Journal of Signal Processing Systems
https://doi.org/10.1007/s11265-020-01563-w

Multicore Performance Prediction with MPET

Using Scalability Characteristics for Statistical Cross-Architecture Prediction

Oliver Jakob Arndt[1] · Matthias Lüders[1] · Christoph Riggers[1] · Holger Blume[1]

Received: 29 November 2019 / Revised: 10 March 2020 / Accepted: 26 May 2020

Abstract

Multicore processors serve as target platforms in a broad variety of applications ranging from high-performance computing to embedded mobile computing and automotive applications. But, the required parallel programming opens up a huge design space of parallelization strategies each with potential bottlenecks. Therefore, an early estimation of an application's performance is a desirable development tool. However, out-of-order execution, superscalar instruction pipelines, as well as communication costs and (shared-) cache effects essentially influence the performance of parallel programs. While offering low modeling effort and good simulation speed, current approximate analytic models provide moderate prediction results so far. Virtual prototyping requires a time-consuming simulation, but produces better accuracy. Furthermore, even existing statistical methods often require detailed knowledge of the hardware for characterization. In this work, we present a concept called *Multicore Performance Evaluation Tool* (MPET) and its evaluation for a statistical approach for performance prediction based on abstract runtime parameters, which describe an application's scalability behavior and can be extracted from profiles without user input. These scalability parameters not only include information on the interference of software demands and hardware capabilities, but indicate bottlenecks as well. Depending on the database setup, we achieve a competitive accuracy of 20% mean prediction error (11% median), which we also demonstrate in a case study.

Keywords Parallelization · Performance Prediction · Scalability · Multicore Software Migration

1 Introduction

All common general-purpose processing architectures in high-performance computing clusters, desktop CPUs, and the embedded field are designed with parallel processors. As they represent powerful and energy-efficient architectures, embedded multicores serve as target platforms in a broad range of applications like mobile computing or advanced driver-assistance systems [1]. Applying programmable chips, in contrast to dedicated hardware, also allows the use of flexible and maintainable software with the opportunity of minimizing the time-to-market for new products. But, the development of parallel software, required for parallel processors, opens up a huge design space of varying programming models and parallelization strategies [2], which becomes even more complex for heterogeneous devices like MPSoCs.

A variety of potential bottlenecks as well as the parallelization strategy (e.g., data- or task-level parallelization, and task granularity) may reduce an application's performance. Hence, not only classic parallelization errors, but also insufficient programming strategies as well as improper combinations of software demands and hardware characteristics can easily deteriorate the runtime. Hardware limitations like memory hierarchy and bandwidth, NoC topology, or instruction-set architecture need to be considered. Additionally, the demands of software like memory usage or synchronization behavior also influence the application. Therefore, an early performance estimation is an

✉ Oliver Jakob Arndt
arndt@ims.uni-hannover.de

Matthias Lüders
lueders@ims.uni-hannover.de

Christoph Riggers
riggers@ims.uni-hannover.de

Holger Blume
blume@ims.uni-hannover.de

[1] Leibniz University Hannover, Institute of Microelectronic Systems, Appelstr. 4, 30167, Hannover Germany

Published online: 01 July 2020

important additive for parallel programming tools in hardware/software co-design to calculate the chance of success for optimizations. On the one hand, a naive estimation, which only divides the total execution time by the number of available cores can indeed mispredict the parallel execution time by orders of magnitude. On the other hand, due to the manifold influencing factors, it is impossible to create a perfect model of parallel executions. Therefore, in order to assist programmers during their parallelization process with useful programming hints, more sophisticated approaches with less prediction errors are needed.

Existing prediction methods can be clustered into three categories, which vary in their modeling effort, simulation speed, and resulting prediction accuracy: *Analytic methods* use approximate mathematical models of the architectural capabilities like cache size or memory bandwidth, and software characteristics like data locality or branch distances. While existing analytic models require low parameterization effort and serve good simulation speed, they provide moderate prediction results. *Virtual prototyping* approaches run existing software on emulated hardware, which either requires a full detailed model of the executing hardware or neglects important information in an abstract simulation. Virtual prototyping requires a time-consuming simulation, but produces better accuracy. *Statistical methods* can also be seen as machine learning approaches, as they are based on a database of performance measures (i.e., profiles) and thereof estimate the performance. Potentially requiring a long training phase, the accuracy of statistical methods relies on database and feature quality.

Common prediction scenarios usually reflect one of the following perspectives, also visualized in Figure 1.

- *Migration prediction*: The parallelized software is fixed, but a suitable target platform is needed – either to be built or selected from existing platforms.
- *Scalability prediction*: The platform is fixed, but a sequential software needs to be parallelized, which requires a well scaling parallelization strategy.

In this work, we present the concept of a statistical approach on predicting the performance of parallel software based on its scalability: *Multicore Performance Evaluation Tool* (MPET). The scalability of a parallel program describes the capability of distributing work over increasing numbers of cores and the tendency of simultaneously involving parallelization overhead, finally resulting in an individual parallelization speedup. In our earlier research, we found that all relevant behavioral characteristics and bottlenecks, like the susceptibility to NUMA-node distance (communication delay between distinct processor chips), increase of work imbalance per additional core, and even upper bounds of concurrency, can be found in the scalability graph. Therefore, we use abstract parameters, which describe the scalability behavior, to search proper prediction candidates from a database and thereof reconstruct the target performance. In contrast to other prediction methods, this approach is purely based on characteristics extracted from profiles and requires no modeling effort and user input. Indeed, we do not train a prediction model, but use distance metrics from feature vectors to use a weighted interpolation without any training time. A simple mathematical model is used to describe scaling curves, of which trends can be expressed by only few parameters that form the descriptive feature vector. Furthermore, our model is able to consider effects like NUMA-node distances and usage of hyperthreading as well, which enables precise predictions with varying numbers of cores n instead of only predicting the maximum speedup at $n = n_{max}$. While we also present a benchmark set to create a database of reference architectures, we specifically introduce the following prediction steps:

1. Extraction of the feature vector from scaling curves
2. Selection of prediction candidates by distances
3. Prediction by weighted interpolation, reconstruction

Up to now, this prediction method requires benchmark results of the target platform stored in a database. Therefore, it is most accurate in predicting the performance on existing platforms. Predicting a workload's performance on a hypothetic platform (e.g., by varying the number of available CPU cores or its susceptability to memory bottlenecks) reduces the reliability of prediction results. By adding more prediction candidates and improving the significance of provided benchmarks in our future research, we aim at an improved confidence of predictions even on hypothetic not-existing platforms.

The remainder of this article is structured as follows. In Section 2, we give an overview on virtual prototyping

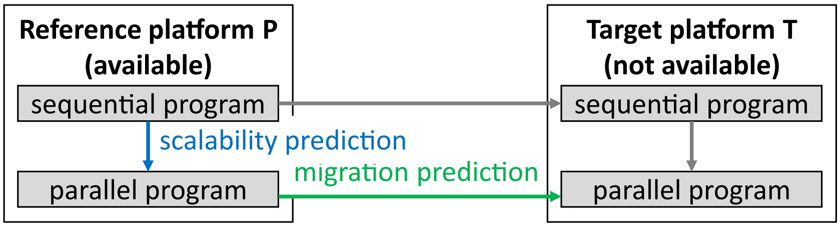


Figure 1 Common performance prediction scenarios: scaling behavior of not yet parallelized software and parallel performance after migrating parallel software to another platform.

techniques as well as analytic and statistical prediction methods. Section 3 presents the contributed statistical prediction method, which is evaluated in Section 4, while we present prediction error measures, an evaluation on the database quality, and a case study, in which we compare our method with two reference methods. Section 5 concludes this paper.

2 Related Work

In this work, we mainly focus on algorithms, which are not intentionally limited in their scalability (e.g., due to a defined maximum of the number of threads or tasks in the implementation), but either make use of flexible task sizes or dynamic numbers of tasks. Consequently, limitations of the given workloads can be classified into the following aspects, while realistic workloads can also suffer from an unfavorable combination of those:

- *Memory bound:* The workload is mainly consuming high amounts of memory and stresses the buses between memory levels (i.e., RAM, caches, and registers). These workloads require optimizations in data locality (spatial and temporal) for both sequential and parallel execution.
- *Communication bound:* The workload is mainly stressing the inter-core communication network (e.g., NoC) for instance due to synchronizations and locks or high data-transfer rates between cores. These workloads require optimizations in their parallelization strategies and could for instance profit from more long-running and independent tasks.
- *Computation bound:* The workload is mainly consuming processor time and only refers to memory and communication networks at a minimum. This is the best scenario for a parallel execution, as it enables independent tasks. Optimizations should focus on sequential optimizations like SIMD vectorization, loop optimization, and pointer-arithmetic.

These aspects are often used to evaluate a worst-case estimation of the performance using a roof-line model as for instance adapted to a so-called *boat-hull* prediction model for GPU and multicore performance by Nugteren and Corporaal [3]. But, parallel programming rather requires a realistic prediction that neither over- nor under-estimates the parallel execution. Furthermore, detailed information of characteristics of the parallel runtime behavior is needed in order to produce concrete optimization hints and specific bottleneck analysis.

System modeling and prediction is a complex task in all research areas, because models either need to be extremely complex in order to cover most aspects or potentially neglect important influences [4]. On the way of defining an appropriate level of abstraction, the search for basic parameters (principle components) with maximum correlation to the system behavior is most decisive. However, many systems exhibit interfering parameters, which complicates the modeling – even more, when the system appears as black box. If the database consists of a sufficient number of measurement points, machine learning (e.g., deep learning) algorithms that automatically extract features, principle parameters, and correlations are often used in other research fields. In the field of processor and software modeling, the number of measurement points is limited due to the costs of varying hardware platforms and the development time of new parallel software. Therefore, even statistical methods for multicore performance prediction often use more traditional (manually created) models for feature extraction. Hence, instead of predicting arbitrary workloads on any processor types, some approaches focus on characterization of basic blocks like *stencil codes*, which define repetitive processing patterns. Existing performance modeling techniques can be clustered into three categories namely *virtual prototyping*, *analytic modeling*, and *statistical methods* [5, 6], which vary in their modeling and simulation effort as well as in the resulting prediction accuracy, as compared in Figure 2.

