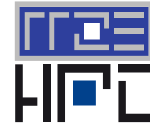
Professur für
Höchstleistungsrechnen
Friedrich-Alexander-Universität
Erlangen-Nürnberg



MASTER THESIS

Cross-Architecture Automatic Critical Path Detection For In-Core Performance Analysis

Jan Laukemann

Erlangen, January 31, 2020

Examiner: Prof. Dr. Gerhard Wellein
Advisor: Julian Hammer

Declaration of Collaboration

OSACA was developed in collaboration between Jan Laukemann and Julian Hammer. The first prototype was implemented by Jan Laukemann during his Bachelor Thesis, based on concepts developed by Julian Hammer. The latest version of OSACA was created in close collaboration during this Master Thesis and Julian Hammer's PhD Thesis. Both software and concept development related to OSACA since November 2018 can not be separated into individual contributions anymore and both authors consider this jointed work.

Eidesstattliche Erklärung / Statutory Declaration

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbständig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

I hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Der Friedrich-Alexander-Universität, vertreten durch die Professur für Höchstleistungsrechnen, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

Erlangen, February 5, 2020

Jan Laukemann

Zusammenfassung

Bei der Optimierung wissenschaftlicher Software stellt Leistungsmodellierung einen essenziellen Teil dar. Um statisch Leistungsanalysen von Programmcode zu erstellen, ist es wichtig, über eine präzise Vorhersage der in-core Laufzeit zu verfügen. Diese ist jedoch stark von der jeweiligen Prozessorarchitekture abhängig. Die in früheren Arbeiten entwickelte Software *Open Source Architecture Code Analyzer (OSACA)* ist ein statisches Leistungsanalysetool zur Vorhersage von Laufzeiten einer sequentiellen Programmschleife. Es unterstützte bereits die einfache Durchsatzanalyse bei x86 (Intel und AMD) Mikroarchitekturen. Wir haben den Funktionsumfang von OSACA erheblich erweitert, sodass nun Abhängigkeitsketten innerhalb und über Schleifen hinweg identifiziert werden können und so ein kritischer Pfad und schleifenübergreifende Abhängigkeiten erkannt werden. Dies führt einer deutlichen Verbesserung der Laufzeitvorhersage. Des Weiteren wurde die Durchsatzvorhersage optimiert und die Unterstützung von ARM-basierten Mikroarchitekturen hinzugefügt, was OSACA zu einem vielseitig einsetzbaren architekturübergreifenden Modellierungswerkzeug werden lässt. Während die Durchsatzvorhersage und schleifenübergreifende Abhängigkeitsanalyse eine untere Grenze der Laufzeit bilden, kann die Latenz des kritischen Pfads als obere Grenze der Ausführungsdauer angesehen werden. Wir evaluieren die Qualität unserer Analyse für Programmcode auf Intel Cascade Lake, AMD Zen und Marvell ThunderX2 Mikroarchitekturen. Die Modellparameter basieren auf Herstellerdokumentationen und kleinteiligen Messungen. Die Vorhersagen werden sowohl den tatsächlichen Messungen der Laufzeit als auch den Analyseresultaten der vergleichbaren Tools *Intel Architecture Code Analyzer (IACA)* und *LLVM Machine Code Analyzer (LLVM-MCA)* gegenübergestellt. Dies zeigt, dass OSACA das aktuell leistungsfähigste und vielseitigste Vorhersagewerkzeug für in-core Laufzeiten darstellt.

Abstract

The creation of performance models is an essential part of optimizing scientific software. To run static performance analyses on code snippets, it is crucial to obtain an accurate in-core execution time prediction, which is highly dependent on the micro-architecture of the chip. Our previously developed *Open Source Architecture Code Analyzer (OSACA)* tool is a static performance analyzer for predicting the execution time of sequential loops. It previously supported only x86 (Intel and AMD) micro-architectures and simple, full-throughput prediction. We heavily extended its functionality by the detection of dependencies within and across assembly loops to identify the critical path and loop-carried dependencies. This enables a much improved runtime prediction in steady state execution. Furthermore, we enhanced the throughput prediction and added support for ARM-based micro-architectures, which turns OSACA into a versatile cross-architecture modeling tool. While its throughput and loop-carried dependency analysis give a lower bound runtime prediction, its critical path analysis can function as an upper bound for the execution time. We evaluate the quality of the analysis for code on Intel Cascade Lake, AMD Zen, and Marvell ThunderX2 micro-architectures based on machine models from available documentation and semi-automatic benchmarking. The predictions are compared with measurements and the analysis results from the related tools *Intel Architecture Code Analyzer (IACA)* and *LLVM Machine Code Analyzer (LLVM-MCA)*. The comparison shows that OSACA is to date the most capable and versatile in-core runtime prediction tool available.

Acknowledgement

I would like to thank my advisors Julian Hammer and Georg Hager for their great support far beyond average, the successful guidance and their helpful discussions at any time during day and night. I would also like to thank my professor Gerhard Wellein for supporting me and my work throughout the whole process of development and research. Finally, I want to thank the whole HPC team at the University of Erlangen-Nürnberg for the enjoyable time during working on this thesis.

Annotation

Parts of the results presented in this thesis were already published in [1] during the 10th IEEE International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS19) held as part of ACM/IEEE Supercomputing 2019 (SC19). We will not refer to that source explicitly.

CONTENTS

1. Introduction 1
1.1. Motivation . . . 1
1.2. Scope of Work . . . 6
1.3. Related Work . . . 6
1.4. Results . . . 7
1.5. Outline . . . 10

2. Background 11
2.1. Modern Micro-Architectures . . . 11
2.1.1. Branch Prediction and Speculative Execution . . . 11
2.1.2. Out-of-order Execution . . . 12
2.1.3. Macro-Op Fusion . . . 12
2.1.4. Vectorization . . . 12
2.2. Simplified Port Model . . . 13
2.2.1. Intel Cascade Lake micro-architecture . . . 13
2.2.2. AMD Zen micro-architecture . . . 14
2.2.3. Marvell ThunderX2 micro-architecture . . . 14
2.3. Throughput, Critical Path and Loop-Carried Dependencies . . . 15
2.3.1. Throughput . . . 15
2.3.2. Critical Path . . . 16
2.3.3. Loop-Carried Dependencies . . . 17

3. Implementation 19
3.1. Prerequisites . . . 19
3.1.1. Benchmarking Throughput and Latency . . . 19
3.1.2. Benchmarking Port Utilization . . . 21
3.1.3. Instruction Form Data . . . 22
3.2. Analysis . . . 22
3.2.1. Throughput Analysis . . . 22
3.2.2. Critical Path Analysis . . . 23
3.2.3. Loop-Carried Dependency Analysis . . . 24

4. Evaluation 25
4.1. Data acquisition for IACA and LLVM-MCA . . . 25
4.1.1. IACA . . . 26
4.1.2. LLVM-MCA . . . 27

4.2. Analysis and Comparison of OSACA Results 27
4.2.1. Copy 29
4.2.2. Vector add 30
4.2.3. Vector update 32
4.2.4. Sum reduction 33
4.2.5. DAXPY 36
4.2.6. STREAM triad 36
4.2.7. Schönauer triad 37
4.2.8. Gauss-Seidel method 37
4.2.9. 2D-5pt stencil 39

5. Conclusion and Future Work 41

5.1. Summary 41
5.2. Future Work 42

A. OSACA manual 43

B. Assembly Code Summary Table 49

Bibliography 51

1 INTRODUCTION

This work intends to enhance the previously developed performance analysis tool OSACA [2] in the function of supporting ARM-based, i.e., non-x86, micro-architectures and of applying a critical path and loop-carried dependency detection on assembly code snippets. OSACA focuses on loop-based, scientific codes in steady-state execution.

1.1. Motivation

In areas of computer science and computational applications in general, one of the most important metrics for efficient software is performance. Optimization of modern programs can be extremely difficult in a world of different and more and more heterogeneous systems which got not only more powerful, but also more complex from generation to generation. Indeed, in order to gain good performance, i.e., work done in a specific amount of time, it is crucial to obtain insight knowledge of the executing systems. One way of predicting the behavior of a CPU is to apply a *performance model*. Performance models not only provide users with information about memory traffic, execution time and software-hardware interaction, but help to identify bottlenecks in the design or to improve power and energy models and therefore to determine trade-offs between performance and energy consumption. Thus, they are powerful tools for all kind of developers, processor designers and researchers.

