DOI: 10.14529/jsfi200204

# Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors

*Johannes Hofmann*[1], *Christie L. Alappat*[2], *Georg Hager*[2], *Dietmar Fey*[1], *Gerhard Wellein*[2]



We propose several improvements to the execution-cache-memory (ECM) model, an analytic performance model for predicting single- and multicore runtime of steady-state loops on server processors. The model is made more general by strictly differentiating between application and machine models: an application model comprises the loop code, problem sizes, and other runtime parameters, while a machine model is an abstraction of all performance-relevant properties of a processor. Moreover, new first principles underlying the model's estimates are derived from common microarchitectural features implemented by today's server processors to make the model more architecture independent, thereby extending its applicability beyond Intel processors.

We introduce a generic method for determining machine models, and present results for relevant server-processor architectures by Intel, AMD, IBM, and Marvell/Cavium. Considering this wide range of architectures, the set of features required for adequate performance modeling is surprisingly small.

To validate our approach, we compare performance predictions to empirical data for an OpenMP-parallel preconditioned CG algorithm, which includes compute- and memory-bound kernels. Both single- and multicore analysis shows that the model exhibits average and maximum relative errors of 5 % and 10 %. Deviations from the model and insights gained are discussed in detail.

*Keywords: microarchitecture comparison, Intel, AMD, ARM, IBM, performance evaluation, performance modeling, analytic modeling, execution-cache-memory model.*

## Introduction

The architectural differences among processor models of different vendors (and even among models of a single vendor) lead to a diverse server-processor landscape in the high-performance computing market. On the other hand, several analytic performance models, such as the Roofline model [10, 25] and the execution-cache-memory (ECM) model [6, 13], show that many relevant performance features can be described using a few key assumptions and a small set of numbers such as bandwidths and peak execution rates. In this work we introduce a structured method of establishing and describing those assumptions and parameters that best summarize the features of a multicore server processor. It has satisfactory predictive power in terms of performance modeling of (sequences of) steady-state loops with regular access patterns but is still simple enough to be carried out with pen and paper. The overarching goal is to allow comparisons among microarchitectures not based on benchmarks alone, which have narrow limits of generality, but based on abstract, parameterized performance models that can be used to attribute performance differences to one or a few parameters or features. As a consequence, reasoning about code performance from an architectural point of view becomes rooted in a scientific process.

### Main contributions

We describe an abstract workflow for predicting the runtime and performance of sequential and parallel steady-state loops (or sequences thereof) with regular access patterns on multicore

[1]Friedrich-Alexander-University Erlangen-Nuremberg, Germany
[2]Erlangen Regional Computing Center, Germany

server CPUs. The core of the method is an abstract formulation of the ECM model, which is currently the only analytic model capable of giving accurate single- and multicore estimates.

We show that a separation between the *machine model*, which contains hardware features alone, and the *application model*, which comprises loop code and execution parameters, is possible with some minor exceptions.

We describe a formalized way to establish a machine model for a processor architecture and present results for Intel Skylake SP and, for the first time, for AMD Epyc, IBM Power9, and Marvell/Cavium ThunderX2 CPUs. The degree of data-transfer overlap in the memory hierarchy is identified as a key parameter for the single-core in-memory performance of data-bound code.

The feasibility of the approach is demonstrated by predicting runtime and performance of a preconditioned conjugate-gradient (PCG) solver and comparing estimates to empirical data for all investigated processors. ECM predictions for the AMD, Cavium, and IBM CPUs have not been published before.

**Outline**

This paper is structured as follows. In Section 1 we detail our testbed and methodology. Section 2 describes, in general terms, our modeling approach including application model, machine model, and the modeling workflow. Section 3 shows how machine models can be constructed by analyzing data from carefully chosen microbenchmarks and gives results for the four CPU architectures under consideration. In Section 4 we validate the model by giving runtime and performance predictions for a PCG solver and comparing them to measurements. Finally, Section 5 puts our work in the context of existing research and Section 5 summarizes and concludes the paper.

## 1. Methodology and Testbed

In this section we point out some relevant high-level properties, while details will be discussed later. Note that we generally take care to run the optimal instruction mix for all benchmark kernels (i.e., using the most recent instruction sets available on the hardware at hand, with appropriate unrolling in place to enable optimal instruction-level parallelism). Compiler peculiarities are commented on where necessary. To minimize interference from the operating system, NUMA balancing was disabled. Transparent huge pages were used by default. Measurements were carried out on repeated loop traversals so timer resolution was not an issue. Run-to-run variations were small (generally below 2 %) and will thus not be reported.

An overview of the investigated processors is provided in Tab. 1. The AMD Epyc 7451 (EPYC) has a hierarchical design comprising four ccNUMA nodes per socket and six cores per domain. L3 cache segments of 8 MiB each are shared among the three cores of a core complex (CCX). The Uncore of the processor (i.e., the L3 cache, memory interface, and other I/O circuitry) is clocked at a fixed 2.66 GHz. Although the cores support the AVX2 instruction set, 32-byte (B) wide SIMD instructions are executed in two chunks of 16 B by only 16-B wide hardware, so that an effective SIMD width of 16 B applies.

Although the Intel Xeon Skylake Gold 6148 (SKL) has a base core frequency of 2.4 GHz and a wide range of Turbo settings, we fix the clock speed to 2.2 GHz in all our experiments in order to avoid the automatic clock-speed reduction when running AVX-512 code [16]. The AVX-512 SIMD extensions were introduced with the SKL architecture and provide 64-B wide vector registers and execution units. The Uncore frequency is set to its nominal value of 2.4 GHz.

**Table 1.** Key specifications of testbed machines

| Microarchitecture | Zen (EPYC) | Skylake-SP (SKL) | Vulcan (TX2) | POWER9 (PWR9) |
|---|---|---|---|---|
| Chip Model | Epyc 7451 | Gold 6148 | ThunderX2 CN9980 | 8335 GTX EP0S |
| Supported core freqs | 1.2–3.2 GHz | 1.2–3.7 GHz | 2.2–2.5 GHz | 2.8–3.8 GHz |
| De-facto freq. | 2.3 GHz | 2.2 GHz | 2.2 GHz | 3.1 GHz |
| Supported Uncore freqs | 2.66 GHz | 1.2–2.4 GHz | 1.1 GHz | N/A |
| Cores/Threads | 24/48 | 20/40 | 32/256 | 22/88 |
| SIMD extensions | AVX2 | AVX-512 | NEON | VSX-3 |
| L1 cache capacity | 24×32 KiB | 20×32 KiB | 32×32 KiB | 22×32 KiB |
| L2 cache capacity | 24×512 KiB | 20×1 MiB | 32×256 KiB | 11×512 KiB |
| L3 cache capacity | 8×8 MiB | 27.5 MiB | 32 MiB | 110 MiB |
| Memory Configuration | 8 ch. DDR4-2666 | 6 ch. DDR4-2666 | 8 ch. DDR4-2400 | 8 ch. DDR4-2666 |
| Theor. Mem. Bandwidth | 170.6 GB/s | 128.0 GB/s | 153.6 GB/s | 170.6 GB/s |

These choices are not a limitation of generality since all procedures described in this work can be carried out for any clock-speed setting. SKL also features a boot-time configuration option of sub-NUMA clustering (SNC), which splits the 20-core chip into two ccNUMA nodes, each comprising ten cores (while the full L3 is still available to all cores). This improves memory-access characteristics and is thus a recommended operating mode for HPC in our opinion. The last-level cache (LLC) prefetcher was turned on for the same reason.

The Cavium/Marvell ThunderX2 CN9980 (TX2) implements the ARMv8.1 ISA with 128-bit NEON SIMD extensions that support double-precision floating-point arithmetic for a peak performance of two 16-B wide FMA instructions per cycle and core. The 32-core chip runs at a fixed 2.2 GHz clock speed, while the L3 cache runs at half the core speed. The victim L3 cache is organized in 2 MiB slices but shared among all cores of the chip.

The POWER9 processor used for our investigations is part of an IBM 8336 GTX data analytics/AI node. Being an implementation of the Power ISA v3.0, the core supports VSX-3 SIMD instructions, corresponding to 16-B wide vector registers. A 512 KiB L2 cache is shared between each pair of cores. The victim L3 cache is segmented, with eleven slices of 10 MiB each, and each slice can act as a victim cache for others [20].

High-level language code for both the Intel and AMD processors was compiled with the Intel C compiler (version 19.0 update 2). On the Marvell and IBM processors the ARM CLANG (version 19) and the IBM XL C (version 16.1.0) compilers were used, respectively. To get the compiler generate an appropriate instruction mix, the `-O3`, `-xHost`, `-mavx2`, and `-mavx` compiler flags are required for the AMD Epyc processor. For the Intel Skylake processor, the `-O3`, `-xCORE-AVX512`, and `-qopt-zmm-usage=high` flags were used. For the Marvell TX2 processor, the `-Ofast` and `-mtune=native` flag were employed. Finally, for the IBM POWER9 processor, the `-O5`, `-qarch=pwr9`, and `-qsimd=auto` flags were used.

Note that the particular choice of compilers was to some extent arbitrary, because it is not our intention to provide a comprehensive compiler comparison. It must be understood that compilers may fail to produce "optimal" code for a loop, but modeling procedures like the one we show here can be used to pinpoint such deficiencies.

The LIKWID suite [5] version 4.3.3 was used in several contexts: `likwid-pin` for thread-core affinity, `likwid-perfctr` for counting hardware performance events, and `likwid-bench` for low-level loop benchmarking (with customized kernels for TX2 and PWR9). Instruction latency and

throughput were measured using the `ibench` tool [11]. Where compiled code was required, we used the compiler versions and flags indicated in Tab. 1.

## 2. Modeling Approach

Just like the Roofline model, the ECM model is an analytic performance model for streaming loop kernels with regular data-access patterns and a uniform amount of work per loop iteration. Unlike Roofline, however, ECM favors an analytic approach. As a result, the model can give single- and multicore estimates with high accuracy without relying on a large number of measurements. Moreover, the analytic nature enables the evaluation of different hypotheses with respect to a processor's performance behavior by investigating which of them lead to a model that best describes empirical performance, thereby enabling deeper insights than measurement-based approaches such as Roofline. See Section 5 for a more thorough comparison of the models and their predictive powers.

Two major shortcomings of the ECM model concern its loose formulation and the resulting lack of portability: in its current form, the model mixes general first principles and Intel-specific microarchitectural behavior into a set of rules that make it difficult to apply it to other processors. In the following, we untangle the original model: First, several truly general (i.e., microarchitecture-independent) first principles and their rationales are laid out. Next, application and machine models that address code- and microarchitecture-specific properties are covered (in addition, we provide general instructions on how to determine machine models for new microarchitectures in Section 3). Finally, the workflow of the new model is demonstrated.