2.1 Virtual Prototyping

Virtual prototypes emulate a full functioning hardware platform in a software simulation that executes compiled software binaries or even entire operating systems. While these platforms can differ in their abstraction, a less precise timing, but more function-oriented model can speedup the simulation at reduced timing accuracy. Pluggable components like memory controllers, CPUs, or network-on-chips are implemented in system-level design languages like SystemC, hardware description languages like VHDL, or even in RTL. In order to model interactions between

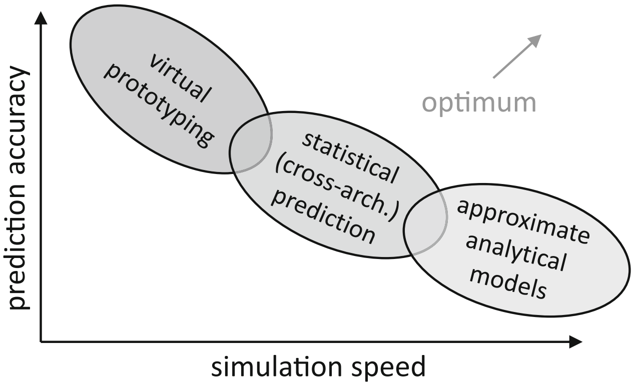

prediction accuracy
virtual prototyping
statistical (cross-arch.) prediction
approximate analytical models
optimum
simulation speed

Figure 2 Prediction accuracy versus simulation speed of different prediction methods, existing models presupposed.

components on varying abstraction levels, TLM offers a communication centric approach.

Existing virtual prototyping frameworks like *GEM5* [7], *Open Virtual Platforms* [8], *QEMU (quick emulator)* [9], or *Cadence Incisive* [10] also offer complete out-of-the-box useable virtual platforms. QEMU for example offers the possibility of quickly configuring an emulation of different instruction-set architectures and also emulates entire platforms like RaspberryPi or the Xilinx Ultrascale+. QEMU uses a high level of abstraction (functional simulation) and thus allows to evaluate an application's functionality in short simulation times. However, there are existing approaches to add timing information to QEMU by adding a timing database or TLM and SystemC modules.

GEM5 [7] is an open-source tool, which is widely used in academia as well as in productive context. GEM5 uses an extendable and modular system for virtual prototypes and some components are even provided by original vendors like ARM. Based on C++ components, a virtual platform can be easily configured with a python API and even heterogeneous architectures like heterogeneous CPU clusters or GPU-equipped platforms can be created. As configured systems represent full functioning platforms, applications can be executed as bare-metal executable or in a full operating system like Linux or Android. GEM5 also offers the possibility to extend the virtual prototypes by TLM or SystemC modules. Butko et al. [11] presented a case study for predicting the performance of an ARM big.LITTLE platform (Cortex-A15 + Cortex-A7) with an average prediction error of 20%.

2.2 Approximate Analytic Models

Analytic predictors build separate models of the hardware capabilities and characterization of software demands [12] to be combined in a mathematical evaluation. Carlson et al. [13] evaluated mechanistic core models for analytic predictions, which are usually based on architecture-independent characterizations of the workload (e.g., cache reuse histograms [14]). Other software metrics like branch probability or loop trip count, which are used in many approaches, can be extracted for example using the LLVM compiler. Van de Steen et al. [15] predicted the single-thread performance with an average error of 13%, which was then extended for multicore prediction by De Pestel et al. [16]. Another analytic processor model by Jongerius et al. [17] (used in IBM Exabounds) predict power and performance, and was evaluated with a Xeon E5-2697 v3 and an ARM Cortex-A15. In this model, the singlecore performance was predicted with an error of 59% and the impact of using multicores with an additional error of 11%.

IBM Exabounds also creates platform independent application profiles in LLVM. The module comes with a small set of existing processor models, while the creation of a new model requires detailed knowledge about the architecture. Exabounds creates comprehensive predictions for runtime, CPI, cache-miss rates, and many more. A related approach is integrated in the Silexica SLX parallelization tools. SLX predicts performance from LLVM-based application profiles and an XML processor-model. Additionally, SLX also predicts promising parallelizations, vectorizations, and offloadable sections of sequential programs. LLVM not only offers platform independent profiles for third-party prediction approaches, but also integrates the so-called MCA prediction tool since version 7. The MCA tool uses profiles offered by LLVM and adds a processor-pipeline model to predict the parallel runtime behavior.

2.3 Statistical Methods

Statistical predictions make use of a database of previously extracted runtime metrics, of which descriptive feature vectors for workload and hardware are extracted separately to be processed by machine learning (ML) (see Figure 3). Hence, the search for descriptive features that allow a good prediction such as performance counters or register usage behavior is key. Statistical methods rely on significance of features, but often extracted metrics of profiles lack of interpretability due to their variations between architectures. Therefore, many approaches focus on certain architecture families, such as GPUs [18–20].

In [21], Ardalani et al. use microarchitecture-independent characteristics as descriptive workload-features of CPU code (see [22]) to predict performance of the corresponding but not yet implemented GPU-kernel. In this method, a set of machine learning algorithms is used in an ensemble technique to predict with an average error of 27% on a Maxwell and 36% on a Keppler architecture. Another approach uses skeletons from CPU code to predict the according performance on a GPU [23]. Wang et al. [24] use performance results and processor classifications from two common benchmarks *SPEC* and *Geekbench* to predict the performance of x86-architectures for new platforms as well as for new workloads. While these benchmarks basically provide single-threaded sample codes, a principle component analysis is used in this work to identify meaningful feature vectors. While deep neural networks showed better results than linear regression, an average error of 5% (SPEC) and 11% (Geekbench) for predicting a new core configuration and 26% (SPEC) and 14% (Geekbench) for predicting a new workload were evaluated.

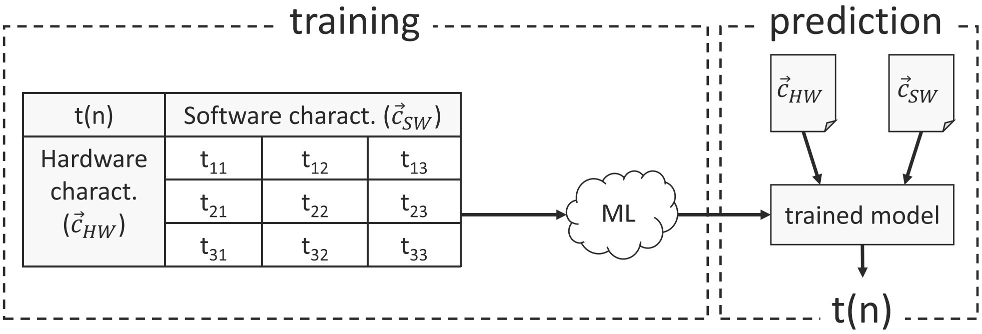


Figure 3 Statistical prediction using previously trained model from machine learning (ML).

3 Scalability Based Prediction

Previous statistical predictions are based on separated features $\vec{c}_{HW}$ of a platform P and $\vec{c}_{SW}$ of algorithmic workload A, such that the parallel execution time of A running on P is:

$$t = f(\vec{c}_{HW}, \vec{c}_{SW}). \tag{1}$$

First, we propose to not only consider the characteristics under use of all available cores, but to also observe entire scalability trends. Therefore, we profile the workload under use of varying numbers of cores n, starting from sequential execution $t(n = 1)$ up to all cores $n = n_{max}$, resulting in t depending on n

$$t(n) = f(\vec{c}_{HW}, \vec{c}_{SW}, n). \tag{2}$$

As many runtime influencing effects play a role in the parallel runtime behavior, we also propose to not predict the execution time in one step, but to split it up by abstract parallelization effects like *work imbalance* or *scheduling overhead* and predict trends of all effects separately. Thereby, we do not define explicitly separated characteristics for hardware and software, but instead we use from profiles extracted parameters, which

- have hardware and software influences,
- represent an abstract *behavioral* perspective,
- directly point out potential bottlenecks, and
- are easy to extract from profiles (no modeling effort).

Other statistical prediction approaches train a model using machine learning that predicts from characteristics. In contrast, we make use of the database as a whole, thereof search prediction candidates by distance metrics between characteristics and perform an interpolation between candidates. Thereby, a characterization of a platform or workload is defined as the combination of characteristics of multiple benchmarks on a platform respectively the combination of characterizations of a workload profiled on different platforms.

Figure 4 presents the prediction workflow that uses a database, which exhibits characterizing profiles from reference and target platforms. To characterize a platform, we prepared benchmark workloads (parallel implementations) B_i, each with varying characteristics (e.g., memory access behavior, parallelization strategy). On an already known reference platform P (profiled with benchmarks B_i), a new workload A can be tested to be ready to predict its performance on any target platform T_j from database. A more detailed visualization of this prediction process is shown in Figure 6 and enabled prediction scenarios are depicted in Figure 7. Consecutive parallel sections (e.g., separated with barriers or explicit spawns) are considered as separate workloads, as they mostly vary in their characteristics.

3.1 Scalability Characteristics

In our earlier work [25], we presented the middleware layer MPAL (*Modular Parallelization Abstraction Layer*) for parallel programming, which automatically extracts the metrics that we use to build descriptive characterizations (features), summarized in Table 1.

- *Redundancy*: Percentage increase of execution time (sum of all parallel tasks $\sum_i t_i(n)$) induced by memory bottlenecks or caching (lock times excluded).
- *Synchronizations*: Time $t_{lock}(n)$ that a task spends on waiting for a lock relative to available CPU time.
- *Work imbalance*: Percentage of the available CPU time, which can not be used for task execution due to improper work split (processor idle time).
- *Scheduling overhead*: Fraction of the available CPU time needed to manage tasks. Considers task creation $t_{tc}(n)$, task distribution $t_{dist}(n)$, task switch $t_{sw}(n)$, and task team synchronization (join) $t_{join}(n)$.