In general, performance models can be divided into two main groups [3]: *White-box* or *first principles* models, i.e., constructed from specifications and information visible for third-parties and *black-box* or *statistical* models, which are based on empirical data and therefore neither need nor additionally give insight into an investigated hardware. Furthermore, within those two domains, given a code snippet to investigate (which will be called *kernel* in this work), one can differentiate between a *static* analysis or a *simulation* of the given kernel. While for the latter one a more precise model and prediction of the system's behavior is possible, it may also lead to a overly complex and time consuming approach and sometimes cannot be developed due to insufficient information. With a static analysis applying a model of an underlying hardware is mostly much easier and requires less detailed a priori knowledge and therefore in many cases is more

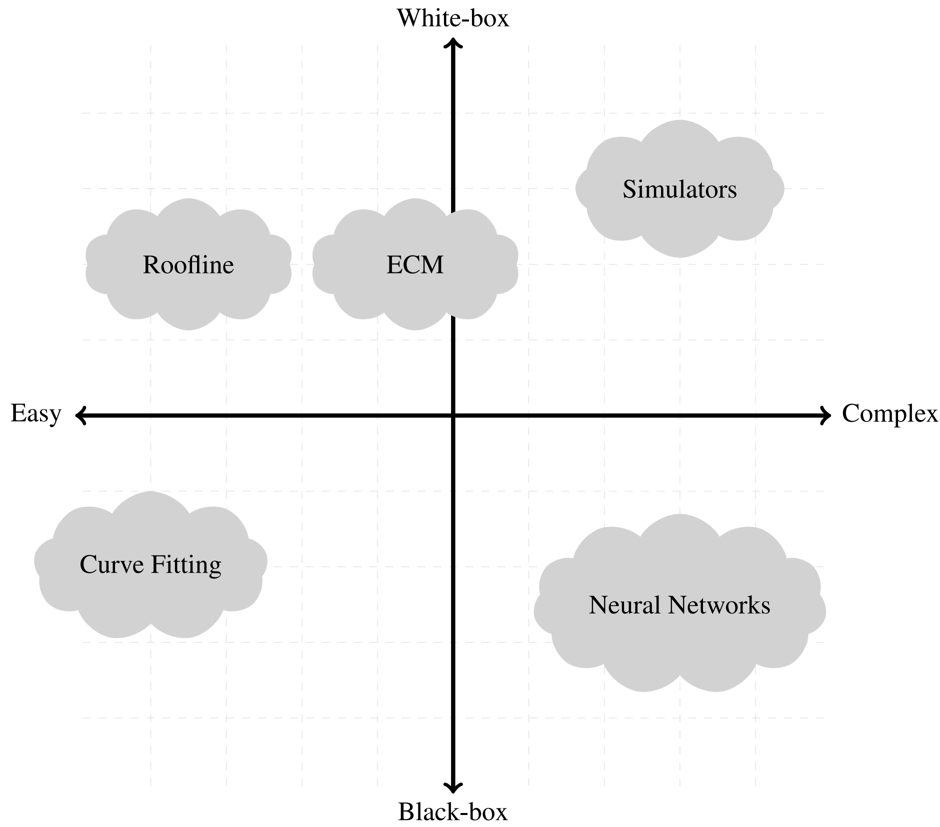


Figure 1.1.: Possible domains of performance models including some sample performance evaluation methods in an approximate order.

light-weight and faster than simulators. An overview of general metrics for performance models including example evaluation methods can be found in Figure 1.1.

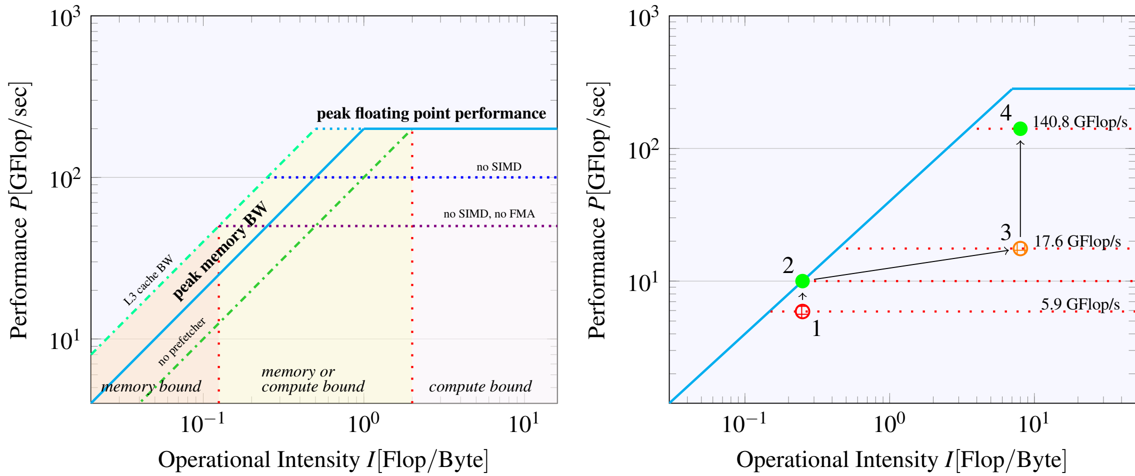
The most famous, yet, simplest model mentioned here and part of the static analyses is the *Roofline* [4] model. It puts in-core floating-point performance and off-chip memory traffic as part of a machine model and *operational intensity* from a code model in a relation. The peak processor performance can be determined by the hardware specifications or through micro-benchmarks and the memory traffic traditionally — despite of the fact nowadays other cache-aware models exist [5] — refers to the main memory bandwidth, assuming normal workloads do not fit fully in a cache size. Therefore, the peak memory bandwidth can be either determined by using the hardware specifications or by running benchmarks like STREAM [6]. The operational intensity is the number of operations per byte of main memory traffic, i.e., the ratio of executed floating-point operations F and the total traffic of bytes B for a certain kernel [7]:

$$I = \frac{F}{B}$$

Therefore, the investigated loop body can be summed up to one metric I. With that, the Roofline model gives an upper bound for the attainable performance P (in general in Flop/sec). It assumes the peak floating-point performance P_{peak} as single in-core bottleneck and the peak memory bandwidth B_{peak} to be the single data traffic bottleneck for streaming kernels in steady-state execution. It can be written as the minimum of P_{peak} and the product of B_{peak} and the operational intensity:

$$P = \min\left(P_{peak}, B_{peak} \times I\right)$$

Both P_{peak} and B_{peak} therefore determine an upper bound of the Roofline model. By looking at the inten-



(a) Graphical representation of possible ceilings in the Roofline model.

(b) Exemplary optimization process of a kernel by reaching the memory limit, increasing the operational intensity of the code and finally doing core-based optimizations.

Figure 1.2.: Example Roofline models based on [8].

sity I, a user can separate their code into one of the domains and, therefore, determine being a memory-bound (below the inclined ceiling) or compute-bound (below the flat ceiling). Unoptimized kernels often fall short of the expectation given by the Roofline model because the code is limited by in-core effects (that can be visualized as a lower flat ceiling). A graphical representation of such general ceilings in the Roofline model is shown in Figure 1.2a. While different P_{peak} can higher or lower the flat ceiling, e.g., by using SIMD or FMA instructions, different B_{peak}, like taking cache levels into account or using prefetching, affect the inclined ceiling. Consequently, the optimization goals consists of (i) reaching the Roofline limit of the inclined ceiling, (ii) increasing the intensity until the code is not memory bound anymore and finally (iv) reaching the flat ceiling Roofline limit again by doing core-level optimizations. Thus, the overall goal is to achieve thorough understanding of performance bottlenecks. This process is shown in Figure 1.2b as an example of applying a Roofline model to a specific architecture and kernel.

For a better understanding of how to apply the Roofline model, we will look at an exemplary code snippet. Here we want to execute a large (i.e., too big to fit into any cache) single precision vector reduction on an 8-core 2.2 GHz Intel Sandy Bridge (SNB) socket with the following kernel:

```
float a[N];
float s;

for(i=0; i<N; ++i) {
    s = s + a[i];
}
```

Using the hardware specifications we can determine a maximum memory bandwidth of $B_{peak} = 40\,\text{GB/s}$ and a peak performance of $P_{peak} = 2.2\,\text{GHz} \times 16 \times 8 = 281.6\,\text{GFlop/s}$. For the latter we multiplied the CPU clock frequency (2.2 GHz), the number of floating-point operations (MUL/ADD) per one cycle per core (16) and the number of cores per socket (8). These two numbers define the rooflines for our model. Additionally, we know the operational intensity $I = 1\,\text{Flop}/4\,\text{Byte} = ¼\,\text{Flop/Byte}$ resulting from one ADD and one four-byte LOAD per iteration of our most naive implementation. Thus, we can assume the kernel takes 3 cycles per iteration (*cy/it*) due to an ADD-pipeline depth of 3 cy combined with the loop-carried dependency on the target variable `s` and therefore has a performance of $8 \times 2.2\,\text{GHz}/3\,\frac{\text{cy}}{\text{it}} \times 1\,\text{Flop/it} = 5.9\,\text{GFlop/s}$. By optimizing the code, e.g., by using unrolling with modulo variable expansion, we can shift the bottleneck from the compute part to the memory, which allows us to get to the inclined ceiling roofline at $B_{peak} \times I = 40\,\text{GB/s} \times ¼\,\text{Flop/Byte} = 10\,\text{GFlop/s}$. After this, we cannot do any further optimization since

there is no option to increase the operational intensity of the code without having any further use of the variables.

Another, more complex approach for static, first-principles-based performance modeling is the *Execution-Cache-Memory (ECM)* model [9]. Similar to the Roofline model, it consists of an in-core execution and a data transfer part, taking into account not only several cache levels, but also victim, write-through, and multi-ported caches [10]. The in-core execution T_{core} assumes that all data is available in the innermost cache and ignores all possible bottlenecks defined by inter-cache transfers and main memory. It is the maximum of overlapping and non-overlapping execution time

$$T_{\text{core}} = \max\left(T_{\text{comp}}, T_{\text{RegL1}}\right)$$

and in this case refers to a work load of 16 iterations. This is due to the fact assume one cache line (CL) of 64 B as smallest unit of data loaded from the caches and using single-precision floating-points, i.e., 4-byte-elements, we need 16 iterations to fill one CL. Non-overlapping execution time T_{RegL1} refers to the cycles in which LOAD instructions are retired and data is loaded from L1 cache to the registers, while overlapping execution time T_{comp} refers to in-core computation without LOAD.