### 2.1. Model Assumptions

The model assumes that the single-core runtime is composed of different runtime components. These include the time required to execute instructions in the core ($T_{\mathrm{core}}$) and the runtime contributions that result from carrying out the necessary data transfers in the memory hierarchy (e.g., $T_{\mathrm{RegL1}}$ the time to transfer data between the register file and the L1 cache, $T_{\mathrm{L1L2}}$ for L1-L2 transfers, and so on). Depending on the architecture, some or all of these components may overlap. The single-core runtime estimate is therefore derived from the runtime components by putting them together according to the architecture's overlap capabilities.

If no shared resources are involved, single-core performance is assumed to scale linearly with the number of active cores for the multicore estimate. In practice, however, at least one shared resource (the memory interface) will be involved. The model takes conflicts on shared resources into account by modeling contention and the resulting waiting times in an analytical way. In the following, some particularities of modern server processors that simplify runtime modeling are discussed.

Today's server processors typically feature superscalar, out-of-order cores that support speculative execution and implement pipelined execution units. Figure 1 shows the execution of instructions corresponding to a simple vector sum (`C[i]=A[i]+B[i]`) for a data set in the L1 cache on a hypothetical core. The core has a two-cycle latency for add and load instructions. When the loop begins execution, each of the two load units can execute a load instruction. Since there is a two-cycle load latency, inputs for the add instruction will only be available after two cycles. However, due to speculative execution, the core can continue to execute two load instructions from the next loop iterations in each cycle. Once input data is available, the

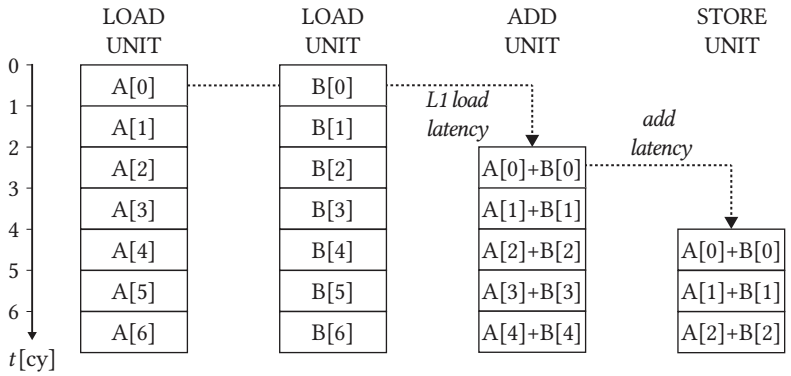



**Figure 1.** Loop execution on a hypothetical core with load and add latencies of two cycles each

core can begin executing an add instruction in each cycle. Eventually, after another two-cycle latency (that of the add instruction), the core can begin executing a store instruction in each cycle. Once this latency-induced wind-up phase of four cycles is complete, instruction latency no longer impacts runtime; instead, the runtime is determined by the throughput of instructions. Although latencies might be higher on real processors, the wind-up phase can be neglected even for short loops with only hundreds of iterations. This leads to one of the key assumptions of the ECM model: in the absence of loop-carried dependencies and data-access delays from beyond the L1 data cache, the runtime of a single loop iteration can be approximated by the time that is required to retire the instructions of a loop iteration. With loop-carried dependencies in place, the inter-iteration critical path is a good estimate of the runtime. Due to speculative execution, load/store instructions are decoupled from the arithmetic instructions of a particular loop iteration. This leads to the further assumption that the time to retire arithmetic instructions and the time to retire load/store instructions can overlap.

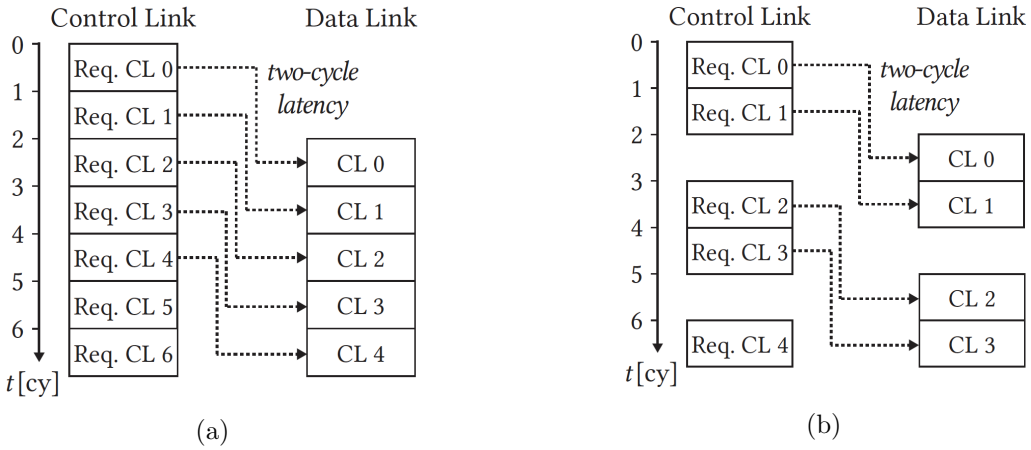



**Figure 2.** (a) Inter-cache data transfers for a design with more than two buffers to track outstanding cache-line (CL) transfers; (b) design with only two buffers

The next set of assumptions concerns data transfers in the memory hierarchy. The relationship between latency and bandwidth is well understood, so most designs typically provide a sufficient number of buffers to track outstanding cache-line transfers to allow for the saturation of the data-transfer link between adjacent cache levels. Figure 2a shows such a design with more than two buffers to track outstanding transfers to hide a two-cycle latency. Sometimes, however, the number of buffers is insufficient, leading to a deterioration of bandwidth. Figure 2b shows a variant with only two buffers: after two cycles, no more transfer-tracking buffers are available,

which prevents the initiation of new transfers. Only after a previous transfer completes and the buffer tracking this transfer is freed can a new transfer request be initiated. As a result, the data link is idle for one cycle, reducing the attainable bandwidth in practice to two-thirds of the theoretical value. On some of the investigated processors this problem can be observed for transfers between the LLC and main memory. This can be attributed to significant latencies caused by the increasingly complex on-chip networks required to accommodate the growing number of cores of modern CPUs.

The model assumes that data links can typically be fully saturated because a sufficient amount of buffers are available and adequate prefetching (be it hardware, software, or both) results in full utilization of these buffers. As a result, runtime contributions of data transfers can typically be calculated by dividing data volumes by the theoretical bandwidths of the corresponding links; the model does, however, include an optional latency penalty to cover edge cases such as the one shown in Fig. 2b. Therefore, the runtime contribution of data transfers between memory hierarchy levels $i$ and $j$ is the sum of the actual data transfer time and an optional latency penalty: $T_{ij} = T_{ij}^{\text{data}} + T_{ij}^{\text{p}}$.

## 2.2. Application Model

An application model condenses all of the code-related information required to give runtime estimates for a particular loop.

It comprises all operations carried out during one loop iteration as well as parameters that influence data transfers in the memory hierarchy. Most prominently, the latter includes the dataset size(s), which determine in which level of the memory hierarchy data resides, yet it may also cover information about blocking size(s) and the scheduling strategy.

## 2.3. Machine Model

Machine models comprise selected key information about processors. Despite being limited to few architectural properties, the data included in machine models is sufficient to give meaningful performance estimates. With respect to scope, the contents of machine models can be separated into two parts: the execution capabilities of cores, and details about the memory hierarchy. In the following, each of the two components is discussed in detail.

The part concerning in-core execution capabilities deals with the cores' properties that determine the runtime contribution of instruction execution. As discussed in Section 2.1, throughput is a key determinant for single-core runtime, so throughput limits (in operations per cycle) of relevant operations are included. To address loop-carried dependencies, latencies for the corresponding instructions must be included. Moreover, the machine model includes information about potential bottlenecks that limit operation throughput: On most architectures, different functional units share the same execution port, which implies that operations associated with units served by the same port cannot cannot begin execution in the same cycle. Finally, most modern core designs have some architectural deficiency that prevents them from fully utilizing the core's load/store units[3].

[3]Most modern cores feature one store and two load units but only have two address-generation units (AGUs), which means that in each cycle only two of the three load/store units can be supplied with memory addresses if complex addressing modes (e.g., base plus scaled offset) are used. In addition to the two-AGU shortcoming, the EPYC's cores have only two data paths between the register file and the L1 cache.

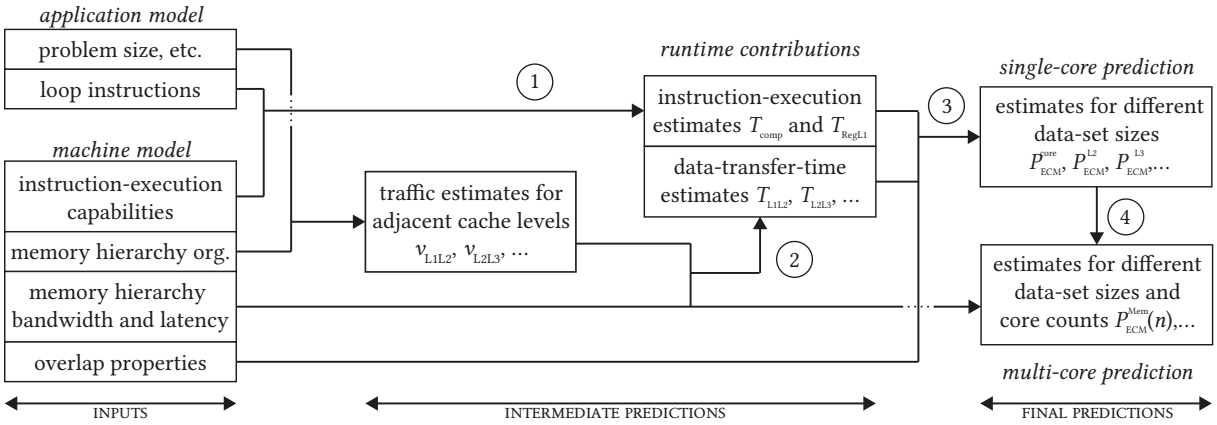



**Figure 3.** Overview of the performance prediction workflow, including application model, machine model, and runtime contributions

The second part of the machine model covers information about the cache hierarchy. This entails everything needed to calculate the volume of data transfers for a loop: the number of cache levels, their effective[4] sizes, write-through vs. write-back policy, victim/exclusive vs. inclusive, etc. For example, a victim cache typically implies additional traffic since it receives both modified and unmodified cache lines (CLs) from the overlying cache, whereas a non-victim cache only receives modified CLs. In order to get from data volumes to runtime contributions of individual data paths, the machine model also requires data about the available bandwidth between adjacent caches, and whether transfers take place over a single bi-directional link or over two uni-directional links. Moreover, if an architecture provides an inadequate number of buffers to track outstanding transfers, the corresponding latency penalties must be included. Finally, the second part of the machine model contains a description of which transfers in the memory hierarchy can occur simultaneously[5].

