimal for several reasons.

First, latencies and bandwidths(maximum achievable throughput) should not be considered the inverse of each other [124]. While this might be valid for high-throughput streams, when accessing single values remotely, latency is not inverse to bandwidth. On the other hand, when interconnects are affected by congestion, the last-level cache usage is high, and memory controllers are flooded with local and remote requests. Then, latencies can become much higher than the inverse of the denoted bandwidth. Thus, it is crucial to note both latencies and bandwidths for each possible path to memory.

Second, SLIT does not denote adjacency information but instead the combined latencies and bandwidths along the shortest paths from point to point. This is not a problem for fully connected graph topologies, which allow for point-to-point connections only. Yet, if data routing were done in a multi-hop fashion in the future, summed distances would not help in determining shortest paths or cost.

Third, the table does not incorporate information about whether the connection is full-duplex or whether it suffers from cross-talk behavior. This information is crucial when exchanging information bidirectionally, though [95].

2.4.2 Program Execution Phases

Since program behavior can change over sampling time, modeling phases might be a good strategy for creating a NUMA model. In each phase, different machine properties might limit the program's efficiency. Especially for problems typically solved on NUMA hardware, there are many memory allocations.

As a proposal, this thesis suggests to analyze different phases of a program exeuction based on its memory footprint. In this way, the phases of *ramp-up* and *calculation* can be distinguished, which are typical of high-performance computing [113]. Previous models did not consider these program phases and often times ignored initial resource allocations for complex calculations. This thesis proposes to at least split the ramp-up and calculation into to two distinct phases to consider when optimizing performance.

Phasenprüfer, a tool that automatically detects the ramp-up and the calculation phases, is later described in Section 4.2.

2.4.3 Creating NUMA Cost Models Based on Empirical Observations

Aside from creating theoretical models, cost functions could also be found by learning from prior empirical observations. While this approach might report numerical costs conveniently, this section shows that purely learning such functions is inadvisible.

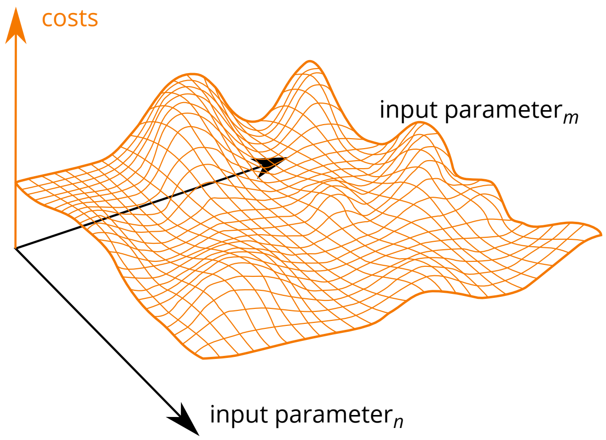


Figure 2.10: The cost model landscape. A cost function is modeled from multiple input dimensions. Note that the figure shows only two of the possibly hundreds of input parameters of such a function.

The cost functions mentioned before represent a mapping from a multi-dimensional space of environment parameters (forming a *cost model landscape*) to multiple cost values. This landscape is highly complex because of the interplay between mutually influencing input–output relations. Analyzing and operating on such a high-dimensional space is considered difficult and often referred to as the *curse of dimensionality* [11]. An illustration of how such a landscape could look like is shown in Figure 2.10.

Techniques to learn these mappings based on samples are plentifully found in the areas of machine learning and regression, yet this thesis argues that these learning techniques are not able to reproduce the landscape without an enormous amount of samples. Minimizing least squares in a linear fashion might give a good approximation of the landscape yet results in hyperplanes only. Though hyperplanes are a first approximation of the problem, they do not resemble the actual landscape but result in an averaged gradient only. Note that samples, when understood as vectors, must be pairwise linearly independent. This implies that if there is a dimension that is constant, the hyperplane cannot be determined correctly for all dimensions. Additionally, the sample count must at least match the number of dimensions of the problem space, that is, rank(input + output). There exist methods for approximating higher-order manifolds (non-planar landscapes), yet they require even more samples as an input [10].

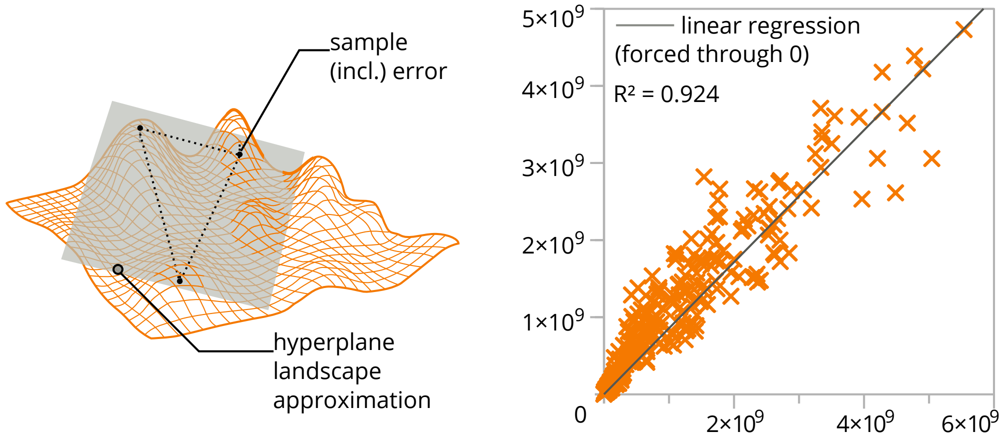
Although there may be many input dimensions (describing code, hardware, measured indicators, and the topology along all their properties), pairs of dimensions exist that depend on each other and thus overlap in their influence on the costs. As an example, the numbers of instructions and cycles are correlated fairly linearly. Since for all instructions, at least one and at maximum a specific worst-case number of cycles have to be spent, there exist upper and lower bounds for this linear relationship.

Mechanisms such as *principal components analysis* (PCA) [73] that leverage the most prevalent eigenvectors of the measured data are able to eliminate such linear dependencies. Since only linear dependencies can be extracted in this fashion, PCA cannot be of much help in the case of more complex interdependencies. Figure 2.11 shows both space sampling and an example of two correlated indicators for PCA.

Overall, this thesis suggests to further develop cost models incorporating numerical data yet does not recommend learning cost functions in this way. Understanding the mechanisms and reproducing them via simulation or even roughly-formed models is much more feasible in this regard.

2.4.4 Limitations of the Applicability of NUMA Cost Models

To make precise cost predictions with respect to the ever-increasing hardware complexity, NUMA cost models incorporate more and more parameters. Unfortunately, complex behavior such as the topological description cannot be described with few numerical parameters. If there are many parameters, users are obliged to invest much time in obtaining them in order to determine the costs. Even functions have to be con-



(a) Hyperplane manifold. As a minimum approximation, #dim independent samples are taken to form a hyperplane. Samples can still contain errors that lie within the measurement's error margin. Note that this only pictures the three-dimensional case. Real-world data resides in much higher-dimensional spaces, requiring more and higher-dimensional measurements.

(b) PCA. Although many counters exist, their data does not span the whole measurement space. Here, instructions (x axis) and cycles (y axis) are shown. Data was sampled from 300 1-second intervals of the rather complex browser program Firefox. Choosing only one axis along the regression would still explain more than 90 % of the variance in the measured data in this case.

Figure 2.11: Problem space considerations. The problem suffers from the *curse of dimensionality*. To add more indicators, which help to explain more of the cost, a higher number of samples would be needed to describe the cost landscape. Conversely, correlated parameters could be fused by projecting their data onto their eigenvectors.

sidered as inputs to models. This makes it hard for engineers to understand and apply modern models of parallel computation when analyzing performance.

At the same time, cost models are very theoretical and often only available in textual form [114]. This makes it impossible for computers to automatically determine costs with these cost models. There are only few simulators and actual applications that compute costs based on these complex cost models. In Section 2.6, this thesis addresses these shortcomings by proposing a strategy for assessing performance that is based on the practically available performance indicators, measured at runtime.

2.5 Extracting Costs from Program Code

Instead of employing cost models, programmers may perform static code analyses on their software implementation. This section describes common methodologies when it comes to extracting costs from program code only in an automated fashion. However, with the current state of the presented tools, such analyses are not applicable to moderately complex programs.

2.5.1 Abstract Interpretation and Symbolic Execution

The execution costs of a program are first and foremost dominated by the executed instructions. Because of this, an analysis of the program's code should be an important information source for creating accurate cost functions. In practice, however, determining costs from code is very complex and not feasible with current computational resources for reasonably sized sequences of code [110, 55]. This thesis explains two major methods for static code analysis: abstract interpretation and symbolic execution.

Abstract interpretation aims at extracting a specific property of the program by neglecting non-relevant information in the process. This might be the case when just the flow of the variables and not their contents are considered for proving a specific property of a program. This is famously used in compilers, which optimize out variables that are provably not influencing the end result. Another case are the signs of numerical values, which can be deduced for operations such as multiplications or squares without having exact knowledge about their operands. In this way, some information about the CPU's status register can be extracted from the code beforehand [43].

At the same time, abstract interpretation is bound by *Rice's theorem*, which forbids simple property deduction in programs [104]. The theorem states that for any non-trivial property of partial functions, no *effective* method can decide if the program under test computes the function such that it satisfies this property. This theorem holds for all methods of static code analysis.

Conversely to abstract interpretation, *symbolic execution* aims at covering all possible execution paths by replacing the variables' actual values with symbols. This method is thus able to determine various properties of these symbols, even backwards in time [75]. For example, dead code paths can be found with this method. Symbolic execution can be performed either fully (on all code paths) or by sampling code paths that are heuristically identified as most probable. In the area of software verification, *concolic execution* is a popular representative symbolic execution method [59].

2.5.2 Code Representations

When analyzing programs, different forms of one and the same behavior can be studied along the different levels of code abstraction (for instance, C++, IR, and assembly). In the exemplary case of a C++ program, determining abstract costs is hard yet fairly attributable to variables and lines of code. Analyzing the Intel x86 assembly of the same program might yield better results in terms of cost precision. However, optimized assembly lacks the connection between costly instructions and specific lines that caused them, since multiple instructions represent multiple lines of code and vice versa (for instance, fused mathematical expressions).

A viable option for analyzing smaller program sections is the LLVM *intermediate representation* (IR) [80]. With IR, a translation of code into the form of *static single assignments* (SSA) is done [45]. In the static single assignment form, variables are defined exactly once before the first usage. Static analyzers such as *Klee* offer functionalities to prove constraints that can also be upper bounds for execution runtime [22]. There even exist so-called *superoptimizers* such as *Souper* [102], which find shortest possible instruction sequences for a given piece of code. Both Klee and Souper make use of Clang's intermediate representation in their analysis steps.

2.5.3 Limitations of Static Code Analysis

Unfortunately, as mentioned in the introduction to this chapter, static code analysis is not really applicable for assessing the performance of complex real-world programs. The exploding number of execution paths renders it nearly impossible to find assumptions and constraints for larger programs. As a practical example, consider even the comparably small piece of code below:

```
while (n != 1)
  n = (n % 2 == 0) ? (n / 2) : (3 * n) + 1;
```

Despite its length, this code snippet's runtime depending on `n` cannot be determined universally. This code represents the *Collatz conjecture*, proven in 1972 by Conway et al. not to be decidable [30]. Another simplistic counterexample is the following:

```
if (md5(x) == 0xdeadbeef)
       x.f();
```

Because inherently, the MD5 hash function cannot be reversed efficiently, the information whether or not the member function `f()` is called on `x` is very hard to obtain [105].

In general, there exists an unavoidable problem to create exact universal runtime cost functions based on code only. If there was a cost function applicable to all programs, it would solve the *Halting problem* [31]. This is a contradiction and thus, such a universal function cannot be formulated.

Caused by all mentioned theoretical constraints, this thesis puts less emphasis on static code analysis and focuses on information retrieved from actual program runs instead.

2.6 A NUMA Performance Strategy Based on Hardware Event Counters

This thesis introduces a new NUMA performance strategy based on measurements made during prior program runs. In this section, this two-step strategy is developed, which enables portability across physical systems. This strategy circumvents impracticable static code analyses by using low-level performance indicators as an indirection.

2.6.1 Hidden Variables and Performance Indicators

Inside a computing system, there are parameters that can be observed and others that cannot. *Hidden variables* are entities of the system's mechanism that cannot be determined directly. Software engineers know that the branch predictor exists in modern CPUs, for instance. Yet, they cannot know about the CPU's internal state, which is influenced by prior events and will also affect the program's runtime behavior.

Indicators, on the other hand, are observable values in the system's mechanism. If they relate to costs, they are referred to as *performance indicators* [26]. Through *hardware counters*, CPUs reveal a part of their internal hidden state. When assuming a functional dependency between input parameters, performance indicators, and costs, a strategy using low-level hardware counters for program analysis can be devised.

One example for such performance indicators is the instruction count of a program run. Given the costs of each instruction in CPU cycles, the overall number of clock cycles may be estimated. Assuming a fixed clock frequency, the costs in execution time may in turn be deduced. This example illustrates how all steps require knowledge (cycle costs of instructions and CPU frequency) about the causal link among indicators and finally from indicators to costs. At the same time, this method introduces simplyfing abstractions (worst-case, fixed clock cycles are assumed), which blur the resulting costs. The interplay of all these key terms was shown at the beginning in Figure 2.1.

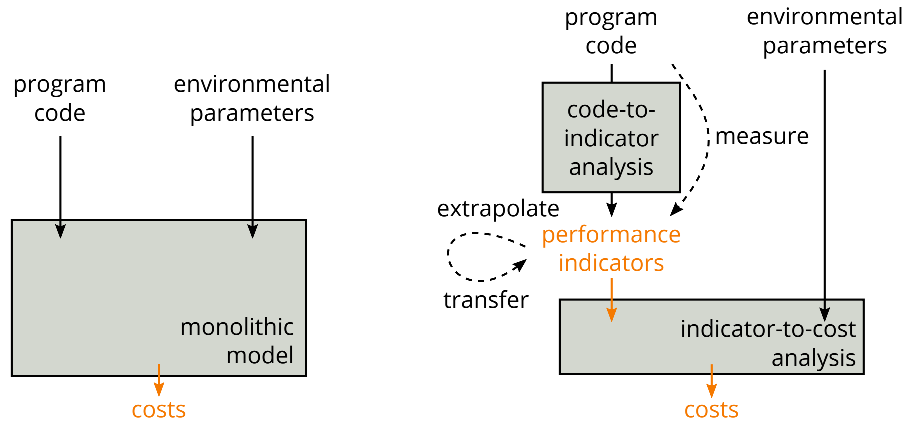
2.6.2 A Two-Step Strategy for Analyzing Performance

With the massively available number of hardware indicators as input parameters (see Chapter 3), a new strategy for performance optimization is developed. A classic performance model might determine costs in a single step. This monolithic approach needs to translate all input parameters (concerning the program's code, its configuration, and so on) to actual costs. This thesis, however, proposes a two-step performance deduction strategy consisting of a *code-to-indicator* and an *indicator-to-cost* analysis, as seen in Figure 2.12. As discussed later, this split increases versatility and portability.

First, a code-to-indicator analysis is performed. As this step resembles most aspects of static code analysis, it can be considered complex and infeasible for large code bases.

Yet, for many programs, measurements with common workloads can be perfomed beforehand. First, programmers would measure small yet typical workloads. Based on these measurements, programmers could extrapolate performance indicators by continuously increasing the workload sizes or measuring varying workloads [60]. Alternatively, they could transfer well-known hardware performance indicators from a different computer system to simulate the performance behavior on new hardware. In this way, the infeasible direct code-to-cost deduction can be circumvented.

The second step consists of an indicator-to-cost analysis. Performance indicators act as the interface between the two analysis steps hereby. The indicator-to-cost analysis



(a) monolithic model

(b) two-step strategy

Figure 2.12: Classical and proposed strategy. By selecting performance indicators as an intermediate step, the proposed strategy overcomes several issues. Static code analysis can be bypassed by measuring hardware counters and either extrapolating them for different workload sizes or transferring them between physical systems.

can be considered less complex compared to the first step, since hardware performance indicators relate to costs much more directly.

Even if some indicators are not directly connected to execution time performance, some of them—measuring wattage, for instance—can provide valuable insights about the system's hidden variables such as thermal conditions. In this example, knowledge about thermal conditions helps estimating the costs, because clock frequencies might correlate with thermal conditions [14].

Hardware vendors could perform indicator-to-cost analyses for all their hardware systems. Before deciding upon new hardware, enterprises and researchers could fingerprint their applications' key indicators as well as their performance. The vendors' models could then provide the customers with a more precise speed-up estimation, without ever having to run the program on new hardware.

Limitations—Input Selection, Sampling Bias, *Multiple Comparisons Problem*

This thesis suggests to consider all available hardware performance indicators as an intermediate phase for inferring costs yet aims at ensuring portability among machines. Because not all machines offer the same performance indicators, a selection of fairly common ones is necessary to begin with. Additionally, not all performance indicators are equally important, and some might even be redundant.

Because the strategy employs indicators actually related to the problem's scaling behavior, a further selection needs to be performed. In this case, the selection of a subset of indicators might diminish the strategy's expressive power, thus reducing its flexibility and the soundness of the performance analysis (*sampling bias*) [137]. As a response, this thesis later introduces an event selection program called *EvSel* (see Section 4.1). Often times, performance indicators are zero or do not significantly change within the execution of a specific program. These candidates should be considered for removal, along with other uncommon indicators only existing on a specific platform.