The data transfer time T_{data} describes the bandwidth (in cycles) of transferring data between L1 and L2 caches, L2 and L3 caches, and L3 cache and main memory, respectively, for an architecture with three inclusive cache levels. It depends on the micro-architecture if the single data transfer times overlap, partly overlap or can be seen as completely separated transfer times. For example, for Intel x86 micro-architectures up to Broadwell, T_{data} can be written as the sum of the data transfer times:

$$T_{\text{data}} = T_{\text{L1L2}} + T_{\text{L2L3}} + T_{\text{L3Mem}}$$

Including this equation, an overall ECM performance model estimating an execution time can be given as the maximum of the in-core computation time and the overall data transfer time, i.e., the sum of the in-core data transfer T_{RegL1} and the bandwidth T_{data}:

$$T_{\text{ECM}} = \max\left(T_{\text{comp}}, T_{\text{RegL1}} + T_{\text{data}}\right)$$

A shorthand notation for the execution and data transfer time—shown in [11]—can be written as

$$\left\{T_{\text{comp}} \,||\, T_{\text{RegL1}} \,|\, T_{\text{L1L2}} \,|\, T_{\text{L2L3}} \,|\, T_{\text{L3Mem}}\right\}.$$

Furthermore, an ECM model prediction for various levels of cache and memory hierarchy, which is calculated out of the formula above using different levels of cache, can be summarized in a similar way as

$$\left\{T^{\text{ECM}}_{\text{core}} \rceil T^{\text{ECM}}_{\text{L2}} \rceil T^{\text{ECM}}_{\text{L3}} \rceil T^{\text{ECM}}_{\text{Mem}}\right\}.$$

The subscript indicates the memory level taken into account for calculating T_{data}. We therefore differentiate above between the in-core execution using only the L1 cache ($T^{\text{ECM}}_{\text{core}}$), using L1 and L2 cache ($T^{\text{ECM}}_{\text{L2}}$), all caches up to L3 ($T^{\text{ECM}}_{\text{L3}}$) or the main memory ($T^{\text{ECM}}_{\text{Mem}}$).

Applying the ECM model on the same hardware system (SNB) to the same kernel shown earlier in its optimized version, i.e., including modulo variable expanded unrolling and SIMD instructions and with sufficient unrolling to achieve full throughput for these instructions, results in an overlap in-core computation time of $T_{\text{comp}} = 2\,\text{cy}/16\,\text{it}$. Due to fact of a workload of 16 it, i.e., 64 B, we have two vector ADDs including eight single-precision floating-points running in parallel, each with a latency of 3 cy. The non-overlapping time T_{RegL1} equals $2\,\text{cy}/16\,\text{it}$ for two AVX LOAD instruction done in parallel. From the hardware specification we can see the possible L1-L2 transfer time as well as the L2-L3 transfer time is 32 B/cy [12]. But since for our SNB system the smallest unit of loads from the cache is one CL, i.e., 64 B, we need to take the time for loading one CL into account, therefore, we define $T_{\text{L1L2}} = 2\,\text{cy}/16\,\text{it}$ and $T_{\text{L2L3}} = 2\,\text{cy}/16\,\text{it}$. Using the peak memory bandwidth of $B_{peak} = 40\,\text{GB/s}$ with the clock speed of 2.2 GHz, we can convert it to $T_{\text{L3Mem}} = 3.5\,\text{cy}/16\,\text{it}$. Thus, for this kernel an ECM model of $\{2.0||2.0|2.0|2.0|3.5\}$ cy/16 it can be determined. The overall in-core execution time emerges to $T_{\text{core}} = \max(2.0, 2.0)\,\text{cy}/16\,\text{it} = 2\,\text{cy}/16\,\text{it}$. Sum-

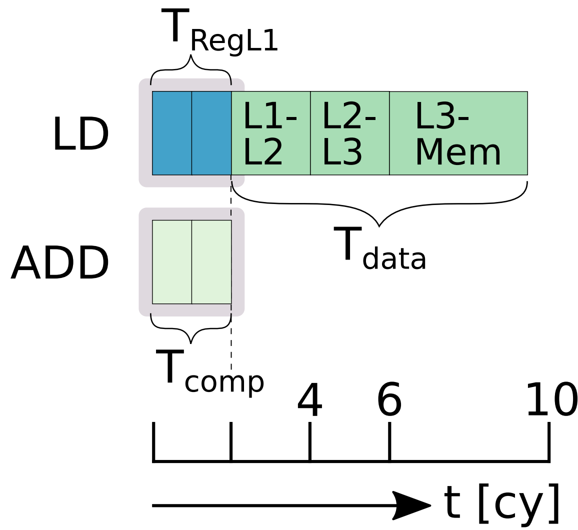


Figure 1.3: Single-core ECM model for a single-precision floating point vector reduction loop kernel on Intel SNB. We assume a fully optimized code, i.e., (i) an sufficient unrolled kernel including modulo variable expansion to overcome all ADD latencies and (ii) SIMD vectorization to use 256-bit `vaddps` instructions. This way, this visualization represents one CL of work.

marizing the levels of memory hierarchy, we can finally state the model as $\{2.0\rceil 4.0\rceil 6.0\rceil 9.5\}$ cy/16 it for single-core, which subsequently leads in the case of 16 Flop/16 it to a prediction of

$$\frac{1\,\text{Flop/it}}{\{2.0\rceil 4.0\rceil 6.0\rceil 9.5\}\ \text{cy}/16\,\text{it}} \times 2.2\,\text{GHz} = \{17.6\rceil 8.8\rceil 5.9\rceil 3.7\}\ \text{GFlop/s}$$

one core. Finally, an (optimistic) saturation assumption can be calculated out of the number of cores needed until a shared main memory bandwidth bottleneck is hit [13]:

$$T_{\text{ECM}}(n) = \max\left(\frac{T_{Mem}^{ECM}}{n}, T_{\text{L3Mem}}\right) \Rightarrow n_s = \left\lceil \frac{T_{Mem}^{ECM}}{T_{\text{L3Mem}}} \right\rceil = \left\lceil \frac{9.5}{3.5} \right\rceil = 3$$

For further understanding, a visualization of the separation of the individual in-core and data transfer parts for a eight times unrolled vector reduction kernel is shown in Figure 1.3.

While the parameters for the machine model can be found in hardware specifications, the quantification of the code parameters often requires manual analysis, which may become unsolvable by hand quickly as soon as the kernel complexity grows. Hammer et al. [14] provides the *Kerncraft* tool[1], which is capable of calculating static performance analyses like the Roofline model or the ECM model. Within Kerncraft, the in-core execution part can be predicted by different tools: The *Intel Architecture Code Analyzer (IACA)* [15], an Intel proprietary, but free tool for their own architectures, the *Open Source Architecture Code Analyzer (OSACA)* [2], and LLVM Machine Code Analyzer (LLVM-MCA). OSACA is an open source tool inspired by IACA with the motivation to overcome its limitations and nontransparent performance analysis. Before this work, both tools provided for a given, marked innermost loop body an analysis of the block throughput and the instruction binding to the CPU ports. In IACA version 2.2 (Dec 2016), Intel dropped the support for latency analysis, which limited the in-core runtime prediction to a lower-bound estimation. Furthermore, in April 2019, Intel announced the end-of-life of their tool[2], resulting inevitably in no further support for any micro-architecture. OSACA, on the other hand, supported various x86 architectures in a transparent and generic way based on micro-benchmarking, but was also not able to provide a more detailed latency analysis for more complex kernels.

To overcome all these limitations, we enhanced OSACA tremendously. The goal of this thesis is to extend the functionality of OSACA by supporting critical path (CP) analysis and detecting loop-carried dependencies (LCDs). Furthermore, OSACA should be able to perform analysis on ARM-based platforms and be made more generic and flexible to enable further enhancements.

[1]The current version of Kerncraft is available at https://github.com/RRZE-HPC/kerncraft/
[2]See https://software.intel.com/en-us/articles/intel-architecture-code-analyzer (accessed January 29, 2020)

1.2. Scope of Work

This thesis covers two main tasks:

(i) After OSACA was already enhanced to support non-Intel micro-architectures in form of AMD Zen in [16], we proceeded to provide support for ARM architectures, which included to extend the OSACA internal assembly parser to understand AArch64 syntax, the 64-bit execution state of the ARMv8 instruction set architecture (ISA) as well as the formulation of a port model for the ARM-based Marvell ThunderX2 (TX2) system. Because of the lack of a testbed for ARM's new scalable vector extension (SVE) instruction set[3], we could not ensure a support of any of these *instruction forms*. An instruction form, as introduced in [2], is a unique expression of an assembly instruction mnemonic and its operand types in a specific order.

(ii) To refine the in-core runtime prediction of a kernel, OSACA supports the automatic detection of the critical path, i.e., the longest dependency chain of instruction forms within one loop kernel, and (simple) loop-carried dependencies. This can improve the overall prediction tremendously in case of kernels which are not throughput bound and provides the user with an upper-bound prediction of their code.

Furthermore, we wanted to alter the database structure to store more hardware-specific information and develop it into a machine model, being able to store more details about instruction form data and distinguish between ISA-specific and micro-architecture-specific peculiarities. Instruction forms containing memory addressing in their operands should be uncoupled to model overlapping and non-overlapping of LOAD instructions. For micro-benchmarking, the ability of generating benchmark files and importing results to and from *ibench* [17], a tool for micro-benchmarking assembly instruction forms in x86, ARM and Power syntax, should be extended to also support the instrumentation of the *asmbench* framework [18].