## 2.4. Performance Prediction Workflow

An overview of the performance-prediction workflow is provided in Fig. 3. As indicated in the figure, the process can be divided into four steps: first, the runtime contribution of performing operations in the core (with all data coming from L1) is determined. Next, the runtime contributions of data transfers in the memory hierarchy are calculated (to this end, data transfer volumes in the memory hierarchy need to be determined). In a third step, the previously determined runtime contributions are put together to arrive at a single-core runtime estimate. Finally, based on the single-core estimate from the previous step, multicore predictions can be given. In the following, each of the steps is discussed in detail.

[4]For several reasons (imperfect cache replacement strategies, prefetchers preempting data that could have otherwise been reused, etc.) the effective capacity of a cache is lower than its nominal size. In practice, the heuristic of halving the theoretical cache size delivers good estimates for the effective size.

[5]As will be demonstrated later, we find that in practice, this rarely discussed architectural feature turns out to be much more important for single-core in-memory performance than other more prominent features such as SIMD width or cache bandwidths.

### *2.4.1. Contributions of instruction execution in the core*

The fact that some architectures cannot overlap data transfers between the register file and the L1 cache on one hand and the L1 and L2 caches on the other makes it necessary to separate the runtime contribution of operations into two components: $T_{\text{comp}}$, which are cycles in which no data transfers between registers and L1 cache occur, and $T_{\text{RegL1}}$, which are cycles in which at least one load or store operation retires. Unless otherwise indicated, all arithmetic and load-store operations handle double-precision floating-point operands.

To estimate $T_{\text{RegL1}}$, first the numbers of load and store operations ($n_{\text{LD}}$ and $n_{\text{ST}}$) are determined by counting their occurrences in the loop body; the numbers are then divided by the respective throughputs, $\tau_{\text{LD}}$ and $\tau_{\text{ST}}$, taking additional constraints specified in the machine model into account (e.g., a limited throughput for the overall number of load/store operations per cycle, $\tau_{\text{LD/ST}}$, caused by a limited number of AGUs). The corresponding runtime contribution is the maximum of all components:

$$T_{\text{RegL1}} = \max\left(\frac{n_{\text{LD}}}{\tau_{\text{LD}}}, \frac{n_{\text{ST}}}{\tau_{\text{ST}}}, \frac{n_{\text{LD}} + n_{\text{ST}}}{\tau_{\text{LD/ST}}}\right). \tag{1}$$

The number of cycles in which no load/store operations are carried out is determined in a similar way: operation counts are found in the loop body. Each count is then divided by the operation's throughput documented in the machine model. As before, additional constraints have to be considered: For example, execution-port conflicts (cf. Section 2.3) can be addressed by summing up the contributions of functional units that share the same execution port (this is demonstrated in the equation below, where MUL and DIV units are assumed to be assigned to the same execution port). The fact that cores have an upper limit to the number of instructions they can retire per cycle can be modeled by dividing the total number of operations by a corresponding instruction-throughput limit $\tau_{\text{total}}$. Finally, loop-carried dependencies are accounted for by including the contribution of the longest cross-iteration dependency chain, $T_{\text{dep}}$, when determining the overall runtime by applying the maximum to all individual contributions:

$$T_{\text{comp}} = \max\left(\frac{n_{\text{ADD}}}{\tau_{\text{ADD}}}, \frac{n_{\text{MUL}}}{\tau_{\text{MUL}}} + \frac{n_{\text{DIV}}}{\tau_{\text{DIV}}}, \ldots, \frac{\sum_i n_i}{\tau_{\text{total}}}, T_{\text{dep}}\right). \tag{2}$$

### *2.4.2. Contributions of data transfers in the memory hierarchy*

Before the runtime contributions of data transfers can be determined, the data volumes transferred over the various data paths in the memory hierarchy need to be established. To this end, the location of the data set(s) in the memory hierarchy is derived from the data-set size(s) specified in the application model. Then, the load/store operations documented in the application model are revisited: for each operation, the corresponding data set is identified, and the transfers required to get the data from its current location in the memory hierarchy to the L1 cache are recorded. Along with the required transfers, the data volume is determined (e.g., four bytes per single- or eight bytes per double-precision floating-point number). Note that full CL transfers need to be taken into account even when CLs are only partially read or written (e.g., for strided but regular access). In case of truly random access patterns, latency contributions will dominate. This case is not part of the ECM model yet, although it is possible to incorporate it in a phenomenological way [2]. Extending the analytic model towards random accesses is part of future work.

Note that determining data-transfer volumes requires keeping track of previous data accesses to detect possible data reuse. While this can be done manually for kernels with simple data-access patterns, analysis of complex patterns is best left to cache simulators (e.g., pycachesim [8]). To this end, per-loop traffic estimates from cache simulators can be used as inputs in Eq. 3. If necessary, the resulting numbers can be validated by measuring the actual data volumes using hardware performance events (e.g., with PAPI [23] or LIKWID [5]).

Once the data volumes have been established, the runtime contribution $T_{ij}$ of data transfers between levels $i$ and $j$ of the memory hierarchy can be calculated:

$$T_{\mathrm{ij}} = \mathrm{max/sum}\left(\frac{v_{i\to j}}{b_{i\to j}}, \frac{v_{i\leftarrow j}}{b_{i\leftarrow j}}\right) + T^{\mathrm{p}}_{ij} = T^{\mathrm{data}}_{\mathrm{ij}} + T^{\mathrm{p}}_{ij}. \tag{3}$$

The process works by first calculating the time the data link(s) connecting levels $i$ and $j$ are actually busy transferring data. To calculate this data-link busy time, $T^{\mathrm{data}}_{ij}$, the data volumes, $v$, transferred in each direction are divided by the bandwidth, $b$, of the link over which the data is transferred. The two directional components $T_{i\to j}$ and $T_{i\leftarrow j}$ are then combined according to the information provided in the machine model. If there is a single bi-directional link over which transfers in both directions take place, the combined data-link busy time is the *sum* of both contributions. If there are two dedicated uni-directional links over which the transfers can take place, the overall data-link busy time is the *maximum* of both contributions. The overall data-transfer time, $T_{ij}$, is given by the sum of the previously determined data-link busy time and (if applicable) the corresponding latency penalty specified in the machine model.

### *2.4.3. Combination of runtime contributions for single-core estimate*

To arrive at a single-core runtime prediction, the previously determined components are put together according to the overlap capabilities specified in the machine model. To this end, first, all non-overlapping components are added up. The result is then included in the set of overlapping components, and the total runtime estimate is the maximum of the resulting set:

$$T = \max\left(\overbrace{T_{\ldots}, \cdots, T_{\ldots}}^{\text{overlapping}}, \overbrace{T_{\ldots} + \cdots + T_{\ldots}}^{\text{non-overlapping}}\right). \tag{4}$$

The following example will clarify the process: when discussing the model assumptions in Section 2.1, it was established that $T_{\mathrm{comp}}$ and $T_{\mathrm{RegL1}}$ overlap on all processors. Let us further assume that the architecture under consideration has a multi-ported L1 cache, which enables the cache to simultaneously communicate with the register file and the L2 cache. Assuming no overlap of other transfers, the runtime estimate for an in-memory data set on this processor would be $T = \max(T_{\mathrm{comp}}, T_{\mathrm{RegL1}}, T_{\mathrm{L1L2}}, T_{\mathrm{L2L3}} + T_{\mathrm{L3Mem}})$.

The runtime estimate $T$ can be converted into a performance estimate $P$ by dividing the amount of work $W$ carried out in one loop iteration by the runtime estimate for the same, and multiplying the result with the core frequency: $P = f_{\mathrm{core}} \cdot W/T$.

For our investigations $f_{\mathrm{core}}$ was fixed, so converting from runtime to performance estimates is trivial. In practice, however, $f_{\mathrm{core}}$ is often set dynamically on the authority of the operating system, the processor, or even the user. However, $f_{\mathrm{core}}$ is virtually constant during the execution of a particular steady-state loop. This is because the metric used by the underlying mechanism (e.g., DVFS) to select $f_{\mathrm{core}}$ does not change while the processor is in a steady state. For a particular kernel, $f_{\mathrm{core}}$ can thus be measured via hardware performance events. For each kernel of

a multi-loop application, $f_{\text{core}}$ value must be determined individually. See [13] for an investigation of the model's ability to deal with different core and Uncore frequencies.

#### *2.4.4. Multicore prediction based on single-core estimate*

Multicore estimates require as inputs the single-core runtime estimate $T$, and the time the memory interface is busy transferring data $T_{\text{Mem}}^{\text{data}}$, which is the sum of all data-link busy times that involve the main memory (e.g., in a memory hierarchy with a victim L3 cache, where memory sends data to L2 and receives modified CLs from L3, $T_{\text{Mem}}^{\text{data}} = T_{\text{L2Mem}}^{\text{data}} + T_{\text{L3Mem}}^{\text{data}}$).

In the absence of shared resources (e.g., if the entire data set fits into core-private or scalable[6] shared caches), single-core performance $P$ is expected to scale linearly with the number of active cores $n$, so the multicore estimate for $n$ active cores is just $P(n) = nP$. If shared resources, such as the main memory interface, are involved, resource conflicts and the resulting waiting times must be considered. Here we employ a statistical model that is motivated by first principles: the utilization of the memory bus $u$ is the probability of another core encountering a busy bus. For a single core, the utilization is given by the ratio of the time the memory interface is busy transferring data and the overall runtime estimate: $u(1) = T_{\text{Mem}}^{\text{data}}/T$. If multiple cores are active, the utilization is expressed recursively:

$$u(n) = \min\left(1, \frac{nT_{\text{Mem}}}{\max(T_{\text{comp}}, \ldots, T_{\text{Mem}} + \underbrace{u(n-1)(n-1)p_0)}_{T_{\text{conf}}})}\right). \tag{5}$$

In the numerator, the memory-bus busy time is multiplied with the number of active cores, $n$, since multiple cores are using the memory interface. The denominator is the expanded expression for the runtime estimate, $T$, where a conflict time has been added to the data-transfer time involving the memory interface. This conflict time represents the average time that a core encountering a busy memory bus has to wait for the bus to become available to it. The conflict time encountered in a scenario with $n$ active cores is given by multiplying the probability of a core hitting a busy memory bus, which corresponds to the memory utilization of the remaining cores, $u(n-1)$, with the time the other $n-1$ cores are using the interface. This results in $T_{\text{conf}} = u(n-1)(n-1)p_0$, with $p_0$ being an empirical fit parameter[7].

For performance estimates, the memory-bus utilization is multiplied with the performance to be expected with fully saturated bandwidth: $P(n) = u(n)P^{\text{Sat}}$. The memory-saturation performance, $P^{\text{Sat}}$, corresponds to the bandwidth limitation of the Roofline model and is determined by dividing the amount of work per loop iteration by the memory-bus busy time, and multiplying the result with the core frequency: $P^{\text{Sat}} = W/T_{\text{Mem}} \cdot f_{\text{core}}$.