Conversely, when correlating a lot of input parameters to end costs, the sheer amount of parameters might reveal some seemingly well-fitting correlations. However, these correlations might not represent actual interdependencies but are instead caused by the high statistical possibility resulting from many measurement values. This problem is known as the *multiple comparisons problem* (or the *multiple hypotheses problem*) and states that if researchers just add more and more data to a data set, at one time, they will eventually find correlations [66]. To tackle these problems, statistics uses methods such as *Bonferroni correction*, which requires more samples when the possibility of a multiple comparisons problem exists [8]. Although this might be a candidate for future work, the tools presented in this thesis do not react to this issue, currently. Users should be aware of this fact when employing these tools to investigate performance.

3 Measuring Hardware Counters on Multiple Platforms

This chapter focuses on the technical foundations and related work for the tools later introduced in Chapter 4. After describing existing performance measurement utilities of CPUs, performance counters of different hardware platforms are discussed in detail. This chapter aims at giving practical usage examples for all mentioned event counters.

In the second part of this chapter, existing tools for performance optimization are presented, which build upon these low-level performance counters. Some of these tools allow for measuring counters solely, while others process this information further to allow for a detailed performance analysis of programs.

3.1 PMU Setups and Performance Counters

This section analyzes the *performance measurement units* (PMUs) of different hardware vendors. With each vendor, more sophisticated features are introduced and explained.

Performance counters are the lowest-level performance indicators provided by the PMU. They are measured in dedicated registers with very low overhead and are thus optimal for fine-grained performance analyses. Although the Linux measurement tool `perf` lists many performance counters by default, it can measure even more counters through *raw* registers by accessing the hardware directly. These counters are addressed with hex codes to the *model-specific control registers* (MSRs) of the CPU. To obtain all these hex codes for raw counters, the performance library `libPAPI` can be utilized.

supported hardware platforms
x86, x86_64 (Intel, AMD)
ARM
MIPS
Intel Itanium II, Montecito, Montvale
IBM POWER4, 5, 6, 7, 8
IBM PowerPC970, 970MP

Table 3.1: Selected hardware platforms supported by `libPAPI` [99]. The first two rows are supported via the `perf_events` driver of the Linux kernel. Itanium is supported via Perfmon2. IBM's platform-specific counters are obtained using the `perfctr` interface.

3.1.1 Obtaining Raw Event Counters with `libPAPI`

The *performance application programming interface* (PAPI) is a source of raw event counters that is frequently updated [92]. PAPI offers support and portability for many platforms ranging from IBM System/390 processors to up-to-date IBM Power8 and Intel Skylake architectures. PAPI does so by employing the model-specific registers via the `msr-tools` kernel module for Linux [32]. Furthermore, graphics and accelerator performance (for some architectures, power consumption only) can be measured for Intel's Xeon Phi and Nvidia's Tesla and Kepler architectures via their drivers and CUDA interfaces, respectively [42, 41]. With PAPI, it is even possible to measure the InfiniBand interconnect performance. In this case, the events are queried via the system's `sysfs` under `/sys/class/infiniband/`. Overall, PAPI can be seen as an integrator for many event sources. Selected and supported platforms for the Linux operating system family are shown in Table 3.1.

`libPAPI` further offers abstractions for vendor-specific event counters across the platforms. This enables comparability and portability. Because the project is supported by all major hardware vendors, its information quality can be considered very high. Many auto-optimization tools and scientific tools base their measurements on PAPI [20].

On many hardware platforms, the concepts of event codes, unit masks, and further event modifiers are found. *Event codes* describe the general class of performance events, whereas *unit masks* allow programmers to filter events by the location of their occurrences. Additional modifiers allow for specifying event properties such as inverted, user-mode, kernel-mode, system-wide, or edge counting. All available event modifiers are listed in the Linux `sysfs` under `/sys/bus/event_source/devices/cpu/format`.

`libPAPI` comes together with tools (which are named *examples* in the repositories) that allow for exploring the options of the available counters. As a remnance of the discontinued `perfmon2` project, `libpfm4` is one major source for this type of event information within PAPI. Below, an output of `showevtinfo`, which can be found in the `libpfm4` examples, is shown for determining all available event sources.

```
Detected PMU models:
[ix86arch, "Intel X86 architectural PMU", 7 events, 1 max encoding, core PMU]
[perf, "perf_events generic PMU", 80 events, 1 max encoding, OS generic PMU]
[hsw, "Intel Haswell", 73 events, 2 max encoding, core PMU]
[rapl, "Intel RAPL", 3 events, 1 max encoding, uncore PMU]
[perf_raw, "perf_events raw PMU", 1 events, 1 max encoding, OS generic PMU]
```

Listing 3.1: Part 1. Exemplary output of `libpfm4`'s PMU source detection. Although multiple PMU sources are identified in the system, their events might overlap. Note the fourth entry where the *running average power limit* (RAPL) as an energy measurement facility is listed additionally.

Next, an exemplary event code is shown with its umask configuration options and modifiers, as determined by `showevtinfo`.

```
IDX : 218103839
PMU name: hsw (Intel Haswell)
Name     : L2_RQSTS
Equiv : None
```

```
Flags    : None
Desc     : L2 requests
Code     : 0x24
Umask-00: 0x21: PMU: [DEMAND_DATA_RD_MISS]: Demand Data Read requests that miss L2
Umask-01: 0x41: PMU: [DEMAND_DATA_RD_HIT]: Demand Data Read requests that hit L2
Umask-02: 0x22: PMU: [DEMAND_RFO_MISS]: RFO requests that miss L2
Umask-03: 0x22: PMU: [RFO_MISS]: Alias to DEMAND_RFO_MISS
Umask-04: 0x42: PMU: [DEMAND_RFO_HIT]: RFO requests that hit L2
Umask-05: 0x42: PMU: [RFO_HIT]: Alias to DEMAND_RFO_HIT
Umask-06: 0x24: PMU: [CODE_RD_MISS]: L2 misses when fetching instructions
Umask-07: 0x27: PMU: [ALL_DEMAND_MISS]: All demand requests that miss the L2
Umask-08: 0x44: PMU: [CODE_RD_HIT]: L2 hits when fetching instructions, code reads
Umask-09: 0x30: PMU: [L2_PF_MISS]: Requests from the L2 prefetchers that miss L2
Umask-10: 0x3f: PMU: [MISS]: All requests that miss the L2
Umask-11: 0x50: PMU: [L2_PF_HIT]: Requests from the L2 prefetchers that hit L2
Umask-12: 0xe1: PMU: [ALL_DEMAND_DATA_RD]: Any data read request to L2
Umask-13: 0xe2: PMU: [ALL_RFO]: Any data RFO request to L2
Umask-14: 0xe4: PMU: [ALL_CODE_RD]: Any code read request to L2
Umask-15: 0xe7: PMU: [ALL_DEMAND_REFERENCES]: All demand requests to L2
Umask-16: 0xf8: PMU: [ALL_PF]: Any L2 HW prefetch request to L2
Umask-17: 0xff: PMU: [REFERENCES]: [default]: All requests to L2
Modif-00: 0x00: PMU: [k]: monitor at priv level 0 (boolean)
Modif-01: 0x01: PMU: [u]: monitor at priv level 1, 2, 3 (boolean)
Modif-02: 0x02: PMU: [e]: edge level (may require counter-mask >= 1) (boolean)
Modif-03: 0x03: PMU: [i]: invert (boolean)
Modif-04: 0x04: PMU: [c]: counter-mask in range [0-255] (integer)
Modif-05: 0x05: PMU: [t]: measure any thread (boolean)
Modif-06: 0x07: PMU: [intx]: monitor only inside transactional memory (boolean)
Modif-07: 0x08: PMU: [intxcp]: no count in aborted transactional memory (boolean)
```

Listing 3.2: Part 2. Output of a single event class description. One can see how the event code denotes the general group (here, L2 requests) and how the so-called *unit mask* describes all possible variants such as hits, misses, and prefetch requests. Lastly, *event modifiers* are listed, which change the measurement properties of the counter.

Plain counters simply accumulate event occurrences that match the specified mask and modifier configuration. When measuring a higher amount of events than there are counting registers, tools like `perf` fall back to round-robin measurements. Round-robin measurements (also called *event cycling*) can only be considered to be *sampled* over time, since not all events are counted continuously.

More complex events exist (for instance, denoting branching information or program counter status), which often times use more than one register to store information. Some trigger an *interrupt service routine* (ISR) to write data to memory instantly.

This thesis showcases event counters and special features of four major hardware vendors in detail to give an overview of different performance counter capabilities and their utility for performance evaluation.

3.1.2 ARM—Cortex-A8 and A9

In the most basic setup, ARM PMUs follow a general architecture of three registers, two of them configuring the event and one containing the number of counted events [7]. Since the vendor's business model is licensing CPU IP, ARM recommends PMU building blocks, which might be implemented by licensing companies to different extents.

The popular ARMv8 architecture incorporates one 64-bit-wide counter and up to 31 further 32-bit-wide counters. The wider, 64-bit counter is dedicated for cycle counting (cycles naturally amount to the highest numerical values). ARM guarantees counters to act non-invasively (not influencing program execution), yet ARM does not guarantee overall measurement accuracy. Thus, an inaccuracy of 5 % has to be expected on ARM systems [7]. This is caused by missing event validation facilities for speculative execution, which can amount to an »unreasonable overhead,« according to ARM.

In general, all PMU counter registers can handle overflows. On ARMv8, the generic `PMUIRQ` interrupt is triggered and an according overflow bit is set in the status register, which can be queried in the interrupt service routine later. Events are only counted depending on certain states of the CPU. If the PMU is disabled, no events are counted. If the CPU is in hardware debug state or ARM security state, specialized bits enable or disable counting. Additionally, all events can be filtered according to the *execution level* (`EL0` to `EL3`) of the ARM processor, which is similar to Intel's better-known *security ring* concept. Overall, ARM suggests 48 standardized events for their ARMv8 IP [7].

The following showcases how to access and measure the raw performance counters for the newer Cortex-A9 CPU, which supports both the ARMv7 and ARMv8 architectures depending on the model. For this purpose, `libpfm`'s `showevtinfo` is used to provide the hex codes of the registers, later used by Linux `perf` to measure the values.

```
~/papi/papi-5.4.3/src/libpfm4/examples$ ./showevtinfo
[...]
#-----------------------------
IDX  : 138412038
PMU name : arm_ac9 (ARM Cortex A9)
Name     : DREAD
Equiv  : None
Flags    : None
Desc     : Data read architecturally executed
Code     : 0x6
#-----------------------------
IDX  : 138412039
PMU name : arm_ac9 (ARM Cortex A9)
Name     : DWRITE
Equiv  : None
Flags    : None
Desc     : Data write architecturally executed
Code     : 0x7
#-----------------------------
[...]
#-----------------------------
IDX  : 138412050
PMU name : arm_ac9 (ARM Cortex A9)
Name     : JAVA_SW_BYTECODE_EXEC
Equiv  : None
Flags    : None
Desc     : Software Java bytecodes decoded, including speculative (approximate)
Code     : 0x41
#-----------------------------
[...]
```

Listing 3.3: Three exemplary raw PMU events on a Cortex-A9-enabled Parallela board.

```
~/papi/papi-5.4.3/src/libpfm4/examples$ perf stat -e r06,r07,r41 ls ~
papi

 Performance counter stats for 'ls /home/christoph.sterz':

            601975      r06
            261973      r07
                 0      r41

       0.009860732 seconds time elapsed
```

Listing 3.4: Exemple measurement of memory reads and writes using `perf`. Outputs for raw registers, as obtained by `libpapi` in the listing before, show that about 2.3 times as many data reads as writes happen. No Java bytecode is decoded, as expected for `ls`.

Note that for all architectures, the CPU counter accuracy depends on the thermal budget of the respective CPU. The amount of jitter in all cycles depends on the aforementioned measurement inaccuracies as well as the fluctuating CPU clock frequency. Some vendors, such as Intel, react to this by offering so-called *ref cycle* events that stem from an independent, non-scaling clock.

3.1.3 AMD—Family 15h (*Bulldozer*)

Compared to ARM, AMD provides many more counter events, which can be specialized even further. Modern AMD Family 15h processors (code-named *Bulldozer*) incorporate up to six available 48-bit-wide hardware counters with about 100 available events, depending on the model [1]. AMD calls these events *performance counter events*, which are counted for the core and northbridge sections separately.

Northbridge events describe additional interaction with I/O peripherals and buses. By leveraging these events, the NUMA remote accesses are counted, which are seen as remote I/O by the local node. Additionally, the *cache block commands*, which ensure coherency across NUMA nodes, can be monitored with counters from this category.

With AMD, many more counters can be parametrized with *umask* (unit mask) filters, which in turn allows for characterizing the events further. AMD's unit masks offer dedicated filters for each event. A possible unit mask description is shown in Table 3.2.

Instruction-Based Sampling

Instruction-based sampling (IBS) was introduced to make events attributable to code during execution [48]. In general, all the following hardware vendors—AMD, Intel, and IBM—implement a form of IBS in their PMUs. With IBS, instructions that cause a previously defined event are sampled with a selectable occurrence frequency. This frequency is continuously altered to avoid sampling the same instructions over and over

unit mask bit	description
0	IC fill
1	DC fill
2	TLB fill (page table walks)
3	NB probe request
4	cancelled request
5	*reserved*
6	L2 cache prefetcher request
7	*reserved*

Table 3.2: Exemplary unit mask bits for the event PMC `x07D` (requests to L2 cache). A customization of the counting mode can be achieved by setting the respective bits [1].

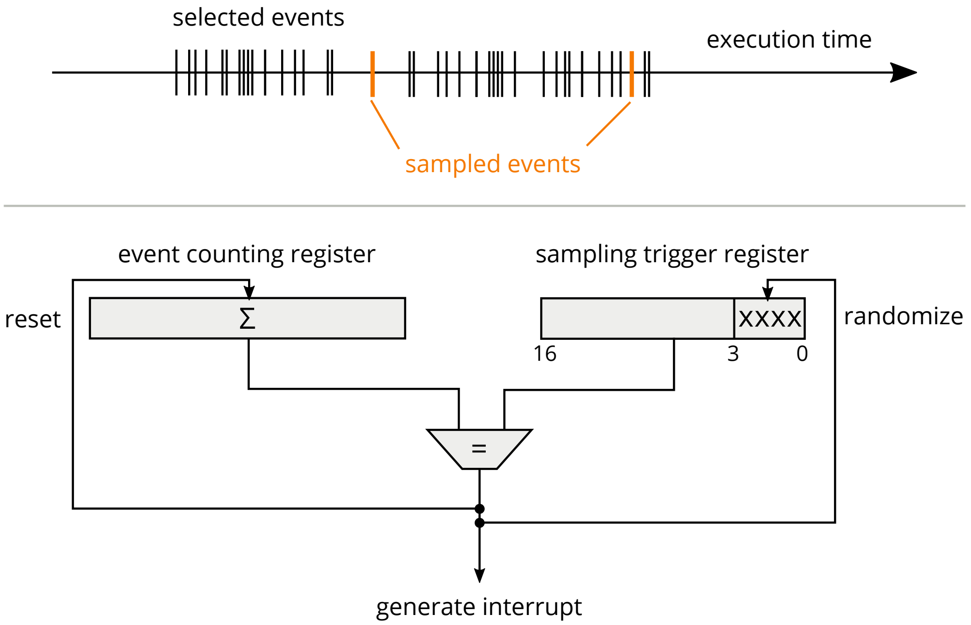


Figure 3.1: IBS sampling. A counter is incremented for every selected event. If the sampling counter is surpassed, an interrupt is generated and the lower 4 bits of the sampling rate are altered randomly.

in tight loops. The method can subsequently be considered an empirical measurement. A block diagram explaining this sampling method further is shown in Figure 3.1.

The resulting IBS record of a captured sample comprises a virtual or even physical address. Additionally, information about the instructions' hits and misses in the caches and the TLB are recorded. A typical IBS ISR copies all these values to a preallocated kernel memory region for later analysis.

Because modern CPUs are designed in a pipelined fashion, there is a small delay of unknown duration called *skid*, which is the time between issuing the instruction and detecting the PMU event at an arbitrarily delayed point in time. This causes the address of the instruction pointer to be incorrect and spread around the actual address of the instruction. In their paper, Drongowski et al. are the first to present IBS in the context of AMD's Catalyst analysis suite [48]. They explain how IBS is able to reduce jitter in skid dramatically for newer architectures, resulting in more accurate measurements.

As an opposite effect to skid, Chen et al. identify an additional phenomenon called *shadow effect*, which impairs the sampling results [27]. Shadow is caused by instructions with long stalls such as TLB misses or NUMA remote memory loads, which get sampled disproportionally more often than shorter instructions. For a detailed investigation on more effects and overheads, the works of *CERN openlab* (Andrzej Nowak et al.) are a good source of information [98].

There are two different kinds of instruction-based sampling in modern AMD processors: fetch sampling and op sampling [1].

Fetch sampling describes the loading process of an instruction until it reaches the CPU's decoding unit. A term closely related to this is *frontend-stalled*, which describes that the processor suffers from the delay in retrieving and decoding an instruction. In the IBS records for fetch sampling, one can find hit and miss information for all instructions with respect to caching and TLBs, as depicted in Figure 3.2.