In general, all changes should be made keeping in mind the generic approach of OSACA and the possibility of extend it for other micro-architectures and ISAs in the future. To fulfil this, we refrained from the approach of analysing ELF files as in previous versions and focused solely on the analysis of marked assembly files. This allows as additionally to use comment line markers for the convenience of the user. Nevertheless, to stay consistent with the widely used IACA tool, we further want to support byte markers in the assembly for identifying the kernel, since IACA works on compiled object files.

All code of this project continues to be available and freely accessible for further collaboration as an open source project at GitHub[4].

1.3. Related Work

As mentioned in Section 1.1, OSACA was inspired by IACA, the Intel Architecture Code Analyzer. Developed by Israel Hirsh and and Gideon S. [sic], Intel released IACA in 2012. At that time it included both throughput and critical path analysis. In 2017, Intel dropped the latency analysis support for unknown reasons. Finally, in April 2019, the developers announced the end-of-life of their tool, therefore no future enhancements or new micro-architecture support can be expected. At the time of writing, IACA supports in its current version 3.0 Intel micro-architectures from Haswell (HSW) to Skylake-X (SKX). One of the biggest limitations of IACA is the entirely closed-source development, so no insight or validation of single performance values is possible.

LLVM-MCA [19] is a powerful performance analysis tool based on LLVM's existing scheduling models. It can output various information about the kernel on a specific architecture like latency information, the block throughput, and a timeline view. The timeline view emulates the kernel execution and can be used to manually identify critical paths based on instruction dependencies, but is bound to some pen-and-paper effort. Unfortunately, results are not always accurate or need refinement and in its current release of version 9.0.0,

[3]For more information see the ARM SVE manual, available at: https://developer.arm.com/docs/ddi0584/latest/arm-architecture-reference-manual-supplement-the-scalable-vector-extension-sve-for-armv8-a (accessed January 29, 2020)

[4]https://github.com/RRZE-HPC/OSACA/

some HPC-relevant ARM architectures such as the TX2 are lacking. Nevertheless, in ARM's LLVM HPC version 19.2.0, LLVM-MCA is capable of analysing some of its own compiled kernels, but does not work reliably.

With its *EXEgesis* project [20], Google created an open source project to improve code generation by parsing ISA information out of vendor or unofficial manuals of the hardware. This information could be used to build similar models as Intel IACA or OSACA. On their website, the developers state the support of Intel x86-64 and ARM A64 micro-architectures and the current development of IBM Power. Nonetheless, already six months passed since the last commit, so the future and continuity of the project is unsure.

With *Ithemal*, Mendis et al. [21] provided a hierarchical LSTM-based machine learning approach to predict the block throughput of x86-64 loop kernels and show a higher accuracy than LLVM-MCA and IACA [22]. They support Intel HSW to Skylake (SKL) (without AVX-512) and are able to use IACA byte markers for indicating the code block, but are not capable of detecting a critical path or loop-carried dependencies. Furthermore, as a black-box model, it only returns a number of cycles as runtime prediction and does not provide any insight on the investigated kernel or hardware.

Charif-Rubial et al. [23] introduced the *Code Quality Analyzer (CQA)*, a static performance analysis tool to give the developer a quality estimation of the code based on static binary analysis and out-of-order execution simulation done by *Uop Flow Simulation (UFS)* [24], therefore, it is not focused on predicting an actual runtime like OSACA and only supports x86 micro-architectures.

There exist various simulators like *gem5* [25], *ZSim* [26], and *MARSSx86* [27], which can be all considered as full system simulators. They go beyond the scope of this work and provide more a coarse overview on complete (multi- or many-core) systems, rather than detailed insights pinpointing a bottleneck.

For obtaining instruction form performance data as needed by OSACA, there a several options: Ibench by Johannes Hofmann [17] is a micro-benchmark tool for measuring latency and throughput for single assembly instructions for x86, A64 and IBM Power syntax and already used by OSACA. The asmbench framework by Julian Hammer [18] uses the LLVM just-in-time and cross-platform compilation capabilities to abstract instructions and operands and provides throughput and latency information for specific instruction forms. Furthermore, databases like Intel's Software Optimization Reference Manual [12], uops.info [28] and Agner Fog's "Instruction Tables" [29] give helpful insight about throughput and latency of instruction forms, but need to be treated carefully, since they are know to be not fully correct and therefore should be used only as additional validation to micro-benchmarks.

After the in-core analysis done by OSACA, the Kerncraft tool suite by Julian Hammer [14] is capable of applying further knowledge about the hardware on the analyzed loop kernel to provide a complete ECM prediction for performance analysis.

1.4. Results

A structural overview of OSACA in its current version can be found in Figure 1.4 with the example analysis of a STREAM triad on an Intel Cascade Lake X (CSX) micro-architecture:

```c
double a[N], b[N], c[N];
double s;

for(i=0; i<N; ++i) {
    a[i] = b[i] + s * c[i];
}
```

For starting the analysis, OSACA needs a marked assembly loop kernel. In contrast to IACA, the file does not need to be compiled and can therefore consist only of the marked kernel code, as shown in the top left part of Figure 1.4. By running the command `osaca --arch CSX triad.s` OSACA parses the assembly file and extracts and analyzes the marked kernel. OSACA applies each single assembly instruction form to its hardware model, which consists of two main parts: On the one hand it checks the micro-architecture-specific machine file containing the generic L1 load and store costs for instruction form specific performance data, i.e., the latency and port pressure based on the determined port model. On the

x86/ARM marked assembly

```
movl   $111,%ebx        #START MARKER
.byte  100,103,144      #START MARKER
.L22:
  vmovapd      0(%r13,%rax),%ymm0
  vfmadd213pd  (%r14,%rax),%ymm1,%ymm0
  vmovapd      %ymm0,(%r12,%rax)
  addq         $32,%rax
  cmpq         %rax,%r15
  jne          .L22
movl   $222,%ebx        #END MARKER
.byte  100,103,144      #END MARKER
```

```
mov    x1, #111       //START MARKER
.byte  213,3,32,31    //START MARKER
.L18:
  ldr  q2, [x20, x0]
  ldr  q1, [x21, x0]
  fmla v1.2d, v2.2d, v0.2d
  str  q1, [x19, x0]
  add  x0, x0, #16
  cmp  x22, x0
  bne  .L18
mov    x1, #222       //END MARKER
.byte  213,3,32,31    //END MARKER
```

Machine Files / Databases

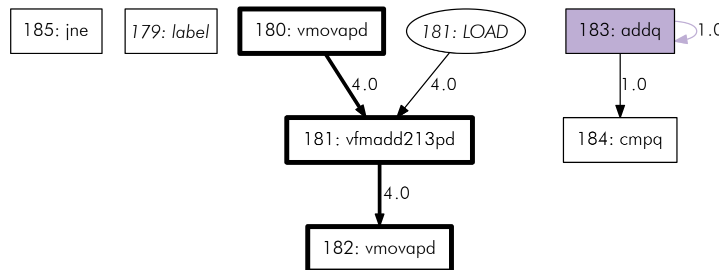
```
generic load     load_latency: {gpr: 4, xmm: 4, ymm: 4, zmm: 4}
 information     load_throughput: {port_pressure: [0,0,0,0.5, ...,0]}
                 ...
mnemonic         - name: vfmadd213pd
                   operands:
                     - class: "register"
                       name: "ymm"
                       source: true
specific operand       destination: false
 description         - class: "register"
                       name: "ymm"
                       source: true
                       destination: false
                     - class: "register"
                       name: "ymm"
                       source: true
                       destination: true
performance        throughput: 0.5
 information       latency: 4    # 0  DV 1  2 D 3 D 4 5 6 7
                   port_pressure:[0.5,0,0.5,0,0,0,0,0,0,0,0,0]
```

In-Order
Out-of-Order
Out-of-Order Scheduler

Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7
ALU	ALU	LOAD	LOAD	STORE	ALU	ALU	AGU
2nd Branch	Fast LEA	AGU	AGU		Fast LEA	Branch	
AVX DIV	Slow LEA				AVX SHUF		
AVX FMA	AVX FMA				AVX-512 FMA		
AVX MUL	AVX MUL				AVX-512 ADD		
AVX ADD	AVX ADD				AVX-512 MUL		
AVX ALU	AVX ALU				AVX-512 ALU		
AVX Shift	AVX Shift						
VNNI	VNNI						

Memory Control

Throughput & Critical Path Analysis




```
 * - Instruction micro-ops not bound to a port

Combined Analysis Report
------------------------
                                    Port pressure in cycles
     |  0   - 0DV |  1   |  2 - 2D  |  3  - 3D |  4  |  5   |  6   | 7 || CP  | LCD |
--------------------------------------------------------------------------------------
 179 |            |      |          |          |     |      |      |   ||     |     |   .L22:
 180 |            |      | 0.5  0.5 | 0.5  0.5 |     |      |      |   || 4.0 |     |   vmovapd 0(%r13,%rax),%ymm0
 181 | 0.50       | 0.5  | 0.5  0.5 | 0.5  0.5 |     |      |      |   || 4.0 |     |   vfmadd213pd (%r14,%rax),%ymm1,%ymm0
 182 |            |      | 0.5      | 0.5      | 1.0 |      |      |   || 5.0 |     |   vmovapd %ymm0,(%r12,%rax)
 183 | 0.25       | 0.25 |          |          |     | 0.25 | 0.25 |   ||     | 1.0 |   addq $32,%rax
 184 | 0.0        | 0.0  |          |          |     | 0.5  | 0.5  |   ||     |     |   cmpq %rax,%r15
 185 |            |      |          |          |     |      |      |   ||     |     | * jne .L22
       0.75         0.75   1.5  1.0   1.5  1.0   1.0   0.75   0.75       13.0   1.0

Loop-Carried Dependencies Analysis Report
-----------------------------------------
 183 |  1.0 | addq $32, %rax                   | [183]
```