In Section 4, we will present an evaluation of the meaningfulness and validity of these parameters to be used as descriptive features for characterization of parallel workloads. Thereby, exemplary workload characteristics from our current database as well as an analysis of the correlation

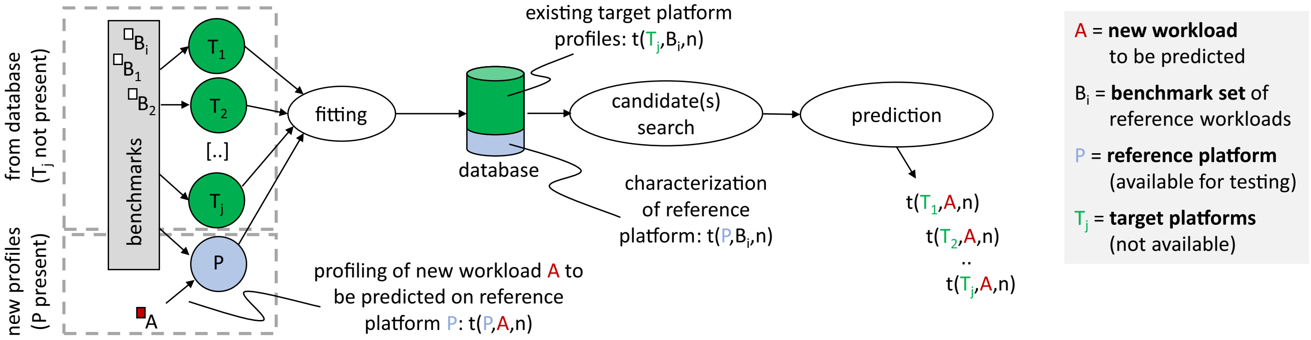

from database
(T_j not present)
new profiles
(P present)
benchmarks
B_i
B_1
B_2
T_1
T_2
[..]
T_j
P
A
fitting
existing target platform profiles: t(T_j,B_i,n)
database
characterization of reference platform: t(P,B_i,n)
profiling of new workload A to be predicted on reference platform P: t(P,A,n)
candidate(s) search
prediction
t(T_1,A,n)
t(T_2,A,n)
..
t(T_j,A,n)
A = new workload to be predicted
B_i = benchmark set of reference workloads
P = reference platform (available for testing)
T_j = target platforms (not available)

Figure 4 Prediction workflow that predicts a new workload A, profiled on a reference platform P, onto a known but not present target platform T from database.

between derived distances and corresponding prediction errors will be given.

These abstract parameters provide direct information on the program's parallel runtime behavior and bottlenecks. For example, the scalability profile in Figure 5 illustrates a strong influence of redundancy and work imbalance on the parallel execution at only small scheduling overhead (sum of scheduling influences: $sched(n) = c(n) + d(n) + s(n) + j(n)$) and no locks, which can be the opposite in other profiles. We predict these scalability behavior metrics separately, while the resulting parallel execution time can be reproduced as follows.

$$t(n) = \frac{t(1) \cdot R(n)}{n \cdot U(n)} \text{, with Utilization } U(n) \tag{3}$$

$$U(n) = 1 - sum(l(n), w(n), c(n), d(n), s(n), j(n)) \tag{4}$$

We found, that all scalability curves for a parameter $p \in \{R, l, w, c, d, s, j\}$ have a simple trend over increasing n that can be described with a function p with $p(n) = m \cdot g(n) + b$. While $g(n)$ represents a general gradient function, most parameters are currently modeled with a linear increase ($g(n) = n$). Only the task distribution is modeled with $g(n) = \frac{n\cdot(n+1)}{2}$ (Gaussian sum), as the waiting time for each additional thread, that needs to be served with tasks, adds another delay $n \cdot x$. b defines an offset, which can be set to zero for some parameters. Only additional offsets (sudden increases) and adjustments of the gradient need to be taken into account, when NUMA-node communications occur or hyperthreads are used. Therefore, after fitting these curves with the model in Equation 5, each scalability parameter's trend is fully described with quantitative numbers in a 6-dimensional descriptive feature $\vec{s}_p$ that is used to concatenate the final characterizing feature vectors per profile.

Table 1 Characterizing scaling parameters in parallel workloads. All measures are normalized by the available CPU time $t(n){\cdot}n$ respectively $t(1)$ and have no units

Redundancy:	$R(n) =$	$\left[\sum_i t_i(n) - t_{lock}(n)\right]/t(1)$
Lock delays:	$l(n) =$	$t_{lock}(n)/(t(n) \cdot n)$
Work imbalance:	$w(n) =$	$t_{wi}(n)/(t(n) \cdot n)$
Task creation:	$c(n) =$	$t_{tc}(n)/(t(n) \cdot n)$
Task distribution:	$d(n) =$	$t_{dist}(n)/(t(n) \cdot n)$
Task switching:	$s(n) =$	$t_{sw}(n)/(t(n) \cdot n)$
Task synchronization:	$j(n) =$	$t_{join}(n)/(t(n) \cdot n)$

$$p(n) = m_{base} \cdot g(n) + b_{base} + nu(n) + ht(n) \tag{5}$$

$$nu(n) = \begin{cases} m_{nu} \cdot g(n - n_{nu}) + b_{nu} & n > n_{nu} \\ 0 & else \end{cases} \tag{6}$$

$$ht(n) = \begin{cases} m_{ht} \cdot g(n - n_{ht}) + b_{ht} & n > n_{ht} \\ 0 & else \end{cases} \tag{7}$$

In order to create the characterizing *descriptive feature vector* $\vec{sc}$ of a profile, we concatenate all *parameter description features* $\vec{s}_p$ (column vectors) and the vector of extracted performance counters $\vec{pc}$ (column vector) (see Equation 9). Indeed, the performance counter vector $\vec{pc}$ not only includes the pure extracted performance counters themselves, but also some relative values like *L1-Load-Misses/Instructions*. In current configuration, the resulting descriptive column vector $\vec{sc}$ includes up to 84 elements, but some elements can be invalid as some platforms do not have multiple NUMA-nodes, provide hyperthreads, or do not offer requested performance counters. In this case, the particular elements in $\vec{sc}$ are tagged as invalid.

$$\vec{s}_p = [m_{base}, b_{base}, m_{nu}, b_{nu}, m_{ht}, b_{ht}]^{\mathsf{T}} \tag{8}$$

$$\vec{sc} = [\vec{s}_R^{\mathsf{T}}, \vec{s}_l^{\mathsf{T}}, \vec{s}_w^{\mathsf{T}}, \vec{s}_c^{\mathsf{T}}, \vec{s}_d^{\mathsf{T}}, \vec{s}_s^{\mathsf{T}}, \vec{s}_j^{\mathsf{T}}, \vec{pc}^{\mathsf{T}}]^{\mathsf{T}} \tag{9}$$

For load balancing purposes, the Linux kernel assigns software threads to cores by increasing core-ID, with respect to scheduling domains [26]. Thereby, CPUs typically represent their core-IDs in an order, which first assigns one core-ID per physical core (no hyperthreading) over all

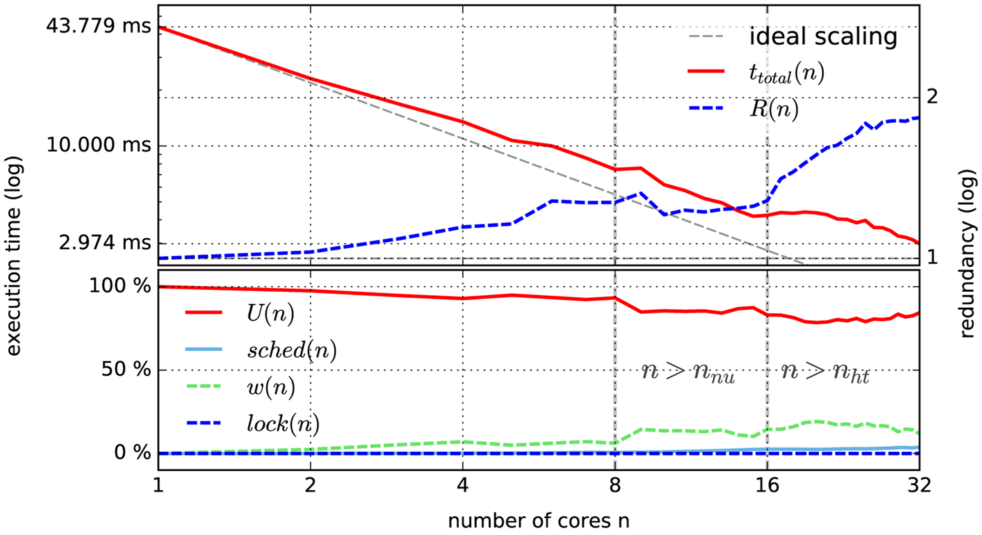


Figure 5 Exemplary profile that shows the introduced scalability curves of a parallel section of a stereo-vision algorithm on a Xeon E5-2630 v3 processor.

NUMA-nodes. Hyperthreads are numbered with core-IDs subsequently, such that $n_{nu} < n_{ht}$ and effects induced by communications between NUMA-nodes (separate processor chips) typically appear earlier in scalability plots than hyperthreading effects. In the example in Figure 5, the processor consists of two distinct processor dies (NUMA-nodes), each with 8 physical cores (NUMA-communication effective from $n > n_{nu}$ with $n_{nu} = 8$) and two hyperthreads per core (hyperthreading effective from $n > n_{ht}$ with $n_{ht} = 16$).

As mentioned before, this prediction approach basically covers the modeling of theoretically scalable implementations (i.e., flexible task sizes or a dynamic number of tasks). In fact, most realistic implementations are designed to be scalable in order to realize a performance-portable application. Therefore, this model covers all possible scalability limitations with the mentioned scalability parameters to represent the full variety of scaling curves. In case of a not parallelizable workload, the lack of scalability is also represented in scalability parameters and thus considered for performance prediction.

3.2 Prediction

The prediction is split into two separate steps, which can also be used separately: (1) prediction of the sequential execution time $t(1)$ and (2) prediction of the scalability behavior $\vec{sc}$. Basically, both are predicted in a similar process, which is schematically presented in Figure 6 that shows signals, parameters, and switches, while all processing steps will be explained separately in the following pages. The proposed method enables the following predictions, while concrete prediction scenarios and therefore required information from database will be explained later and summarized in Table 3:

a. Performance prediction of a present workload A, measured on a reference platform P, to another benchmarked target platform T that is not present ($t_{P,A}(n) \rightarrow t_{T,A}(n)$) (see Figure 7(a)). That includes the prediction of $t(1)$ and $\vec{sc}$.
b. Scalability prediction of not yet parallel workload A on platform P using references from database ($t_{P,A}(1) \rightarrow t_{P,A}(n)$) (see Figure 7(b)). Adopting scaling features of existing benchmarks, clustered by parallelization strategies, estimates the parallelization success. As sequential profiles provide no scaling features ($\vec{pc}$ only), accuracy is limited.
c. Prediction of virtual workload V, where free parameters are for instance *memory usage*, *lock synchronizations*, *work distribution* ($t_{P,A}(n) \rightarrow t_{P,V}(n)$) (see Figure 7(c)). That allows to predict performance impacts due to changes in current implementation of workload A (e.g., after bottleneck analysis).

As a platform characterization basically consists of the set of tested benchmarks, a virtual platform is hard to predict. Free parameters for virtual platforms are those, which are directly used in reconstruction of $t(n)$: n_{nu}, n_{ht}, n_{max}.