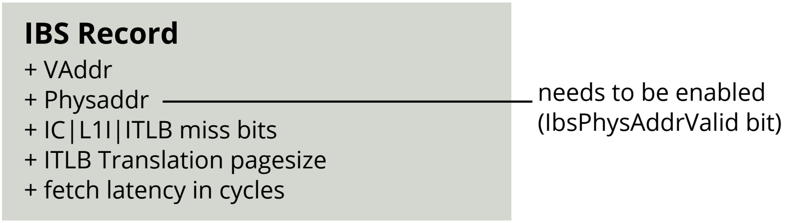


Figure 3.2: Fetch record. When a certain amount of events has passed, one instruction is sampled in detail. The IBS record contains all information valuable to the programmer, such as miss addresses, miss information bits, and fetch latencies [1].

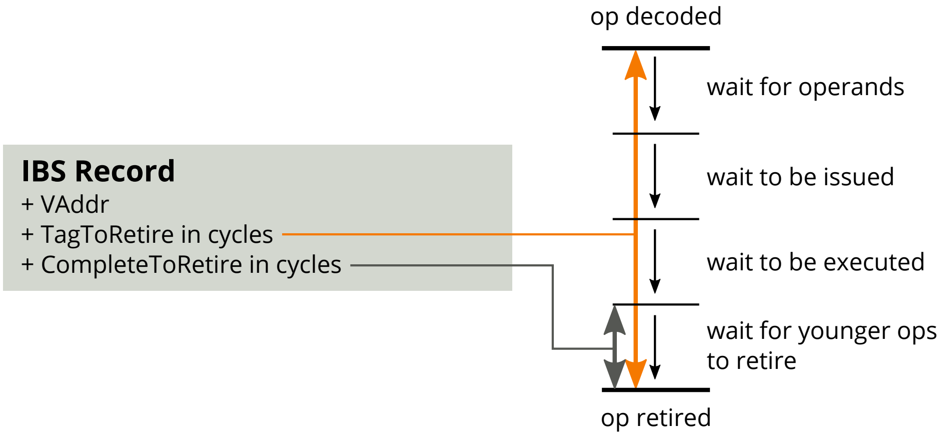


Figure 3.3: Op record. For op sampling, only a virtual address is recorded. As shown in the graphic, the latencies `TagToRetire` and `CompleteToRetire` denote different timings in the instruction's execution lifetime.

After fetch sampling, *op sampling* describes the subsequent life cycle of the instruction, that is, from reaching the decoder until causing the last effect on other operations inside the execution pipeline. The corresponding term here is *backend-stalled*. Op sampling can be done for events that issue a branch, resync (pipeline flush), return, load, or store. When combining fetch sampling and op sampling, the overall observable life cycle of an instruction can be documented accurately. The exact information stored in the IBS record for op sampling can be found in Figure 3.3.

To further reduce the overhead of interrupts with IBS, AMD introduced *lightweight profiling* (LWP). With few more instructions, a userland control block can be defined that triggers LWP for IBS. By using LWP, IBS measurement results are written directly to a userland ring buffer through the PMU interrupt service routine. Unfortunately, this feature is discontinued with the AMD Zen architectures. The Linux kernel, however, is able to implement the same behavior when using the `perf_event_open` facility.

3.1.4 Intel—Haswell

The performance counters of Intel processors are similar to the ones described for AMD [70]. Modern Intel CPUs (Haswell, 3rd generation) provide up to eight individual counters per core (if engineers attribute events to hyperthreads, these split to 2 × 4). There are about 300 plain performance events available on recent processors [37].

At first, a close-up look on *precise event-based sampling* (PEBS) is taken, which is similar to AMD's IBS. This section then further focuses on the performance-relevant NUMA-related events within PEBS. A short introduction to CPU-supported power measurements using Intel *running average power limit* (RAPL) facility and the *last branch record* (LBR) concludes Intel's PMU specifics [135].

Precise Event-Based Sampling

Intel *precise event-based sampling* (PEBS) is a sampling-based monitoring tool capable of recording the location and the operator information of specific instructions. As with AMD IBS, a sampling counter determines the measurement rate. Especially the in-depth measurement capabilities of PEBS for determining memory load latencies are relevant to NUMA, since they also incorporate costs for remote instruction and operand fetching as well as NUMA-induced cache coherency.

For PEBS events, engineers can choose a so-called *precise level* of skid that they consider acceptable for their measurements.

Level 0 (no perf extension) allows for arbitrary skid

Level 1 (:p) only allows for a fixed amount of skid

Level 2 (:pp) skid requested to be 0

Level 3 (:ppp) skid forced to be 0

PEBS includes the instruction's virtual address in its record only. Intel argues that the most relevant information about cache lines can still be retrieved, since the lower bits are identical in virtual and physical addresses. Note that for better readability, all events are denoted in Intel `perfmon` mnemonics rather than in hexadecimal representation.

Instead of specifying event codes and unit masks, Intel denotes each of these combinations with capital letter identifiers. The variety in memory load specifics for PEBS is high. For memory accesses, three events exist: `MEM_INST_RETIRED` for any instruction containing an operand access, `MEM_LOAD_RETIRED` for load instructions only, and `MEM_STORE_RETIRED` for instructions comprising a store operation.

Regarding NUMA, all interaction concerning L3 is relevant, since both data transfers as well as cache coherency between cores happen at that level. Since Intel incorporates a snooping-based approach for cache coherency, each NUMA node's L3 cache contains consistency information for each cache line concerning the other NUMA nodes [91]. The PEBS record contains information about the data source and cache coherency state for data of a given instruction in its lower 4 bits. Table 3.3 depicts the data source locations for the cache protocols in detail.

Load latency is usually determined for `MEM_INST_RETIRED`. With Intel, it is only possible to determine if the latency of a certain memory request is above a predefined threshold (which is called `ldlat` in this case). The load latency threshold is written to the MSR at `0x3f6`. If an instruction has a higher latency, the event counter is increased. Only one counter can be used while determining the latency, so that two events may never be recorded simultaneously. This latency measurement facility is used by the tool *Memhist*, which is described in Section 4.3.

Last Branch Record

PEBS further provides detailed information on the recent branching behavior of a branch instruction (*precise branch events*) by filling a buffer that Intel calls the *last branch record* (LBR). The LBR contains branching source and target addresses for the last 16 branches. This is explained in more detail in Section 3.2.1.

Running Average Power Limit

With Sandy Bridge, Intel introduced the *running average power limit* (RAPL) to its *power control unit* (PCU), which is able to estimate power usage with an internal model [35]. The power usage since start-up can be queried via `sysfs` under `/sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj`, for usage as yet another low-level performance indicator. To estimate the power consumption, the PCU measures the CPU's frequency and current. Power consumption is determined for each core individually to spend more thermal budget during a short period of time if possible. Settings concerning the thermal budget can be found and changed via the `sysfs`, too [129].

encoding	description
`0x0`	unknown L3 cache miss
`0x1`	L1 hit
`0x2`	data was in fill buffer (on the way to L2)
`0x3`	L2 hit
`0x4`	L3 hit, no coherency measure necessary
`0x5`	L3 hit remote, no modified copies
`0x6`	L3 hit remote, modified copies where present
`0x7`	reserved
`0x8`	L3 miss remote, no modified copies (forwarded)
`0x9`	reserved
`0xA`	L3 miss, serviced by local DRAM (go to shared)
`0xB`	L3 miss, serviced by remote DRAM (go to shared)
`0xC`	L3 miss, serviced by local DRAM (go to exclusive)
`0xD`	L3 miss, serviced by remote DRAM (go to exclusive)
`0xE`	I/O, request of input/output operation
`0xF`	the request was to uncacheable memory

Table 3.3: Location records for memory loads. Event masks for selecting specific memory load events. NUMA-relevant events are highlighted.

3.1.5 IBM—Power8

The vendor IBM offers rich features within the PMUs, often times covering the ones presented for the previous CPU architectures [126]. This thesis presents a summary of the hardware counter facility of IBM's Power8 architecture. IBM's counting facility is called *performance counter monitor* (PCM) [23].

For Power8 systems, there are six performance counters per hardware thread. The first four ones are bound to specific event groups, the fifth one measures instructions, and the sixth one measures cycles. IBM recommends measuring *cycles per instruction* (CPI) as one of the key performance indicators, using the latter two registers at all times [87].

In addition to the usual control registers, there are three registers with further information concerning the last event [69]:

SIAR the sampled instruction address register

SDAR the sampled data address register

SIER the sampled instruction event register

SIAR holds the approximate virtual address of the sampled instruction. SDAR provides information about the instructions and operands addresses alike. With SIER, IBM offers a functionality to determine events related to the instruction's execution. SIER contains information about the class of events, cache fetching information, branching information, and even reasons for mispredictions of speculative branches.

As in the case of Intel's LBR, IBM provides a so-called *branch history rolling buffer*, which represents a record of the last branches [107].

Threads running on a system can be flagged to be either counted by the PMU or ignored. In this way, threads can be preselected for performance analyses. Similar to Intel and AMD architectures, IBM offers a *performance monitor interrupt*. This interrupt is either triggered by hardware or artificially invoked by executing the (`and 0,0,0`) NOP [126].

IBM does not specify the skid of general events. This implies that most counters cannot be considered precise. There are exceptions, however, that allow exact precision for counters on Power8. If the instruction forces a synchronization (such as *transactional memory instructions* do), they can be considered precise. IBM introduced the concept of verified *marked event counters* (often annotated with MRK), which can also be considered precise [126].

IBM provides various event counting modes. Default, continuous counting simply increments associated counters upon every event occurrence. Randomized sampling can be applied to all instructions. Loading or branching instructions can be sampled with even more options. As with AMD's sampling, a small random change in the sampling period is employed to avoid sychronizations of events inside tight loops. There exists a third method that employs time points for sampling. This so-called *time-based transition event* allows for better attributability among threads in parallel applications, because sampling intervals are triggered using time information instead of cycles.

All counters can be *frozen* inside the Power8 PMU with the help of bits in the PCM control registers. This either allows for emitting events when reaching a certain counter value or for excluding parts of executables from being counted or analyzed by the PCM.

3.2 Existing Tools for Measuring Hardware Performance Counters

Chapter 4 presents novel tools that analyze performance based on hardware counters. With this in view, this section introduces related tools for comparison. Since this thesis focuses on execution time performance mainly, tools optimizing for other aspects such as memory usage are mentioned briefly only.

3.2.1 The Linux perf Utility

After describing the hardware facilities that offer information about the system's performance state, this thesis explains the Linux perf utility. During the course of this work, perf became the centerpiece for performance measurements. The author recommends all programmers concerned with performance to learn the basic perf functionalities. Many tools presented in this thesis are built upon the Linux perf utility.

perf constitutes an integrating as well as an abstracting layer over raw counters and kernel events. It became a part of the Linux kernel in version 2.6.31 [106]. perf abstracts from the individual hardware counters on different platforms. Additionally, Linux perf offers tracepoints and counters for all kernel-mode activities such as networking, scheduling, or filesystems. Kernel facilities as well as arbitrary function symbols can be injected with a perf kernel probe.

In the immediate measurement mode (perf stat), events can be counted for program runs. Repetitions can be specified such that event counts are averaged and basic statistical errors are calculated and presented. Available abstract events can be listed with perf list. This list subprogram additionally shows all aforementioned kernel tracepoints when executed as a superuser.

perf is able to attribute all measurements to specific locations in a fine-grained fashion. Programs can be measured on the entire system, on specific CPUs, and on sockets. Also, a system-wide mode exists, which traces all processes on a CPU or even across the whole system, again requiring superuser permissions. This allows for detecting imbalanced workloads between NUMA nodes, for instance.

Through the event-based sampling facilities (if existing on the processor), perf is able to record (perf record) events and attribute them to individual addresses. In this way, assembly—or code if debug symbols are available—can be viewed, which is then annotated with the number of event occurrences (perf report).

Branching Information from Last Branching Records

`perf` offers options to monitor the branching behavior of programs using CPU facilities such as the *last branching record* (LBR). The branching information enables engineers to extract short traces of the last branching decisions leading to a specific instruction. Since branching records contain both the source and target addresses of branches, the most frequently taken branches can be analyzed statistically. All branches are also tagged as function calls, regular branches, or returns.

Compilers are able to generate compiliation hints for *profile-guided optimization* (PGO)[1] from the LBR [96]. Through LBR, the causes of errors originating from the improper use of transactional memory instructions can also be debugged.

Note that on most Linux systems, one PMU counter is used as a watchdog for *non-maskable interrupts* (NMI). Because of this, `perf` measurements that require all available registers do not work when this watchdog is enabled. Non-maskable interrupts cannot be interrupted by any other event inside the system. Infinite loops inside NMI ISRs can thus bring the system to a complete halt state, which is non-recoverable. With the help of the watchdog, the system is able to detect the loops and recover, though. Although `perf` warns about the NMI, it does not automatically disable it before measurements. One can disable the NMI watchdog when executing the line below or by disabling it in the system's bootloader.

```
echo 0 > /proc/sys/kernel/nmi_watchdog
```

Listing 3.5: Disabling the Linux NMI watchdog.

[1] `-fprofile-generate` for gcc

Linux API: `perf_event_open`

Linux `perf_event_open` is a programming API for `perf`'s functionality, which can be embedded into software to measure counted or sampled events [134]. After setting up an event, a file descriptor is returned, which can be queried with the `read` call from then on. Sampled events are periodically written to a ring buffer specified beforehand. Example code for counted events can be found in Appendix A.

While `perf_event_open` is the library counterpart of the `perf` command and does not add new functionality to `perf`, programmers are enabled to measure specific parts of their code without needing to run `perf` in an additional process. Yet, this approach requires modifying the program code and might thus also involve a management overhead, such as aggregating the measured values or printing them. `perf_event_open` might help in making dynamic decisions in the program's execution based on reflection upon its own behavior.

3.2.2 Intel VTune Amplifier and AMD CodeAnalyst

Intel's *VTune Amplifier* is a product commercially available on Linux and Windows, which allows for analyzing the performance of code [34]. VTune offers a graphical user interface as well as a command-line interface and can be run on a remote host for performance analyses over the network. VTune makes use of program instrumentation, hardware counters, and precise event-based sampling to attribute performance measurements to lines of code. Performance is recorded during program runtime, and thus, a temporal analysis of a program is possible. The utilization of multi-threading is analyzed and classified from poor to excellent. When using VTune, programmers can gain information at different levels of abstraction. As a first overview, a performance summary is created automatically, which points out major hotspots and performance issues in the programs execution. Lines of code, individual variables, as well as called functions can be further investigated with respect to their performance.

As with VTune, AMD's *CodeAnalyst* is available on Linux and Windows [47]. CodeAnalyst is able to attribute previously recorded performance-relevant measurements, such as function sampling through instrumentation and hardware performance counters, to lines of code. This tool is additionally applicable when analyzing performance problems in heterogeneous computing, namely CPU–GPU interaction in OpenCL [121]. For Linux systems, all `perf` events can be used as an input to CodeAnalyst, which enables it to also trace kernel profiling probes.

3.2.3 PMU Tools

The most prominent open-source tool for aggregating the information available to `perf` is `pmuevents` [77]. It is restricted to Intel CPUs and consists of a set of Python scripts. Before start-up, `pmuevents` automatically downloads processor-specific PMU information from Intel's open-source web platform [38]. `pmuevents` combines measurements with general recommendations and performance drilldowns recommended in the software optimization guides of the vendor [37].

Engineers new to the field of performance measurements should use `toplev.py`, which points out main program bottlenecks such as front-end or back-end stalling issues. The tool `ocperf.py` eases the measurement of Intel's symbolic event notation, which is much more readable than raw event codes and thus easy to understand.

3.2.4 Other Tools

Tools for Windows

On Windows, Microsoft Visual Studio offers profiling capabilities, which are based on recording instrumented code samples that can be analyzed temporally [72]. Visual Studio, however, does not consider hardware counters in its analysis. To measure raw PMU counters on Windows, *Xperf*, a part of the *Windows Performance Toolkit*, can be used, which yet only records counters for a fixed amount of time after start-up. The results recorded with Xperf can later be visualized with *Xperfview* [36].

Valgrind

Valgrind is a collection of tools concerning memory debugging and profiling, which is available on Linux, Android, as well as OS/X. Though it does not rely on hardware performance counters for its performance measurements, Valgrind can analyze caching behavior through its subtools `callgrind` and `cachegrind` [94]. Valgrind can obtain this information because it co-emulates all instructions and memory behavior in a virtualized processing system during execution. On the one hand, this allows for introspection but on the other hand, this slows down measurements by factors of up to 100.

Heaptrack

As an upcoming open-source tool for finding unneccessary heap allocations, `heaptrack` can trace most heap allocations through the interception of `malloc` calls. Results are presented in a graphical user interface, offering top-down and bottom-up analyses and attribution up to individual variables [138].

Intel Memory Latency Checker

To assess properties inherent to the NUMA interconnects, the Intel *Memory Latency Checker* (`mlc`) can be utilized [33]. The tool runs a custom test suite between all locations memory could reside in. In this way, latency and bandwidth information for caches, memory accesses, and even NUMA remote accesses can be determined. On a command-line interface, node-to-node interconnect information is presented in the form of latency matrices.

4 New Tools for Analyzing Performance

This chapter introduces four novel tools, which support the performance optimization strategy formulated in Section 2.6. The strategy proposes splitting cost deduction into two steps, using low-level hardware counters (see Section 3.2) intermediately.