## 3. Machine Model Construction

### 3.1. Method to Determine Machine Models

In the ideal case, all of the data required for a machine model would be available in vendor data sheets. In practice, however, this is rarely the case because important information is

[6]Scalable means a parallel efficiency close to one for all relevant degrees of parallelization (i.e., up the maximum number of cores sharing the cache).

[7]Although $p_0$ can also be modeled analytically employing the data used to derive $T_{\text{Mem}}$, we find that the level of detail required to reliably estimate the parameter outweighs the benefits of using an analytical approach.

deemed irrelevant or, more likely, intellectual property and therefore omitted from specifications. Moreover, the interaction of different parts of the processor might lead to situations in which vendor-specified numbers are not attainable (see, e.g., the discussion on load/store throughput in Section 2.3). In the following, a method is presented that allows to establish machine models in cases where relevant information is missing, or the documented specifications turn out to be impractical for some reason.

### *3.1.1. Instruction throughput and latency*

At fixed core clock speed $f_{\text{core}}$, the time $t$ it takes the core to execute a large number $n$ of independent[8] instructions of type $i$ is measured. The throughput of the instruction is then $\omega_i = n/(tf_{\text{core}})$. Since we will usually use a work unit of one (high-level) loop iteration in the modeling procedure, the instruction throughput is multiplied by the appropriate SIMD width $w_{\text{SIMD}}$ to get the *operation throughput*:

$$\tau_i = w_{\text{SIMD}} \times \omega_i = w_{\text{SIMD}} \times n/(tf_{\text{core}}). \tag{6}$$

To measure latency, an artificial data-dependency chain is introduced by making each instruction use the output of the previous instruction as its input. This forces each new instructions to be held at a reservation station until the previous instruction has completed. The holding time, $\Lambda = n/(tf_{\text{core}})$, corresponds to the instruction's latency. The measured instruction latency is divided by the appropriate SIMD width to get the *operation latency*:

$$\lambda_i = \Lambda_{\text{i}}/w_{\text{SIMD}} = n/(tf_{\text{core}}w_{\text{SIMD}}). \tag{7}$$

While implementing these two strategies sounds simple in theory, deriving a suitable instruction mix from a high-level language implementation can be difficult in practice because compiler optimizations get in the way. We solve this problem by side-stepping the compiler and hand-crafting the necessary code in assembly language. To automate the process of determining latencies and throughputs, the `ibench` tool [11] was developed, which comprises a measurement framework and a number of assembly-code files for the most widespread instructions of AMD, IBM, ARM, and Intel processors.

### *3.1.2. Topology and data flow in the memory hierarchy*

Information about the topology of the memory hierarchy, such as the number of caches, their sizes and properties (write-back vs. -through, victim vs. non-victim) are often well documented in vendor data sheets. Even if this is not the case, the data is easy to obtain, for most processors provide access to it over a well-defined interface. In case of x86, for instance, the `cpuid` instruction can be used to extract detailed information about the memory hierarchy, including the capacity, associativity, number of sets, inclusiveness, cache-line size, and more for each level in the hierarchy. Other processors offer similar mechanisms, and the Linux `sysfs` file system provides an architecture-independent interface to obtain the necessary data.

Information about data flow (i.e., the path data takes from a particular level in the memory hierarchy to reach a core's L1 cache) can be derived from the topology information. In most cases, only stores require special attention to determine whether store-misses trigger a write-allocate

[8]Independent means that there are no data dependencies between the different instances of the instruction.

for the missed CL or if some optimization (as the one implemented in Marvell's ThunderX2) detects whether a full CL is written to avoid the write-allocate. Such details can be derived with the help of hardware performance events, which can be used to record the data volumes exchanged between different levels of the memory hierarchy.

### *3.1.3. Bandwidth, latency, and overlap in the memory hierarchy*

Typically, cache bandwidths are well-documented by vendors. In some cases, however, vendors only specify bandwidths for selected caches. In these instances, cache bandwidths can be determined by selecting a set of reasonable bandwidth candidates (e.g., 16, 32, and 64 B/cy), and examining which of the corresponding estimates best agrees with empirical data. To the best of our knowledge, no vendor publishes data on the overlap properties of their processors' memory hierarchies, so this data needs to be determined in a similar way.

The process of comparing estimates to empirical data is iterative: once the bandwidth and overlap properties for a particular memory level have been established, the numbers can be used as input for different bandwidth and overlap assumptions in the next memory level. In the following, the process is demonstrated on the SKL processor for the well-known STREAM triad [17].

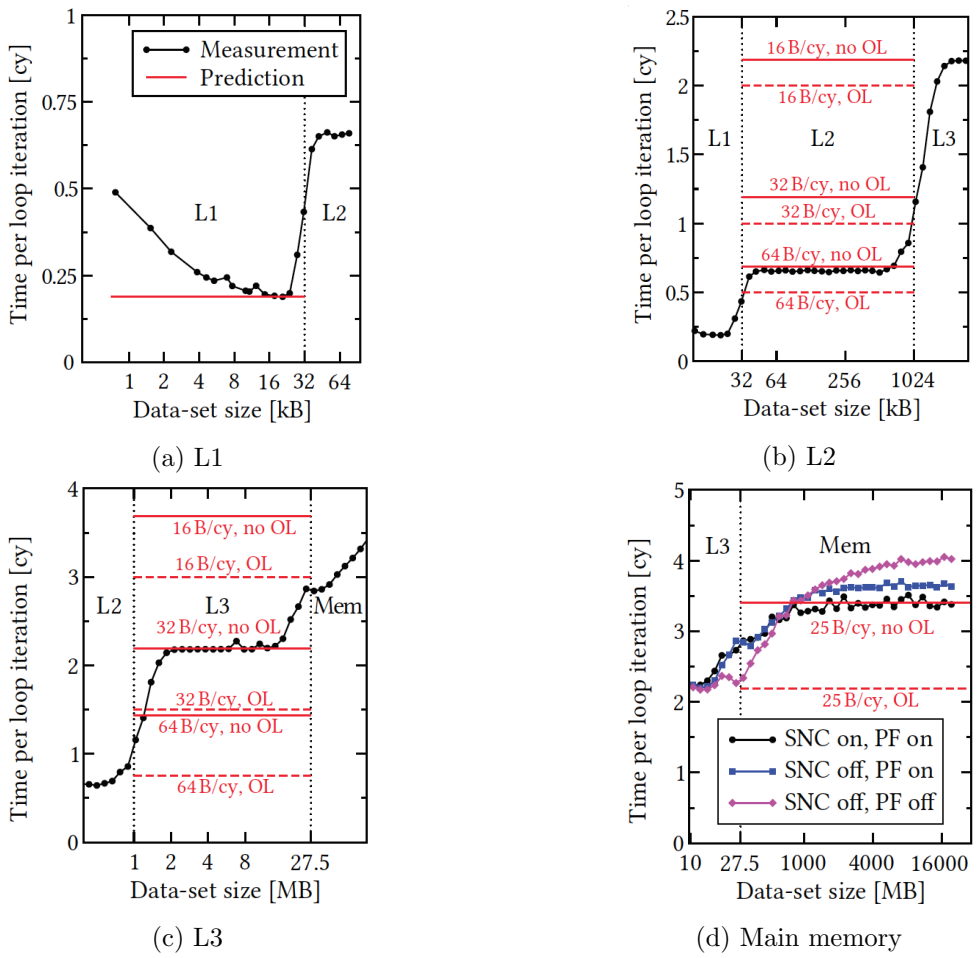



**Figure 4.** Comparison of model estimates to empirical data for the STREAM triad on SKL for data sets in (a) L1, (b) L2, and (c) L3 caches, and (d) main memory

On the SKL processor, one loop iteration of the STREAM triad (`A[i]=B[i]+s*C[i]`) comprises two loads (one from each of the input arrays `B` and `C`), one fused multiply-add (FMA) (to calculate the result), and one store (to write the result to the output array `A`). Using `ibench`, the following operation throughputs were established: $\tau_{\mathrm{FMA}} = 16/\mathrm{cy}$, $\tau_{\mathrm{LD}} = 16/\mathrm{cy}$, $\tau_{\mathrm{ST}} = 8/\mathrm{cy}$, and $\tau_{\mathrm{LD/ST}} = 16/\mathrm{cy}$. According to Equations (1), (2), and (4), for a data set in the L1 cache this leads to a single-iteration runtime estimate of

$$\begin{aligned} T_{\mathrm{L1}} &= \max\left(\overbrace{\frac{1\,\mathrm{FMA/it}}{16\,\mathrm{FMA/cy}}}^{T_{\mathrm{comp}}}, \overbrace{\frac{2\,\mathrm{LD/it}}{16\,\mathrm{LD/cy}}, \frac{1\,\mathrm{ST/it}}{8\,\mathrm{ST/cy}}, \frac{3\,\mathrm{LD/ST/it}}{16\,\mathrm{LD/ST/cy}}}^{T_{\mathrm{RegL1}}}\right) \\ &\approx 0.19\,\mathrm{cy/it}. \end{aligned}$$

In Fig. 4a we compare this prediction to measurements. Note that the estimate corresponds to the lower limit of runtime, which is actually attained by the running code if the loop is long enough.

If the data set resides in the L2 cache, a total of 32 B are transferred between the L1 and L2 caches per iteration: 8 B for each of the double-precision floating-point numbers from the input arrays `B` and `C`, 8 B for the write-allocate to `A`, and 8 B for evicting the updated element of `A` to the L2 cache. Bandwidth assumptions of 16, 32, and 64 B/cy yield estimates for $T_{\mathrm{L1L2}}$ of two, one, and one-half cycle, respectively. Figure 4b compares the estimates to empirical data. The assumptions of no overlap and a bandwidth of 64 B/cy match the measurements strikingly well; incidentally, the L1-L2 cache bandwidth as advertised by Intel is also 64 B/cy. With L1-L2 cache bandwidth and overlap properties established, we can move on to the L3 cache. The data exchanged between the L2 and L3 caches is 48 B because each of the three eight-byte reads from L3 (two from the input arrays `B` and `C`, one write-allocate from the target array `A`) triggers the eviction of data replaced in the L2 cache to the victim L3. L2-L3 bandwidth assumptions of 16, 32, and 64 B/cy yield estimates for $T_{\mathrm{L2L3}}$ of 3, 1.5, and 0.75 cy, respectively. Figure 4c compares estimates derived from the different bandwidth and overlap assumptions to empirical data for a data set in the L3 cache. In this case we find that that assumptions of no overlap and a bandwidth of 32 B/cy agree very well with the measurement. Finally, for in-memory data sets, only different overlap assumptions must be made, since the sustained memory bandwidth is determined by measurement (55 GB/s for one SNC domain, which for $f_{\mathrm{core}} = 2.2\,\mathrm{GHz}$ is 25 B/cy). Figure 4d compares the resulting estimates to empirical data (black line) and we find that in memory, too, no overlap of data transfers occurs.

