Assessing NUMA Performance Based on Hardware Event Counters

Max Plauth, Christoph Sterz, Felix Eberhardt, Frank Feinbube and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute for Software Systems Engineering
University of Potsdam, Germany
Email: {firstname.lastname}@hpi.uni-potsdam.de

***Abstract*—Cost models play an important role for the efficient implementation of software systems. These models can be embedded in operating systems and execution environments to optimize execution at run time. Even though *non-uniform memory access* (NUMA) architectures are dominating today's server landscape, there is still a lack of parallel cost models that represent NUMA system sufficiently.**

Therefore, the existing NUMA models are analyzed, and a two-step performance assessment strategy is proposed that incorporates low-level hardware counters as performance indicators. To support the two-step strategy, multiple tools are developed, all accumulating and enriching specific hardware event counter information, to explore, measure, and visualize these low-overhead performance indicators. The tools are showcased and discussed alongside specific experiments in the realm of performance assessment.

***Index Terms*—Parallel programming, Performance analysis, Memory management**

I. Introduction

With the advent of *non-uniform memory access* (NUMA) architectures, theoretical cost models have become increasingly complex because they need to incorporate various topology characteristics of NUMA-based computer systems [1]. Likewise, recent developments in performance assessment 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 work addresses the need for new strategies and tools for performance assessment and optimization in NUMA systems. Existing cost models for parallel computation are surveyed and categorized. NUMA models are discussed and motivated as another, distinct class of cost models. Additionally, a new, two-step strategy for performance assessment is presented, which makes use of low-level hardware performance counters.

Furthermore, three 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. *Memhist* reveals the latency cost distribution for memory accesses as a major cost factor of recent programs. *Phasenprüfer* automatically attributes indicator measurement records to distinct ramp-up and computation phases. All tools are available online[1]. Figure 1 relates the three presented tools to the proposed two-step strategy.

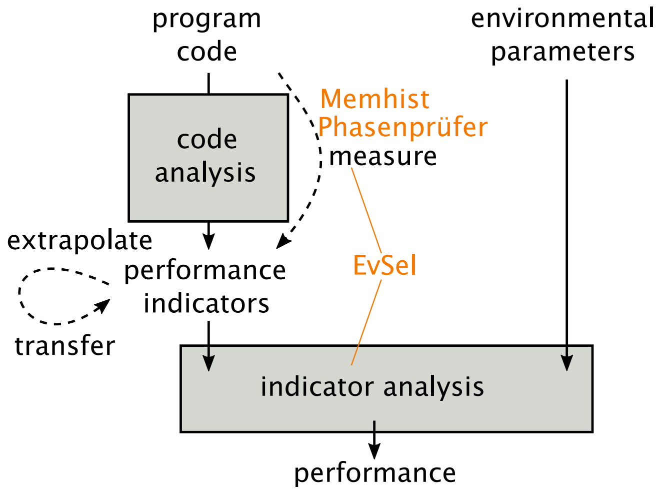


Fig. 1: The three tools presented in this paper (orange) incorporate hardware counters and help in adhering to the presented two-step strategy.

This work is structured as follows: Section II introduces basic concepts and terminologies. We then survey existing cost models for parallel computation and, specifically, NUMA. In Section III, we present a novel two-step strategy for assessing performance in NUMA systems. Section IV introduces three novel performance analysis tools, which support the performance assessment strategy formulated in Section III. 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. *Memhist* reveals the latency cost distribution of memory accesses. *Phasenprüfer* automatically identifies ramp-up and computation phases of a tested program. In this section, we explain development considerations of the tools and discusses their scope of applicability. In Section V, we provide a preliminary evaluation and demonstrate exemplary usage scenarios. Finally, Section VI concludes and summarizes this work's contributions.

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

II. Related Work

This section portrays the development of parallel computation models in three historical eras as categorized by Zhang et al. [2]. Additionally, this section presents an overview of cost models specifically designed for NUMA architectures and introduces the Linux `perf` utility. A rough timeline of the three historical eras of models in parallel computation can be seen in Figure 2. A few representative examples of each era are explained in more detail in the following.