EvSel covers all hardware counters to compare program runs and to perform program parameter regressions. *Phasenprüfer* automatically attributes the counters to distinct ramp-up and calculation phases of a tested program. *Memhist* leverages the load latency events for load instructions to reveal the latency cost distribution of memory accesses. *Hydralisk* artificially loads individual interconnect links while simultaneously measuring their bandwidths. In contrast to the other tools, which concentrate on performance indicators, Hydralisk influences and assesses the environmental parameters.

As all tools are built upon Linux `perf`, they are adoptable to hardware platforms where according performance counters are exposed. For each tool, development considerations and the resulting scope of applicability are explained. This chapter provides examplary usages of the presented tools in characteristic experiments.

4.1 Selection Through Correlation: *EvSel*

This section introduces *EvSel*, a tool that retrieves, measures, and presents all physically available hardware counters to the user. EvSel allows for identifying and excluding counters that are zero or that do not matter for performance considerations. EvSel also enables programmers to compare two versions or parameter configurations of a program with respect to all performance counter information.

Often times, programmers hypothesize that certain performance indicators correlate to input parameters of a program. With EvSel, such individual correlations can be detected automatically. Therefore, EvSel varies specified input parameters in order to determine functional dependencies between the input parameters and each measured indicator. As a qualitative assessment, EvSel computes statistical confidence values both for comparisons and correlations.

Altogether, EvSel allows programmers to pinpoint relevant bottlenecks stemming from the underlying mechanism (represented by a specific indicator) to investigate further.

4.1.1 Retrieving Performance Counters

To allow for interoperability and portability across multiple hardware platforms, EvSel is built on top of `perf`. The event codes available on the platform are read from a JSON file that provides descriptions for the events. EvSel presents event codes with all possible unit masks alongside the resulting semantic description.[1] Additionally, a detailed description of the events is shown, which can later be used for identifying the corresponding performance problem.

EvSel was designed to measure all performance counters during the whole program run, and does not perform event cycling thus. Because only a limited number of registers is available for measuring, however, program runs are repeated to circumvent this problem. This thesis argues that collecting counters over identically configured program runs instead of performing event cycling during execution (through often times more than 100 counters) might be the better choice when measuring many counters.

[1] Using all event/umask combinations sometimes results in configurations that have identical effects. It is the aim of the program, not to exclude possible sources of information, though.

All retrieved values are stored as raw as possible with their event identifiers for a single measurement run. To process the data further, a chain of lazily evaluated filtering and aggregating C++11 functors (lambdas) and functions is applied to the raw data. This architecture does not preaggregate or reject values and thus aims for extensibility. For the task of regression, the data is represented as raw matrices for faster computation using linear algebra libraries.

The user interface is written in C++ with the addition of the Qt GUI toolkit. Events are separated into groups for core and offcore events.

Several visual cues help engineers understand data more quickly. If a value remains zero for all measurements, it is greyed out. Correlations are color-coded for a quick overview. Tooltips are added to reveal more detailed information.

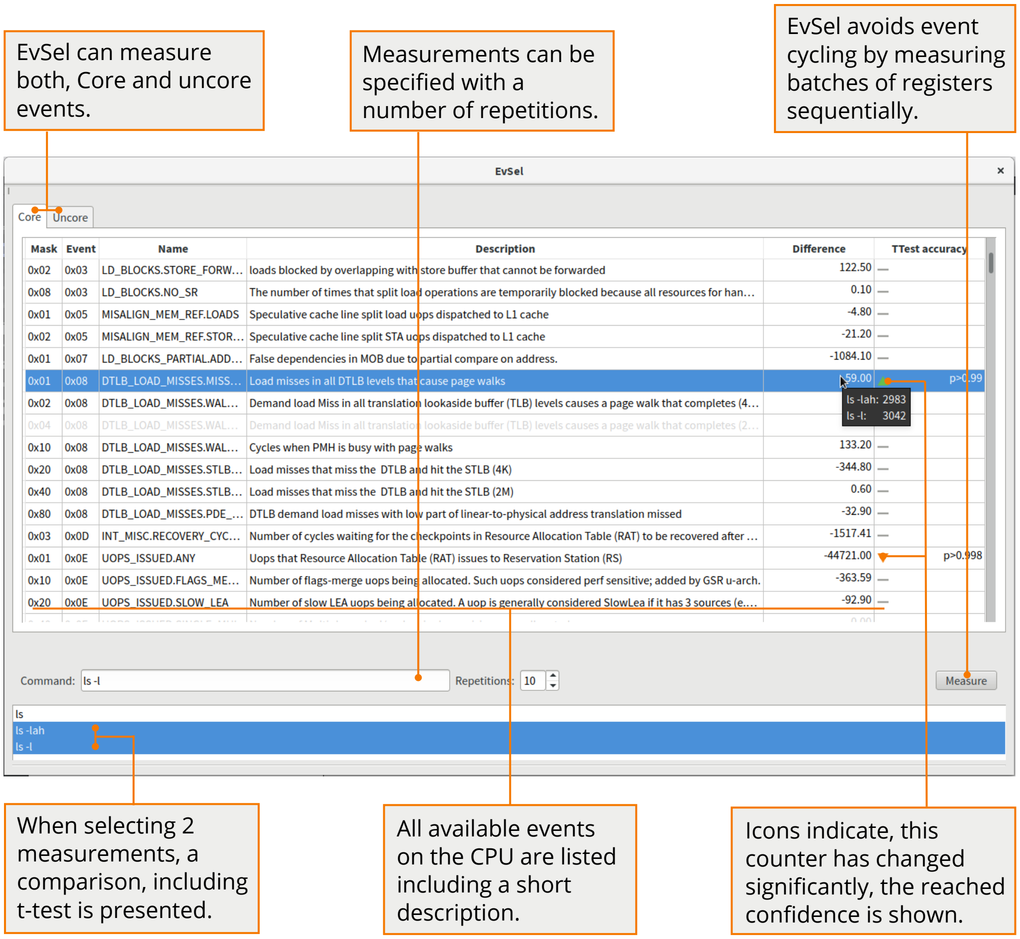
The main functionality of EvSel is depicted in Figure 4.1. Interested readers may try out EvSel, improve it, or add desired functionalities using the repository.[2]

4.1.2 Comparison and Correlation

EvSel uses regressions to correlate parameters with event counters. To find interdependencies, linear, quadratic, and exponential regressions are created and evaluated. The library *Eigen* 3 is used to retrieve both regression parameters and errors by means of linear algebra [49]. This method is also used for other purposes in the tool *Phasenprüfer* and explained in more detail in Section 4.2.

For single comparisons, *Student's t-test* is applied to the measurements [123]. Upon selecting more than one measurement, Student's t-test is conducted among all events of the two measurements. In the rest of this section, this thesis shortly outlines the statistical decisions made for this method.

[2] https://github.com/chsterz/performance-tools.git



EvSel can measure both, Core and uncore events.

Measurements can be specified with a number of repetitions.

EvSel avoids event cycling by measuring batches of registers sequentially.

Mask	Event	Name	Description	Difference	TTest accuracy
0x02	0x03	LD_BLOCKS.STORE_FORW…	loads blocked by overlapping with store buffer that cannot be forwarded	122.50	—
0x08	0x03	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for han…	0.10	—
0x01	0x05	MISALIGN_MEM_REF.LOADS	Speculative cache line split load uops dispatched to L1 cache	-4.80	—
0x02	0x05	MISALIGN_MEM_REF.STOR…	Speculative cache line split STA uops dispatched to L1 cache	-21.20	—
0x01	0x07	LD_BLOCKS_PARTIAL.ADD…	False dependencies in MOB due to partial compare on address.	-1084.10	—
0x01	0x08	DTLB_LOAD_MISSES.MISS…	Load misses in all DTLB levels that cause page walks	59.00	▲ p>0.99
0x02	0x08	DTLB_LOAD_MISSES.WAL…	Demand load Miss in all translation lookaside buffer (TLB) levels causes a page walk that completes (4…		
0x04	0x08	DTLB_LOAD_MISSES.WAL…	Demand load Miss in all translation lookaside buffer (TLB) levels causes a page walk that completes (2…		
0x10	0x08	DTLB_LOAD_MISSES.WAL…	Cycles when PMH is busy with page walks	133.20	—
0x20	0x08	DTLB_LOAD_MISSES.STLB…	Load misses that miss the DTLB and hit the STLB (4K)	-344.80	—
0x40	0x08	DTLB_LOAD_MISSES.STLB…	Load misses that miss the DTLB and hit the STLB (2M)	0.60	—
0x80	0x08	DTLB_LOAD_MISSES.PDE_…	DTLB demand load misses with low part of linear-to-physical address translation missed	-32.90	—
0x03	0x0D	INT_MISC.RECOVERY_CYC…	Number of cycles waiting for the checkpoints in Resource Allocation Table (RAT) to be recovered after …	-1517.41	—
0x01	0x0E	UOPS_ISSUED.ANY	Uops that Resource Allocation Table (RAT) issues to Reservation Station (RS)	-44721.00	▼ p>0.998
0x10	0x0E	UOPS_ISSUED.FLAGS_ME…	Number of flags-merge uops being allocated. Such uops considered perf sensitive; added by GSR u-arch.	-363.59	—
0x20	0x0E	UOPS_ISSUED.SLOW_LEA	Number of slow LEA uops being allocated. A uop is generally considered SlowLea if it has 3 sources (e.…	-92.90	—

ls -lah: 2983
ls -l: 3042

Command: ls -l

Repetitions: 10

ls
ls -lah
ls -l

When selecting 2 measurements, a comparison, including t-test is presented.

All available events on the CPU are listed including a short description.

Icons indicate, this counter has changed significantly, the reached confidence is shown.

Figure 4.1: EvSel interface. EvSel presents all retrieved counters alongside their description. In the shown case, the two selected measurements are compared.

First, the implementation assumes a normal distribution. This decision can be considered controversial, since the measurement is clearly biased towards smaller values. The bias is due to the fact that often times, there is a minimum value that cannot be undercut. This value can be thought of as the optimum execution plan of the program. For this reason, EvSel first assumed a log-normal distribution. This decision, however, turned out to favor very small values overly. Conversely, when employing a log-normal distribution, the t-test did not perform well on very large values. Caused by these problems, EvSel relies on a plain normal distribution for now. However, determining the aforementioned minimum with a suitable estimator and employing a gamma distribution starting at this minimum point could capture the underlying process statistically more accurately.

Second, the t-test uses *Bessel's correction* to correct the degrees of freedom when calculating standard deviations for a mean that is not known prior to the measurement [103].

Third, since the test should be possible for different user-chosen program runs, *Welch's method* is employed to compare different population sizes [89]. EvSel assumes similar standard deviations for both measurements, since the mechanisms producing the values are the same—in this case, the hardware counters.

4.1.3 Exemplary Results

This section analyzes and compares selected microbenchmarks with EvSel related to performance issues. In a first example, a comparison of program configurations is presented for a cache miss scenario. A second example shows EvSel's results when finding correlations between input parameters and measured indicators for parallel sorting.

Cache Miss Microbenchmark

In the following code listing, an array is created and also read in column-major order (left), hereby hitting cache lines fairly often. On the right side, the array is read in row-major order instead, causing many more cache misses than before.

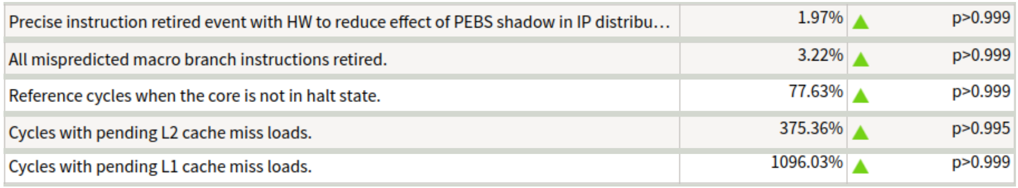
```
const size_t size = 1024;
auto array = new float[size][size];
float altsum = 0;

// fill array with random values

for (size_t y = 0; y < size; y++)
  for (size_t x = 0; x < size; x++)
    if (y % 2 == 0)
      altsum += array[y][x];
    else
      altsum -= array[y][x];

std::cout << altsum << std::endl;
```

```
const size_t size = 1024;
auto array = new float[size][size];
float altsum = 0;

// fill array with random values

for (size_t x = 0; x < size; x++)
  for (size_t y = 0; y < size; y++)
    if (x % 2 == 0)
      altsum += array[y][x];
    else
      altsum -= array[y][x];

std::cout << altsum << std::endl;
```

EvSel reveals interesting insights even beyond cache misses. The difference in the numbers of cycles can be fully explained with execution stalls. As expected, all cache levels suffered from the increased stride length in the accesses. L1, L2, and L3 cache misses rose by over 1000 %, 300 %, and 50 %, respectively. Interestingly, L2 prefetch requests dropped by 90 %, since prefetchers directly accessed the L3 cache (L3 cache accesses increased by a factor of 100). The biggest increase was notable in *rejected fill buffer requests*.[3] While in the example of cache hitting, the fill buffer rarely had to reject a demand (26 occurrences), it rejected nearly all registration attempts in the case of the cache miss example (3 million occurrences). All these values reveal statistical differences with significances of over 99.9 %, which is typical of such large absolute changes.

As expected, branch misses (3.2 %) and instruction-related values (1.9 %) show very small changes of their absolute values. Still, a minor correlation caused the values to be statistically distinguishable through t-tests. Selected results are shown in Figure 4.2.

[3] Instructions that miss the L1D cache can register their fetch demand in the fill buffer, which holds around ten entries, depending on the CPU model.



Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribu…	1.97%	▲	p>0.999
All mispredicted macro branch instructions retired.	3.22%	▲	p>0.999
Reference cycles when the core is not in halt state.	77.63%	▲	p>0.999
Cycles with pending L2 cache miss loads.	375.36%	▲	p>0.995
Cycles with pending L1 cache miss loads.	1096.03%	▲	p>0.999

Figure 4.2: Selected run comparisons with EvSel for a caching microbenchmark.

Parallel Sort Microbenchmark

As a second microbenchmark, the parallelization of `std::sort` using GNU `libstdc++` parallel mode is analyzed. For this purpose, a 4 MiB array of `uint` is filled with pseudorandom numbers using a *linear congruential engine* (LCE), which is essentially a multiply–add ignoring overflows [78]. Figure 4.3 shows the code as well as a chart that depicts the runtime development with regard to the number of threads.

This development can be investigated further using EvSel. Due to an increased use of the cache protocol for the shared data, the regression detects a strong correlation ($R > 0.95$) between thread count and L1 data caches being locked. The L1D cache is locked due to TLB page walks by the uncore, which manages the core interplay.

One can also observe a high negative correlation between the number of threads and retired speculative jumps ($R > 0.99$). This is a bad sign—this means that the CPU was not able to speculatively predict more actually executed instructions. Especially when comparing between hyper-threaded and non-hyper-threaded execution, this becomes clear because for 1–4 threads, relatively constant values were detected and rose in one step by 30 % to a relatively constant level for 5–8 threads. Three selected correlations can be seen in Figure 4.4.

```
omp_set_num_threads(numThreads);

const long size = 1024*1024;
std::vector<uint> data;
data.reserve(size);

//BSD linear congruential engine
const uint lcg_a = 1103515245;
const uint lcg_c = 12345 ;
uint lcg = 1337;
for(long i = 0; i < size ; i++)
{
  lcg = lcg * lcg_a + lcg_c;
  data.emplace_back(lcg);
}

std::sort(data.begin(), data.end());
```

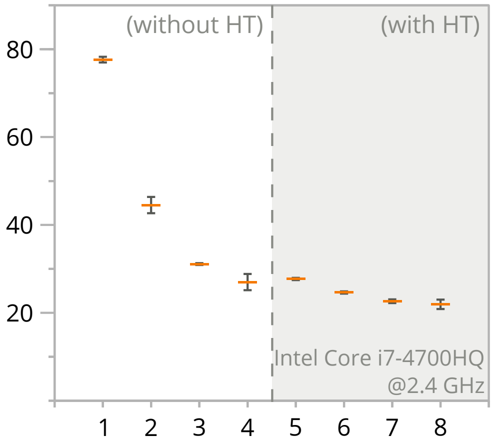
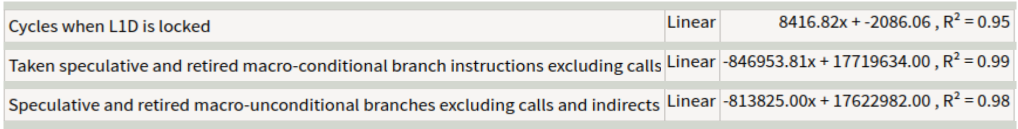


Figure 4.3: Parallel sorting. An array filled with pseudorandom numbers is sorted using GNU `libstdc++` parallel mode with a varying number of threads. As seen in the right diagram, which shows execution times in milliseconds with respect to the number of threads, the scaling behavior is different in the non-hyper-threaded and hyper-threaded cases. Generating the random numbers is included in the measurements because with EvSel, the whole program run is analyzed.



Cycles when L1D is locked	Linear	8416.82x + -2086.06 , $R^2 = 0.95$
Taken speculative and retired macro-conditional branch instructions excluding calls	Linear	-846953.81x + 17719634.00 , $R^2 = 0.99$
Speculative and retired macro-unconditional branches excluding calls and indirects	Linear	-813825.00x + 17622982.00 , $R^2 = 0.98$

Figure 4.4: Selected correlations from EvSel for the parallel sorting microbenchmark. Event types, regression function types, and the regression functions themselves are shown along with their coefficients of determination.