In addition to runtime measurements obtained with SNC mode and the LLC prefetcher (PF) enabled, Fig. 4d also shows data where these features were disabled. This is to demonstrate that in some settings, bandwidth and overlap are not sufficient to describe the empirical behavior in a satisfying manner. Then, a latency penalty must be added to data transfer times (see Section 2.1).

### 3.2. Results for Investigated Processors

Table 2 shows the machine models that result from applying the previously introduced method to the processors from the testbed.

The upper part of the table lists relevant operation throughput ($\tau$) and instruction latency ($\lambda$) values. The center part lists bandwidths and latency penalties (if applicable) in the memory hierarchy. Note that in cases where two numbers are provided (e.g., 64+16 B/cy for PWR9's L1-

**Table 2.** Machine models determined for the investigated processors

| Microarchitecture | Skylake-SP (SKL) | Zen (EPYC) | Vulcan (TX2) | POWER9 (PWR9) |
|---|---|---|---|---|
| $\tau_{\text{ADD}}$, $\tau_{\text{MUL}}$, $\tau_{\text{FMA}}$ [/cy] | 16, 16, 16 | 4, 4, 4 | 4, 4, 4 | 4, 4, 4 |
| $\tau_{\text{LD}}$, $\tau_{\text{ST}}$, $\tau_{\text{LD/ST}}$ [/cy] | 16, 8, 16 | 4, 2, 4 | 4, 2, 4 | 4, 4, 4 |
| $\lambda_{\text{ADD}}$, $\lambda_{\text{MUL}}$, $\lambda_{\text{FMA}}$ | 0.5, 0.5, 0.5 | 1.5, 2, 2.5 | 3, 3, 3 | 3, 3, 3 |
| $b_{\text{L1}\leftrightarrow\text{L2}}$ | 64 B/cy | 32+32 B/cy | 64 B/cy | 64+16 B/cy |
| $b_{\text{L2}\leftrightarrow\text{L3}}$ | 32 B/cy | 32 B/cy | 32 B/cy | 32 B/cy |
| $b_{*\leftrightarrow\text{Mem}}$ | 25–28 B/cy | 13–16 B/cy | 47–56 B/cy | 41–45 B/cy |
| Data-transfer penalties | — | — | — | $T^{\text{p}}_{\text{Mem}} = 0.04$ cy/B |
| Non-overlapping transfers | all | L2-L3, L2-Mem, L3-Mem | all (if Mem involved) | L2-Mem, L3-Mem |
| Write-through/ victim caches | Victim L3 | Victim L3 | Victim L3 | Write-through L1, Victim L3 |

L2 bandwidth), two uni-directional data paths exist between the caches. In such instances, the first number corresponds to the bandwidth of sending data from the underlying to the overlying cache, and second number to the bandwidth in the opposite direction. Note that listed memory bandwidth corresponds to that of a single NUMA node (SNC node on SKL, Zeppelin on EPYC, full-chip on TX2 and PWR9). Memory bandwidths are specified as ranges, since different data-access patterns exhibit slightly varying sustained memory bandwidths. The last part of the table contains overlap capabilities and additional information on cache types.

## 4. Case Study: PCG

We use a matrix-free PCG solver to demonstrate the viability of our approach in real-world scenarios. The solver is preconditioned using the well-known symmetric Gauss-Seidel iteration and relies on the second-order finite-difference method for discretization. We use it to solve the steady-state heat equation in 2D. The sparse matrix entries are not stored explicitly but hard-coded into a 2D five-point stencil representation. Hence, the solver is similar to the well-known HPCG but shows a more interesting phenomenology: as opposed to HPCG, where all loops are limited by data transfers due to explicit matrix storage, our preconditioner is bound by in-core pipeline hazards. All computations and data storage are in double precision.

Algorithm 1 shows the entire PCG method. It is composed of a matrix-free sparse-matrix-vector multiplication (SpMVM) which we refer to as STENCIL, a symmetric Gauss-Seidel preconditioner (GS), and three BLAS-1 routines: DOT product, vector NORM, and DAXPBY. The code is implemented in C++ and parallelized with OpenMP. The Gauss-Seidel kernels, which have loop-carried dependencies, are parallelized using a well-known wavefront technique that preserves the numerical behavior of the serial code [7]. The preconditioner can be vectorized by, e.g., coloring methods, but this would alter the convergence and render the loops data bound, which is not the scenario we want to showcase (see above).

**Algorithm 1** PCG algorithm: solve for $x : Ax = b$

1: $r = b - Ax$
2: $r_{\text{norm}} = \langle r, r \rangle$
3: $p = z = Pr$
4: $\alpha_0 = \langle r, z \rangle$
5: $i = 0$
6: **while** $(i < n_{\text{iter}})$ && $(r_{\text{norm}} > \varepsilon^2)$ **do**
7: $v = Ap$ — STENCIL operation (SpMVM)
8: $\lambda = \frac{\alpha_0}{\langle v, p \rangle}$ — DOT
9: $x = x + \lambda p$ — DAXPBY
10: $r = r - \lambda v$ — DAXPBY
11: $r_{\text{norm}} = \langle r, r \rangle$ — NORM
12: $z = Pr$ — GS preconditioner
13: $\alpha_1 = \langle r, z \rangle$ — DOT
14: $p = z + \frac{\alpha_1}{\alpha_0} p$ — DAXPBY
15: $\alpha_0 = \alpha_1$
16: $i = i + 1$

**Algorithm 2** High-level representation of STENCIL

1: **for** $j = 1 : n_j - 1$ **do**
2: **for** $i = 1 : n_i - 1$ **do**
3: $v_{j,i} = w_c p_{j,i} + w_y(p_{j-1,i} + p_{j+1,i}) + w_x(p_{j,i-1} + p_{j,i+1})$

### 4.1. Application Models

The total problem size ($n_i \times n_j$) was chosen to be $n_i = 25000$ (inner, leading dimension) and $n_j = 2000$ (outer dimension), so that all arrays reside in main memory. In the following, application models for all of the PCG components are presented.

Features important for the considered example include the number of loads and stores, floating-point operations, and loop structures. For simple streaming loops, all of these details can be derived from high-level code. The DAXPBY kernel (`y[i]=a*x[i]+b*y[i]`) entails two loads, one FMA, one multiplication, and one store. The DOT product (`d+=x[i]*y[i]`) and NORM (`n+=x[i]*x[i]`) have two and one load(s), respectively, along with an FMA. These kernels can be fully and effectively vectorized by all compilers.

For kernels with cache reuse such as STENCIL and GS, reuse-distance analysis (best done using the layer condition [3, 22]), blocking factors, parallelization strategies, and scheduling techniques have to be taken into account. The STENCIL kernel is shown in Algorithm 2, with $w_*$ representing different weights obtained from the matrix $A$. The kernel requires two FMAs, two additions, one multiplication, one store, and five load operations. SIMD vectorization is straightforward, but in contrast to the BLAS kernels, different loads can hit different memory hierarchy levels depending on the reuse distance. For the considered inner dimension of $n_i = 25000$ and outer ($j$) loop parallelization employed in our code, the layer condition would require $4n_i$ elements per thread to fit in a cache. The lowest (i.e., outermost) cache that satisfies this criterion will only have a miss for one of the four elements on the right-hand side, while the cache levels above it will have three. On all processors under investigation, the layer condition is satisfied in the last-level cache (LLC). Changing the inner problem dimension would certainly change

**Algorithm 3** High-level representation of GS forward sweep

```
1: for j = 1 : n_j − 1 do
2:     for i = 1 : n_i − 1 do
3:         z_{j,i} = w_c(r_{j,i} + w_y z_{j−1,i} + w_x z_{j,i−1})
```

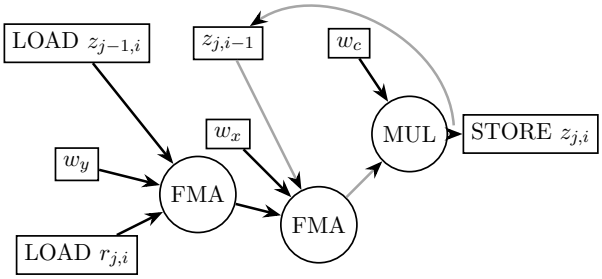



**Figure 5.** Dependency chain of the GS forward kernel when using the Intel compiler. Critical path shown in gray

the prediction; the ECM model has been demonstrated to yield accurate results in all these cases [14, 22], so we restrict ourselves to a single size only here. Storing to $v$ implies a write-allocate through the whole memory hierarchy on all processors, and, at some point, the writing back of the newly-computed data to memory.

The GS kernel is a symmetric operator comprising a forward and a backward sweep. The forward sweep ($GS_F$) is shown in Algorithm 3, and requires two FMAs, one multiplication, one store, and three load operations. The kernel is similar to STENCIL, but it reads from $z_{j,i-1}$ and writes to $z_{j,i}$, causing a loop-carried dependency. A wavefront technique can be used to parallelize the kernel [7], and the corresponding layer-condition criterion requires $3n_i$ elements to fit in a cache. The outermost cache that satisfies this condition will have only two load misses on the right-hand side, while the others would have three. The GS backward sweep (not shown here for brevity) is similar, but loops are traversed in reverse direction and $w_c r_{j,i}$ in $GS_F$ is replaced with $z_{j,i}$. The analysis of the kernel follows the same approach, but there is one less load miss.

Both GS loops have loop-carried dependencies, preventing SIMD vectorization. As a result, a critical path analysis is required. In $GS_F$ the element $z_{j,i}$ written in a particular iteration is read in the next as $z_{j,i-1}$. The actual delay caused by this dependency can vary depending on the code generated by the compiler. Figure 5 shows the result when using the Intel compiler and the critical path of the generated instruction mix includes one FMA and one multiplication. The ARM clang compiler produces code that does not keep $z_{j,i}$ in a register across loop iterations, leading to an extra delay caused by storing and loading the element. Due to its particular unrolling strategy, IBM's XLC compiler generates a combination of the two previous variants.

## 4.2. Runtime Predictions

In the following, the proposed model is validated by comparing the model estimates to empirical performance for the DOT product, DAXPBY and $GS_F$ kernels, as well as the full PCG algorithm. The DOT kernel is also used to exemplify how the simultaneous multi-threading (SMT) feature of modern processors can be incorporated in the model. Note that estimates correspond to the runtime of a single high-level (i.e., scalar) loop iteration.