3.2.1 Distances

The prediction starts with a search of prediction candidates based on distances between scalability vectors. First, all single elements of $\vec{sc}$ (scalar element sc_m with dimension-ID m) are normalized inside of the dimensions of the

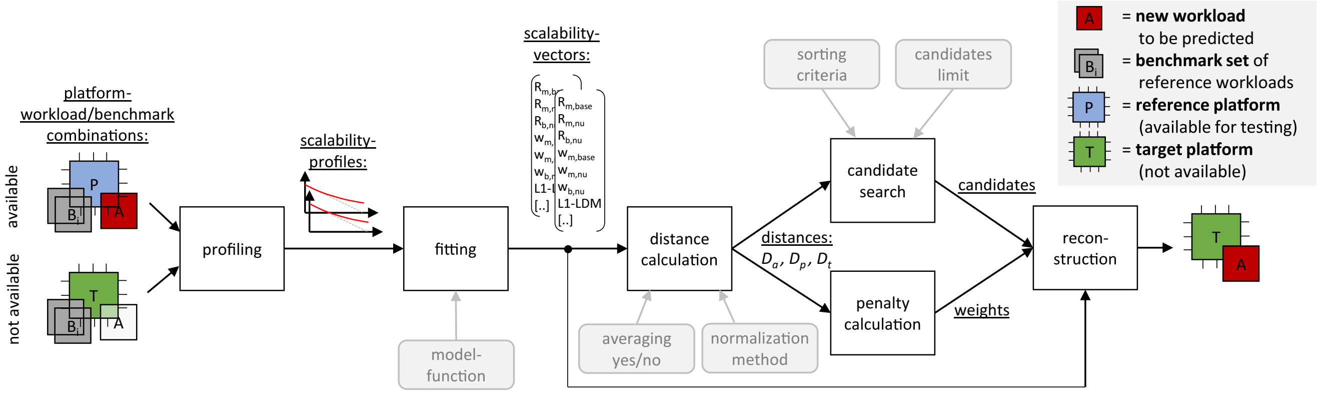


Figure 6 The prediction process with signals and data as well as a selection of parameters and switches that can be used to optimize the prediction. The process on the left shows how the database is constructed. Prior to prediction, the target platform T is characterized with benchmarks B_i. Profiles of A and B_i on reference platform P are added at prediction time.

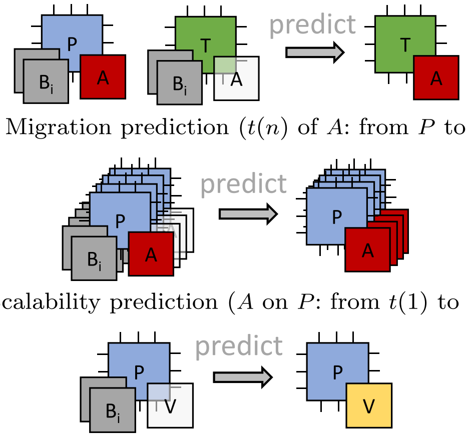
scalability vectors to the range $[0, 1]$ before estimating distances. We only store global absolute minimum and maximum for final reconstruction of $t(n)$ from the predicted target feature vector.

$$max_m = max(sc_m(P, A), \{sc_m(T_j, B_i)\}) \tag{10}$$

$$min_m = min(sc_m(P, A), \{sc_m(T_j, B_i)\}) \tag{11}$$

$$sc_{m,norm} = (sc_m - min_m)/(max_m - min_m) \tag{12}$$

Generally, the distance $\mathcal{D}$ between two vectors is defined as the geometric vector distance, which is normalized to the maximum possible distance depending on the number of valid dimensions, so that $0 \leq \mathcal{D} \leq 1$. In our first experiments, the L2-norm is used, but dimensions are ignored, where one or both elements in the vectors are tagged as invalid. The number of valid dimensions, in which none of both vector elements is tagged as invalid, is in the following written as $dim(\vec{sc}_\alpha, \vec{sc}_\beta)$. Therefore, in a space in which all value ranges are limited to 1, the maximum possible distance is $\sqrt{dim(\vec{sc}_\alpha, \vec{sc}_\beta)}$.



(a) Migration prediction ($t(n)$ of A: from P to T).

(b) Scalability prediction (A on P: from $t(1)$ to $t(n)$).

(c) Virtual workload prediction ($t(n)$ on P: from B_i to V).

Figure 7 Prediction scenarios enabled with the presented statistical MPET predition method.

$$\mathcal{D}(\vec{sc}_\alpha, \vec{sc}_\beta) = \frac{\left\|\vec{sc}_\alpha - \vec{sc}_\beta\right\|_2}{\sqrt{dim(\vec{sc}_\alpha, \vec{sc}_\beta)}} \tag{13}$$

Hypothetical (reduced) example: The following example demonstrates the calculation of the distance between two reduced hypothetical vectors, only consisting of the base-redundancy parameters ($R_{m,base}$, $R_{b,base}$) and last-level cache-misses (LLC-M) as a performance counter measure:

$$\begin{pmatrix} R_{m,base} \\ R_{b,base} \\ LLC - M \end{pmatrix} \vec{sc}_\alpha = \begin{pmatrix} 0.01 \\ 0.04 \\ 0.20 \end{pmatrix}, \vec{sc}_\beta = \begin{pmatrix} 0.20 \\ 0.01 \\ inv. \end{pmatrix} \tag{14}$$

In the Example, the LLC-M performance counters are not available in $\vec{sc}_\beta$ (marked as invalid), which reduces the number of dimensions that are respected in distance calculation to 2, such that

$$\mathcal{D}(\vec{sc}_\alpha, \vec{sc}_\beta) = \frac{\sqrt{(-0.19)^2 + (0.03)^2}}{\sqrt{2}} = 0.136. \tag{15}$$

In the following, we define the distance between scaling vectors of workloads A and B profiled on platform P as $\mathcal{D}_a(P, A, B)$ and the distance between scaling vectors of the same workload A profiled on platforms P and T as $\mathcal{D}_p(A, P, T)$. A distance $\mathcal{D}_a(A, B)$ between workloads A and B can be expressed as the average distance over all platforms T_j in database, where both workloads are profiled. A general distance $\mathcal{D}_p(P, T)$ between platforms P and T can be expressed as the average distance over all workloads B_i from database that are profiled on both platforms.

$$\mathcal{D}_a(P, A, B) = \mathcal{D}(\vec{sc}(P, A), \vec{sc}(P, B)) \tag{16}$$

$$\mathcal{D}_p(A, P, T) = \mathcal{D}(\vec{sc}(P, A), \vec{sc}(T, A)) \tag{17}$$

$$\mathcal{D}_a(A, B) = mean(\mathcal{D}_a(T_j, A, B)) \tag{18}$$

$$\mathcal{D}_p(P, T) = mean(\mathcal{D}_p(B_i, P, T)) \tag{19}$$

While the proposed scalability parameters are always used for distance calculation, the set of performance counters varies between platforms according to their particular availability. Generally, we always use basic performance counters as well as some relative measures, which are briefly (incomplete) summarized in Table 2. According to the central approach of this work to use scalability information as characteristics, we not only use performance counters as a static number, but observe the entire scalability trend and fit the proposed mathematical model to receive 6 derived model parameters per performance counter. In order to characterize a platform, TLB measures could add significant behavioral information, but TLB counters are not available on most platforms. However, the search for a perfect set of derived performance counters is still subject of current research and will be improved in our future work.

3.2.2 Candidates

Given a set of reference profiles (benchmarks B_i) from the database, most promising candidates for predicting a new workload A onto a target platform T are selected by minimum distances. If A has only been profiled on one reference platform P, workload candidates can be selected by $\mathcal{D}_a$. If multiple reference platforms P_x are available (as shown in Figure 8), which improves the prediction accuracy, we consider $\mathcal{D}_p$ as well. While combining algorithmic and platform distances to a *tuple distance* $\mathcal{D}_t$, distances can be seen as inverse likeliness, such that we calculate $\mathcal{D}_t$ as

$$\mathcal{D}_t = 1 - \left[(1 - \mathcal{D}_p) \cdot (1 - \mathcal{D}_a)\right]. \tag{20}$$

Tuples can either be created by using workload and platform individual distances, or by using mean distance for one or both (Figure 8). Prediction candidates are then selected by minimum tuple-distances. For example, a reference platform P can be selected by mean distance $\mathcal{D}_p(P, T)$ first, whereas the workload candidate is selected by individual distances on P by $\mathcal{D}_a(P, A, B_i)$. To eliminate characteristic peculiarity of only a certain prediction candidate, we can use multiple prediction candidates, as described in the subsequent paragraph. We tested multiple configurations, whether candidates should be selected by mean or individual distances, and how many candidates are used. The final setup is given in Section 4.

In addition to the enabled prediction scenarios that we summarized in Figure 7, Table 3 presents an overview of information that is required to be available in the database to enable certain predictions:

1. With only sequential profile information stored in the database (sequential execution time $t(1)$ and performance counters $\vec{pc}(1)$), an approximate estimation of the sequential execution time after migration to another platform can be realized.
2. Using the proposed behavioral scalability metrics (plus performance counters for better results, which are included in $\vec{sc}$), allows to predict the parallelization speedup after a migration.
3. A full and precise migration prediction of the entire scaling trend $t(n)$ on a target platform T can be achieved using both t_1 and $\vec{sc}$.
4. Using classifications of the applied parallelization strategies of the benchmark workloads, the speedup that is to be expected after parallelizing a not yet parallel implementation can be predicted for an available reference platform P as well as for an unavailable target platform T.

Table 2 Some performance counters and relative measures used in the prediction process (if available).

Abbr.	Description
INSTR	no. instructions
BR	no. branches
BR-M	no. branch-misses
L1-LD	no. level-1 cache loads
L1-ST	no. level-1 cache stores
L1-LDM	no. level-1 cache load-misses
L1-STM	no. level-1 cache store-misses
LLC-LD	no. last-level cache loads
LLC-ST	no. last-level cache stores
LLC-LDM	no. last-level cache load-misses
LLC-STM	no. last-level cache store-misses
etc.	
BR / INSTR	regularity of the code
BR-M / BR	loop predictability
BR-M / INSTR	relative induced pipeline stalls
L1-LD / INSTR	data accesses (read)
L1-STM / L1-ST	data regularity / evictions (write)
LLC-LD / INSTR	global data usage (read)
L1-LD / LLC-LDM	data locality
etc.	