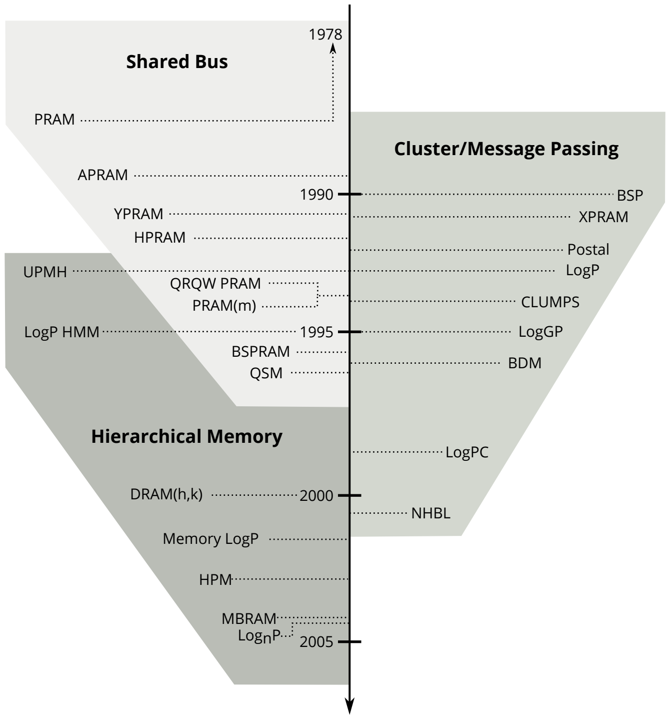


Fig. 2: Historic models of parallel computation. The first era is characterized by a common bus among all processors. The second era addresses clustered environments, whereas the third-era incorporates memory hierarchies.

A. Shared-Bus Cost Models

The first era of parallel computation models describes the dominating shared-bus systems, where multiple concurrent RAM processors execute unit-cost instructions on commonly shared memory in lockstep fashion [3]. PRAM, the most popular model of this era [4], was later enhanced by modeling 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 [5]. 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 [6].

For asynchronous execution, APRAM [7], *asynchronous PRAM* [8], and XPRAM [9] 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 simulating congestion of the shared bus [10]. Hierarchical behavior is implemented in the variants YPRAM [9] and HPRAM [11], where multiple PRAM subunits are simulated inside the machine. These subunits communicate with each other at increased costs, which is modeled using inefficiency factors.

Extensions exist that consider memory access latencies (LPRAM) and bandwidth information (BPRAM) [12]. These extensions can be applied when the application is bound by either property. Concepts of PRAM can be found in the succeeding eras as well. BSPRAM, for example, fuses bulk synchronous behavior with the refinements of concurrent memory accesses [13].

B. Cluster Cost Models

The second era of parallel computation is characterized by distributed memory and moderately slow interconnects. Together with the idea of clusters, *message passing* became part of the models. Notable representatives are *bulk synchronous parallel* (BSP) [14] and LogP [15].

In BSP, a concurrent section is executed by multiple processors. The processors then wait at a global barrier to resynchronize for communication. Lastly, a global synchronization barrier is passed by all threads and a new round of concurrent computation is started. 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. Conversely to costs, the loss of parallelization potential can be determined by summing up the waiting time for synchronization and communication.

LogP can be seen as the asynchronous counterpart of BSP [15]. Four parameters describe computation among processors: latency L, overhead o, the minimum gap between messages g, and the number of processors P.

C. Hierarchical Memory Cost Models

With an ever-increasing speed gap between the execution of instructions and operand fetching, the importance of caching in computer systems increased. Alongside this development, models that embrace memory hierarchies evolved [16].

The models $\mathrm{RAM}(h,k)$ [2] and UPMH [17] 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 memory access operations.

There are LogP representations of caching hierarchies, for instance, *Memory LogP* [18], where caching is modeled using message passing between the hierarchical cache layers. However, neither access patterns nor cache affinity is considered with Memory LogP.

D. NUMA Cost Models

Cost models specifically designed for NUMA architectures usually bear memory access costs as one of their key factors [19], [20], [21]. 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 interconnect [22]. 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.

Schmollinger and Kaufman propose a model named κNUMA, which is aimed at clusters and SMP machines [23]. The model builds on top of the concept of communication in BSP, extending it through submachine functionality (see Figure 3). κNUMA can be thought of as a κ-deep tree hierarchy of processors. The authors present a cost function that integrates sub-processor communication costs into global superstep costs.

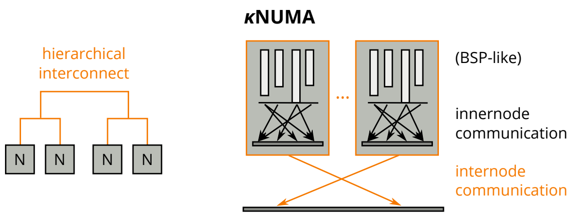


Fig. 3: κNUMA. Schmollinger et al. model nested BSP behavior via tree-interconnected nodes (shown in dark grey) with depth κ [23]. While an internal BSP behavior can be found inside the nodes, an additional, similar behavior is modeled for internode communication.

In his work, Forsell proposes a model for converting workloads with low thread-level parallelism from NUMA to a PRAM counterpart [1]. The paper puts emphasis on hiding latencies by grouping processors to act on a single state, resulting in a PRAM-NUMA model. The work is focused on providing a theoretical contribution, since a rather conceptual machine model is presented instead of actual cost functions.

The work of Zhang and Qin models NUMA interconnects as switching networks [24]. Using the analogy of resistors for memory access costs, the authors predict access times for the matrix multiplication example.