Figure 1.4.: Structural design of OSACA based on an exemplary analysis of the STEAM triad for Intel CSX with the port model based on [12].

```
Open Source Architecture Code Analyzer (OSACA) - v0.3
Analyzed file:      triad.s.csx.O3.s
Architecture:       csx
Timestamp:          2019-12-16 14:21:44

 P - Throughput of LOAD operation can be hidden behind a past or future STORE instruction
 * - Instruction micro-ops not bound to a port
 X - No throughput/latency information for this instruction in data file

Combined Analysis Report
------------------------
                                     Port pressure in cycles
    | 0  - 0DV| 1  | 2 - 2D | 3  - 3D| 4 | 5  | 6  | 7 || CP  | LCD |
---------------------------------------------------------------------
179 |         |    |        |        |   |    |    |   ||     |     |  .L22:
180 |         |    |0.5  0.5|0.5  0.5|   |    |    |   || 4.0 |     |  vmovapd  0(%r13,%rax),%ymm0
181 |0.50     |0.5 |0.5  0.5|0.5  0.5|   |    |    |   || 4.0 |     |  vfmadd213pd (%r14,%rax), \
                                                                          %ymm1,%ymm0
182 |         |    |0.5     |0.5     |1.0|    |    |   || 0.0 |     |  vmovapd  %ymm0,(%r12,%rax)
183 |0.25     |0.25|        |        |   |0.25|0.25|   ||     | 1.0 |  addq     $32,%rax
184 |0.0      |0.0 |        |        |   |0.5 |0.5 |   ||     |     |  cmpq     %rax,%r15
185 |         |    |        |        |   |    |    |   ||     |     |* jne      .L22

     0.75      0.75 1.5    1.0 1.5    1.0 1.0 0.75 0.75        8.0    1.0


Loop-Carried Dependencies Analysis Report
-----------------------------------------
183 |  1.0 | addq       $32, %rax                      | [183]
```

Listing 1.1: OSACA output for analysis of the STREAM triad kernel on an Intel CSX micro-architecture.

other hand it looks up the instruction form in the ISA-specific database to assign a source/destination distributions for each operand. If an instruction form can not be found in a machine file, the user can benchmark the instruction using either ibench or asmbench with automatically created benchmark files by OSACA. The workflow for creating a port model will be explained in detail in Section 2.2.

The resulting analysis of OSACA is shown in detail in Listing 1.1 and is made up of three different subparts:

(i) A block throughput analysis as a lower bound of the kernel runtime from the port distribution during the execution of the kernel for each port 0–7 and includes a separation of the divider pipeline and the data ports. Note that all values are given as reciprocal throughput, i.e., "cycles per instruction" or "cycles per iteration". While in earlier versions of OSACA an equal distribution of all possible ports per functional unit execution was assumed, the tool now tries to make more realistic estimations by explicitly scheduling instruction forms on lower-pressure ports if possible to reduce the overall block reciprocal throughput for these ports. This can be observed for instruction form 184 in Listing 1.1. The `cmp` instruction could be executed on all for ALU ports 0, 1, 5 and 6, but due to the higher pressure on ports 0 and 1 because of the `vfmadd` instruction in line 181, it is only scheduled to ports 5 and 6. The possible execution on the now unused ports is indicated through the 0.0 cy instead of a blank field. The maximum reciprocal throughput of all ports results in the overall block throughput of 1.5 cy in the STREAM triad example.

(ii) In column "CP" all instructions on the critical path are shown with their corresponding latencies. The sum can be taken as an upper bound of the execution time[5] and represents the longest dependency chain within one kernel iteration, i.e., instruction forms which must be executed serially. In Listing 1.1 OSACA predicts a CP of 8 cy.

(iii) In the far right column named "LCD" the longest loop-carried dependency with its corresponding latency values is shown. This describes the longest cyclic dependency chain within the loop. It might be part of or equal to the CP, but not necessarily. If there exist multiple LCDs, all of them can be found as a list in the "Loop-Carried Dependencies Analysis Report" below the main report. In the STREAM triad example we can see there is only one LCD, which is caused by the loop counter increment and

[5]This assumption applies only if no other by the hardware model unconsidered factors inside of the architecture affect the kernel in a way that any penalty is added to the code.

conditional branch with a latency of 1 cy, so it does not influence the overall throughput. Nevertheless, in cases of kernels with a longer LCD than the block throughput, this number serves as a lower runtime limit.

Out of this information we can define T_{TP}, T_{CP}, and T_{LCD} as the predicted runtime for the block throughput, the critical path and the longest loop-carried dependency, respectively. Subsequently, a simple equation for the in-core runtime T_{core} can be formulated:

$$T_{\text{core}} = \max\left(T_{\text{TP}}, T_{\text{LCD}}\right) \quad \text{with} \quad T_{\text{core}} \leq T_{\text{CP}}$$

Furthermore, the CP and LCD can be visualized in a dependency graph using the .dot-file OSACA creates. As in the graph in the bottom left of Figure 1.4, LCDs are marked with colors and the CP is indicated through bold frames. Furthermore, the latencies of the instruction forms label the vertices between the nodes.

1.5. Outline

This thesis is organized as follows: Chapter 2 describes the background of modern micro-architectures and their functionalities. Furthermore, it explains in Section 2.2 the organization of a simplified port model and in Section 2.3 how the throughput, the critical path and loop-carried dependencies are related for a kernel in detail. Chapter 3 covers the technical implementation of OSACA. This includes all prerequisites for creating a new port model for a micro-architecture in Section 3.1, i.e., the benchmarking of instruction forms in Sections 3.1.1–3.1.2, and the structure of the data file for dynamic combination of instruction forms and LOAD instructions in Section 3.1.3, as well as the enhanced throughput analysis, the CP and the LCD analysis in Section 3.2. In Chapter 4 the introduced analysis methodologies will be validated by comparing OSACA's predictions both the actual measurements and results from related tools, in specific IACA and LLVM-MCA. Finally, a summary and an outlook for future work is presented in Chapter 5.

2 BACKGROUND

This chapter documents the structure and functionalities of modern micro-architectures, gives an overview over simplified port models we assume for the analyzed micro-architectures and clarifies the definition of "throughput", "critical path" and "loop-carried dependency".

2.1. Modern Micro-Architectures

Modern micro-architectures as used in the common desktop machines and servers are still exclusively based on the von Neumann architechture [30]. Naturally, many additions have been made since then, so nowadays we can divide a CPU into three main parts: An in-order *front end*, an out-of-order *execution back end* and a *memory subsystem*. While the front end concentrates on (macro-)instruction fetching, decoding and branch prediction, the back end is responsible for renaming and allocating of registers, reordering of instructions, the actual scheduling of micro-ops[1] and finally their execution. The in-core memory subsystem normally includes the core-exclusive caches and the load and store buffer. OSACA currently focuses on modeling the back end up to and including the first core-exclusive cache of processors, therefore we assume a simplified processor model for this work. Hereinafter, some of the most important functionalities of modern micro-architectures will be described.

2.1.1. Branch Prediction and Speculative Execution

An important function for executing looped kernels in an efficient way is *branch prediction*. The processing of instructions is done in pipelined steps. This increases the overall throughput due to a in general higher parallelism of distributed execution parts. In case of conditional jumps, however, this can lead to a decrease in performance due to pipeline stalls caused by instructions, which were not considered during speculative execution and need to be dispatched. In some parts of the code, conditional branches can be avoided, but

[1] Also known as "μ-ops."

Instructions	ADD / SUB / CMP	INC / DEC	TEST / AND
JO / JNO / JS / JNS / JP/JPE / JNP/JPO	✗	✗	✓
JC/JB / JAE/JNB / JNA/JBE / JA/JNBE	✓	✗	✓
JE/JZ / JNE/JNZ / JL/JNGE	✓	✓	✓
JGE/JNL / JLE/JNG / JG/JNLE	✓	✓	✓

Table 2.1.: Macro-Fusible instructions in Intel HSW micro-architecture [12].

at least in loops an end-condition is indispensable. Modern branch predictors can consist of multi-level predictors and take both local and global states into account to reduce the frequency of mispredictions [31]. For loops, which usually tend to be iterated a large number of times, the prediction is relatively easy. Therefore, for loop kernel performance analysis, end-of-loop conditions can be ignored [2].

2.1.2. Out-of-order Execution

Out-of-order execution describes the ability of the processor to execute instructions in a modified order compared to their appearance in the machine code. Nonetheless, data dependencies or control structures must not be violated. This results in two benefits: On the one hand, in case of pipeline stalls, e.g., because of high memory latencies or non-predictable branches, non-related instructions can be scheduled and executed to avoid idle time. On the other hand, the reorder buffer (ROB), scheduler and renaming units try to achieve optimal performance by identifying independence of instruction streams and assigning them to different execution ports to be run in parallel. However, this also complicates static predictions due to unforeseeable stalls or conditional instructions which might even differ in the use of functional units based on the condition. Instructions being able to execute part of their computation, i.e., single μ-ops, before fulfilling all needed preconditions for the whole macro-op or instructions altering hidden registers such as predefined general-purpose or flag registers are just two examples of difficulties in detecting dependency chains within a kernel on an out-of-order system.