#### *4.2.1. Simultaneous multi-threading on SKL*

As discussed in Section 4.1, one loop iteration of the DOT product entails two loads and one FMA. According to the machine model documented in Tab. 2, the SKL processor can perform 16 loads and 16 FMAs per cycle for AVX-512 code. Combining application and machine models according to Eq. (1) yields a contribution of $T_{\text{RegL1}} = n_{\text{LD}}/\tau_{\text{LD}} = 2/16\,\text{cy} = 0.125\,\text{cy}$ for data transfers between the register file and the L1 cache. With respect to computational cycles, it is worth pointing out that the kernel contains a loop-carried dependency. Each FMA uses as one of its inputs the result of the previous FMA. Without modulo-variable expansion (MVE) and SMT, the impact of the dependency corresponds to the FMA latency, so $T_{\text{dep}} = \lambda_{\text{FMA}} = 0.5\,\text{cy}$. According to Eq. (2), computational cycles therefore amount to $T_{\text{comp}} = \max(n_{\text{FMA}}/\tau_{\text{FMA}}, T_{\text{dep}}) = \max(1/16\,\text{cy}, 0.5\,\text{cy}) = 0.5\,\text{cy}$. Since both contributions can overlap, the runtime estimate for a data set in the L1 cache according to Eq. (4) is $T = \max(T_{\text{RegL1}}, T_{\text{comp}}) = 0.5\,\text{cy}$.

Using either two-way unrolling with MVE or 2-SMT, the impact of the dependency is cut in half, so $T_{\text{dep}} = 0.25\,\text{cy}$. The overall runtime estimate in this case becomes $T = 0.25\,\text{cy}$.

The combination of two-way unrolling with MVE and 2-SMT again halves the impact of the dependency, so $T_{\text{dep}} = 0.125\,\text{cy}$, and the overall runtime becomes $T = 0.125\,\text{cy}$. Note that at this point, the contribution of the loop-carried dependency is identical to $T_{\text{RegL1}} = 0.125\,\text{cy}$. This means that additional unrolling will no longer effect a reduction in runtime since runtime is now limited by data transfers between the register file and the L1 cache. Note as well that reducing $T_{\text{dep}}$ to 0.125 cy can also be achieved without SMT by applying four-way unrolling with MVE to the code. In fact, it is possible to run most loop-based streaming codes at the lower runtime limit without SMT if the executed instruction mix is optimized appropriately and sufficient physical registers are available.

**Table 3.** Comparison of model estimates to empirical data (in cycles per loop iteration) for the DOT product on the SKL CPU for data-set sizes of 25 kB (L1), 127 kB (L2), 9772 kB (L3), and 1022 MB (Mem) as function on the degree of simultaneous multi-threading (SMT) and unrolling with modulo-variable expansion (MVE)

| Degree of | | Model estimate for | | | | Measurement for | | | |
|---|---|---|---|---|---|---|---|---|---|
| SMT | MVE | L1 | L2 | L3 | Mem | L1 | L2 | L3 | Mem |
| 1 | 1 | 0.500 | 0.500 | 1.375 | 1.975 | 0.501 | 0.500 | 1.411 | 2.096 |
| 1 | 2 | 0.250 | 0.375 | 1.375 | 1.975 | 0.250 | 0.379 | 1.411 | 2.085 |
| 2 | 1 | 0.250 | 0.375 | 1.375 | 1.975 | 0.250 | 0.359 | 1.411 | 2.028 |
| 2 | 2 | 0.125 | 0.375 | 1.375 | 1.975 | 0.136 | 0.360 | 1.411 | 2.030 |
| 1 | 4 | 0.125 | 0.375 | 1.375 | 1.975 | 0.136 | 0.376 | 1.413 | 2.090 |
| 2 | 4 | 0.125 | 0.375 | 1.375 | 1.975 | 0.136 | 0.364 | 1.411 | 2.029 |

Table 3 summarizes the estimates discussed above, as well as estimates for the remaining levels of the SKL processor's memory hierarchy. As predicted, no unrolling and SMT results in a runtime of 0.5 cy per loop iteration for a data set in the L1. Moreover, either two-way unrolling or SMT results in a halving of the runtime to 0.25 cy. Combining both optimizations further reduces the runtime by a factor of two to 0.125 cy. The data shows that the same result can be achieved without SMT when applying four-way unrolling. Furthermore, the data in the last

row supports the model prediction that additional unrolling (or SMT, if the core supported it) would not lead to further reductions in runtime since at this point $T_{\mathrm{RegL1}}$ dominate.

When the data set resides in the L2 cache, 16 bytes (8 bytes for each of the double-precision floating-point numbers from the two input arrays) must be transferred between the L1 and L2 caches per loop iteration. The machine model (see Tab. 2) lists a L1-L2 bandwidth of 64 B/cy for the SKL processor, so the data-transfer time is $T_{\mathrm{L1L2}} = 0.25\,\mathrm{cy}$. All data transfers are non-overlapping, so the runtime estimate according to Eq. (4) becomes $T = \max(T_{\mathrm{comp}}, T_{\mathrm{RegL1}} + T_{\mathrm{L1L2}})$ with an aggregated transfer time of $T_{\mathrm{RegL1}} + T_{\mathrm{L1L2}} = 0.125\,\mathrm{cy} + 0.25\,\mathrm{cy} = 0.375\,\mathrm{cy}$. For the version without unrolling and SMT, $T_{\mathrm{comp}} = 0.5\,\mathrm{cy}$ is higher than the combined contribution of the runtime, resulting in an overall runtime of $T = \max(0.5\,\mathrm{cy}, 0.375\,\mathrm{cy}) = 0.5\,\mathrm{cy}$. For all other versions, the overall runtime is dominated by the combined data-transfer time, so $T = \max(T_{\mathrm{comp}}, 0.375\,\mathrm{cy}) = 0.375\,\mathrm{cy}$.

With data in the L3 cache, 16 bytes are to sent from the L3 to the L2 cache in each loop iteration. At the same time, 16 bytes are preempted from the L2 cache into the victim L3 cache. The total amount of data transferred is therefore 32 bytes, which takes 1 cy according to the documented bandwidth of 32 B/cy (cf. Tab. 2). Considering that data transfers are non-overlapping, the overall runtime estimate becomes $T = \max(T_{\mathrm{comp}}, T_{\mathrm{RegL1}} + T_{\mathrm{L1L2}} + T_{\mathrm{L2L3}})$. According to the model, the runtime of all variants is dominated by the contribution of data transfers of $T_{\mathrm{RegL1}} + T_{\mathrm{L1L2}} + T_{\mathrm{L2L3}} = 0.125\,\mathrm{cy} + 0.25\,\mathrm{cy} + 1\,\mathrm{cy} = 1.375\,\mathrm{cy}$. Consequently, the runtime estimate for all variants is $T = \max(T_{\mathrm{comp}}, 1.375\,\mathrm{cy}) = 1.375\,\mathrm{cy}$.

Finally, in the case of input data residing in main memory, 16 bytes have to be sent from memory to the L3 cache. The bandwidth of about 26.5 B/cy documented in the machine model (cf., again, Tab. 2) implies a contribution of $T_{\mathrm{L3Mem}} \approx 0.6\,\mathrm{cy}$. As before, the non-overlapping data-transfer contributions dominate the overall runtime for all versions, resulting in an estimate of $T = \max(T_{\mathrm{comp}}, 0.125\,\mathrm{cy} + 0.25\,\mathrm{cy} + 1\,\mathrm{cy} + 0.6\,\mathrm{cy}) = 1.975\,\mathrm{cy}$.

### *4.2.2. Single-core*

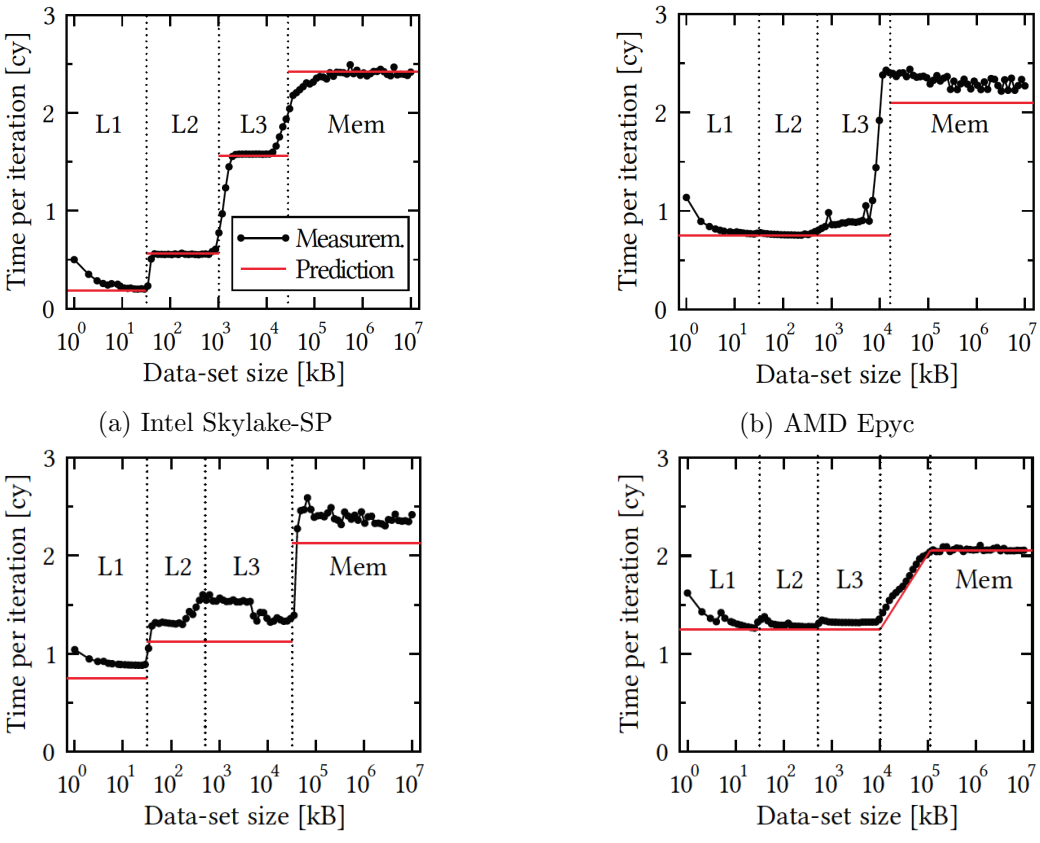
On the SKL processor, retiring the DAXPBY kernel's multiplication and FMA operations takes $T_{\mathrm{comp}} \approx 0.0625\,\mathrm{cy}$. The one store and two load operations take $T_{\mathrm{RegL1}} \approx 0.1875\,\mathrm{cy}$. Per-iteration data-transfer volumes are 24 B between the L1 and L2 caches (one load each from `x` and `y`, one write to `y`), 32 B between the L2 and L3 caches (one load each from `x` and `y`, and two corresponding evicts since the L3 is a victim cache), and 24 B between L3 and main memory (see L1-L2 transfers). Using the bandwidths documented in the machine model, this results in contributions of $T_{\mathrm{L1L2}} = 0.375\,\mathrm{cy}$, and $T_{\mathrm{L2L3}} = 1\,\mathrm{cy}$. For the measured memory bandwidth of 60 GB/s, which for $f_{\mathrm{core}} = 2.2\,\mathrm{GHz}$ corresponds to a bandwidth of 27.3 B/cy, $T_{\mathrm{L3Mem}}$ is 0.88 cy. Since all data transfers are non-overlapping, the runtime estimates are $T_{\mathrm{L1}} = 0.1875\,\mathrm{cy}$, $T_{\mathrm{L2}} = 0.5625\,\mathrm{cy}$, $T_{\mathrm{L3}} = 1.5625\,\mathrm{cy}$, and $T_{\mathrm{Mem}} = 2.4425\,\mathrm{cy}$.