In their work, Ma et al. develop a memory access model for highly-threaded many-core architectures [19]. The authors describe how fast context switching enables memory latency hiding in modern multi-threaded environments. Their *threaded many-core model* (TMM) is validated against four shortest-path algorithms, where it is able to accurately predict performance.

Using a set of parameters, Byna et al. describe the topology or memory hierarchy as well as additional parameters that characterize data access operations [20]. Based on this characterization, they authors are able to estimate the costs of memory access for the widely used matrix transposition algorithm.

Tudor et al. propose an analytical model for estimating the speedup of programs on UMA and NUMA multicore systems [25]. The model uses hardware event counters to predict the performance impact of data access policies and thread placement.

A similar approach has been presented by Cho et al., where the authors provide an online scalability prediction model for applications on NUMA systems [26]. Most notably, a prototypical integration of the model into OpenMP and OpenCL runtimes is used to validate the model.

In other approaches, 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 [27].

E. Applicability of NUMA Cost Models

To enable accurate 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 easily. For a large number of parameters, software developers are obliged to invest much time in obtaining them in order to determine the costs. Even functions have to be considered as inputs to models. This burdens software developers to understand and apply modern models of parallel computation.

At the same time, most cost models are based on theoretical considerations and often are only available in textual form [28]. This makes it impossible for computers to automatically determine costs based on these cost models. There are only a few simulators and actual applications that compute costs based on these complex cost models. In Section III, this paper attempts to address these shortcomings by proposing a strategy for assessing performance based on performance indicators.

F. The Linux `perf` Utility

The Linux `perf` utility represents the centerpiece of all performance measurement tools presented in the course of this work. By providing a layer of abstraction on top of raw hardware event counters, `perf` is able to provide equivalent metrics even if the specific hardware counters may differ across varying 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 using a `perf` kernel probe.

The `perf` utility is able to attribute all measurements to specific code locations in a fine-grained fashion. Programs can be measured on the entire system or on specific CPU cores. Through the event-based sampling facilities, `perf` is able to record events and attribute them to individual addresses. Also, a system-wide mode exists, which traces all processes across the entire system. With these facilities at hands, `perf` enables detecting imbalanced workloads among NUMA nodes.

III. A NUMA PERFORMANCE ASSESSMENT STRATEGY

Here, we suggest a NUMA performance assessment strategy based on measurements made during prior program runs. We propose a two-step strategy, which enables portability across physical systems. The strategy circumvents static code analyses by using low-level performance indicators as an indirection.

A. *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. *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* [29]. Through *hardware counters*, CPUs reveal a part of their internal hidden state. Under the assumption that a functional dependency exists between input parameters, performance indicators, and costs, a strategy using low-level hardware counters for program analysis can be devised.

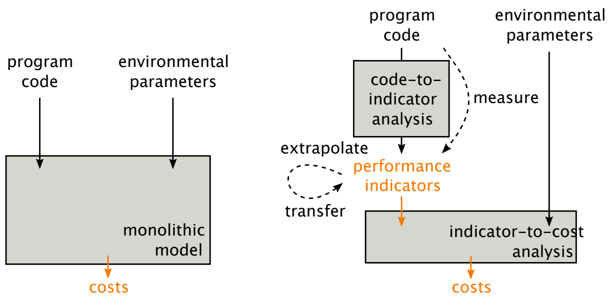
One example for such performance indicators is the instruction count of a program run. Given the costs of each instruction in CPU cycles, the overall number of clock cycles may be estimated. Assuming a fixed clock frequency, the costs in execution time may, in turn, be deduced. This example illustrates 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 simplifying abstractions (worst-case, fixed clock cycles are assumed), which blur the resulting costs.

B. *A Two-Step Strategy*

With hardware event indicators as input parameters, we outlined a new strategy for performance optimization. In contrast to classic single-step (*code-to-cost*) performance models, we propose a two-step performance deduction strategy consisting of a *code-to-indicator* and an *indicator-to-cost* analysis, as seen in Figure 4.

First, the code-to-indicator analysis resembles most aspects of static code analysis, and can thus be considered complex and infeasible for large code bases. Yet, for many programs, measurements with common workloads can be performed offline. For example, programmers would start by measuring small yet typical workloads. Based on these measurements, programmers could extrapolate performance indicators by continuously increasing the workload sizes or measuring varying workloads [30]. In this way, the infeasible direct code-to-cost deduction can be circumvented.

The second step consists of an indicator-to-cost analysis, which 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 to estimate the costs, because clock frequencies might correlate with thermal conditions [31].



(a) monolithic model (b) two-step strategy

Fig. 4: 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 different hardware.

1) Limitations: Because not all machines offer the same performance indicators, a selection of fairly common indicators 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*) [32]. As a response, this paper later introduces an event selection program called *EvSel* (see Section IV-A). Often times, performance indicators 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 [33]. To tackle these problems, statistics uses methods such as *Bonferroni correction*, which requires more samples when the possibility of a multiple comparisons problem exists [34]. Users should be aware of this fact when employing these tools to investigate performance.