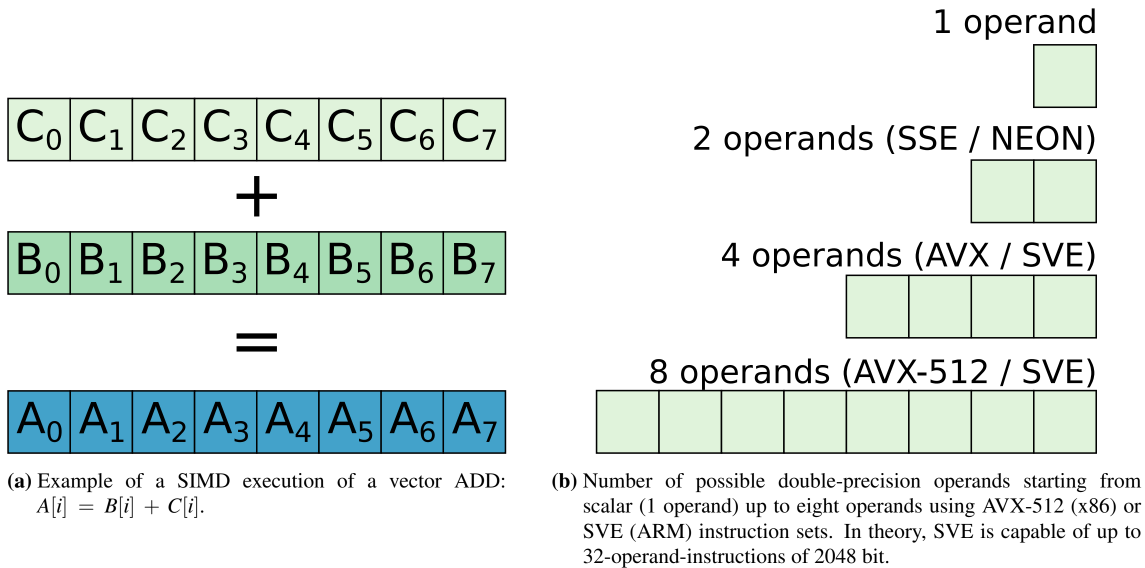
2.1.3. Macro-Op Fusion

For faster computation, modern processors can combine two adjacent instructions before decoding to compute them within the execution time of one, which is called *macro-op fusion*[2]. Macro-op fusion is limited to a small set of instruction combinations and needs to consist out of a flag-modifying instruction, e.g., CMP or ADD, and a conditional jump instruction. Furthermore, if two instruction pairs reach the decoder units in the same clock cycle, only one pair is macro-fused [32]. An overview of macro-fusible instructions for Intel's Haswell (HSW) micro-architecture can be found in Table 2.1

2.1.4. Vectorization

One of the most powerful functions of a modern CPU is what is called *vectorization* or — as introduced by Flynn [33] — *single instruction, multiple data (SIMD)*. It means "basically simultaneous parallel data operations" [31], i.e., one instruction is applied to multiple operands within one register. Based on the width of the register, modern processors operate on up to eight double-precision operands of in total 512 bit. In the x86 ISA, several versions of SIMD extensions were released, e.g., streaming SIMD extension (SSE) for 128-bit instructions, advanced vector extensions (AVX) for 256-bit instructions and AVX-512 for 512-bit instructions. In AArch64, ARM introduced NEON instructions for 128-bit instructions and recently added up to 2048-bit instructions with its new scalable vector extension (SVE) instructions. Figure 2.1 shows a simple vectorized ADD instruction in Figure 2.1a and visualizes the amount of operands within one vector register for x86 and AArch64 in Figure 2.1b.

[2] Sometimes also called *macrofusion*.



(a) Example of a SIMD execution of a vector ADD: $A[i] = B[i] + C[i]$.

(b) Number of possible double-precision operands starting from scalar (1 operand) up to eight operands using AVX-512 (x86) or SVE (ARM) instruction sets. In theory, SVE is capable of up to 32-operand-instructions of 2048 bit.

Figure 2.1.: Example SIMD execution and different possible register widths for vector instructions.

2.2. Simplified Port Model

For performance analysis, OSACA applies a simplified model of the micro-architecture and makes several assumptions:

(i) *All data accesses hit the first-level (L1) cache*
The L1 cache represents the boundary between in-core and data traffic analysis, therefore, for our in-core prediction we assume all data comes from the first-level cache. Properties like replacement strategies, prefetching and line buffering can be therefore ignored. Behavior beyond the L1 cache can be modeled using Kerncraft [14].

(ii) *Steady-state execution*
While the actual execution normally consists of a warm-up and wind-down phase taking care of preparations or the computation of the serial, not SIMD executable rest of the loop, OSACA only predicts the steady-state execution. Since we assume a large number of iterations, all startup and wind-down effects can be neglected.

(iii) *Perfect out-of-order execution and branch prediction*
We assume no misprediction of conditional jumps and perfect out-of-order execution across multiple iterations to let the throughput prediction result in a lower-bound value for the overall in-core runtime. As a consequence, by default every branch instruction is assigned with 0 cy of reciprocal throughput and latency.

In the following, we will outline three of the port models embedded in OSACA and their differences. They are used as a testbed for this work.

2.2.1. Intel Cascade Lake micro-architecture

The Intel Cascade Lake (CSX) micro-architecture is an Intel server micro-architecture and was released in 2019. It is mainly based on the Skylake architecture and, therefore, is considered as an optimization of it in terms of a "Tick" in Intel's Tick-Tock model rather than a huge architectural change. Intel introduced new Vector Neural Network Instructions (VNNI) execution units, mainly for convolutional neural network applications. An overview is shown in Figure 2.2. We can see there are two LOAD units (port 2 and 3), one STORE unit (port 4), three advanced arithmetic units (port 0, 1, and 5) and one simple arithmetic unit (port

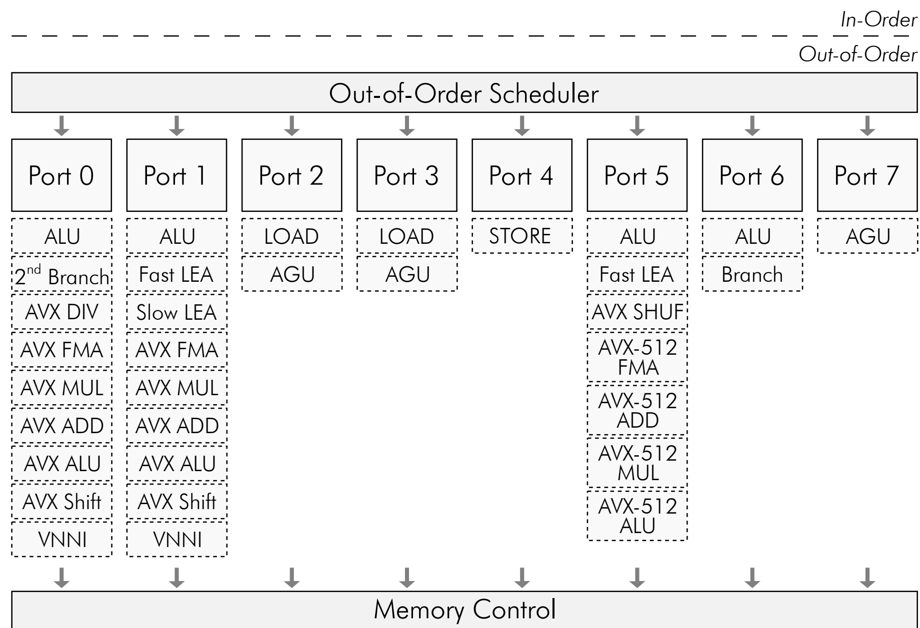


Figure 2.2.: Overview of the out-of-order execution back end components of an Intel Cascade Lake X (CSX)-based core based on [12].

6) to relieve the pressure on the other three. Additionally, there is a third Address Generation Unit (AGU) on port 7 for simple addressing, i.e., “base plus offset” [10].

2.2.2. AMD Zen micro-architecture

The Zen micro-architecture by AMD was released 2017 and is based on a 14 nm manufacturing process. It is a complete redesign of the Excavator architecture. The back end, as shown in Figure 2.3, is separated in a floating-point/vector execution and an integer execution part, i.e., there is a different out-of-order scheduler for each part. One vector execution unit by itself is only capable of executing SSE instructions; nevertheless, for running 256-bit AVX instruction, two SSE units can work — and, therefore, are occupied — together. Further differences are the combined LOAD and STORE units for each of the two AGUs on port 8 and 9, which result in a maximum throughput of either two LOADS or one LOAD and one STORE per cycle.

2.2.3. Marvell ThunderX2 micro-architecture

The ThunderX2 (TX2) micro-architecture, originally developed as “Vulcan” architecture by Cavium, which was acquired by Marvell, was launched in 2018 and can be seen as the first serious ARM-based server technology on the HPC market. Different to the previously mentioned, x86-based CSX and Zen micro-architectures, it inherits an ARMv8.1 ISA and is therefore capable of executing NEON instructions on 128-bit vector registers, comparable to the x86 SSE instructions. As seen in Figure 2.4, those instructions as well as regular floating-point instructions can be scheduled on two ports in parallel (0 and 1), while port 2 is used only for integer arithmetic and branch execution and — similar to Intel’s CSX architecture — it is capable of two LOADS and one STORE per cycle, done on ports 3/4 and 5, respectively.