Intermediate and final single-core estimates for DAXPY on SKL, and all other processors, are given in Tab. 4. Cases where data volumes change in the victim L3 cache (depending on whether the input data resides in the L3 or main memory) are indicated by listing two numbers in the table, the former corresponding to the data-transfer time estimate for data in the L3, the latter for data in memory.

These single-core estimates are compared to empirical data in Fig. 6. The data indicates that the model manages to describe empirical performance on all investigated processors with high accuracy.

**Table 4.** Single-core estimates for DAXPY on all investigated processors

| CPU | SKL | EPYC | TX2 | PWR9 |
|---|---|---|---|---|
| $T_{\text{comp}}$ [cy/it] | 0.0625 | 0.25 | 0.25 | 0.25 |
| $T_{\text{RegL1}}$ [cy/it] | 0.1875 | 0.75 | 0.75 | 0.75 |
| $T_{\text{L1L2}}$ [cy/it] | 0.375 | 0.5 | 0.375 | 0.5 |
| $T_{\text{L2L3}}$ [cy/it] | 1 | 0.75 \| 0.25 | 1 \| 0.5 | 1 \| 0.5 |
| $T_{\text{L2Mem}}$ [cy/it] | — | 1.23 | 0.29 | 0.36 |
| $T_{\text{L3Mem}}$ [cy/it] | 0.88 | 0.62 | 0.14 | 0.18 |
| $T_{\text{L1}}$ [cy/it] | 0.1875 | 0.75 | 0.75 | 1.25 |
| $T_{\text{L2}}$ [cy/it] | 0.5625 | 0.75 | 1.125 | 1.25 |
| $T_{\text{L3}}$ [cy/it] | 1.5625 | 0.75 | 1.125 | 1.25 |
| $T_{\text{Mem}}$ [cy/it] | 2.4425 | 2.1 | 2.06 | 2.1 |




(a) Intel Skylake-SP

(b) AMD Epyc

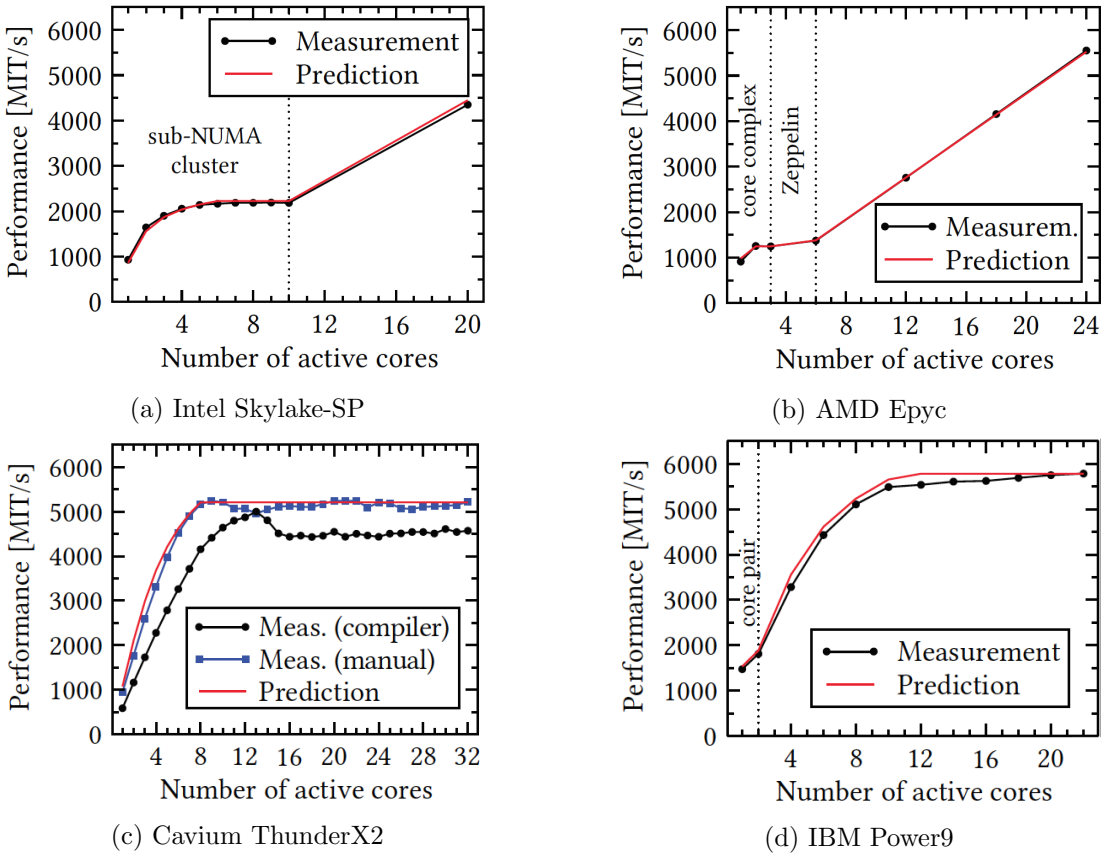
(c) Cavium ThunderX2

(d) IBM Power9

**Figure 6.** Comparison of model estimates to empirical data for DAXPY on (a) SKL, (b) EPYC, (c) TX2, and (d) PWR9

### *4.2.3. Multicore*

The DAXPBY and $\text{GS}_\text{F}$ kernels were selected to investigate the model's capability to accurately describe multicore performance. Being a data-bound streaming kernel, DAXPBY proves particularly suitable to investigate the memory subsystem of the investigated processors and




(a) Intel Skylake-SP

(b) AMD Epyc

(c) Cavium ThunderX2

(d) IBM Power9

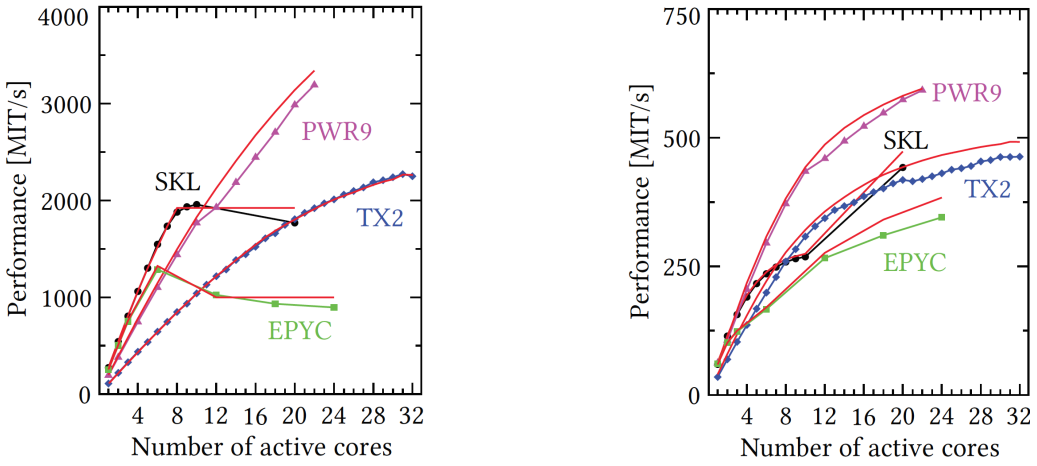
**Figure 7.** Comparison of performance models to empirical data for intra-socket scaling of DAXPBY on (a) SKL, (b) EPYC, (c) TX2, and (d) PWR9. Performance is given in $10^6$ iterations per second (MIT/s)

their scaling behavior. $GS_F$, on the other hand, is core bound for all architectures when executed on a single core. However, when increasing the number of cores, NUMA properties turn out to have a significant impact on performance.

Figure 7 shows the multicore scaling of DAXPBY on all architectures up to a full socket using "close" thread affinity (i.e., filling cores consecutively through ccNUMA domains). For SKL we observe the typical saturation behavior (at $\approx 2.2\,\mathrm{GIT/s} = 53\,\mathrm{GB/s}$) of bandwidth-bound code within a single SNC domain. Using the second SNC domain doubles the bandwidth and hence performance by a factor of two as predicted by the model. The scaling behavior of EPYC exposes its main hardware features: within a single CCX (three cores) the shared L3 bandwidth does not scale across the cores and hits a maximum of 32 B/cy. The best bandwidth attained on a single CCX is 30 GB/s compared to 33 GB/s for the entire ccNUMA domain (a "Zeppelin" die); we speculate that this is a faint echo of non-scalable L3 cache. Scaling across the Zeppelin dies is linear, as expected. On the TX2, we initially observed a significant deviation: The compiler-generated code (black line) fell short of the model by as much as 40 % for a single core and 10 % after saturation. The prompted investigation revealed that the TX2's hardware prefetchers have some deficiency: data was not prefetched in time, so runtime is subject to additional latency. The issue could be resolved by manually adding software prefetch instructions to the compiler-generated code to work around the flawed hardware prefetchers (blue line). This demonstrates how the model can be used to identify bottlenecks or other shortcomings that limit performance (in this case, the compiler). Note that the optimization is not part of our PCG code; we use the

compiler versions for all further comparisons. On PWR9, the scaling within a core pair is similar to that observed within a CCX of EPYC. This is due to the shared and non-scalable L2 and L3 cache segments per core. The multicore model accommodates this behavior by keeping the L2 and L3 data-transfer rates constant for the two cores sharing the resources. Scaling across core pairs (i.e., running with 2, 4, 6, etc. cores) is only limited by bandwidth saturation as can be observed by the measurements and respective model prediction.

The $GS_F$ kernel is latency bound due to the loop-carried dependency discussed in Section 4.1. There are two peculiarities that make predictions of the parallel $GS_F$ kernel challenging: first, the wavefront parallelization requires a barrier synchronization after each inner loop traversal. For the chosen problem size, the corresponding OpenMP-barrier was found to cause non-negligible overhead. We addressed this by benchmarking the OpenMP barrier for all relevant compiler-hardware combinations and included the barrier time as additional overhead. Secondly, although parallel first-touch page placement works fine for all other loops, the parallel-wavefront algorithm accesses data in parallel across the inner dimension. Since data placement is done with static OpenMP scheduling across the outer dimension, this leads to all threads accessing the same ccNUMA domain most of the time during the GS sweeps. It turns out that this effect can be incorporated into the model as well. To this end, the sustained memory bandwidth is measured across all ccNUMA domains with data residing in only one domain. This data can then be used as a bandwidth limit when using multiple ccNUMA domains on SKL and EPYC. Figure 8a compares performance estimates to measurements for $GS_F$ across the cores of a socket on all architectures. The deviation from the model is generally smaller than 10 % when using multiple NUMA domains, and below 5 % when looking at a single ccNUMA domain. The results indicate that the model with enhancements described above (barrier overhead, ccNUMA contention) delivers a good qualitative and quantitative description of the performance behavior.