IV. New Performance Assessment Tools

In this section, we introduce three novel tools, which support the performance assessment strategy formulated in Section III. The strategy proposes splitting cost deduction into two steps, using low-level hardware counters intermediately.

EvSel covers all hardware counters to compare program runs and to perform program parameter regressions. *Memhist* leverages the load latency events for load instructions to reveal the latency cost distribution of memory accesses. *Phasenprüfer* automatically attributes the counters to distinct ramp-up and computation phases of a tested program. As all tools are built upon Linux `perf`, they are adoptable to hardware platforms where according performance counters are exposed.

A. Selection Through Correlation: EvSel

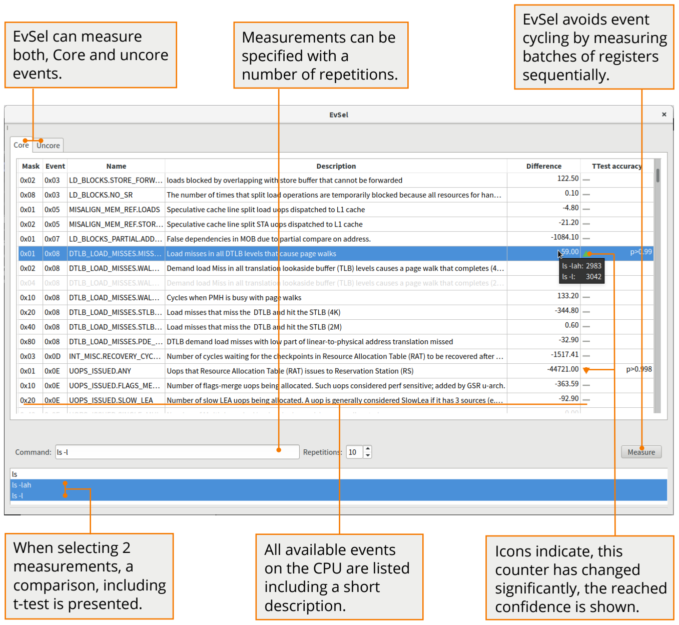
The tool *EvSel* retrieves, measures, and presents all available hardware counters to the user. In addition to identifying relevant performance counters, EvSel helps developers to verify the effectiveness of optimization techniques by comparing two versions or parameter configurations of a program with respect to all performance counter information.

The tool 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 to investigate further.

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. 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. Since only a limited number of registers is available for measuring, program runs are repeated to circumvent this limitation. We argue that collecting counters over identically configured program runs instead of performing event cycling during execution might yield better results when many counters are measured.

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



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.
EvSel
Core
Uncore
Command:
Repetitions:
10
Measure
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.