4.2 Program Run Phases: *Phasenprüfer*

Phasenprüfer automatically detects the transition between execution phases from a recorded program run, namely, ramp-up and computation. While recording, selected events are measured and presented to the user, separately aggregated for both phases.

4.2.1 Functionality and Limitations

Many programs undergo multiple execution phases [113]. The tool Phasenprüfer was built to help gain understandings about the ramp-up and the computation phase. Especially in the case of high-performance workloads on NUMA, a single node is typically generating or reading bigger amounts of data at first, which are then distributed and calculated later. This results in an asymmetry of the measurements for the individual nodes because in the ramp-up phase, all events originate from a single node, whereas in the computation phase, counters on all nodes contribute.

Observations during prior experiments showed that most of the events in the ramp-up phase are caused by I/O activity or memory redistribution among nodes. Consequently, in the case of Phasenprüfer, memory footprint (reserved memory, obtained trough `procfs`) is used to determine the phases.

Because the need of attributing different `perf` event measurements to the phases arose, Phasenprüfer was adapted to also record and analyze information for the two phases separately. Since statistical fluctuations and, often times, too few available samples cannot provide the same input data quality as the memory footprint, phase analysis itself is not yet feasible based on analyzing single or multiple `perf` events.

Currently, Phasenprüfer considers two phases only yet is easily extendable to recognize more phases in future work. In the example of BSP-like programs, where multiple supersteps could be analyzed, recognizing individual steps may be desireable.

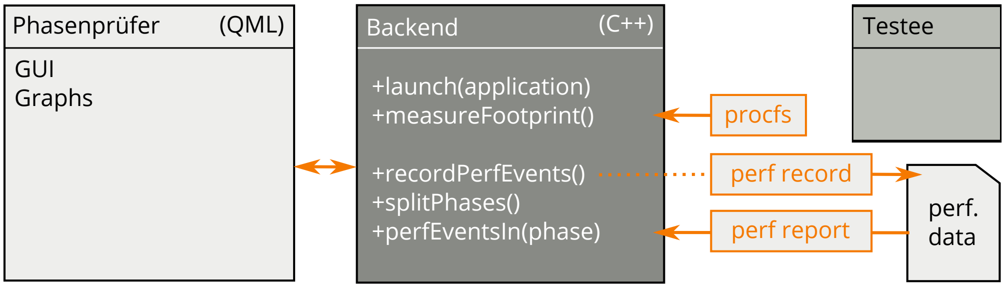


Figure 4.5: Architecture of Phasenprüfer. The frontend written in QML triggers functionalities in the backend (C++). First, the testee is launched. This triggers the recording of `perf` events, which are stored in a temporary file to be analyzed later. During the testee's runtime, memory consumption is measured and visualized. After the test, phases are determined. Finally, users can query the events for the separate phases.

Phasenprüfer is implemented using C++ and Qt's QML language [29]. The first part is a backend that provides the measurement and application launch functionalities. The second part is the graphical user interface, written in QML and communicating with the backend via Qt's signal and slot facilities. Because the phase split cannot be known beforehand, a temporary file records the events during program runtime. Later, the contents of this file can be queried for the individual phases. Architecture and data flows can be seen in Figure 4.5.

4.2.2 Automatic Phase Selection Through Regression

Phasenprüfer needs an automatic and fast way of splitting measurement data into multiple chunks to allow for interactive response times. As with the other tools, Phasenprüfer is designed not to intrude the operation of the program under test.

Instead, the two phases are determined purely based on observation. With the help of segmented regression, Phasenprüfer models the phases as functions and finds the phase transition. To achieve this, all data points are iteratively considered as phase

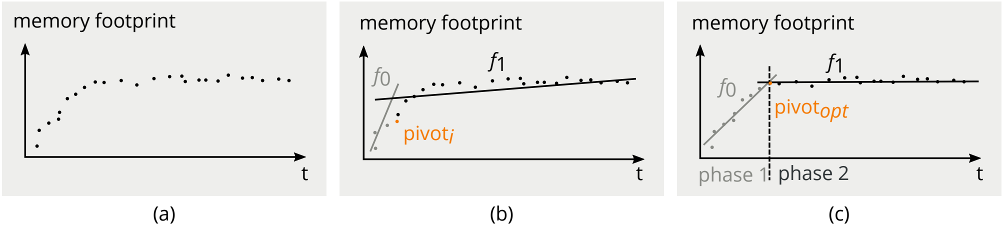


Figure 4.6: Determining phases. Based on the raw data (a), all data points are iteratively considered as pivot elements. Then, all possible combinations of two linear regression fits are tested. After selecting the lowest combined error of both functions that were found previously, the pivot element is selected as a phase transition.

transition points (pivots) first. Next, regression is performed before and after each pivot point. The phase transition is selected as the point where the summed error of both regressions is minimal. This method is depicted in Figure 4.6.

The regression itself is achieved using the linear least squares method with linear algebra. A short deduction of the used method can be found below [122]. For this purpose, the data is modeled as the overdetermined system of linear equations $y = X\beta$.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_0 & 1 \\ x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix}$$

The least squares method now finds the parameter $\hat{\beta}$ such that its squared difference to y (here called S) is minimal.

$$\begin{aligned} S &= \|y - X\beta\|^2 \\ &= (y - X\beta)^T(y - X\beta) \\ &= y^Ty - \underbrace{\beta^TX^Ty}_{\text{(a)}} - \underbrace{y^TX\beta}_{\text{(b)}} + \beta^TX^T\beta X \end{aligned}$$

See how (a) and (b) can be considered equal in this case:

$$(\beta^TX^Ty)^T = y^TX\beta$$

Both sides are scalar, since y and $X\beta$ both have the dimension of y. Scalars are equivalent to their transposes. Hence, the following (non-transposed) also holds:

$$\begin{aligned} \beta^TX^Ty &= y^TX\beta \\ \text{(a)} &= \text{(b)} \end{aligned}$$

To find the minimum of S, we can now minimize the function (with (a) equal to (b)):

$$S = y^Ty - 2\beta X^Ty + \beta^TX^TX\beta$$

We partially derive for β and set it to 0 for the initial criterion:

$$-X^Ty + (X^TX)\beta = 0$$

Since X consists of different x_i coordinates for the different points, the second partial derivation for β will never be 0, as it is positive definite (sufficient criterion):

$$(X^TX) \neq 0$$

From the second but last formula, the optimal parameters for $\hat{\beta}$ can be derived:

$$\hat{\beta} = (X^TX)^{-1}X^Ty \qquad \square$$

$\hat{\beta}$ now contains the parameters a and b of the best-fitting polynomial $y = ax + b$.

Since matrix operations for these small values can be done highly efficiently with the linear algebra library Eigen, the phases can be determined in milliseconds, even for thousands of data points [49]. More complex functions could be fitted by pretransforming the data (for instance, by applying natural logarithms beforehand). For the case of this particular discrimination, only linear memory serving capacity is assumed. This is because programs allocate memory with the maximum possible rate during the ramp-up phase (linearly increasing memory footprint) and commonly keep a relatively flat slope during the calculation phase [113]. Therefore, the need of a non-linear fitting function does not arise here.

4.2.3 Exemplary Results

This thesis showcases the start-up phase for the popular web browser Google Chrome. The typical application usage can be seen in Figure 4.7.

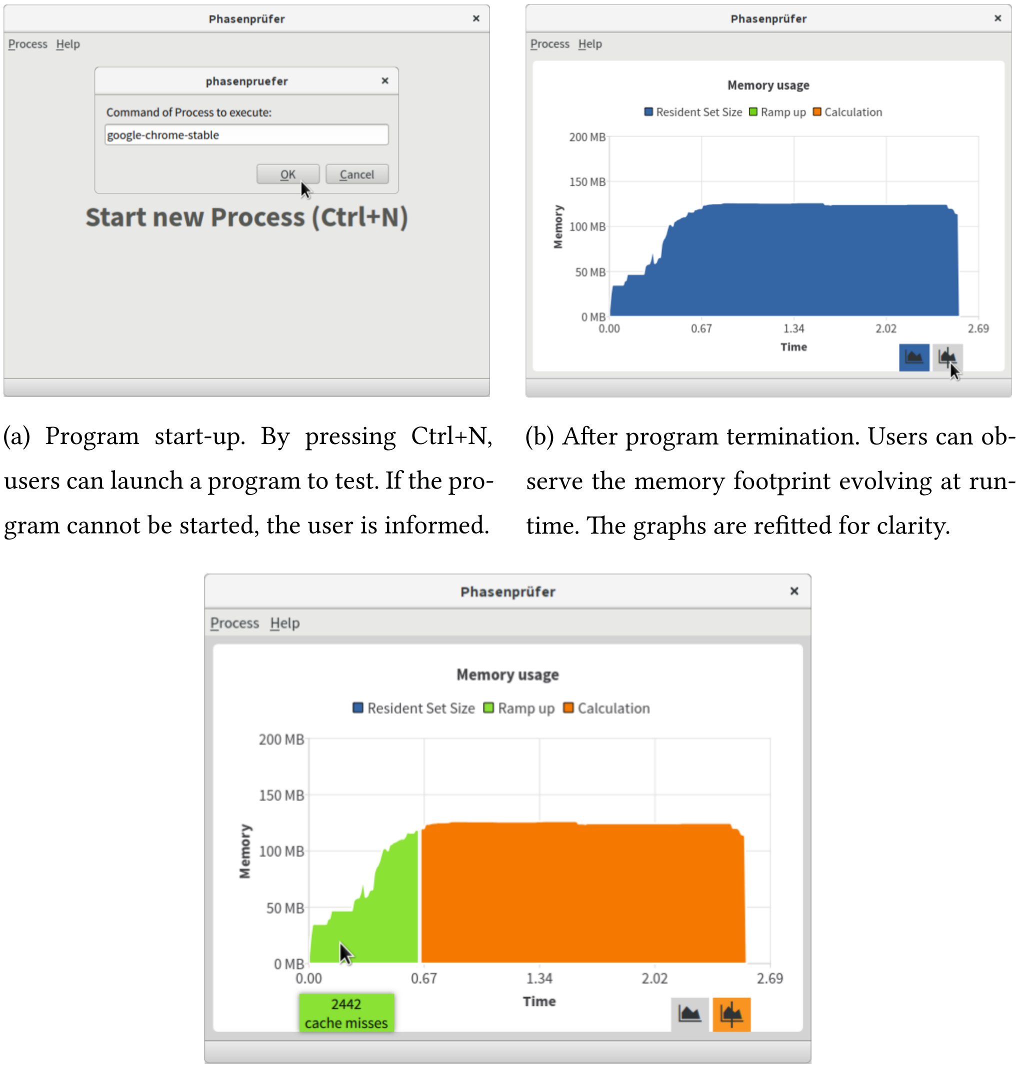
4.3 Latency Analysis: *Memhist*

To better characterize NUMA workloads, a third tool, named *Memhist*, was developed, which summarizes latency penalties of memory load operations in a histogram. The histogram shows how often events occurred, grouped by their latencies. Memhist is available for Intel platforms starting from Haswell. It makes use of the *precise load latency* feature of these processors, which can be used to determine whether a certain cycle threshold was surpassed for a memory load.

4.3.1 Histogram Construction

Events for cache levels, the TLB, and memory can be assumed to bear well-known costs [68, 37]. However, latencies in the range of NUMA remote accesses can vary widely. Memhist provides an estimation of the cost of remote accesses. When measuring exact latencies, even cache accesses spread around their expected peaks. This is caused by the costs of cache evictions and overall, undeterminable jitter in the process. On Intel systems, the so-called *use latency* is determined, which includes the pipeline queuing delays in addition to memory subsystem latency.

Creating the histogram is only possible through empirical sampling over longer time periods. This is due to several drawbacks of *load latency events*. First, as these need to



Command of Process to execute:
google-chrome-stable

(a) Program start-up. By pressing Ctrl+N, users can launch a program to test. If the program cannot be started, the user is informed.

(b) After program termination. Users can observe the memory footprint evolving at runtime. The graphs are refitted for clarity.

(c) Phase split. When clicking the *phase split* button in the bottom right, the phases are separated. By hovering over one of the colored phases, the previously recorded event information is revealed.

Figure 4.7: A typical user interaction with Phasenprüfer. In this example, the start-up behavior of Google Chrome is analyzed.

be measured as PEBS events (blocking the remaining count registers), only a single measurement can be performed at a time. Second, the load latency events denote all the loads that surpassed a threshold value. To retrieve event information for a specific latency interval, two measurements (lower and upper bound) have to be performed and subtracted. However, only one threshold measurement can be done at a time.

This poses the problem that time cycling has to be performed to cover a wider range of latencies. For this reason, negative event occurrences might be observed if the measurements for both bounds vary overly. This poses an error that cannot be avoided, although Memhist cycles with a frequency of 100 Hz (10 ms slices). As another drawback, Intel does not guarantee measurements of under three cycles to be correct. Thus, L1 cache hits cannot really be distinguished from register accesses. Because Memhist targets latencies in the realm of NUMA, which often require around 300 cycles and more, this is a minor issue.

The correctness of the latencies measured with Memhist was verified with the Intel Memory Latency Checker tool, `mlc`.

4.3.2 Implementation

Memhist is also implemented in C++ and the declarative language QML for its GUI. Users can choose latency interval sizes in a specified range. Memhist offers the option to either count events or multiply event occurrences with their respective latencies to gain insights on the amount of cycles spent in a certain latency interval.

With Memhist, latencies can be measured either on a local computer or on a remote system. Server platforms do not always provide all options for a rich graphical interface (Memhist requires OpenGL for its QML interface). Because of this, an additional headless probe was developed, which transfers the measured data via TCP to the GUI. This remote–local architecture is shown in Figure 4.8.

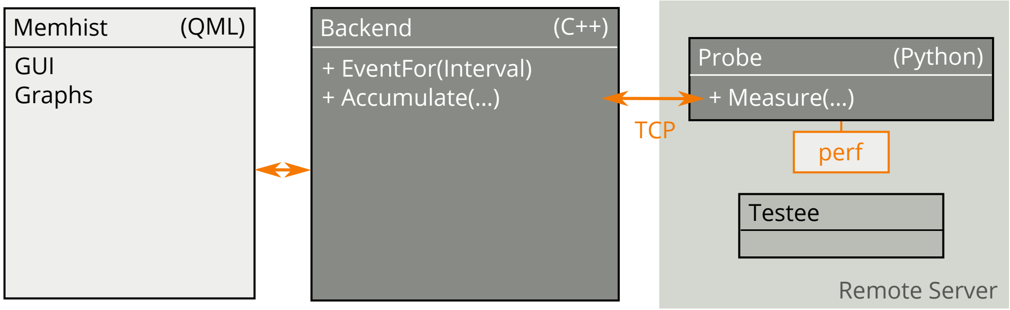


Figure 4.8: Memhist architecture for remote probing.

4.3.3 Exemplary Results

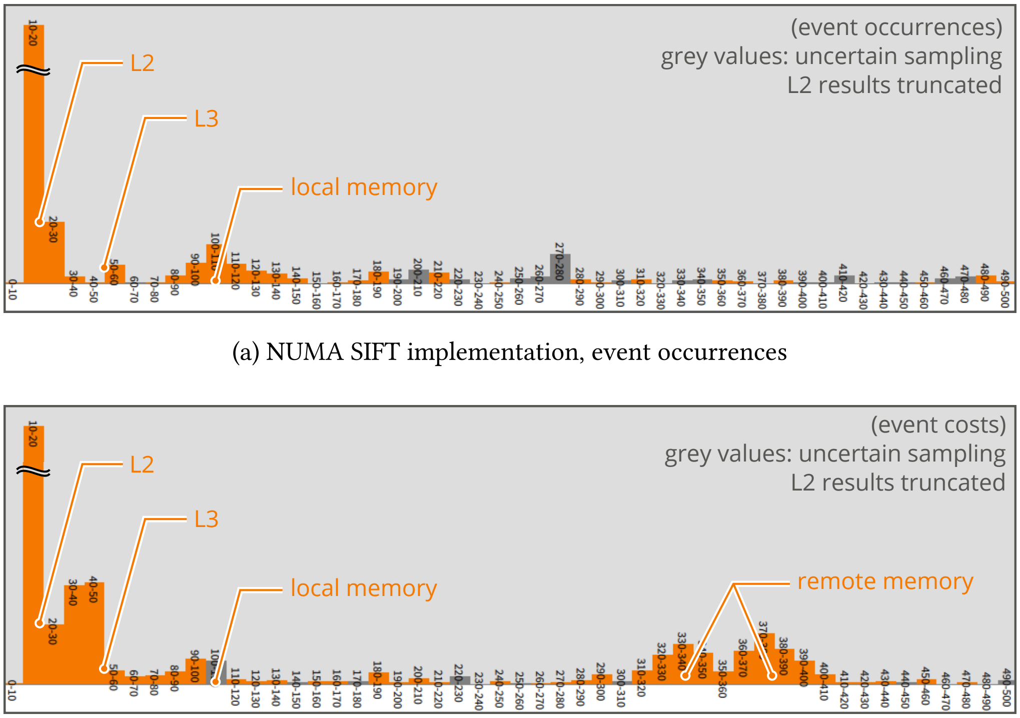
To verify the described measurement approach, this thesis presents results for a NUMA-optimized SIFT implementation [81], which acts almost entirely on local memory, as shown in Figure 4.9. This measurement, as well as all following measurements concerning NUMA, were performed on a four-node topology server.[4]

The peaks in the graphics match the results obtained with the Intel Memory Latency Checker [33]. Note that while `mlc` measures times in nanoseconds, Memhist measures cycles, which can be transformed to times in this case. This is especially easy, since the underlying hardware operated at a fixed frequency and TurboBoost was disabled.