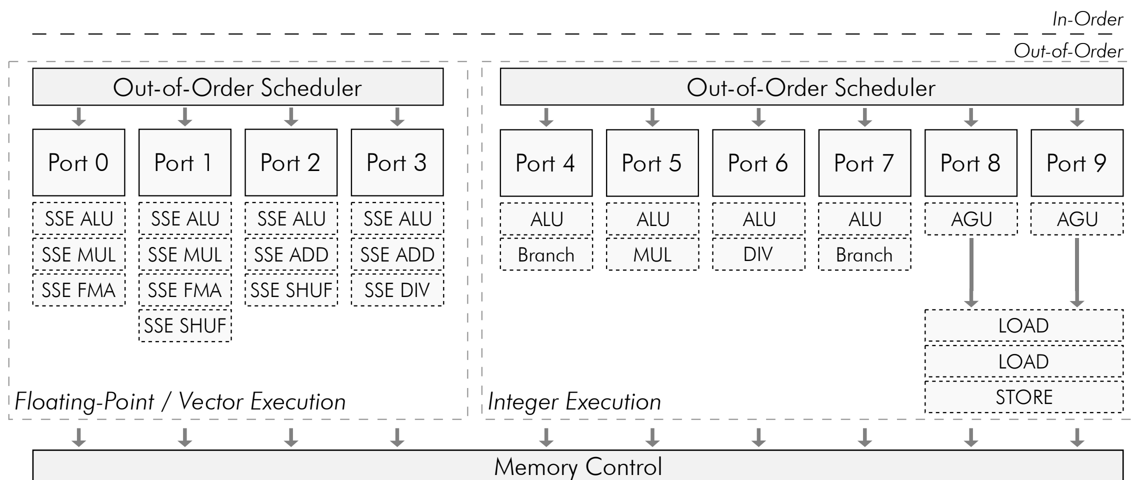


Figure 2.3.: Overview of the out-of-order execution back end components of an AMD Zen-based core [10].

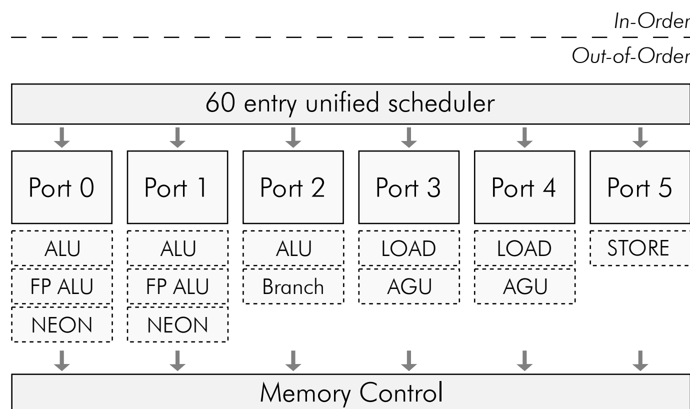


Figure 2.4.: Overview of the out-of-order execution back end components of a core in Marvell's ThunderX2 system (previously known as Vulcan micro-architecture).

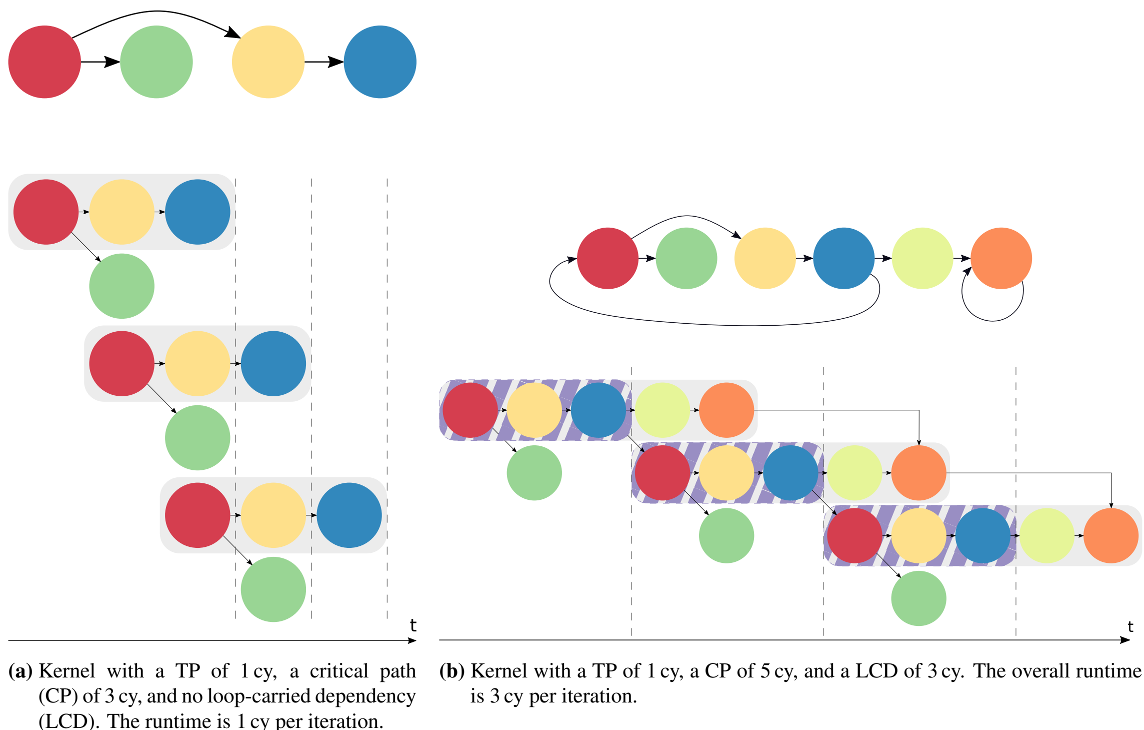
2.3. Throughput, Critical Path and Loop-Carried Dependencies

OSACA gives predictions for the throughput, the critical path and all loop-carried dependency chains in a given kernel. To clarify the meaning of each term in the given context a short explanation will be provided in the following.

2.3.1. Throughput

Even though there might exist dependencies between single instructions of a kernel, it is unlikely to find in a loop body not at least two instruction streams or single instructions independent of each other. Furthermore, one loop iteration can be independent of the previous and the following iteration. These instructions can be run in parallel and — looking at a steady-state execution — scheduled one per cycle for each port.

This behavior is exemplary shown in Figure 2.5a. In the top there is a dependency graph representing a loop kernel, each colored dot stands for one instruction and consists of four differently colored dots, therefore, it has a throughput of one cycle. The CPU can schedule a new iteration every cycle, i.e., after the functional unit executing the red dot is available again. The vertical dotted lines show that in a steady-state execution, we are able to finish one iteration per cycle, which is the throughput of the kernel. Depending on this



(a) Kernel with a TP of 1 cy, a critical path (CP) of 3 cy, and no loop-carried dependency (LCD). The runtime is 1 cy per iteration.

(b) Kernel with a TP of 1 cy, a CP of 5 cy, and a LCD of 3 cy. The overall runtime is 3 cy per iteration.

Figure 2.5.: Example kernels for visualizing throughput, the critical path and loop-carried dependencies of a loop. Each colored dot represents an instruction, e.g., ADD or MUL. Each subfigure consist of a dependency graph of one iteration of the kernel in the top and an execution diagram in the bottom. The steady-state runtime of each kernel is marked by dotted vertical lines. The CP is represented by all dots with a gray background, while the longest LCD is indicated by a purple striped background.

instruction level parallelism (ILP) a port can have up to the full pressure, i.e., it is occupied each cycle of the runtime, or can be unused for the whole time of the kernel execution. Furthermore, if multiple ports have similar functional units, e.g., two LOAD units or multiple integer ALUs, the pressure gets distributed onto all available ports to minimize the overall execution time. We can see in real measurements this does not only mean a separation with fixed probabilities, i.e., 50 % of the time on one port and 50 % of the time on the other in case of two available ports, but the scheduler is able to achieve close to optimal port pressure distribution. The throughput analysis of a kernel therefore describes the lowest intensity of iterations per cycle on one out of all ports of the micro-architecture and is usually given in reciprocal format, i.e., as cycles per iteration.

2.3.2. Critical Path

Each kernel has a critical path, which describes the latency of the longest chain of instruction dependencies within one loop iteration. Therefore, it represents the longest serial sequence of instructions and can be seen as upper bound of a kernel execution as long as there is a sufficient amount of available functional units. Since in most cases the critical paths of different loop iterations can overlap partly or even fully, the resulting runtime may differ from the time needed for the critical path. This can be seen in Figure 2.5, in which the two CPs are represented by a gray background. While the CP for the kernel in Figure 2.5a goes from the red dot via the yellow dot to the blue dot, the CP for the kernel in Figure 2.5b is additionally extended via the bright green dot to the orange dot. In both cases we can see the CP can overlap to reduce the overall runtime in steady state.

OSACA prints the CP as latency of the execution of one path iteration in processor cycles, which is not necessarily the sum of the latencies of all single instruction forms within this path, since inherit LOADs

in arithmetic instruction forms, i.e., addressing memory as an operand and, therefore, loading data before executing the arithmetic instruction, can overlap.