3.2.3 Reconstruction

If only one prediction candidate (B profiled on P and T) is chosen to predict the workload A onto T, a simple factorized transformation is used to either generate the sequential execution time $t(1)$ or elements sc_m of the scalability feature vector $\vec{sc}$ (see Figure 8). Because the predictions of $t(1)$ and $\vec{sc}$ follow the same rules ($t(1)$ can also be interpreted as an element of $\vec{sc}$), we henceforth only describe the transformation for scaling vector elements

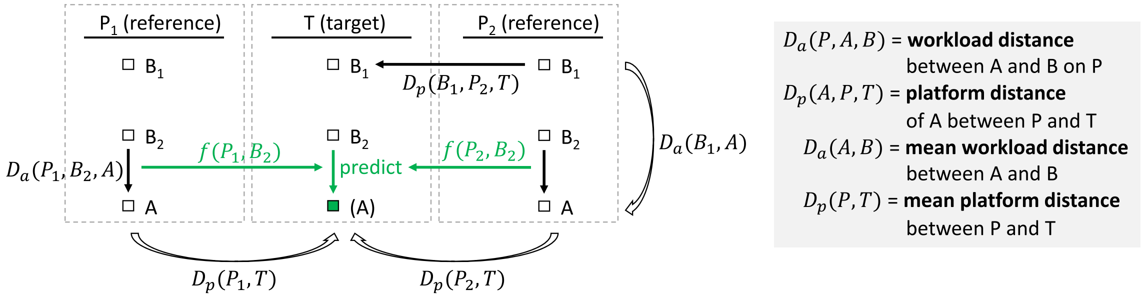

P$_1$ (reference)
T (target)
P$_2$ (reference)
B$_1$
B$_1$
B$_1$
$D_p(B_1, P_2, T)$
B$_2$
B$_2$
B$_2$
$D_a(P_1, B_2, A)$
$f(P_1, B_2)$
predict
$f(P_2, B_2)$
A
(A)
A
$D_a(B_1, A)$
$D_p(P_1, T)$
$D_p(P_2, T)$
$D_a(P, A, B)$ = workload distance between A and B on P
$D_p(A, P, T)$ = platform distance of A between P and T
$D_a(A, B)$ = mean workload distance between A and B
$D_p(P, T)$ = mean platform distance between P and T

Figure 8 Prediction scenario from two reference platforms and two benchmarks. Platform and algorithm individual distances and mean distances are marked.

sc_m. In order to predict an sc_m, a transformation factor f between A and B running on P is taken to reconstruct the target value as follows.

$$sc_m(T, A) = sc_m(T, B) \cdot f \text{ , with } f = \frac{sc_m(P,A)}{sc_m(P,B)} \tag{21}$$

In case that multiple candidates are selected, we use a kind of interpolation, which not allows extrapolations, such that the variations in the characteristics of benchmarks must also include extreme parameter ranges. For each parameter to be predicted, we calculate a series of transformation factors $f(C)$, one for each candidate C. Afterwards, factors can be weighted by their inverse distances ($w = 1 - \mathcal{D}$) to the target and different weights have been tested (mean or individual workload, platform, or tuple distance). Because smaller distanced candidates potentially have more similarities in their scalability, we added a further normalizing penalty $r(w)$, that decreases the influence of more distanced candidates (linearly, square, or cubic) between 100% at minimum distance w_{min} and 0% at maximum distance w_{max}.

$$r(w) = \left(\frac{w - w_{min}}{w_{max} - w_{min}}\right)^k, \quad k \in \{0, 1, 2, 3\} \tag{22}$$

Consequently, a final prediction of an element sc_m of the scalability feature vector (or the sequential execution time $t(1)$) can be calculated out of a number of candidates C with

$$sc_m(T, A) = \frac{\sum_C [sc_m(C) \cdot f(C) \cdot r(1 - \mathcal{D}(C))]}{\sum_C r(1 - \mathcal{D}(C))}. \tag{23}$$

Table 3 Enabled prediction scenarios through different information in database of benchmarks B and the workload A that is to be predicted ($t(1)$, $\vec{pc}(1)$: sequential, $\vec{sc}$: scaling trend).

Scenario	Benchmarks $(P+T, B)$	Workload (P, A)	Predictable performance
migration predictions $(A : P \to T)$:			
1. $(t(1))$:	$t(1), \vec{pc}(1)$	$t(1), \vec{pc}(1)$	$\sim t_{T,A}(1)$
2: $(\vec{sc})$:	$\vec{sc}$	$\vec{sc}$	$\vec{sc}_{T,A}$
3. $(t(n))$:	$t(1), \vec{sc}$	$t(1), \vec{sc}$	$t_{T,A}(1), \vec{sc}_{T,A}$
parallelization prediction $(t(1) \to t(n))$:			
4. $(t(n))$:	$t(1), \vec{sc}$ + class.	$t(1), \vec{pc}(1)$	$\sim \vec{sc}_{P/T,A}$

In the current implementation of the prediction workflow, the number of maximum used candidates is set as a fix number, while there can be database constellations that naturally add further limits to the set of available candidates. For instance, if the parallel performance of a workload is to be predicted on a platform that features hyperthreads, but most candidates in database do not contain ht-parameters, these candidates can therefore not serve as prediction candidates. The choice between using less or more candidates (platforms as well as workloads) is a trade-off between two positions, which will be discussed in Section 4.

3.3 Database

As prediction accuracy relies on database quality, the basic idea is to have a large set of varying benchmarks for more likely providing close prediction candidates. All benchmarks and required libraries like MPAL [25] for parallelization, profiling, and parameter extraction and PAPI [29] for performance counter usage are compressed in a tar-ball. A Makefile automatically extracts and compiles libraries, executes benchmarks one by one, and compresses all results in addition to the extracted characteristics ($t(1)$ and $\vec{sc}$) in a new tar-ball. The composition of benchmarks exhibits implementations with different characteristics and parallelization strategies. We use fix input-data sizes and each parallel section is considered as individual workload. While we used real-world algorithms from the automotive area like stereo vision, or optical flow, we also added standard benchmarks like speckle noise reduction, or k-d tree implementations. Each of which exhibits multiple parallel sections, which are considered as separate workloads. The current set of 17 different workloads includes varying parallelizations like *domain decomposition*, *nested and recursive spawns*, and *wavefront* *parallelism*, as well as different *lock and synchronization* situations.

Figure 9 illustrates the of varying workloads from our benchmark set profiled on different platforms using four selected parameters: work imbalance w, increase of redundancy R, branches per instruction *BR/INSTR*, and level-1 cache-misses per cache-load *L1-LDM/L1-LD* (i.e., miss-rate). Generally, some profiles show good scaling trends and some show poor speedups, while all workloads are influenced by effects of varying magnitudes and origins on different platforms, as discussed in the following (incomplete) list:

- In the example in Figure 9, branches per instructions vary between workloads due to their algorithmic demands and regularity of the code. In fact, the relative number *BR/INSTR* mostly decreases for the Cortex-A15, because its ARM instruction-set requires more instructions for some calculations than x86 processors, which increases the total number of issued instructions.
- The Cortex-A15 processor generally shows a higher susceptability to work imbalance in comparison to the i9-9900K and the Xeon Silver-4114 cores. An analysis of the scaling trends shows a sudden increase of the work imbalance especially at the use of the maximum number of availabe cores, which indicates high influences of the Linux kernel thread that blocks ressources of one core.
- The L1-LDM/L1-LD ratio is different between workloads on the one hand. On the other hand, this measure always slightly decreases for the Cortex-A15 core, which shows the variances of the cache implementations in hardware. Effects, like prefetching internals, set-associativity respectively eviction strategies, or varying compiler optimizations to reduce miss rates, can cause this behavior. The same effect was also measured in [30] in a comparison a Cortex-A53 and an i7 920.

In general, the collection of workloads and the scaling parameter representation can indeed denote platform- and workload-characteristics and interference between software demands and hardware capabilities. The current composition of workloads in database covers different characteristics as well as good and poor scaling trends. All of these effects are represented in the scaling parameters, such that well scaling and not parallelizable workloads can be predicted. Adding microbenchmarks that stress certain hardware features (e.g., inter-core communication) could enhance the prediction, provide information on theoretical performance peaks (e.g., in a roof-line model), and thereby point out bottlenecks and optimization potentials.

The prediction accuracy highly depends on the quality of the database. Therefore, we currently work on the creation of a comprehensive database with significant benchmarks and microbenchmarks as well as on mechanisms to automatically eliminate profile outliers to improve the prediction's robustness.

4 Evaluation

Many prediction error numbers presented in literature lack of comparability, as results are interpreted differently, which was also elaborated by Hoefler and Belli [31] for parallel

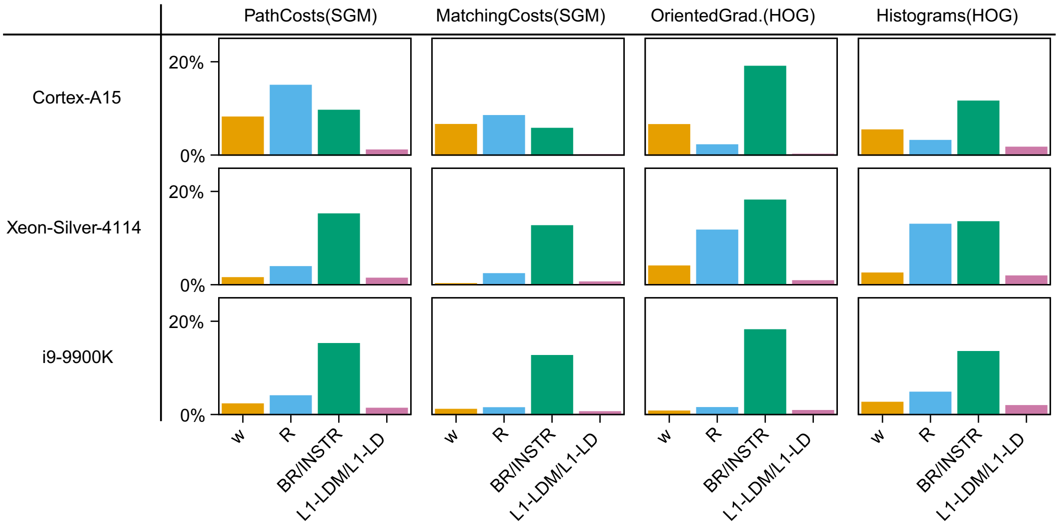


Figure 9 Characteristics of the parallel runtime behavior of selected workloads from database (parallel section of *HOG* pedestrian detection [27] and *SGM* stereo vision [28]), which are profiled on different processors. Scalability parameters w and R denote the percentage increase per core of the available CPU time $t(n) \cdot n$, while the relative performance counter measures are percentage numbers themselves (initial offset b measured at $n = 1$).

benchmarking results. While some authors give absolute errors, others only present correlations of prediction and measured results, or aggregate all predicted parallel sections (positive and negative discrepancies) and only rate the overall error. Therefore, we not only present accuracy of our approach as geometric error (deviations between predicted and measured times $t(n)$ averaged by absolute error), which reflects precision over the entire scaling trend. Additionally, in a case study, we give comparative prediction errors of Exabounds and GEM5.