In the second histogram, a bimodal distribution can be observed for the remote accesses. In comparison to `mlc`'s latency results, this cannot be explained with the phenomenon of remote accesses hitting or missing the remote LLC, as this would have much higher impacts (`mlc` suggests around 500 cycles). This thesis did not investigate the reasons for this observation further.

In future work, many more effects could be investigated, which can now be identified by Memhist: TLB miss costs, cache coherency protocol overheads, costs of remote memory accesses in more complex NUMA toloplogies, and so on.

[4] HPE ProLiant DL580 Gen9 Server, **4**×Intel Xeon 8890v3 @2.4 GHz, **4** × 32 GiB RAM @1600 MHz



(a) NUMA SIFT implementation, event occurrences

(b) Intel `mlc` remote latencies benchmark, event costs

Figure 4.9: Screenshots of Memhist's histogram. While aside from caches, main memory is primarily accessed in the first case, the costs of remote accesses can be observed in the second case. All intervals are denoted in cycles. L2 results are truncated to approximately half their height for readability.

If engineers are interested in profiling even higher latencies, tools like biolatency [61], which traces block device accesses using `perf`, could be used to determine latencies of file I/O requests, for example.

4.4 Limiting and Characterizing Interconnects: *Hydralisk*

Hydralisk is a tool that deploys bandwidth load on NUMA interconnects while simultaneously monitoring throughput. In this way, it provides an overview of the interconnect bandwidth state. Using Hydralisk, engineers can artificially create subtopologies of fully connected graphs by limiting links to the maximum possible extent.

This thesis leverages Hydralisk to conduct experiments on the already mentioned four-node Intel NUMA system available to the lab through the HPI Future SOC initiative.

4.4.1 Limiting Interconnects

Though it is possible to select the interconnect speed coarsely in modern Intel processors through the system's BIOS, influencing single QPI links independently cannot be achieved by these means. Further, there is no concept of priorities in the realm of hardware threads, so that assigning bandwidth shares by changing priorities is not an option. Overall, Intel QPI can be considered fair with respect to threads that demand bandwidth. In the measurements performed in this thesis (shown in Section 4.4.5), the difference between individual threads never surpassed 3 % of the reached bandwidth. The so-called *global queue*, which manages memory accesses arising from remote and local requests inside the CPU's offcore, has been reported to have a slight favor for local accesses, though [86].

Surely, engineers could change the scheduling priorities of the individual threads to reduce their bandwidth, yet this would also limit the CPU times of these threads, which should be considered independently for interconnect measurements.

Consequently, the most practical method to create bandwidth limits artificially is to launch dedicated threads to create load on the links. In the following, multiple of these options are presented and evaluated.

4.4.2 gland: Generating Load on an Interconnect

For generating bandwidth load on the system's interconnect, gland implements the *stream triad* benchmark [88]. Stream triad is essentially executing a multiply–add (crucial to many linear algebra algorithms) with the help of three arrays, a, b, and c. The arrays b and c reside on a remote node, allocated there using libnuma:

```
double *a = (double*) numa_alloc_onnode( N * sizeof(double), localNode );
double *b = (double*) numa_alloc_onnode( N * sizeof(double), remoteNode );
double *c = (double*) numa_alloc_onnode( N * sizeof(double), remoteNode );
[...]
for(long i = 0; i < N; i++)
{
        a[i] = b[i] + scalar * c[i];
}
```

Execution is pinned to the local node using libnuma. The scalar is chosen to be 3.0 for reasons of tradition and to avoid possible compiler optimizations.

Since in the particular case of gland, all arrays are filled with doubles, one run iteration of 2^{24} loop iterations amounts to exactly 256 MiB of data that need to be obtained from the remote arrays b and c. This amount is chosen to be at least twice the size of the overall L3 cache (45 MiB per node) to avoid caching effects on reiterations. Additionally, the increased batch size reduces jitter during the measurement. The value is not increased further, though, to not unnecessarily load the system with memory pressure during start-up and runtime.

Stream triad is suitable for bandwidth load simulations for several reasons [86]. First, caused by its regular data access behavior, the benchmark has a low and constant cache miss behavior of about 53 misses per 1000 instructions. Second, it scales well to multiple cores, since there are little to no intercore caching effects. Third, because the presented implementation designs triad to surpass the LLC sizes, data can be served through neither local nor remote LLC and thus has to be fetched with the QPI interconnect or from a remote memory controller. Researchers consider triad a so-called *cache gobbler* in terms of the program categories defined by Sandberg et al. [108].

In addition to stream triad, there is an alternative implementation using plain memcpy. Essentially, just two arrays are created and permanently copied from a to b in this case:

```
double *a = (double*) numa_alloc_onnode(N * sizeof(double), localNode);
double *b = (double*) numa_alloc_onnode(N * sizeof(double), remoteNode);
[...]
memcpy(a, b, N * sizeof(double));
```

The memcpy version does not reach the same raw throughput as the stream triad benchmark. While gland_memcpy falls down to libc's memcpy_avx_unaligned, which even enriches the code with prefetching information, it only achieves around 4.3 GiB/s.[5] The ordinary stream benchmark can make use of vectorized fused multiply–add instructions, and thus, it does not spend much time in computation but rather in operand fetching (backend-stalled). In this case, single instances reach 5.2 GiB/s. Appendix A presents a comparison of the hot-path assembly of triad and memcpy, which is also used by the Intel Memory Latency Checker. It is important to note that all memory has to be written beforehand as to circumvent symbolic memory management inside the operating system. For checking whether memory is finally committed and bound to a NUMA node, modern versions of the Linux kernel offer a specialized numa_maps file accessible through the proc filesystem:

```
        cat /proc/<pid>/numa_maps
```

There, the memory policies and current locations of all mapped pages can be reviewed.

[5] measured for gcc version 6.1.1, compiler flags: -O3

4.4.3 Managing Load

Hydralisk is built in C++ using the `ncurses` library [58] to visualize nodes and interconnects of the Intel-based test system as well as to allow for user interaction via keyboard inputs. Aside from `libnuma` for memory placement and `ncurses` for the interface, no additional library dependencies are needed. Hydralisk can be compiled from source on all platforms supporting a C++11 compiler.

In the current version, interconnect information is simply hard-coded for the four-node topology that is present on the test system. Later on, tools such as `hwloc` or `lstopo` can be used to identify topologies and adapt the system to different platforms. As a start, the tool can be used for at least any identical fully connected four-node topology.

Hydralisk starts up to 20 instances of the throughput generator `gland` on a each interconnect channel. The general architecture is depicted in Figure 4.10.

Hydralisk uses Unix pipes dispatched by `select` to transfer the troughputs measured within the `gland` instances. It is important to note that user interaction, which is mutiplexed by the same `select` call—namely on `stdin`'s file descriptor—is intercepted and handled separately as not to interfere with the communication of the currently measured throughputs. The system was tested with up to the maximum of 240 threads.

Hydralisk can be recompiled with *gland_memcopy* as a load generator. But since highest throughputs were achieved with the stream implementation, it became the default.

4.4.4 Exemplary Usage

A screenshot of Hydralisk's command-line interface can be seen in Figure 4.11. Engineers are able to select the individual interconnect channels and place up to 20 instances of `gland` load threads on them. As live feedback, the most recently measured bandwidths are presented to the user.

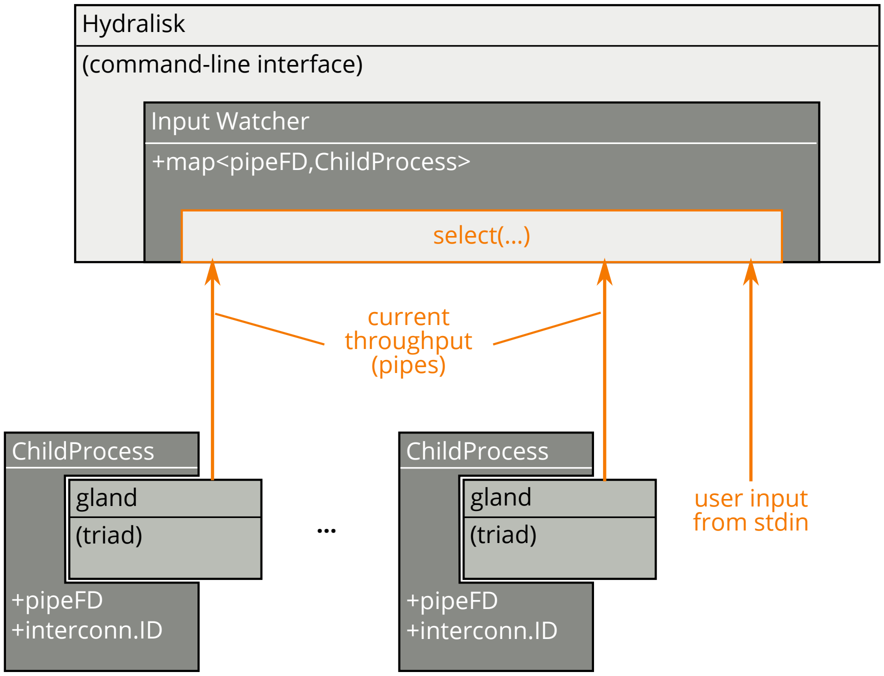
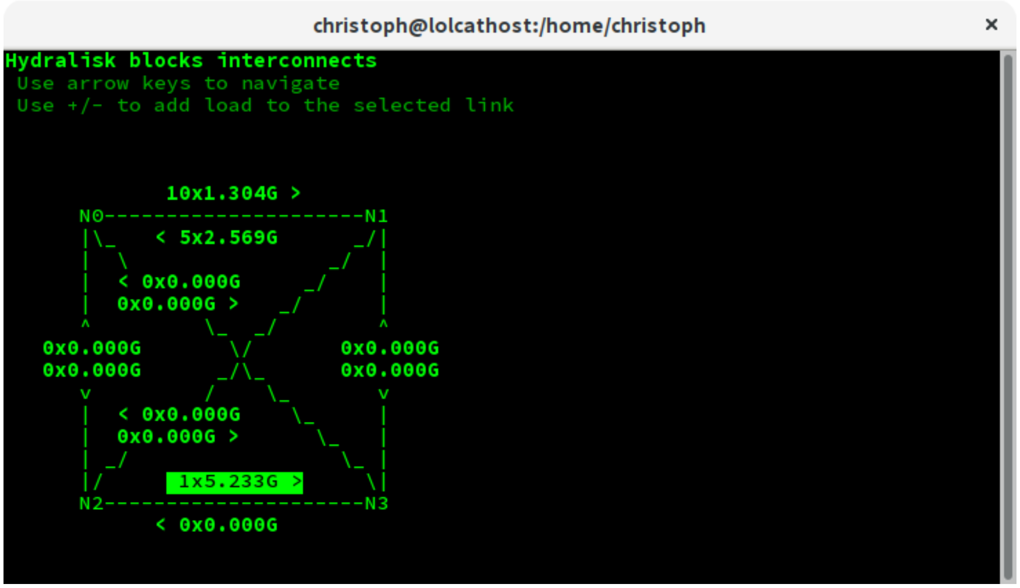


Figure 4.10: Hydralisk architecture. Single instances of the load-producing process `gland` are launched via the Hydralisk command-line interface. `ChildProcess` wrappers hold interconnect information and start the `gland` processes with the according parameters. The resulting throughput measurements obtained by the `gland` processes are communicated via pipes to the `InputWatcher`, multiplexed by using the Unix `select(…)` call. To reidentify `ChildProcesses`, a map translates the file descriptors of each pipe (obtained from `select`) back to `ChildProcess` instances. Finally, the transmitted throughput information is presented on Hydralisk's command-line interface.



```
Hydralisk blocks interconnects
 Use arrow keys to navigate
 Use +/- to add load to the selected link



             10x1.304G >
      N0---------------------N1
      |\_   < 5x2.569G      _/|
      |  \                _/  |
      |  < 0x0.000G     _/    |
      |  0x0.000G >   _/      |
      ^         \_  _/        ^
   0x0.000G       \/       0x0.000G
   0x0.000G      _/\_      0x0.000G
      v         /    \_       v
      |  < 0x0.000G    \_     |
      |  0x0.000G >      \_   |
      | _/                 \_ |
      |/      1x5.233G >     \|
      N2---------------------N3
            < 0x0.000G
```

Figure 4.11: Hydralisk. The command-line interface presents the topology of the test platform (limited to four-node configurations currently) to the user. Engineers are able to select each unidirectional QPI channel and add bandwidth load. In this case, the QPI link between node 0 and node 1 is loaded in a bidirectional fashion. The channel from node 2 to 3 is loaded with one instance of stream triad only, which does not suffice to reach the channel's capacity.

Hydralisk is meant to aid administrators and engineers at two points. First, it enables the simulation of asymmetric hardware topologies on symmetric hardware. As systems grow bigger, fully connected topologies change towards interconnect hierarchies and hypercube layouts. Here, Hydralisk can provide first insights to these locally asymmetric topologies and their interplay with software.

Second, Hydralisk is able to stress-test existing applications and operating systems regarding their behavior when links congest. In an exemplary scenario, database administrators operate a second application (for instance, backup jobs) in addition to their database processes. Hydralisk is able to simulate extra load and let administrators investigate the decline of the quality of service upon diminishing bandwidth availability.

In addition to the use cases envisioned above, Hydralisk can be used for fast investigations of link bandwidth availability. To this end, Hydralisk can also be seen as an educational tool to visualize the effects of link cross-talk and congestion.

4.4.5 Experiments and Results

With the functionality provided by Hydralisk, this thesis conducts several small experiments to characterize the QPI interconnect that is present on the test system. It practically provides insights concerning the reachable bandwidths as well as QPI cross-talk behavior. Hydralisk is also able to answer the question if adjacent nodes in an energy-saving state diminish the bandwidth performance.

At first, internode bandwidths are measured by starting several instances of the load generator `gland` on the same interconnect. The test platform consists of 9.6 GT/s (gigatransfers per second) QPI links, which should amount to 19.5 GB/s in overall performance. However, the Intel Memory Latency Checker, which is used for comparison, is only able to measure about 15.1 GB/s on the system (note the different units here; this equals 14.0 GiB/s). In the benchmark using Hydralisk with the stream-triad-based

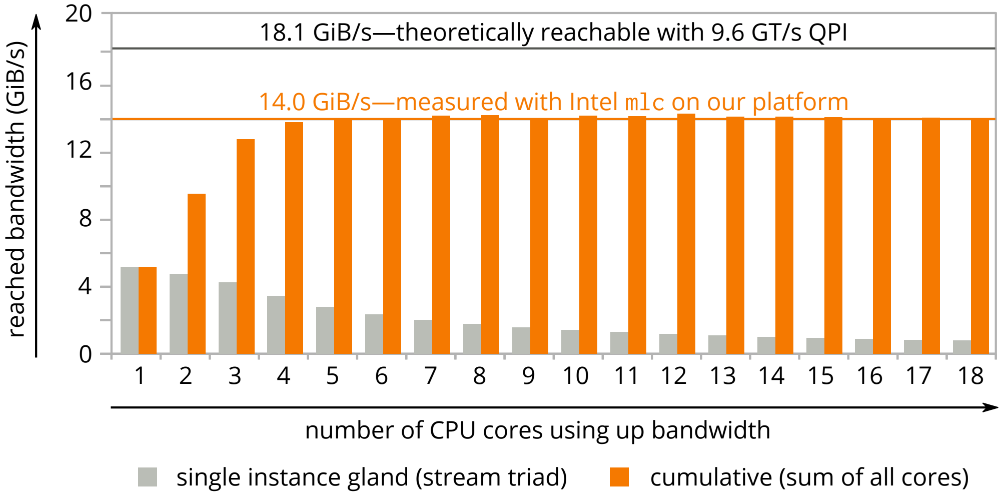


Figure 4.12: Single-channel QPI performance. One and two instances of `gland` are not able to saturate the QPI interconnect between two nodes fully. With more launched instances, the available bandwidth is shared equally (with differences between the cores smaller than 3 %). In this example, the theoretical limits of a 9.6 GT/s interconnect could not be reached. Still, the measurements resemble those of the vendor tool `mlc`.

`gland` load generator, Hydralisk reaches 14.3 GiB/s as a maximum bandwidth on the link, which resembles the results with Intel `mlc`. Results can be seen in Figure 4.12.

A second experiment was conducted to investigate whether energy-saving states of the receiving processors influence the bandwidth behavior. Upon specifying verbose output, `mlc` stated that it used spinloops to keep cores active. The results in Figure 4.13 reveal that this experiment cannot reproduce this mechanism on the target system.

As a last experiment leveraging Hydralisk, this thesis analyzes the duplex properties of the QPI links. Therefore, varying loads are placed on opposing nodes to generate cross-talk on the interconnect. While QPI is conceptionally full-duplex [91], we expect the results to contain diminished bandwidths with high cross-talk behavior, since the same L3 caches are used up when cross-talking. The results are visualized in Figure 4.14.