(a) GS forward kernel

(b) The full PCG algorithm

**Figure 8.** Comparison of estimates to empirical data for (a) GS forward kernel and (b) the full PCG algorithm

### *4.2.4. Composition*

With estimates for individual kernels in place we can now present multicore-scaling data for the full PCG algorithm. Composing the model from single-loop predictions is simple due to the

time-based formulation of the ECM model [21]. In the case of PCG we have three invocations of DAXPBY, two of DOT, one GS forward- and backward-sweep each, as well as one of STENCIL. Figure 8b shows the comparison of the model with measurements for all four architectures. Again, the general model error is below 10 %, and less than 5 % when looking at single ccNUMA domains. The slightly larger deviation beyond 12 cores on TX2 can be attributed to the fact that we use compiler-generated code instead of hand-crafted assembly for the CG solver on this machine. The lack of prefetching causes a 10–15 % performance breakdown of data-bound loops beyond the saturation point (see Fig. 7c), which we ignore in the model. On EPYC and SKL we observe very low performance for OpenMP reductions across ccNUMA domains (much larger than the considered OpenMP barrier) with the Intel compiler, causing the slight deviation beyond one domain.

## 5. Related Work

There are two capable analytic (in the sense of "first principles") performance models for steady-state loop code on multicore CPUs: the Roofline model [10, 25] and the ECM model [6, 13, 22]. Both have been subject to intense study, refinements, and validation, and their areas of applicability are well understood. However, while there is ample data available for Roofline on a wide variety of architectures [15, 18], one drawback of previous applications of the ECM model [2, 4, 12, 21, 22, 24, 26] is that they were mostly restricted to Intel processors. We provide the first thorough cross-architecture study of the model.

The Roofline model has the attractive property that it can be easily separated into a machine part (memory and cache bandwidths, peak performance) and an application part (computational intensity). There is no previous work that has done the same with the ECM model. A comparison between Roofline and ECM for several stencil algorithms can be found in [22]. A drawback of the Roofline model is that it requires a large amount of phenomenological input such as measured bandwidths for all core counts and all memory hierarchy levels [15], while the ECM model only needs the saturated memory bandwidth and the machine model (i.e., overlap assumptions).

Advanced curve-fitting and machine-learning techniques combined with hardware performance monitoring data have been used in the past to model the performance of code [1, 19]. Although these approaches are useful in practical settings, e.g., for predicting program runtimes with a goal of optimized resource scheduling, the deepest insights are gained through first-principles models such as Roofline or ECM.

## Conclusion

We have shown that it is possible to set up a well-defined workflow for modeling the serial and parallel runtime of steady-state (sequences of) loops with regular data access patterns using the analytic ECM performance model. One can, with minor exceptions, cleanly separate machine properties from application properties. Four multicore server processors were investigated, and we could demonstrate that despite their obvious differences the main properties needed to set up a useful machine model can be summarized in a few parameters. The performance, including scalability across cores and ccNUMA domains, of an OpenMP-parallel preconditioned CG solver with wavefront-parallel Gauss-Seidel sweeps could be described with a modeling error of 5 % or less in most cases. We have to emphasize that no other first-principles model is capable of delivering such predictions with comparable accuracy and generality.

We found the overlapping property of transfers across data paths in the cache hierarchy to be the pivotal architectural feature governing single-core performance for data-bound loops. A design with very strong in-core performance (e.g., via wide SIMD execution) but a non-overlapping memory hierarchy may well be inferior to a weak core with strong overlap, as our comparison of Skylake SP and AMD Epyc shows. The architecture with the lowest in-core computational performance, POWER9, came out first in serial and parallel memory-bound performance. The Cavium ThunderX2 processor can compensate its rather low in-core performance with good memory bandwidth and a large core count.

All modeling procedures carried out in this paper were done by hand. Some components, e.g., the construction of a runtime prediction from code and a (given) machine model, can be supported by tools [9]; others, such as the derivation of overlapping properties, would be very hard to automate. However, the purpose of performance modeling is not just prediction but also insight, and manual analysis sharpens the view on the relevant details.

## Acknowledgements

We thank Thomas Gruber for helping to port the LIKWID tool suite to IBM's POWER9 architecture.

We also thank the Center for Information Services and High Performance Computing (ZIH) at TU Dresden for providing access to their POWER9 cluster.



## References

1. Alam, S.R., Bhatia, N., Vetter, J.S.: An exploration of performance attributes for symbolic modeling of emerging processing devices. In: Perrott, R., Chapman, B.M., Subhlok, J., de Mello, R.F., Yang, L.T. (eds.) High Performance Computing and Communications. Lecture Notes in Computer Science, vol. 4782, pp. 683–694. Springer, Berlin, Heidelberg (2007), DOI: 10.1007/978-3-540-75444-2_64

2. Cremonesi, F., Hager, G., Wellein, G., Schrmann, F.: Analytic performance modeling and analysis of detailed neuron simulations. The International Journal of High Performance Computing Applications 34(4), 428–449 (2020), DOI: 10.1177/1094342020912528

3. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Review 51(1), 129–159 (2009), DOI: 10.1137/070693199

4. Gmeiner, B., Rüde, U., Stengel, H., Waluga, C., Wohlmuth, B.: Performance and scalability of hierarchical hybrid multigrid solvers for Stokes systems. SIAM Journal on Scientific Computing 37(2), C143–C168 (2015), DOI: 10.1137/130941353

5. Gruber, T., et al.: LIKWID performance tools (2019), `http://tiny.cc/LIKWID`

6. Hager, G., Treibig, J., Habich, J., Wellein, G.: Exploring performance and power properties of modern multicore chips via simple machine models. Concurrency Computat.: Pract. Exper. 28(2), 189–210 (2013), DOI: 10.1002/cpe.3180

7. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Inc., Boca Raton, FL, USA, 1st edn. (2010)

8. Hammer, J.: pycachesim – Python Cache Hierarchy Simulator (2019), `https://github.com/RRZE-HPC/pycachesim`

9. Hammer, J., Eitzinger, J., Hager, G., Wellein, G.: Kerncraft: A tool for analytic performance modeling of loop kernels. In: Niethammer, C., Gracia, J., Hilbrich, T., Knüpfer, A., Resch, M.M., Nagel, W.E. (eds.) Tools for High Performance Computing 2016: Proceedings of the 10th International Workshop on Parallel Tools for High Performance Computing, October 2016, Stuttgart, Germany. pp. 1–22. Springer International Publishing, Cham (2017), DOI: 10.1007/978-3-319-56702-0_1

10. Hockney, R.W., Curington, I.J.: $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. Parallel Computing 10(3), 277–286 (1989), DOI: 10.1016/0167-8191(89)90100-2

11. Hofmann, J.: ibench – measure instruction latency and throughput (2019), `https://github.com/hofm/ibench`

12. Hofmann, J., Fey, D.: An ECM-based energy-efficiency optimization approach for bandwidth-limited streaming kernels on recent Intel Xeon processors. In: Proceedings of the 4th International Workshop on Energy Efficient Supercomputing, 14 Nov. 2016, Salt Lake City, UT, USA. pp. 31–38. IEEE Press, Piscataway, NJ, USA (2016), DOI: 10.1109/E2SC.2016.010

13. Hofmann, J., Hager, G., Fey, D.: On the accuracy and usefulness of analytic energy models for contemporary multicore processors. In: Yokota, R., Weiland, M., Keyes, D., Trinitis, C. (eds.) High Performance Computing. pp. 22–43. Springer International Publishing, Cham (2018), DOI: 10.1007/978-3-319-92040-5_2

14. Hornich, J., Hammer, J., Hager, G., Gruber, T., Wellein, G.: Collecting and presenting reproducible intranode stencil performance: INSPECT. Supercomputing Frontiers and Innovations 6(3), 4–25 (2019), DOI: 10.14529/jsfi190301

15. Ilic, A., Pratas, F., Sousa, L.: Cache-aware roofline model: Upgrading the loft. IEEE Comput. Archit. Lett. 13(1), 21–24 (2014), DOI: 10.1109/L-CA.2013.6

16. Intel Corporation: Intel Xeon Processor Scalable Family (2019), `http://tiny.cc/IntelSP`

17. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (1995)

18. Ofenbeck, G., Steinmann, R., Cabezas, V.C., Spampinato, D.G., Püschel, M.: Applying the roofline model. In: IEEE International Symposium on Performance Analysis of Systems and Software, 23-25 March 2014, Monterey, CA, USA. pp. 76–85. IEEE (2014), DOI: 10.1109/ISPASS.2014.6844463

19. Peraza, J., Tiwari, A., Laurenzano, M., Carrington, L., Ward, W.A., Campbell, R.: Understanding the performance of stencil computations on Intel's Xeon Phi. In: 2013 IEEE International Conference on Cluster Computing, 23-27 Sept. 2013, Indianapolis, IN, USA. pp. 1–5. IEEE (2013), DOI: 10.1109/CLUSTER.2013.6702651

20. Sadasivam, S.K., Thompto, B.W., Kalla, R., Starke, W.J.: IBM Power9 processor architecture. IEEE Micro 37(2), 40–51 (2017), DOI: 10.1109/MM.2017.40

21. Seiferth, J., Alappat, C., Korch, M., Rauber, T.: Applicability of the ECM performance model to explicit ODE methods on current multi-core processors. In: Yokota, R., Weiland, M., Keyes, D., Trinitis, C. (eds.) High Performance Computing, 24-28 June 2018, Frankfurt, Germany. Lecture Notes in Computer Science, vol. 10876, pp. 163–183. Springer International Publishing, Cham (2018), DOI: 10.1007/978-3-319-92040-5_9

22. Stengel, H., Treibig, J., Hager, G., Wellein, G.: Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. In: Proceedings of the 29th ACM International Conference on Supercomputing, June 2015, Newport Beach, CA, USA. ACM, New York, NY, USA (2015), DOI: 10.1145/2751205.2751240

23. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with PAPI-C. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009. pp. 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), DOI: 10.1007/978-3-642-11261-4_11

24. Wichmann, K.R., Kronbichler, M., Löhner, R., Wall, W.A.: Practical applicability of optimizations and performance models to complex stencil based loop kernels in CFD. International Journal of High Performance Computing Applications 33(4), 602–618 (2018), DOI: 10.1177/1094342018774126

25. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (2009), DOI: 10.1145/1498765.1498785

26. Wittmann, M., Hager, G., Zeiser, T., Treibig, J., Wellein, G.: Chip-level and multi-node analysis of energy-optimized lattice Boltzmann CFD simulations. Concurrency and Computation: Practice and Experience 28(7), 2295–2315 (2016), DOI: 10.1002/cpe.3489