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
```

Fig. 5: EvSel interface. EvSel presents all retrieved counters alongside their description. In the shown case, the two selected measurements are compared.

The user interface of EvSel is depicted in Figure 5. Several visual cues help engineers understand data more quickly. If a value remains zero for all measurements, it is grayed out. Correlations are color-coded for a quick overview. Tooltips are added to reveal more detailed information.

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 [35]. For single comparisons, *Student's t-test* is applied to the measurements [36]. Upon selecting more than one measurement, Student's t-test is conducted among all events of the two measurements.

The implementation assumes a normal distribution. This decision can be considered controversial since the measurement is clearly biased towards smaller values. The bias is inherent to the fact that for many metrics, there is a lower bound that cannot be undercut. 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.

Moreover, 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 [37].

Lastly, since the test should be possible for any user-chosen program runs, *Welch's method* is employed to compare different population sizes [38]. EvSel assumes similar standard deviations for both measurements since the mechanisms producing the values are the same.

B. *Latency Analysis:* Memhist

Memhist was developed to better characterize NUMA workloads by summarizing latency penalties of memory load operations in a histogram. The tool makes use of the *precise load latency* feature, which can be used to determine whether a certain cycle threshold was surpassed for a memory load.

When measuring exact latencies, even cache accesses spread around their expected peaks. This is caused by the costs of cache evictions and 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.

1) Histogram Construction: Creating the histogram is only possible through empirical sampling over longer time periods, as only a single PEBS event can be measured at a time. Furthermore, the load latency events denote all the loads that surpass a threshold value. To retrieve event information for a specific latency interval, two measurements (lower and upper bound) have to be performed and subtracted.

As a consequence, 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 excessively. This poses an error that cannot be avoided, although Memhist cycles with a frequency of $100\,\mathrm{Hz}$ ($10\,\mathrm{ms}$ slices). Unfortunately, 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 using the Intel Memory Latency Checker tool `mlc`.

Memhist is implemented in C++ and employs the declarative language QML for its GUI. Memhist offers the option to either count events or to multiply event occurrences with their respective latencies to gain insights on the number 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. Because of this, an additional headless probe was developed, which transfers the measured data via TCP to the GUI application. The details of the remote–local architecture are documented in Figure 6.

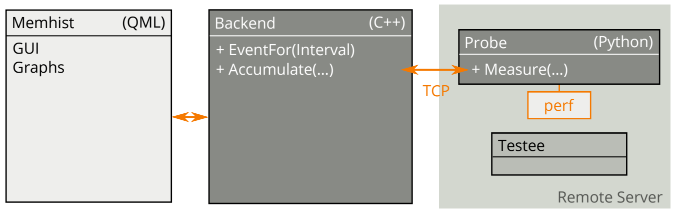


Fig. 6: Memhist architecture for remote probing.

C. *Program Run Phases:* Phasenprüfer

During their execution, most programs pass through multiple execution phases [39]. The tool *Phasenprüfer* was developed to gain insights about the ramp-up and the computation phase of an application. For many workloads, nodes are accumulating large amounts of data during the ramp-up phase. Afterwards, the data is processed during the computation phase.

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, the memory footprint (reserved memory, obtained trough `procfs`) is used to determine the phases.

In order to attribute `perf` event measurements to different phases, Phasenprüfer records and analyzes performance counters for the two phases separately. Attempts at using performance counters for phase detection failed due to strong statistical fluctuations and few available samples. Hence, Phasenprüfer performs phase detection based on the memory footprint as input data.

At this moment, Phasenprüfer merely considers two phases, however, it can be easily extended to recognize additional phases. In the example of BSP-like programs, where multiple supersteps could be analyzed, recognizing individual steps may be desirable.

1) Automatic Phase Selection Through Regression: As with the other tools, Phasenprüfer is designed not to intrude the operation of the program under test. Instead, the two phases are identified based on observation. With the help of segmented regression, Phasenprüfer models the phases as functions and finds the phase transition.

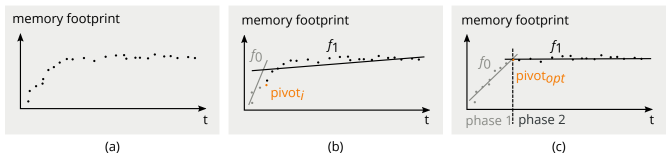


Fig. 7: Detecting phases: Based on the raw data (a), all data points are iteratively considered as pivot elements. Then, all possible combinations of two linear regression fits are tested. The functions with the least combined error determine the pivot element as a phase transition.

To achieve this, all data points are iteratively considered as phase transition points (pivots) first. Next, regression is performed before and after each pivot point. The phase transition is obtained by selecting the point where the summed error of both regressions is minimal. This method is depicted in Figure 7.

The regression itself is achieved using the linear least squares method. A short deduction of the used method can be found below [40]. 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 yields the parameter $\hat{\beta}$ such that its squared difference to y (here called S) is minimal.

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

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

$$(\beta^T X^T y)^T = y^T X\beta$$

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:

$$\beta^T X^T y = y^T X\beta$$
$$\text{(a)} = \text{(b)}$$

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

$$S = y^T y - 2\beta X^T y + \beta^T X^T X\beta$$

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

$$-X^T y + (X^T X)\beta = 0$$

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):

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

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

$$\hat{\beta} = (X^T X)^{-1} X^T y \qquad \square$$

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

Since matrix operations for such small values can be computed efficiently with the linear algebra library Eigen, the phases can be determined in milliseconds, even for thousands of data points [35]. More complex functions could be fitted by transforming 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 computation phase [39]. Therefore, the need of a non-linear fitting function does not arise here.

V. Preliminary Evaluation

A preliminary evaluation of the tools discussed in Section IV is provided in this section. Unless noted otherwise, all tests were conducted using a 4-socket NUMA system as specified in Table I.

TABLE I: Specifications of the test systems.

Server Model	HPE ProLiant DL580 Gen9 Server
Processor	4×Intel Xeon 8890v3 @2.4 GHz
NUMA Topology	Fully interconnected
Memory	4 × 32 GiB RAM @1600 MHz
Operating System	Ubuntu Linux 16.04.1 LTS
Kernel Version	4.4.0-64

A. *Selection Through Correlation:* EvSel

This section analyzes and compares selected micro-benchmarks 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 EvSels results when finding correlations between input parameters and measured indicators for parallel sorting.

1) Cache Miss Micro-Benchmark: In Listing 1, an array is created and also read in column-major order, hereby hitting cache lines fairly often. In Listing 2, the array is read in row-major order instead, causing many more cache misses than before.

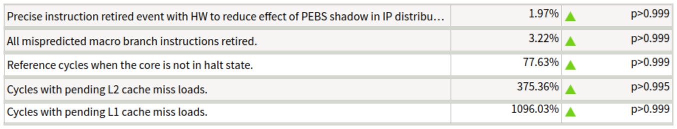
```
const size_t size = 1024;
auto array = new float[size][size];
float altsum = 0;

// 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;
```

Listing 1: Cache miss micro-benchmark example A