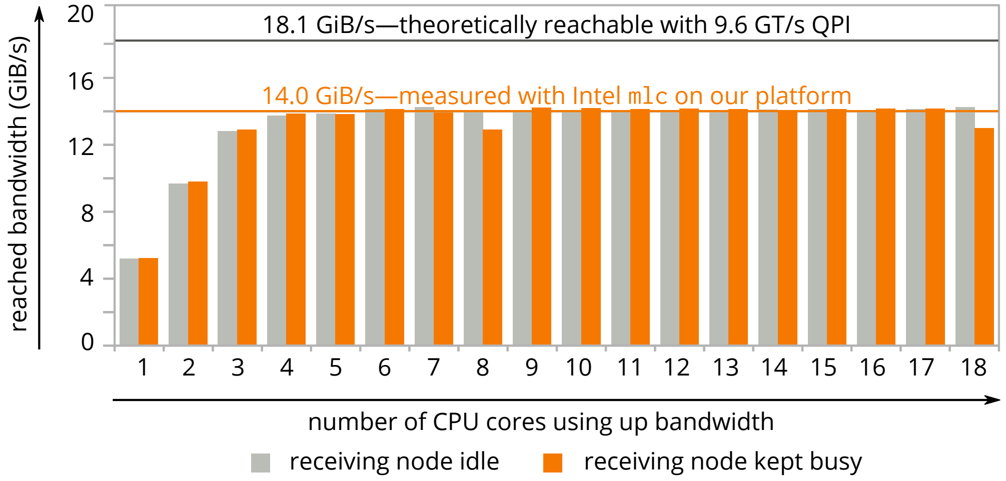


Figure 4.13: Idle and busy receivers. When measuring bandwidths, the adjacent receiving nodes were configured to be either idle (grey) or busy-waiting (orange). The one-shot data reveals that no wake-up effect can be seen in the busy cases. Conversely, some cases even showed reduced bandwidths when receivers were busy.

4.4.6 Creating Worse Scenarios, More Fine-Grained Limits

The following explains how worst-case scenarios could be generated. These might be used to stress-test an application's capabilities to react to transient load.

Caused by fair scheduling, creating worst-case scenarios is very hard on these kinds of systems. The worst thinkable scenario using the methods presented above would be to share the bandwidth among all 36 threads on the socket, effectively reducing one thread's share of the forward channel bandwidth to a 36th. When adding cross-talk, the lowest achievable bandwidth would be about 0.36 GiB/s. While this bandwidth would be the minimum by purely adding load, LLC interferences and increased access scheduling to the local queue would further hinder a real-world process.

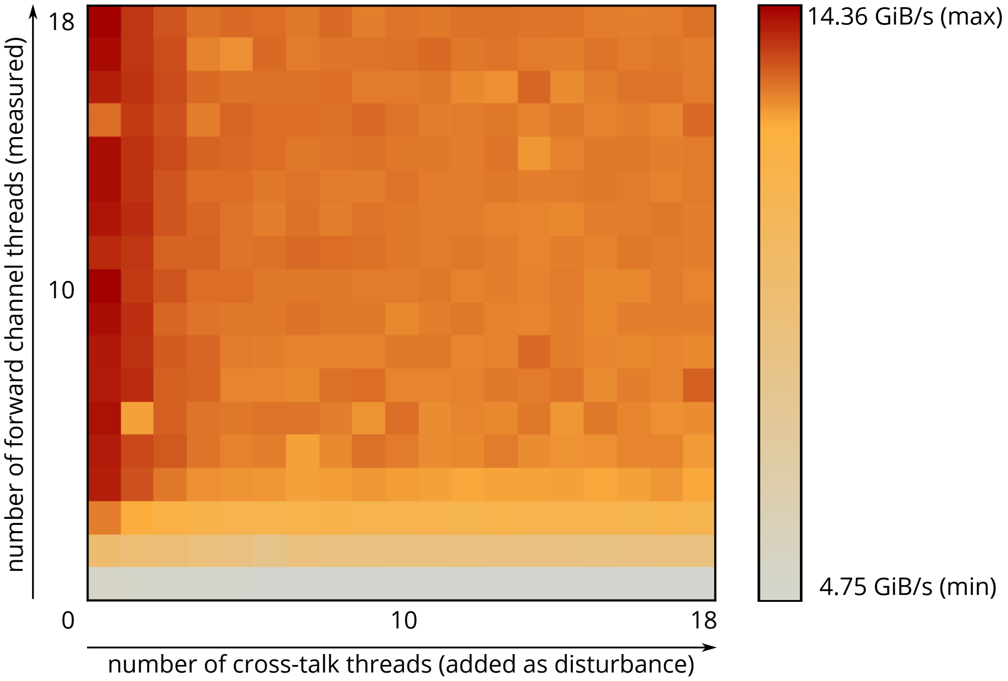


Figure 4.14: Cross-talk. As hypothesized, links behave best when there is no cross-talk. After reaching the bandwidth limit, the forward channel is able to transmit about 14.3 GiB/s in the non-cross-talk case. However, when launching an increasing number of cross-talking instances, the bandwidth diminishes gradually at first yet stays constantly reduced by about 1 GiB/s afterwards.

To achieve a more fine-grained limit, stream triad instances can be modified to include sporadic NOP phases between their load phases. It is important to note that inserting NOPs has to be performed in continuous intervals to prevent the CPU from performing out-of-order latency hiding, which would not reduce the overall bandwidth. This modification has not be investigated, though, and can be considered part of future work and improvements on Hydralisk. For further information on the interplay of caches and interconnects, this thesis refers to the work of Molka et al. [91].

5 Results and Conclusions

This last chapter outlines the contribution of this thesis and concludes the work done in the research field. It further presents high-level insights and finally comments on future work and prospects concerning the discussed topics.

Contributions

In the first part of this thesis, a top-down introduction and historical summary on performance modeling in the field of distributed computing was layed out. This thesis proposed and motivated the need for new performance models in recent and upcoming non-uniform memory access systems. The chapter identified both strengths and weaknesses concerning the applicability of current models and suggested proposals for extending and developing future NUMA models.

Low-level hardware counters, acting as the primary performance indicators, were identified as a potential interface to circumvent the more difficult code-to-cost models or to transfer program performance characteristics across machines. The document argued how creating such a model merely on empirical data is inadvisable, though. Instead, a two-step performance analysis strategy was proposed.

To support programmers in investigating performance issues, the second part of this thesis introduced readers to the technical concepts of hardware performance counting for the most popular modern hardware platforms, additionally providing multiple examples. After explaining the hardware counter sources and register formats, existing tools were described that aggregate this raw information to high-level insights. Linux `perf`, serving as the base for most of the tools developed in this thesis, was presented with most detail in this regard.

As a third part, novel tools based on these low-level hardware counters were proposed. The tool *EvSel* helps engineers in exploring the details of raw hardware counters, most of which are not listed by `perf`. EvSel can measure all counters (currently for Intel hardware) and presents the visually enhanced results to the user. Programmers are able to compare pairs of program execution runs or even parameter series statistically. To gain confidence about measurement runs, t-test significances and the regressions' coefficients of determination are displayed.

Because this thesis proposed to consider temporal phases when creating cost models, a deterministic way of splitting a program execution into individual periods had to be found. The presented tool *Phasenprüfer* was proposed to address this issue. To perform and attribute measurements to the identified phases, the memory footprint is used to temporally separate the execution phases.

As memory access cost is crucial for performance considerations nowadays, measuring individual accesses can reveal much information about program performance. On modern Intel CPUs, this can be achieved with so-called *load-latency-enabled* events. *Memhist* uses these low-level counters to allow programmers to obtain a rough understanding of their program's memory access behavior through a cost histogram. In this way, Memhist is able to capture the behavior of programs accessing hierarchical and heterogeneously distributed memory.

As a core component of NUMA systems, the interconnect plays an increasing role when determining performance cost caused by remote accesses. Current related work measures interconnect properties (such as the bandwidth) for exclusively executed programs or for simplistic topologies like fully connected graphs. To generate and explore more complex cases, the presented load generator *Hydralisk* is able to limit individual interconnect bandwidths by deploying load on them. This enables researches to artificially create topologies (bandwidth-wise) that do not resemble the single-cost toplogies often found nowadays.

Challenges—A Personal View

During the course of this thesis and while investigating performance through measurements, the author identified four major high-level challenges:

Mechanism Programmers need to know from where a performance degradation originates inside the computing system or programming framework and how to reproduce the circumstances leading to the issue.

Measurement To validate the effectiveness of changes to a program, measurements need to be employed. This opens the second challenge of which indicators are available for measurement and how to measure them.

Relevance The plenitude of measurement possibilities needs to be filtered for applicability in the specific case. This third challenge is about which performance indicators correlate to a specific program change.

Trust As a fourth challenge, programmers are interested in verifying their findings against the statistical fluctuations or hidden mechanisms that blur their results. The challenge of trust in the measurement came up when working on this thesis and proved to be of significant importance.

This work aimed at addressing all these challenges through the contributions of this thesis and the developed tools.

Future Work and Prospects

In future work, the proposals stated for NUMA models could be set into practice and evaluated. Therefore, a method for simulating accesses to the topology with its denoted bandwidth and latency information needs to be developed. With typical benchmarks, the advantages of a detailed interconnect description such as Linpack might be revealed in this case. Especially simulating and incorporating different topologies is important to investigate further when dealing with large-scale systems, such as SGI UV systems with their complex and dedicated interconnects [40].

The programs developed in this thesis can be generalized to support more hardware platforms. Additionally, Hydralisk could be adapted to topologies other than the typical four-node configurations. Especially for the case of IBM, the PMU counter documentation should be refined and investigated in more depth as to port the existing tools to the Power8 platform. This would also require changing the event codes for Linux-based systems, and since perf does not exist on IBM's AIX, a suitable measurement layer would be needed there.

The mapping from events to lines of code was not covered in much detail in this thesis, yet this is important to developers when searching for performance bottlenecks in their programs. Thus, tools incorporating the program code need to be developed using *event-based sampling* and *last branching record* information.

The future is expected to bring more heterogeneity both in execution as well as in memory hierarchies [112]. Performance models and predictions based on program characteristics will decide upon scheduling a task to the best-fitting processor or relocating its data to the optimal location in the topology.

Although memory bandwidths might increase, since stacked memory and non-volatile technologies become more popular, the gap between CPU and memory speeds will remain, which is simply due to the high production costs of cache memory compared to DRAM. This causes memory loads to still remain interesting for investigation, especially for intercommunication when scaling up problems.

Finally, CPU vendors include increasingly more PMU information in the realms of instruction and branch tracking, which eases the attributability of events to code. In the future, processors can therefore be expected to communicate their behavior or even identified performance bottlenecks more directly to the programmers.

Bibliography

[1] Inc. Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 60h-6Fh Processors*. 2016.

[2] Neha Agarwal et al. "Page placement strategies for GPUs within heterogeneous memory systems". In: *SIGPLAN Notices*. Vol. 50. 4. 2015, pp. 607–618.

[3] Alok Aggarwal, Ashok K Chandra, and Marc Snir. "Communication complexity of PRAMs". In: *Theoretical Computer Science* 71.1 (1990), pp. 3–28.

[4] Ashwin Mandayam Aji et al. "Automatic Command Queue Scheduling for Task-Parallel Workloads in OpenCL". In: *International Conference on Cluster Computing*. 2015, pp. 42–51.

[5] Bowen Alpern et al. "The uniform memory hierarchy model of computation". In: *Algorithmica* 12.2-3 (1994), pp. 72–109.

[6] Francesc Alted. "Why modern CPUs are starving and what can be done about it". In: *Computing in Science & Engineering* 12.2 (2010), pp. 68–71.

[7] ARM architectures. *ARMv8 A Architecture Reference Manual*. 2013.

[8] Richard A Armstrong. "When to use the Bonferroni correction". In: *Ophthalmic and Physiological Optics* 34.5 (2014), pp. 502–508.

[9] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[10] Peter Auer et al. "On the complexity of function learning". In: *Machine Learning* 18.2-3 (1995), pp. 187–230.

[11] Richard Bellman and Robert Kalaba. "On the role of dynamic programming in statistical communication theory". In: *IRE Transactions on Information Theory* 3.3 (1957), pp. 197–203.

[12] Stefan Berchtold et al. "A cost model for nearest neighbor search in a high-dimensional data space". In: *Proceedings of the 16th SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1997, pp. 78–86.

[13] E Ender Bilir et al. "Multicast snooping: a new coherence method using a multicast address network". In: *SIGARCH Computer Architecture News* 27.2 (1999), pp. 294–304.

[14] William Lloyd Bircher and Lizy K John. “Complete system power estimation using processor performance events”. In: *IEEE Transactions on Computers* 61.4 (2012), pp. 563–577.

[15] William Lloyd Bircher et al. “Runtime identification of microprocessor energy saving opportunities”. In: *ISLPED’05. Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005.* IEEE. 2005, pp. 275–280.

[16] Robert D Blumofe and Charles E Leiserson. “Scheduling multithreaded computations by work stealing”. In: *Journal of the ACM* 46.5 (1999), pp. 720–748.

[17] William J Bolosky et al. “NUMA policies and their relation to memory architecture”. In: *SIGARCH Computer Architecture News.* Vol. 19. 2, pp. 212–221.

[18] Carl Friedrich Bolz et al. “Tracing the meta-level: PyPy’s tracing JIT compiler”. In: *Proceedings of the 4th workshop on Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems.* 2009, pp. 18–25.

[19] Ryan Braithwaite, Patrick McCormick, and Wu-chun Feng. “Empirical memory-access cost models in multicore numa architectures”. In: *Virginia Tech Department of Computer Science* (2011).

[20] Martin Burtscher et al. “Perfexpert: An easy-to-use performance diagnosis tool for hpc applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 2010, pp. 1–11.

[21] Surendra Byna et al. “Predicting memory-access cost based on data-access patterns”. In: *International Conference on Cluster Computing.* 2004, pp. 327–336.

[22] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” In: *OSDI.* Vol. 8. 2008, pp. 209–224.

[23] JJ Cahill et al. “IBM Power Systems built with the POWER8 architecture and processors”. In: *IBM Journal of Research and Development* 59.1 (2015), pp. 4–1.

[24] Kirk W Cameron and Xian-He Sun. “Quantifying locality effect in data access delay: memory logP”. In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International.* IEEE. 2003, 8–pp.

[25] Duncan KG Campbell. “A survey of models of parallel computation”. In: *Report-University Of York Department Of Computer Science YCS* (1997).

[26] Pierre Cassier et al. *Effective zSeries Performance Monitoring Using Resource Measurement Facility.* IBM, 2005.

[27] Dehao Chen et al. “Taming hardware event samples for FDO compilation”. In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization.* ACM. 2010, pp. 42–52.

[28] Richard Cole and Ofer Zajicek. “The APRAM: Incorporating asynchrony into the PRAM model”. In: *Proceedings of the 1st annual symposium on Parallel algorithms and architectures.* 1989, pp. 169–178.

[29] The Qt Company. *Qt Quick: CSS and JavaScript like declarative language with a cross-platform IDE.* URL: https://www.qt.io/qt-quick/.

[30] J Conway. *Unpredictable iterations.* 1972.

[31] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the 3rd symposium on Theory of computing.* 1971, pp. 151–158.

[32] Intel Cooperation. *MSR-Tools.* URL: https://01.org/msr-tools.

[33] Intel Cooporation. *Intel Memory Latency Checker Website.* URL: https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[34] Intel Cooporation. *Intel VTune Amplifier XE.* URL: https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[35] Intel Cooporation. *Running Average Power Limit – RAPL.* URL: https://01.org/blogs/tlcounts/2014/running-average-power-limit-%E2%80%93-rapl.

[36] Microsoft Cooporation. *Windows Performance Toolkit Technical Reference.* URL: https://msdn.microsoft.com/en-us/library/windows/hardware/hh162945.aspx.

[37] Intel Coorporation. *Intel 64 and IA-32 optimization reference manual.* 2016.

[38] Intel Coorporation. *Intel Open Source Technology Center.* URL: https://download.01.org/perfmon/.

[39] Thomas H Cormen and Michael T Goodrich. "A bridging model for parallel computation, communication, and I/O". In: *ACM Computing Surveys (CSUR)* 28.4es (1996), p. 208.

[40] Silicon Graphics International Corp. *SGI UV—The World's Most Powerful In-Memory Supercomputers.* URL: https://www.sgi.com/products/servers/uv/.

[41] Intel Corporation. *Intel Xeon Phi Coprocessor Performance Monitoring Units.* 2012. URL: https://software.intel.com/sites/default/files/forum/278102/intelr-xeon-phitm-pmu-rev1.01.pdf.

[42] NVIDIA Corporation. *The PAPI CUDA Component.* URL: https://developer.nvidia.com/papi-cuda-component.

[43] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM. 1977, pp. 238–252.

[44] David Culler et al. "LogP: Towards a realistic model of parallel computation". In: *Proceedings of the 4th SIGPLAN Symposium on Principles & Practice of Parallel Programming.* 1993, pp. 1–12.

[45] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[46] Frank Denneman. *Memory Deep Dive Series.* URL: http://frankdenneman.nl/2015/02/18/memory-configuration-scalability-blog-series/.

[47] Paul J Drongowski, AMD CodeAnalyst Team, and Boston Design Center. "An introduction to analysis and optimization with AMD CodeAnalyst Performance Analyzer". In: *Advanced Micro Devices, Inc* (2008).

[48] Paul Drongowski et al. "Incorporating instruction-based sampling into AMD CodeAnalyst". In: *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 119–120.

[49] *Eigen: A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms*. URL: http://eigen.tuxfamily.org.

[50] Christopher M Farrar et al. *Database query cost model optimizer*. US Patent 6,330,552. Dec. 2001.