4.1 Prediction Accuracy

To evaluate the prediction, we set up a database of varying platforms from three different fields (server, desktop, and embedded) of different ages and microarchitectures (Table 4), to demonstrate the prediction across similar and widely dissimilar platforms: old and new as well as varying microarchitecture-types. We evaluated the prediction in three steps: $t(1)$, $\vec{sc}$ (reconstructed parallelization speedup), and full prediction ($t(1)$ plus $\vec{sc}$). Since predictions of $t(1)$ and $\vec{sc}$ can have positive or negative deviations, the error of a full prediction does not represent the aggregation of both, but errors may compensate each other. A prediction is evaluated by eliminating the ground-truth from database to be used for error-calculation after a prediction (1.6 s per prediction), while all profiles in database are tested.

As mentioned before, there is a theoretical trade-off between two options of either using more candidates for an interpolative prediction or using less candidates with the following qualities:

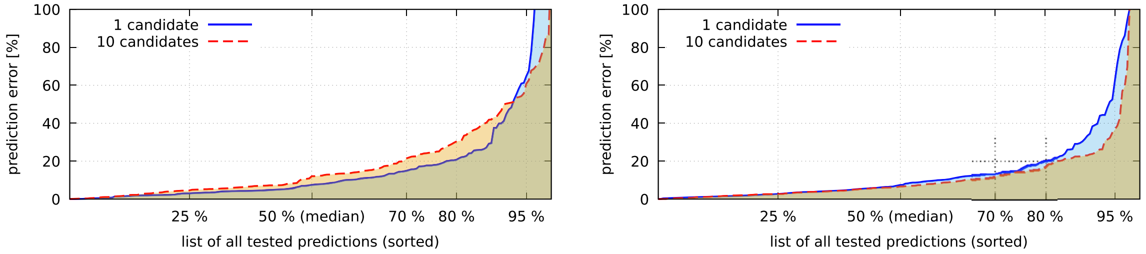
- *More candidates:* This enables an interpolation of more candidates, which on one hand reduces the impact of outlier-candidates and thus reduces the probability of high prediction errors. On the other hand, more candidates with a potentially larger distance need to be taken into account, which generally increases the mean prediction error.
- *Less candidates:* This allows a more precise selection of only few best matching candidates and thus most often returns a very accurate prediction. But, in case of a wrongly selected candidate, its impact on the prediction is even higher, such that outliers can produce a much higher prediction error in total.

To evaluate the impact of both options, we tested the aforementioned database with different configurations, while Figure 10 presents the prediction error of all (sorted) prediction tests for two opposite options: using 1 or 10 candidates. The prediction of $t(1)$ (Figure 10(a)) verifies the theory, as the use of more candidates results in a higher error for most candidates, but maximum errors of outliers are limited in contrast to the use of only 1 prediction candidate. Figure 10(b) shows a different behavior, where the use of more candidates seem to improve the prediction in general. In contrast, to the prediction of $t(1)$, the scalability prediction with multiple candidates uses a cubic distance penalty ($k = 3$), which reduces the influence of more distanced candidates. Because, this prediction approach is especially optimized to predict scalability, calculated distances show better correlations to resulting errors in scalability prediction. Therefore, some outliers that produce incorrect predictions of $t(1)$ are wrongly rated with small distance in the current configuration, which makes the prediction of $t(1)$ slightly less robust, which will be optimized in our future work. This is why candidates are differently weighted in the prediction of $t(1)$ and $\vec{sc}$, as can be seen in the following paragraph. In fact, the optimum configuration appeared to be somewhere in between both extreme configurations, such that we empirically selected the limits somewhere between both trade-off extrema as described in the following.

First, we tested a database composition in which all platforms served as prediction candidates, which increases the candidate count and potentially enables predictions from other platform types (e.g., from a server core to embedded). Then we predicted only inside each platform type (e.g., only desktop). For predicting $t(1)$, we used 5 workloads and 5 platforms to search for candidates, which were rated by their averaged workload distance, but individual platform distances. The transformation factor was weighted (w) by averaged workload distances without extra penalty ($k = 0$). The scalability was predicted by sorting candidates

Table 4 Characterized platforms for prediction evaluation.

CPU	cores	thr.	μ -arch.	word size
Xeon Gold 6148	2 × 20	80	Skylake	64 bit
Xeon E5-2630 v3	2 × 8	32	Haswell	64 bit
Xeon E5-2680	2 × 8	32	Sandy Bridge	64 bit
Opteron 6220	2 × 8	16	Bulldozer	64 bit
Opteron 6172	2 × 12	24	K10	64 bit
Xeon E5620	2 × 4	16	Westmere	64 bit
i9-9900K	8	16	Coffee Lake	64 bit
i5-8500T	6	6	Coffee Lake	64 bit
i7-8550U	4	8	Kaby Lake	64 bit
i5-7300HQ	4	4	Kaby Lake	64 bit
i7-4771	4	8	Haswell	64 bit
i7-3667U	2	4	Ivy Bridge	64 bit
A53 (S905)	4	4	ARMv8	64 bit
A15 (Exynos 5422)	4	4	ARMv7	32 bit
A7 (BCM2836)	4	4	ARMv7	32 bit



(a) Error of the predicted sequential execution time $t(1)$. (b) Error of the predicted scalability trend **sc**.

Figure 10 Prediction errors of all tested predictions with either 1 or 10 candidates sorted by their prediction error.

by tuples of individual workload and platform distances (3 platforms, 10 workloads). Transformation factors were weighted (w) by workload individual distances with cubic penalty ($k = 3$).

Table 5 presents geometric mean and median prediction error, as well as min, max, and 70%, 80%, and 95% percentiles over all predictions. While $t(1)$ is predicted with a geometric mean error of 15.4%, for all platforms in database, the scalability achieves a mean error of 12.7%. Both predictions combined result in a full migration prediction of a parallel workload with a mean error of 19.9%. As can be seen, the database setup influences the prediction accuracy, such that a database of only server cores reaches a mean error of 25.5%, but a prediction with only desktop processors achieves 9.3%. Desktop CPUs show good accuracy, but predictions of server processors suffer from high variances and cores counts, which increases prediction uncertainties. The small number of reference platforms affects the prediction of embedded processors. Overall, in contrast to a naive assumption of an ideal speedup of n (resulting in 217% error in the example in Figure 5), our method can precisely predict varying concurrency influences and bottlenecks on all multicore architectures.

Table 5 Prediction errors of different database-configurations. The 70% percentile denotes that 70% of all predictions return errors lower or equal to the given value, etc)

error [%]	av.	med.	min	max	70%	80%	95%
all: $t(1)$	15.4	9.5	0.0	107.8	15.5	23.7	59.0
all: $\vec{sc}$	12.7	6.7	0.1	195.4	11.5	17.8	38.9
all: full	**19.9**	11.1	0.0	185.6	20.4	29.7	69.8
server: $t(1)$	18.0	10.7	0.0	116.6	20.8	23.7	70.5
server: $\vec{sc}$	16.0	11.0	0.3	101.1	17.7	22.9	57.8
server: full	25.5	16.3	0.2	103.7	29.7	46.4	72.1
desktop: $t(1)$	6.8	4.4	0.2	41.8	6.8	9.9	23.6
desktop: $\vec{sc}$	8.0	4.5	0.1	136.6	7.6	9.8	21.5
desktop: full	9.3	5.3	0.1	129.2	9.4	13.7	29.0
embed.: $t(1)$	28.9	20.4	2.0	112.6	33.9	39.1	108.6
embed.: $\vec{sc}$	3.2	2.4	0.0	16.6	4.1	4.5	11.7
embed.: full	29.0	21.8	0.0	111.2	32.8	39.7	100.8

Besides the given prediction-accuracy evaluation, which already produces good predictions, which verify the MPET method, we analyzed the correlation between the determined distance between candidates and the resulting prediction error. Figure 11 shows the average prediction errors for a number of tested predictions while using candidates of varying algorithmic distances $\mathcal{D}_a$ for either using only one candidate or an interpolation of multiple candidates (same configuration as above). Considering single-candidate predictions in Figure 11 shows, that there is a clear correlation between the determined distance and the resulting prediction error. The relation can be described as: *the lower the determined distance, the higher the probability of a good prediction*. This correlation proves the significance of the used scalability features and thereof derived distances. Furthermore, the use of the presented interpolation technique enhances the prediction even at the use of candidates with higher distances, as can be seen in Figure 11.

As most influencing parameters on the parallel runtime behavior, we determined the redundancy $R(n)$ ($\sim$ 50%),

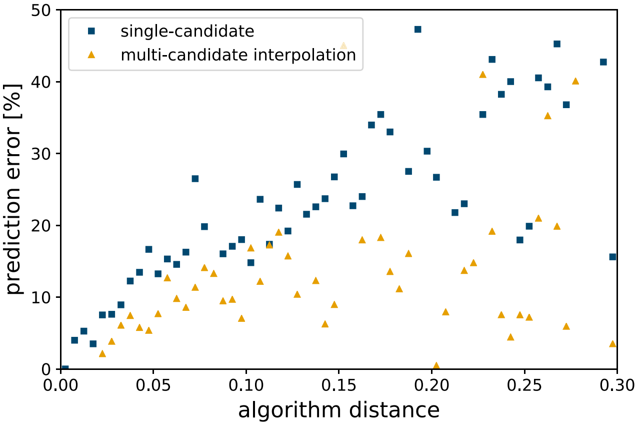


Figure 11 Average prediction error for candidates of varying algorithmic distances $\mathcal{D}_a(P, A, B)$. (Distances of predictions with multiple candidates have been averaged with candidate-individual weights.)

work imbalance $w(n)$ ($\sim$ 35%), and task distribution $d(n)$ ($\sim$ 5%) on average. In addition to the scaling parameters $R(n)$, $w(n)$, etc., we found the L1 cache accesses and misses, as well as LLC cache accesses and misses as most significant descriptive features for calculating distances.