```
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;
```

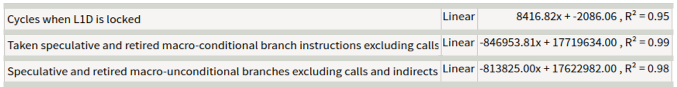
Listing 2: Cache miss micro-benchmark example B

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



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

Fig. 8: Selected run comparisons with EvSel for a caching micro-benchmark.



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	$-813825.00x + 17622982.00$, $R^2 = 0.98$

Fig. 9: Selected correlations from EvSel for the parallel sorting micro-benchmark. Event types, regression function types, and the regression functions themselves are shown along with their coefficients of determination.

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 most significant increase was notable in *rejected fill buffer requests*. In the cache-hit case, the fill buffer rarely had to reject a demand (26 occurrences), whereas it rejected nearly all registration attempts in the cache-miss case (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. Still, a minor correlation caused the values to be statistically distinguishable through t-tests. Selected results are shown in Figure 8.

2) Parallel Sort Micro-Benchmark: As a second micro-benchmark, the parallelization of `std::sort` using GNU `libstdc++` parallel mode is analyzed (see Listing 3). For this purpose, a 4 MiB array of `uint` is filled with pseudo-random numbers using a *linear congruential engine* (LCE), which is essentially a multiply–add ignoring overflows [41].