[51] Faith E Fich, Prabhakar Ragde, and Avi Wigderson. "Relations between concurrent-write models of parallel computation". In: *SIAM Journal on Computing* 17.3 (1988), pp. 606–627.

[52] Martti Forsell. "A PRAM-NUMA model of computation for addressing low-TLP workloads". In: *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–8.

[53] Michael Frasca. "Model-driven memory optimizations for high performance computing: From caches to I/O". PhD thesis. 2012.

[54] Georges Gardarin, Jean-Robert Gruser, and Zhao-Hui Tang. "A cost model for clustered object-oriented databases". In: *Proceedings of the 21th International Conference on Very Large Data Bases*. 1995, pp. 323–334.

[55] Andy German. "Software static code analysis lessons learned". In: *Crosstalk* 16.11 (2003), pp. 19–22.

[56] Phillip B Gibbons. "A more practical PRAM model". In: *Proceedings of the 1st annual symposium on Parallel algorithms and architectures*. 1989, pp. 158–168.

[57] The GNU Project. *Auto-Vectorization in GCC*. URL: http://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[58] Free Software Foundation Inc. GNU. *Announcing ncurses 6.0*. URL: https://www.gnu.org/software/ncurses/ncurses.html.

[59] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *Sigplan Notices*. Vol. 40. 6. 2005, pp. 213–223.

[60] Juan Gonzalez, Judit Gimenez, and Jesus Labarta. "Performance data extrapolation in parallel codes". In: *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. IEEE. 2010, pp. 155–163.

[61] Brendan Gregg. *BCC: Dynamic Tracing Tools for Linux*. URL: https://iovisor.github.io/bcc/.

[62] Jose A. Gregorio, Ramon Beivide, and Fernando Vallejo. "Modeling of interconnection subsystems for massively parallel computers". In: *Performance Evaluation* 47.2 (2002), pp. 105–129.

[63] Todd Heywood and Sanjay Ranka. "A practical hierarchical model of parallel computation". In: *Journal of Parallel and Distributed Computing* 16.3 (1992), pp. 212–232.

[64] John E Hopcroft. *Introduction to automata theory, languages, and computation.* Pearson Education India, 1979.

[65] Kenneth Hoste and Lieven Eeckhout. "Cole: compiler optimization level exploration". In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization.* ACM. 2008, pp. 165–174.

[66] Jason Hsu. *Multiple comparisons: theory and methods.* CRC Press, 1996.

[67] Mohammad Hussein, Franck Morvan, and Abdelkader Hameurlain. "Embedded cost model in mobile agents for large scale query optimization". In: *The 4th Symposium on Parallel and Distributed Computing.* 2005, pp. 199–206.

[68] Advanced Micro Devices Inc. *Advanced Micro Devices Software Optimization Guide for AMD Family 15h Processors.* 2014.

[69] Hiroshi Inoue and Toshio Nakatani. "How a Java VM can get more from a hardware performance monitor". In: *SIGPLAN Notices.* Vol. 44. 10. 2009, pp. 137–154.

[70] Intel. *Intel® 64 and IA-32 Architectures Software Developer Manuals.* 2016.

[71] Matevz Jekovec and Andrej Brodnik. *Survey of the sequential and parallel models of computation.* 2012.

[72] Brian Johnson, Craig Skibo, and Marc Young. *Inside Visual Studio. NET.* 2003.

[73] Ian Jolliffe. *Principal component analysis.* Wiley Online Library, 2002.

[74] Richard M. Karp and Vijaya Ramachandran. "A survey of parallel algorithms for shared-memory machines". In: (1988).

[75] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[76] Andi Kleen. "A numa api for linux". In: *Novell Inc* (2005).

[77] Andi Kleen. *PMU Tools.* URL: https://github.com/andikleen/pmu-tools.

[78] Donald Ervin Knuth. *The Art of Computer Programming: Seminumerical Algorithms. II.* Addison-Wesley, 1969.

[79] Christoph Lameter. "Numa (non-uniform memory access): An overview". In: *Queue* 11.7 (2013), p. 40.

[80] Chris Lattner. "LLVM and Clang: Next generation compiler technology". In: *The BSD Conference.* 2008, pp. 1–2.

[81] David G Lowe. "Object recognition from local scale-invariant features". In: *Computer vision, 1999. The proceedings of the seventh IEEE international conference on.* Vol. 2. Ieee. 1999, pp. 1150–1157.

[82] Jochen Ludewig. "Models in software engineering–an introduction". In: *Software and Systems Modeling* 2.1 (2003), pp. 5–14.

[83] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping”. In: *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE. 2009, pp. 45–55.

[84] Lin Ma, Kunal Agrawal, and Roger D Chamberlain. “A memory access model for highly-threaded many-core architectures”. In: *Future Generation Computer Systems* 30 (2014), pp. 202–215.

[85] Bruce M Maggs, Lesley R Matheson, and Robert Endre Tarjan. “Models of parallel computation: A survey and synthesis”. In: *Proceedings of the 28th International Conference on System Sciences*. Vol. 2. 1995, pp. 61–70.

[86] Zoltán Majó. “Modeling memory system performance of NUMA multicore-multiprocessors”. PhD thesis. 2014.

[87] Richard E Matick, Thomas J Heller, and Michael Ignatowski. “Analytical analysis of finite cache penalty and cycles per instruction of a multiprocessor memory hierarchy using miss rates and queuing theory”. In: *IBM Journal Of Research And Development* 45.6 (2001), pp. 819–842.

[88] John D McCalpin. “A survey of memory bandwidth and machine balance in current high performance computers”. In: *TCCA Newsletter* (1995), pp. 19–25.

[89] Mehmet Mendeş, Erkut Akkartal, et al. “Comparison of ANOVA F and WELCH tests with their respective permutation versions in terms of type I error rates and test power”. In: *Kafkas Univ Vet Fak Derg* 16.5 (2010), pp. 711–716.

[90] Matthew C Merten et al. “An architectural framework for runtime optimization”. In: *IEEE transactions on Computers* 50.6 (2001), pp. 567–589.

[91] Daniel Molka et al. “Memory performance and cache coherency effects on an intel nehalem multiprocessor system”. In: *18th International Conference on Parallel Architectures and Compilation Techniques*. 2009, pp. 261–270.

[92] Philip J Mucci et al. “PAPI: A portable interface to hardware performance counters”. In: *Proceedings of the department of defense HPCMP users group conference*. 1999, pp. 7–10.

[93] JM Nash. “A study of the XPRAM Model for Parallel Computing”. PhD thesis. PhD thesis, School of Computer Studies, University of Leeds, 1993.

[94] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *SIGPLAN notices*. Vol. 42. 6. 2007, pp. 89–100.

[95] Ethiopia Enideg Nigussie et al. “Exploration and design of high performance variation tolerant on-chip interconnects”. PhD thesis. 2010.

[96] Diego Novillo. “SamplePGO-The Power of Profile Guided Optimizations without the Usability Burden”. In: *LLVM Compiler Infrastructure in HPC (LLVM-HPC), 2014*. IEEE. 2014, pp. 22–28.

[97] Andrzej Nowak, David Levinthal, and Willy Zwaenepoel. "Hierarchical cycle accounting: a new method for application performance tuning". In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE. 2015, pp. 112–123.

[98] Andrzej Nowak et al. "Establishing a base of trust with performance counters for enterprise workloads". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 541–548.

[99] *PAPI Supported Platforms*. URL: http://icl.cs.utk.edu/papi/custom/index.html?lid=62&slid=96.

[100] Stefan Poledna. *Fault-tolerant real-time systems: The problem of replica determinism*. Vol. 345. Springer Science & Business Media, 1996.

[101] Vijaya Ramachandran. "QSM: A general purpose shared-memory model for parallel computation". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1997, pp. 1–5.

[102] John Regehr. *Souper: a superoptimizer for LLVM IR*. URL: http://github.com/google/souper.

[103] William John Reichmann. *Use and abuse of statistics*. 1964.

[104] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[105] Ronald Rivest. *The MD5 message-digest algorithm*. 1992.

[106] Lawrence Berkeley National Laboratory Roberto A. Vitillo. *Performance Tools Developments*. 2011.

[107] Satish Kumar Sadasivam. *Presentation: POWER8 Performance Analysis - OpenPOWER*. URL: http://openpowerfoundation.org/wp-content/uploads/2015/03/Sadasivam-Satish_OPFS2015_IBM_031615_final.pdf.

[108] Andreas Sandberg, David Eklöv, and Erik Hagersten. "Reducing cache pollution through detection and elimination of non-temporal memory accesses". In: *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 2010, pp. 1–11.

[109] John E Savage. *Models of computation—exploring the power of computing*. 1998.

[110] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In: *SIGARCH Computer Architecture News*. Vol. 41. 1. 2013, pp. 305–316.

[111] Martin Schmollinger and Michael Kaufmann. "κnuma: A model for clusters of smp-machines". In: *Parallel Processing & Applied Mathematics*. 2001, pp. 42–50.

[112] Amar Shan. *Heterogeneous Processing: a Strategy for Augmenting Moore's Law*. 2006. URL: https://www.linuxjournal.com/article/8368.

[113] Timothy Sherwood et al. "Discovering and exploiting program phases". In: *IEEE micro* 23.6 (2003), pp. 84–93.

[114] David B Skillicorn. “Models for practical parallel computation”. In: *International Journal of Parallel Programming* 20.2 (1991), pp. 133–158.

[115] David B Skillicorn and Wentong Cai. “A cost calculus for parallel functional programming”. In: *Journal of Parallel and Distributed Computing* 28.1 (1995), pp. 65–83.

[116] David B Skillicorn and Domenico Talia. “Models and languages for parallel computation”. In: *ACM Computing Surveys (CSUR)* 30.2 (1998), pp. 123–169.

[117] Marc Snir. *MPI–the Complete Reference: The MPI core*. Vol. 1. MIT press, 1998.

[118] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. “Maestro: data orchestration and tuning for OpenCL devices”. In: *Euro-Par*. 2010, pp. 275–286.

[119] Robert Springer et al. “Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster”. In: *Proceedings of the 11th symposium on Principles and practice of parallel programming*. 2006, pp. 230–238.

[120] Herbert Stachowiak. *Allgemeine Modelltheorie*. 1973.

[121] John E Stone, David Gohara, and Guochun Shi. “OpenCL: A parallel programming standard for heterogeneous computing systems”. In: *Computing in science & engineering* 12.1-3 (2010), pp. 66–73.

[122] Gilbert Strang and Wellesley-Cambridge Press. *Introduction to linear algebra*. Vol. 3. Wellesley-Cambridge Press Wellesley, MA, 1993.

[123] Student. “The probable error of a mean”. In: *Biometrika* (1908), pp. 1–25.

[124] Aater Suleman. *Clarifying Throughput vs. Latency*. URL: http://www.futurechips.org/thoughts-for-researchers/clarifying-throughput-vs-latency.html.

[125] Herb Sutter. “The free lunch is over: A fundamental turn toward concurrency in software”. In: *Dr. Dobb’s journal* 30.3 (2005), pp. 202–210.

[126] IBM Systems and Technology Group. *Power ISA- POWER Instruction Set Architecture (ISA) Specification*. 2015. URL: http://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf.

[127] Jie Tao, Wolfgang Karl, and Martin Schulz. “Memory access behavior analysis of NUMA-based shared memory programs”. In: *Scientific Programming* 10.1 (2002), pp. 45–53.

[128] *The Clang Vectorizer Facility*. URL: http://llvm.org/docs/Vectorizers.html.

[129] *The Linux Sysfs API*. URL: https://www.kernel.org/doc/Documentation/thermal/sysfs-api.txt.

[130] Alexandre Tiskin. “The bulk-synchronous parallel random access machine”. In: *Theoretical Computer Science* 196.1 (1998), pp. 109–130.

[131] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *J. of Math* 58.345-363 (1936), p. 5.

[132] Leslie G Valiant. *Bulk-synchronous parallel computers*. 1989.

[133] John Von Neumann. “First Draft of a Report on the EDVAC”. In: *IEEE Annals of the History of Computing* 4 (1993), pp. 27–75.

[134] Vincent M Weaver. “Linux perf_event features and overhead”. In: *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*. 2013, p. 80.

[135] Vincent M Weaver et al. “Measuring energy and power with PAPI”. In: *41st International Conference on Parallel Processing Workshops*. 2012, pp. 262–268.

[136] Paul R Wilson et al. “Dynamic storage allocation: A survey and critical review”. In: *Memory Management*. Springer, 1995, pp. 1–116.

[137] Christopher Winship and Robert D Mare. “Models for sample selection bias”. In: *Annual review of sociology* (1992), pp. 327–350.

[138] Milian Wolff. *Heaptrack - A Heap Memory Profiler for Linux*. URL: http://milianw.de/blog/heaptrack-a-heap-memory-profiler-for-linux.

[139] Qiang Wu et al. “Exposing memory access regularities using object-relative memory profiling”. In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 315–323.

[140] Xiaodong Zhang and Xiaohan Qin. “Performance prediction and evaluation of parallel processing on a NUMA multiprocessor”. In: *Software Engineering, IEEE Transactions on* 17.10 (1991), pp. 1059–1068.

[141] Xiaodong Zhang and Yong Yan. “Comparative modeling and evaluation of CC-NUMA and COMA on hierarchical ring architectures”. In: *IEEE transactions on parallel and distributed systems* 6.12 (1995), pp. 1316–1331.

[142] Yunquan Zhang et al. “Models of parallel computation: a survey and classification”. In: *Frontiers of Computer Science in China* 1.2 (2007), pp. 156–165.

A Appendix

perf_event_open Counting Setup

```
#include <unistd.h>
#include <string.h>
#include <sys/ioctl.h>
#include <linux/perf_event.h>
#include <asm/unistd.h>
static long perf_event_open(struct perf_event_attr *hw_event, pid_t pid,
                            int cpu, int group_fd, unsigned long flags)
{
   return syscall(__NR_perf_event_open, hw_event, pid, cpu, group_fd, flags);
}
int main(int argc, char **argv)
{
   struct perf_event_attr pe;
   long long count;
   int fd;
   memset(&pe, 0, sizeof(struct perf_event_attr));
   pe.type = PERF_TYPE_HARDWARE;
   pe.size = sizeof(struct perf_event_attr);
   pe.config = PERF_COUNT_HW_INSTRUCTIONS;
   pe.disabled = 1;
   pe.exclude_kernel = 1;
   pe.exclude_hv = 1;
   fd = perf_event_open(&pe, 0, -1, -1, 0);
   if (fd == -1) {
      fprintf(stderr, ”Error opening leader %llx\n”, pe.config);
      exit(EXIT_FAILURE);
   }
   ioctl(fd, PERF_EVENT_IOC_RESET, 0);
   ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
   printf(”Measuring instruction count for this printf\n”);
   ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
   read(fd, &count, sizeof(long long));
   printf(”Used %lld instructions\n”, count);
   close(fd);
}
```

Listing A.1: perf_event_open. When using perf_event_open, only a system call is provided by the libraries. To use it as a function, a function wrapper, as shown in the code, has to be built. Exemplary settings are applied to the perf event later on.

Comparison of `gland`, `gland memcpy`, and Intel `mlc`

The following listings show the *hot path* in assembly, annotated with spent cycles left of the causing instructions.[1]

```
               xor    %eax,%eax
               xor    %ecx,%ecx
 0.40% 272:    mov    -0x80(%rbp),%rdi
 0.20%         add    $0x1,%rcx
 1.80%         vmovap (%r9,%rax,1),%ymm4
37.92%         vmovup (%rdi,%rax,1),%ymm0
47.90%         vfmadd 0xa02(%rip),%ymm4,%ymm0
 9.38%         mov    -0x88(%rbp),%rdi
 0.60%         vmovup %ymm0,(%rdi,%rax,1)
 1.60%         add    $0x20,%rax
 0.20%         cmp    $0x3ffffe,%rcx
             ^ jbe    272
```

Listing A.2: Stream triad. Fused multiply–add instructions are used to perform the calculations in one step. All costs amount to operand fetching and writeback.

Both Intel `mlc` and `gland_memcpy` result in `libc`'s internal `memcpy_avx_unaligned`.

```
 0.45%          prefet 0x1c0(%rsi)
34.73%          prefet 0x280(%rsi)
31.31%          vmovdq (%rsi),%ymm0
 0.20%          vmovdq 0x20(%rsi),%ymm1
15.51%          vmovdq 0x40(%rsi),%ymm2
 0.69%          vmovdq 0x60(%rsi),%ymm3
15.14%          sub    $0xffffffffffffff80,%rsi
 0.12%          vmovnt %ymm0,(%rdi)
 0.16%          vmovnt %ymm1,0x20(%rdi)
 0.24%          vmovnt %ymm2,0x40(%rdi)
 0.73%          vmovnt %ymm3,0x60(%rdi)
 0.20%          sub    $0xffffffffffffff80,%rdi
 0.16%          add    $0xffffffffffffff80,%rdx
 0.33%        v jb     9810
```

Listing A.3: `memcpy_avx_unaligned`. All four AVX registers are used to move the data. One can see how prefetching instructions are used to explicitly request succeeding memory (offset by four AVX registers) first. Non-temporal moves are used to signal the CPU to circumvent the caches for the writeback. One can see once more how operand fetching results in the highest overhead in cycles, here.

[1] for gcc version 6.1.1, compiler flags: `-O3`, values obtained with `perf`

I certify that the material contained in this thesis is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation.

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, 22. Juli 2016

Christoph Sterz