4.2 Case Study

In a case study, which has been published as an extended version in [32], we predicted the performance of a Xilinx Zynq Ultrascale+ EG platform with a virtual prototype in GEM5, an analytic processor model in Exabounds, and our proposed statistical approach. This platform exhibits four ARM Cortex-A53 cores running at 1.2 GHz and 4 GB RAM. We used all parallel sections (S_i) of the linear algorithmic pipelines of the *Semi-Global Matching* (SGM) [28] for stereo-vision and the *Histograms of Oriented Gradients* (HOG) [27] algorithm for pedestrian detection as target workload.

ARM provides the ARM-HPI CPU-models [33] for an ARMv8 architecture as ready-to-use simulation components for GEM5, which only need to be configured with timing parameters that we extracted from benchmarks and data sheets. In spite of the provided processor models, the creation of a virtual prototype takes some effort and requires experienced developers. We set up a similar OS-environment in the simulator and on the original board. Simulating all algorithmic stages lasts over five hours, while we used the MPAL profiling inside the simulation to extract timings. The prediction with IBM Exabounds also necessitates a parameterization of the target platform, such that we used the same configuration as we used for the virtual platform. In addition, an architecture-independent profile of the algorithms was created in a 6 hours evaluation with IBM Pisa (using LLVM). Feeding both independent descriptions the platform configuration and workload profiles to the Exabounds tool, the actual prediction only takes seconds. For predicting the performance with the proposed statistical approach, we make use of the database of prediction candidates and add benchmark profiles of the target platform. We also compared these methods against a naive approach in which another platform is used as reference, which exhibits the same processor (A53), and the measured performance is simply scaled by frequencies of both, reference and target platform. Because, this naive approach resulted in much higher prediction errors, it is not further discussed in the following.

We observe absolute deviation in percent per parallel section as well as the mean error over all algorithmic stages (see Figure 12). The presented MPET approach predicts the performance with a mean error of 19.0% between both reference methods. The costly simulation of GEM5 produces most accurare predictions with an average error of 16.1%, while the approximate analytical model in Exabounds produces the highest average error of 25.3%. All presented error numbers in Figure 12 can be positive or negative deviations, such that all three approaches neither under- nor over-estimate the performance. Furthermore, all three methods produce some accurate as well as erroneous predictions. To summarize the results, we visualized the mean prediction error, modeling effort (for creating a prediction environment), and prediction time (simulation or mathematical model scoring) in Figure 13.

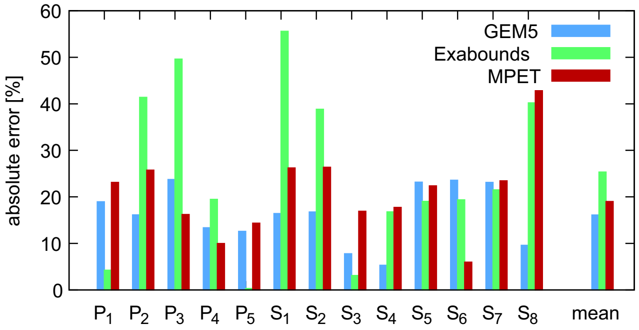


Figure 12 Prediction errors of all stages of a HOG pedestrian detection (P_x) and an SGM stereo-vision (S_x) predicted with GEM5, Exabounds, and MPET.

4.3 Exemplary Predictions

Figure 14(a) presents the scaling curve of an algorithmic stage S_7, which was predicted as target workload A onto a Ryzen 2400G target processor. The scaling curves of both candidate workloads (here: S_1 and S_3 of SGM written as B_1 and B_2) profiled on the target platform need to be stored in database already before prediction. Profiles of A and B_i on two reference platforms (here: i9-9900K and E5-2630_v3 written as P_1 and P_2) were used to calculate the transformation factors $f(P_1/P_2, B_1/B_2)$ to firstly predict the sequential execution time $t(1)$ and the scaling trend $\vec{sc}$

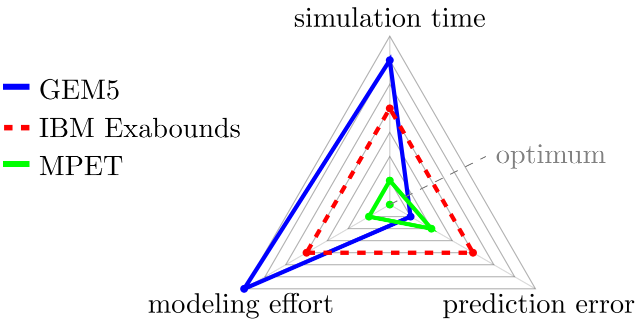
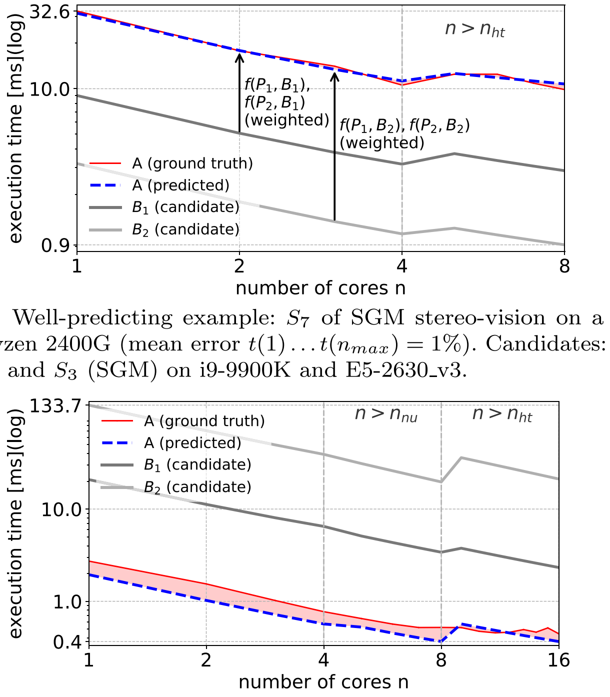


Figure 13 Classification of prediction tools used in the case study.



(a) Well-predicting example: S_7 of SGM stereo-vision on a Ryzen 2400G (mean error $t(1)\dots t(n_{max}) = 1\%$). Candidates: S_1 and S_3 (SGM) on i9-9900K and E5-2630_v3.

(b) Erroneous predition: P_3 of HOG pedestrian detection on a Xeon E5620 (mean error $t(1)\dots t(n_{max}) = 18\%$). Candidates: S_2 (HOG) and SRAD on Opteron-6172 and Opteron-6220

Figure 14 Two exemplary predicted scaling curves with visualization of two selected, most relevant candidated.

subsequently. In this example, the prediction of $t(1)$ and of the parallel runtime behavior $\vec{sc}$ shows only minimum deviations with a mean error over the trend $t(1)\dots t(n_{max})$ of only 1%. It is worth noting, that even the hyperthreading effects (for $n > n_{ht}$ with $n_{ht} = 4$) could be predicted very accurately. However, at the maximum number of cores $n = n_{max} = 8$, the prediction error increases up to 9%, which could be an influence of the kernel thread (prediction: $t(1) = 31.6$ ms, $t(n_{max}) = 10.8$ ms, measured: $t(1) = 32.6$ ms, $t(n_{max}) = 9.9$ ms).

In the second example (shown in Figure 14(b)), the algorithmic stage P_3 of HOG pedestrian detection was used as target workload A to be predicted onto a Xeon E5620 as target platform. This prediction, which used S_2 (HOG) and SRAD as B_1 and B_2 measured on an Opteron-6172 and an Opteron-6220, resulted in a higher prediction error of $t(1)$ of 28%. For small numbers of cores, the scaling behavior is predicted accurately, but the prediction assumes more influences of the hyperthreading and NUMA-effects as they were measured in the ground-truth for higher numbers of cores. This results in a mean error over the trend $t(1)\dots t(n_{max})$ of 18%, while the prediction of $t(n_{max})$ shows an error of 25% (prediction: $t(1) = 1.95$ ms, $t(n_{max}) = 0.37$ ms, measured: $t(1) = 2.72$ ms, $t(n_{max}) = 0.45$ ms).

4.4 Discussion and Future Work

In its current version, the prediction process respects all elements in the scaling vector with the same influence on the resulting distance. Also, candidates are only weighted by their distance, regardless of their number of valid elements in the scaling vector, which indeed influences the significance of the generated distance. In future work, it is to be evaluated, whether additional weights can improve the prediction (statically or dynamically adapted to the current database).

– *Candidate weights:* As mentioned before, profile outliers and the quality of the fitting process influences parameter constellations and the prediction error thereafter. Therefore, the error after reconstructing the original profile from fitted parameters needs to be taken into account and be used as confidence level in order to decrease the influence of candidates of bad certainty in their scaling behavior and according distance metrics.
– *Parameter weights:* In current state, after normalization, distances are calculated by using all elements in $\vec{sc}$ with the same weight and influence. Depending on many characteristics of the database composition, like parallelization strategy and resulting parameters' absolute influence on the scaling behavior, these weights could be adapted in order to emphasize the importance of certain parameters.

The presented concept demonstrated the feasibility of a statistical cross-architecture prediction for multicore applications even with small databases. Because only few reference platforms are required to setup an initial prediction environment that allows a prediction of the parallel performance in a sufficient precision, this method can help to support developers with information on potential bottlenecks and parallelization hints. In our ongoing research, we plan to integrate confidence levels for instance by making use of model-fitting errors, used candidates' distances, or number of used candidates. Providing a confidence with each prediction allows users of this prediction method to decide to either accept a prediction within the current range and trust-level or to add more reference platforms to improve the significance of predictions. Also, confidence levels of multiple prediction results, each evaluated with different configurations (as for instance shown in Figure 10), could be combined to extract benefits of each configuration: reduced mean error plus eliminated outliers. In addition, microbenchmarks will be used in our future work to determine the theoretical peak performance. Using these measures in a roof-line model, helps to classify workloads by their limitations (computation bound, communication bound, or memory bound) and identify optimization potentials.

The use of a too small database, which does not offer a sufficient number of candidates, leads to a more likely selection of outliers as candidates. Therefore, we aim at a comprehensive benchmark set, which covers most parallelization strategies and varying scaling trends. Our vision is to create a shared online database, where the benchmark set and the prediction method can be downloaded to be used to characterize more platforms and new results can be uploaded to be added to the public database. This can help to offer a comprehensive and growing database to the community.