```
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());
```

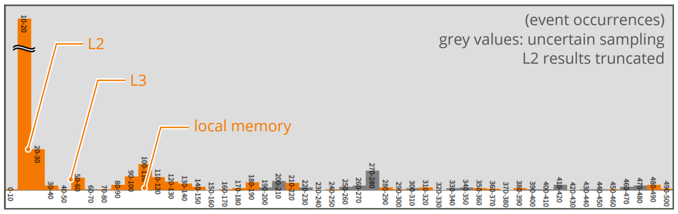
Listing 3: Parallel Sort micro-benchmark

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.

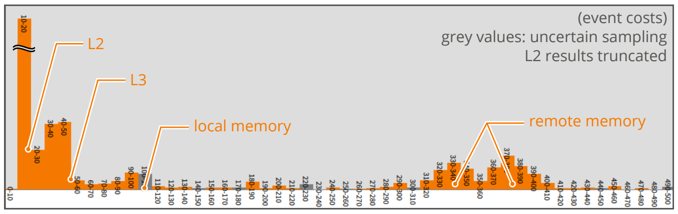
A high negative correlation can be observed between the number of threads and retired speculative jumps ($R > 0.99$), indicating that the CPU was not able to speculatively predict more instructions. Three selected correlations can be seen in Figure 9.

B. *Latency Analysis:* Memhist

To verify the described measurement approach, we present results for a NUMA-optimized SIFT implementation [42], which acts almost entirely on local memory, as shown in Figure 10a. The annotated peaks were verified using the Intel Memory Latency Checker [43]. To provide a contrast to the NUMA-optimized SIFT implementation, we induced remote memory accesses using Intel `mlc` (see Figure 10b).



(a) NUMA SIFT implementation, event occurrences

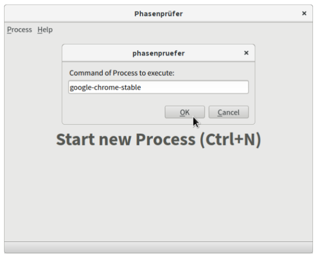


(b) Intel `mlc` remote latencies benchmark, event costs

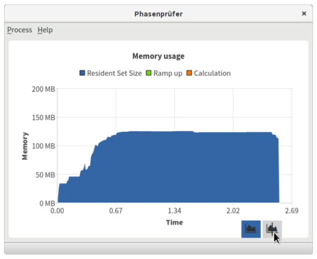
Fig. 10: Screenshots of Memhist's histogram. While aside from caches, main memory is primarily accessed in the first case, the costs of remote accesses can be observed in the second case. All intervals are denoted in cycles. L2 results are truncated to approximately half their height for readability.

C. *Program Run Phases:* Phasenprüfer

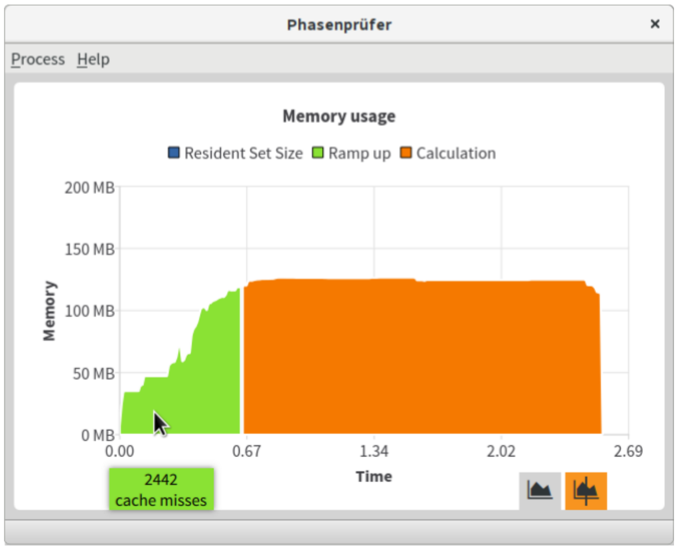
To evaluate the proper isolation of the ramp-up and workload phases, we used several end-user applications such as web browsers and office-suites as exemplary workloads. Here, we showcase the start-up phase for the web browser Google Chrome. Figure 11 demonstrates the typical application usage. With the tool still missing remote probing features, our preliminary results are restricted to end-user workloads. However, we are planning to conduct additional experiments with more complex, server-sided workloads such as the NUMA-optimized SIFT implementation used in Section V-B.



(a) Program start-up.



(b) After program termination.



(c) Upon clicking the *phase split* button, the phases are separated.

Fig. 11: Screenshot of Phasenprüfer, analyzing the start-up behavior of the Google Chrome webbrowser.

VI. Conclusion & Outlook

In the first part of this paper, a brief overview of performance modeling in the field of parallel and distributed computing was laid out. We proposed and motivated the need for new performance models for non-uniform memory access (NUMA) systems. We identified both strengths and drawbacks concerning the applicability of current models and proposed a strategy for extending future NUMA performance models.

Acting as primary performance indicators, low-level hardware counters were identified as a potential interface to circumvent the complexity of code-to-cost models or to transfer program performance characteristics across machines. As a result thereof, a two-step performance analysis strategy was proposed, which is comprised of a code-to-indicator step and an indicator-to-cost step.

We presented three novel tools based on low-level hardware counters. 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 and presents the visually enhanced results to the user. Programmers are enabled 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.

As memory access cost is crucial for performance considerations, measuring individual access latencies is a good performance indicator. On modern Intel CPUs, this can be achieved with so-called *load-latency-enabled* events. *Memhist* uses these low-level counters allowing 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. In future work, many more effects could be investigated, which can now be identified by Memhist: *Translation Lookaside Buffer* (TLB) miss costs, cache coherency protocol overhead, costs of remote memory accesses in more complex NUMA topologies, and so on.

As this paper proposed to consider temporal phases when creating cost models, a deterministic way of splitting a program execution into individual periods had to be identified. 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. With Phasenprüfer currently being limited to detecting two phases, this leaves room for further improvements, including a more elaborate evaluation of the tool.

In the future, the proposals stated for NUMA models could be set into practice. To provide sound means of evaluation, a method for simulating latency and bandwidth characteristics of various systems has to be developed. Especially, simulating and incorporating different topologies should be investigated further when dealing with large-scale systems. Furthermore, the mapping from events to lines of code was merely covered in this paper, yet this information is important to developers when searching for performance bottlenecks in their applications.

Acknowledgement

This paper is based on the master's thesis by Christoph Sterz [44], which contains additional implementation details and background information. We would like to thank Christoph Sterz for having shared his enthusiasm for performance counters with many of his fellow students in the course of writing his thesis. Furthermore, we thank the HPI Future SOC Lab for granting us access to the hardware resources used in this paper.

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

Disclaimer

References

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

[2] Y. Zhang, G. Chen, G. Sun, and Q. Miao, "Models of parallel computation: a survey and classification," *Frontiers of Computer Science in China*, vol. 1, no. 2, pp. 156–165, 2007.

[3] S. Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Springer Science & Business Media, 1996.

[4] D. K. Campbell, "A Survey of Models of Parallel Computation," University Of York, Dept.of Computer Science, Tech. Rep., 1997.

[5] R. M. Karp, "A Survey of Parallel Algorithms for Shared-Memory Machines," Berkeley, CA, USA, Tech. Rep., 1988.

[6] F. E. Fich, P. Ragde, and A. Wigderson, "Relations between concurrent-write models of parallel computation," *SIAM Journal on Computing*, vol. 17, no. 3, pp. 606–627, 1988.

[7] R. Cole and O. 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.

[8] P. B. Gibbons, "A more practical PRAM model," in *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM, 1989, pp. 158–168.

[9] J. Nash, "A study of the XPRAM Model for Parallel Computing," Ph.D. dissertation, PhD thesis, School of Computer Studies, University of Leeds, 1993.

[10] V. 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.

[11] T. Heywood and S. Ranka, "A practical hierarchical model of parallel computation," *Journal of Parallel and Distributed Computing*, vol. 16, no. 3, pp. 212–232, 1992.

[12] A. Aggarwal, A. K. Chandra, and M. Snir, "Communication complexity of PRAMs," *Theoretical Computer Science*, vol. 71, no. 1, pp. 3–28, 1990.

[13] A. Tiskin, "The bulk-synchronous parallel random access machine," *Theoretical Computer Science*, vol. 196, no. 1, pp. 109–130, 1998.

[14] L. G. Valiant, *Bulk-synchronous parallel computers*. Harvard University, Center for Research in Computing Technology, Aiken Computation Laboratory, 1989.

[15] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken, "LogP: Towards a realistic model of parallel computation," in *Proceedings of the 4th SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1993, pp. 1–12.

[16] F. Alted, "Why Modern CPUs Are Starving and What Can Be Done about It," *Computing in Science & Engineering*, vol. 12, no. 2, pp. 68–71, March 2010.

[17] B. Alpern, L. Carter, E. Feig, and T. Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, no. 2-3, pp. 72–109, 1994.

[18] K. W. Cameron and X.-H. Sun, "Quantifying locality effect in data access delay: memory logP," in *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, April 2003, p. 8.

[19] L. Ma, K. Agrawal, and R. D. Chamberlain, "A memory access model for highly-threaded many-core architectures," *Future Generation Computer Systems*, vol. 30, pp. 202–215, 2014.

[20] S. Byna, X.-H. Sun, W. Gropp, and R. Thakur, "Predicting memory-access cost based on data-access patterns," in *2004 IEEE International Conference on Cluster Computing*, Sept 2004, pp. 327–336.

[21] M. Frasca, "Model-driven Memory Optimizations for High Performance Computing: From Caches to I/O," Ph.D. dissertation, The Pennsylvania State University, 2012.

[22] R. Braithwaite, P. McCormick, and W.-c. Feng, "Empirical Memory-Access Cost Models in Multicore NUMA Architectures," *ICCP Virginia Tech Department of Computer Science*, 2011.

[23] M. Schmollinger and M. Kaufmann, "κnuma: A model for clusters of smp-machines," in *International Conference on Parallel Processing and Applied Mathematics*, 2001, pp. 42–50.

[24] X. Zhang and X. Qin, "Performance prediction and evaluation of parallel processing on a numa multiprocessor," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1059–1068, Oct 1991.

[25] B. M. Tudor and Y. M. Teo, "A Practical Approach for Performance Analysis of Shared-Memory Programs," in *Intl. Parallel Distributed Processing Symposium*. IEEE, May 2011, pp. 652–663.

[26] Y. Cho, S. Oh, and B. Egger, "Online scalability characterization of data-parallel programs on many cores," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sept 2016, pp. 191–205.

[27] Q. Wu, A. Pyatakov, A. Spiridonov, E. Raman, D. W. Clark, and D. I. August, "Exposing Memory Access Regularities Using Object-Relative Memory Profiling," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, p. 315.

[28] D. B. Skillicorn, "Models for practical parallel computation," *International Journal of Parallel Programming*, vol. 20, no. 2, pp. 133–158, 1991.

[29] P. Cassier, R. Korhonen, P. Mailand, and M. Teuffel, *Effective zSeries Performance Monitoring Using Resource Measurement Facility*. IBM Corporation.

[30] J. Gonzalez, J. Gimenez, and J. Labarta, "Performance Data Extrapolation in Parallel Codes," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, Dec 2010, pp. 155–163.

[31] W. L. Bircher and L. K. John, "Complete System Power Estimation Using Processor Performance Events," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 563–577, April 2012.

[32] C. Winship and R. D. Mare, "Models for Sample Selection Bias," *Annual Review of Sociology*, vol. 18, pp. 327–350, 1992.

[33] J. Hsu, *Multiple Comparisons: Theory and Methods*. CRC, 1996.

[34] R. A. Armstrong, "When to use the Bonferroni correction," *Ophthalmic and Physiological Optics*, vol. 34, no. 5, pp. 502–508, 2014.

[35] "Eigen: A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms." [Online]. Available: http://eigen.tuxfamily.org

[36] Student, "The Probable Error of a Mean," *Biometrika*, 1908.

[37] W. J. Reichmann, *Use and Abuse of Statistics*. Oxford University Press, 1961.

[38] M. Mendeş and E. Akkartal, "Comparison of ANOVA F and WELCH Tests with Their Respective Permutation Versions in Terms of Type I Error Rates and Test Power," *Kafkas Univ Vet Fak Derg*, vol. 16, no. 5, pp. 711–716, 2010.

[39] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE micro*, vol. 23, no. 6, pp. 84–93, Nov 2003.

[40] G. Strang, *Introduction to Linear Algebra*. Wellesley-Cambridge.

[41] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms. II*. Addison-Wesley, 1969.

[42] M. Plauth, W. Hagen, F. Feinbube, F. Eberhardt, L. Feinbube, and A. Polze, "Parallel Implementation Strategies for Hierarchical Non-uniform Memory Access Systems by Example of the Scale-Invariant Feature Transform Algorithm," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2016, pp. 1351–1359.

[43] Intel Corporation, "Intel Memory Latency Checker." [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[44] C. Sterz, "Analyzing NUMA Performance Based on Hardware Event Counters," Masters Thesis, Hasso Plattner Institute for Software Systems Engineering, University of Potsdam, Jul. 2016.