5 Conclusion

This paper presents a statistical cross-architecture prediction methodology MPET, which is based on abstract characteristics that describe the parallel runtime behavior. It enables a migration prediction of a parallel or sequential workload from one platform to another and a scalability prediction of a not yet parallelized implementation. Having reference workloads of different parallelization strategies available, a scalability of a sequential implementation can be estimated even in early development phases. Fully virtual workloads and virtual architectures can be modeled by modifying the abstract scaling parameters, which directly form the scaling trend. In spite of the low mathematical model complexity compared with other prediction methods, we achieve better results than analytical models like Exabounds. While a large database with potentially closely located prediction candidates can enhance the prediction accuracy, even the relatively small database used in our evaluation generated competitive precisions $< 20\%$ mean error. And a growing benchmark set will further increase the accuracy. In contrast to the reference methods that we used in the case study, our method neither requires time-consuming simulations, nor any user input for modeling and thus no developer training.

Funding Open Access funding enabled and organized by Projekt DEAL.

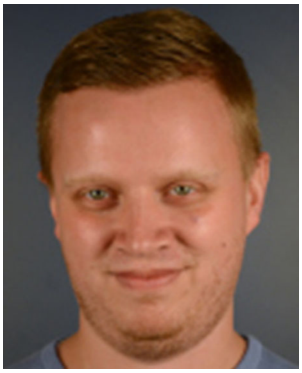
References

1. Payá Vayá, G., & Blume, H. (2017). Towards a common software/hardware methodology for future advanced driver assistance systems: River Publishers.
2. Varbanescu, A., Van Nieuwpoort, R., Hijma, P., E. Bal, H., Badia, R.M., & Martorell, X. (2017). *Programming models for multicore and many-Core computing systems*, (pp. 29–58). New York: Wiley.
3. Nugteren, C., & Corporaal, H. (2012). The boat hull model: Enabling performance prediction for parallel computing prior to code development. In *Proceedings of the 9th conference on computing frontiers* (pp. 203–212). ACM.
4. Teich, J. (2012). Hardware/Software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, *100*(Special Centennial Issue), 1411–1430. https://doi.org/10.1109/JPROC.2011.2182009.
5. Madougou, S., Varbanescu, A., De Laat, C., & van Nieuwpoort, R. (2016). The landscape of GPGPU performance modeling tools. *Parallel Computing*, *56*, 18–33.
6. Gill, D., & Pandey, H.M. (2015). Approaches for software performance modelling, cloud computing and openstack. *Computer Applications*, *119*, 31–35.
7. Menard, C., Castrillón, J., Jung, M., & Wehn, N. (2017). System simulation with gem5 and SystemC: The keystone for full interoperability. In *Int. Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (pp. 62–69). IEEE.
8. Imperas Software (2019). Open virtaul platforms – the source of Fast Processor Models & Platforms. http://www.ovpworld.org/.
9. Software Freedom Conservancy (2019). QEMU. https://www.qemu.org/.
10. Cadence (2019). Virtual System Platform – An open, connected, and scalable virtual prototyping solution. https://www.cadence.com/en_US/home/tools/system-design-and-verification/software-driven-verification/virtual-system-platform.html.
11. Butko, A., Gamatie, A., Sassatelli, G., Torres, L., & Robert, M. (2015). Design Exploration for next Generation High-Performance Manycore On-chip Systems: Application to big.LITTLE Architectures. In *Annual Symposium on VLSI* (pp. 551–556). IEEE.
12. Marin, G., & Mellor-Crummey, J. (2004). Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. In *SIGMETRICS Performance Evaluation Review, 32* (pp. 2–13). ACM.
13. Carlson, T.E., Heirman, W., Eyerman, S., Hur, I., & Eeckhout, L. (2014). An Evaluation of High-Hevel Mechanistic Core Models. *ACM Transactions Architectures and Code Optimization (TACO)*, *11*, 28.
14. Xu, C., Chen, X., Dick, R. P., & Mao, Z.M. (2010). Cache Contention and Application Performance Prediction for Multi-Core Systems. In *International Symposium on Performance Analysis of Systems Software (ISPASS)* (pp. 76–86). IEEE.
15. Van den Steen, S., De Pestel, S., Mechri, M., Eyerman, S., Carlson, T., Black-Schaffer, D., Hagersten, E., & Eeckhout, L. (2015). Micro-Architecture Independent Analytical Processor Performance and Power Modeling. In *International Symposium on Performance Analysis of Systems Software (ISPASS)* (pp. 32–41). IEEE.
16. De Pestel, S., Van den Steen, S., Akram, S., & Eeckhout, L. (2018). RPPM: Rapid Performance Prediction of Multithreaded Applications on Multicore Hardware. *Computer Architecture Letters*, *17*, 183–186.

17. Jongerius, R., Anghel, A., Dittmann, G., Mariani, G., Vermij, E., & Corporaal, H. (2018). Analytic Multi-Core Processor Model for Fast Design-Space Exploration. *Transactions on Computers*, *67*, 755–770. IEEE.
18. Lopez-Novoa, U., Mendiburu, A., & Miguel-Alonso, J. (2015). A Survey of Performance Modeling and Simulation Techniques for Accelerator-Based Computing. *Transactions on Parallel and Distributed Systems*, *26*, 272–281. IEEE.
19. Wu, G., Greathouse, J. L., Lyashevsky, A., Jayasena, N., & Chiou, D. (2015). GPGPU Performance and Power Estimation Using Machine Learning. In *International Symposium on High Performance Computer Architecture (HPCA)* (pp. 564–576). IEEE.
20. Madougou, S., Varbanescu, A. L., de Laat, C., & van Nieuwpoort, R. (2014). An Empirical Evaluation of GPGPU Performance Models. In *European Conference on Parallel Processing* (pp. 165–176). New York: Springer.
21. Ardalani, N., Lestourgeon, C., Sankaralingam, K., & Zhu, X. (2015). Cross-architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance. In *International Symposium on Microarchitecture* (pp. 725–737). ACM.
22. Hoste, K., & Eeckhout, L. (2007). Microarchitecture-Independent Workload Characterization. *Micro*, *27*, 63–72. https://doi.org/10.1109/MM.2007.56.
23. Meng, J., Morozov, V. A., Kumaran, K., Vishwanath, V., & Uram, T.D. (2011). GROPHECY: GPU Performance Projection from CPU Code Skeletons. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11* (pp. 14:1–14:11). ACM.
24. Wang, Y., Lee, V., Wei, G.-Y., & Brooks, D. (2019). Predicting New Workload or CPU Performance by Analyzing Public Datasets. *ACM Transactions on Architecture and Code Optimization*, *15*, 53:1–53:21. https://doi.org/10.1145/3284127.
25. Arndt, O. J., Lefherz, T., & Blume, H. (2015). Abstracting Parallel Programming and Its Analysis Towards Framework Independent Development. In *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)* (pp. 96–103). IEEE.
26. Bovet, D. P., & Cesati, M. (2006). *Understanding the Linux Kernel*, 3rd edn. Cambridge: O'Reilly.
27. Arndt, O. J., Becker, D., Giesemann, F., Payá-Vayá, G., Bartels, C., & Blume, H. (2014). Performance Evaluation of the Intel Xeon Phi Manycore Architecture Using Parallel Video-Based Driver Assistance Algorithms. In *International Conference Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE.
28. Arndt, O. J., Becker, D., Banz, C., & Blume, H. (2013). Parallel Implementation of Real-Time Semi-Global Matching on Embedded Multi-Core Architectures. In *International Conference Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)* .IEEE.
29. Terpstra, D., Jagode, H., You, H., & Dongarra, J. (2010). Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing* (pp. 157–173). New York: Springer.
30. Patterson, D. A., & Hennessy, J. L. (2016). *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Burlington: Morgan Kaufmann.
31. Hoefler, T., & Belli, R. (2015). Scientific Benchmarking of Parallel Computing Systems: Twelve ways to tell the masses when reporting performance results. In *International Conference on High Performance Computing, Networking, Storage and Analysis* (pp. 73:1–73:12). ACM.
32. Lüders, M., Arndt, O. J., & Blume, H. (2019). Multicore Performance Prediction – Comparing Three Recent Approaches in a Case Study. In *European Conference on Parallel Processing Workshops*. New York: Springer.
33. Arm Research (2019). Research Enablement Kits. https://developer.arm.com/solutions/research/research-enablement-kits.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Oliver Jakob Arndt received his Dipl.-Ing. in electrical engineering in 2013 from the Leibniz University in Hanover, Germany. Since then, he worked for the Institute of Microelectronic Systems at the Leibniz University Hanover as a PhD student and research engineer. His work focused on parallel programming methodologies and modeling of multicore runtime behavior with specialization on camera based driver-assistance systems. His latest research investigated the prediction of multicore performance in the context of (heterogeneous) parallel architectures for autonomous driving systems, which was part of the collaborative research project "PARIS" funded by the German ministry of education and research (Bundesministerium für Bildung und Forschung).



Matthias Lüders received his M.Sc. degree in electrical engineering in 2018 from the Leibniz University in Hanover. Since March 2019, he is a Ph.D. student and research engineer at the Institute of Microelectronic Systems at the Leibniz University in Hanover. His research focuses multicore performance prediction methods in the context of (heterogeneous) parallel architectures for autonomous driving systems, which was part of the collaborative research project "PARIS" funded by the German ministry of education and research (Bundesministerium für Bildung und Forschung).

Christoph Riggers received his B.Sc. degree in electrical engineering from the Leibniz University of Hanover, Germany in 2017. His bachelor thesis dealt with the implementation and optimization of synthetic aperture radar image data generation algorithms on a heterogeneous multicore CPU. Since then, he has been working as a research assistant at the Institute of Microelectronic Systems at the Leibniz University Hanover and worked on multicore optimization of a neural network used for autonomous driving. He is currently working on his master thesis on multicore performance prediction.

Holger Blume received his Dipl.-Ing. degree in electrical engineering from the University of Dortmund, Germany in 1992. There he also finished his PhD on nonlinear fault tolerant interpolation of intermediate images in 1997. From 1998 to 2008 he worked as a senior engineer for the Chair of Electrical Engineering and Computer Systems at the RWTH Aachen University. There he finished his habilitation degree on design space exploration for heterogeneous architectures in 2008. In July 2008 he was appointed professor for architectures and systems at the Institute of Microelectronic Systems at Leibniz University Hanover. Prof. Blume is chairman of the German chapter of the IEEE Solid State Circuits Society. His research interests are in design space exploration for algorithms and architectures for digital signal processing. Main application fields, which are addressed, are biomedical applications and driver assistance systems.