2.3.3. Loop-Carried Dependencies

While there is always a value for the block throughput and the critical path, the existence of loop-carried dependencies, i.e., dependencies in between iterations, is not guaranteed. Given a loop body with an arbitrary number of instructions I_0 to I_N, LCDs are created when one instruction I_x inside of the loop kernel is dependent on an instruction I'_{x+y} executed in one of the previous iterations and additionally continues a dependency chain within its own iteration up to the current instance of the instruction I_{x+y}. It therefore creates a cyclic dependency. This prevents the full overlap of different iterations of the LCD instruction chain and leads to a forced serial execution of it. While there is no cyclic dependency in Figure 2.5a, the longest LCD in Figure 2.5b is represented with a purple striped background. When looking at the dependency graph of the kernel in the top of the figure, we can even identify two LCDs: While the longest LCD is formed by the red, the yellow and the blue dot and is built through the dependency of the red dot on the blue dot of the previous iteration, the orange instruction of each iteration is dependant on the orange instruction of the previous iteration, e.g., a pointer increment while looping through an array. This way, no dependency chain can be executed before the end of the previous one and the kernel is not throughput bound anymore. Since the execution time of the LCD including the orange dot is only one cycle, it can be completely covered by the runtime of the longest purple LCD and does not need to be taken into account. We therefore can observe a runtime of the kernel of three cycles which is the duration of the longest LCD.

OSACA shows the LCD as latency of the execution of one cyclic path within a kernel in the unit of processor cycles. Similar to the prediction of the CP, combined LOAD/arithmetic instructions of CISC architectures may overlap, thus, it is not inevitably the sum of latencies of all instructions included.

3 IMPLEMENTATION

This chapter gives details about the technical implementation of the *Open Source Architecture Code Analyzer* (OSACA) tool. The source code, example code kernels and the databases can be found in the OSACA GitHub repository[1]. The project itself is licensed under the GNU Affero General Public License version 3 (AGPL-3.0).

The specific usage including different command line parameters will not be part of this chapter since they were already covered in [2] and can be found in Appendix A.

3.1. Prerequisites

In many cases, a micro-architecture is not completely unknown and there exist official or unofficial documentation on existence, amount and distribution of functional units or even already a full port model. This tremendously simplifies work for integrating it into OSACA, but does not relieve from validating the model and identifying the port usage of single instruction forms.

The way of constructing a model for a specific micro-architecture and individual instruction forms was first introduced in [16] and can be separated in the benchmarking of throughput and latency and the benchmarking for the port utilization. This information gets gathered in the databases of OSACA, divided into architecture specific data files containing the performance data for a specific model, and ISA specific data files containing information about the source and destination operands of an instruction form.

3.1.1. Benchmarking Throughput and Latency

While in theory throughput and latency values can be obtained in many different ways, OSACA supports the automatic import of benchmark results by the two micro-benchmarking tools ibench and asmbench and

[1] https://github.com/RRZE-HPC/OSACA

Number of parallel instructions per iteration	Reciprocal throughput [cy/it]
1	4.01
2	2.01
3	1.34
4	1.00
5	0.82
6	0.67
7	0.57
8	0.50
9	0.50
10	0.50

Table 3.1.: Benchmark results of the `vaddpd xmm, xmm, xmm` instruction form with increasing number of parallel instructions per iteration done by asmbench.

can automatically create the assembly files used as import for ibench. Asmbench, on the other hand, by default is able to generate assembly code for its benchmarking in every necessary way.

For throughput benchmarking of an instruction form, a kernel including multiple instances of this instruction form with independent source and destination operands is created. This can be done by not using any destination register twice, but that would quickly exhaust all available registers. Since we do not want to rely on the register renaming capabilities of the core, multiple dependency chains with the instruction form are created. For this it is important to have a sufficient amount of dependency chains which need to be of adequate length to ensure enough independent instructions are available to utilize all functional units and overheads created by the benchmarking loop will be compensated. By doing this, we can assume one port will reach a throughput of one instruction per cycle in a steady state. An extract of an inner loop of such a benchmark file for x86 assembly of the instruction form `vaddpd xmm, xmm, xmm` is shown in Listing 3.1.

```
loop:
  inc       %eax
  vaddpd    %xmm5, %xmm0, %xmm0
  vaddpd    %xmm6, %xmm1, %xmm1
  vaddpd    %xmm7, %xmm2, %xmm2
  vaddpd    %xmm8, %xmm3, %xmm3
  vaddpd    %xmm9, %xmm4, %xmm4
  vaddpd    %xmm5, %xmm0, %xmm0
  vaddpd    %xmm6, %xmm1, %xmm1
  vaddpd    %xmm7, %xmm2, %xmm2
  vaddpd    %xmm8, %xmm3, %xmm3
  ...
  cmp       %eax, %edx      # loop count
  jl loop
```

Listing 3.1: Inner loop of x86 micro-benchmark file to evaluate the throughput of the `vaddpd` instruction.

With asmbench we can increase the number of parallel instructions inside of the loop body to observe the convergence of the throughput. This is done exemplarily for the instruction form stated above on an Intel Cascade Lake X (CSX) system and shown in Table 3.1.

Here the benchmark converges to a reciprocal throughput of 0.5 cy. Hence, we can infer that there exist two independent ports, and thus pipelines, for executing the instruction form on that micro-architecture.

For obtaining the latency, we need to create a dependency chain of instructions in such a way that only a serial execution is possible. This can be achieved by alternating the operands so that the destination register of one instruction is used as source register of the subsequent instruction in the inner loop of the benchmark. Therefore, we create a read-after-write hazard between those two register operands. The code

Interleaved instruction form	μ-ops per port (resulting single TP)	Number of vaddpd instructions	Measured TP [cy/it]
vtestpd xmm, xmm	1*p0 (1 cy)	1	1.00
bsr r64, r64	1*p1 (1 cy)	1	1.00
vpermilps i8, xmm, xmm	1*p5 (1 cy)	1	1.00
vtestpd xmm, xmm	1*p0 (1 cy)	2	1.51
bsr r64, r64	1*p1 (1 cy)	2	1.50
vpermilps i8, xmm, xmm	1*p5 (1 cy)	2	1.02

Table 3.2.: Benchmark results on a Intel CSX system of different instruction forms interleaved with `vaddpd xmm, xmm, xmm`, which has a single throughput of 0.5 cy. Note that all instruction forms are written in AT&T syntax.

for measuring the latency of `vaddpd xmm, xmm, xmm` may look as in Listing 3.2. This way the latency of register-only instruction forms is achievable easily by dividing the benchmark duration by the length of the dependency chain in instructions.

```
loop:
  inc       %eax
  vaddpd    %xmm0, %xmm1, %xmm0
  vaddpd    %xmm1, %xmm0, %xmm1
  vaddpd    %xmm0, %xmm1, %xmm0
  vaddpd    %xmm1, %xmm0, %xmm1
  vaddpd    %xmm0, %xmm1, %xmm0
  vaddpd    %xmm1, %xmm0, %xmm1
  cmp       %eax, %edx      # loop count
  jl loop
```

Listing 3.2: Inner loop of x86 micro-benchmark file to evaluate the latency of the `vaddpd` instruction.

3.1.2. Benchmarking Port Utilization

To identify not only the number of ports used but also the specific ports, we can systematically interleave the instruction form to analyze with other instructions. Thereby, the change of throughput indicates the ports used. This requires some previous knowledge of the architecture since we need to know the port utilization of the other instruction form, which allocates preferably a minimum amount of functional units and has a low latency. If the combination of two instruction forms does not surpass the overall throughput, we can infer that the investigated instruction form does not make use of the same ports as the known one. If the overall throughput of the interleaved benchmark is higher than the standard one, at least one port utilized by the known instruction form is also used by the investigated instruction form. This way, by combining multiple known instruction forms for interleaving, we can identify the ports and, therefore, validate the hardware model and performance data of the instruction form to investigate.

Continuing the example with `vaddpd xmm, xmm, xmm` from above on Intel CSX, useful instruction forms for interleaving would be, e.g.:

- `vtestpd xmm, xmm` with a known reciprocal throughput of 1 cy, 1 μ-op and only executable on port 0 (written as *1*p0*).
- `bsr r64, r64`, with a known reciprocal throughput of 1 cy, utilizing port 1 (*1*p1*).
- `vpermilps i8, xmm, xmm`, using *1*p5*, thus, with a reciprocal throughput of 1 cy.

The result of the benchmarking is shown in Table 3.2.

As we can see, all benchmarks result in a different overall throughput compared to a single `vaddpd`, since the `vtestpd`, `bsr` and `vpermilps` instructions have a reciprocal throughput of 1 cy. The fact that all benchmarks with one instance of the investigated instruction form have a reciprocal throughput of

1 cy means we can infer that vaddpd is neither solely bound on a execution on port 0 or 1 nor port 5. When we run the the three instruction forms with two instances of vaddpd, the overall throughput changes compared to the previous measurements only in one case, namely, for the instruction running on port 0 and 1. The test with another instruction running on port 5 keeps the reciprocal throughput of 1 cy. This is due to the reciprocal throughput of 0.5 cy of the investigated instruction form. While vaddpd does not interfere with port 5, another instruction can be scheduled on any other port without affecting the execution of the vpermilps. For vtestpd and bsr, the second vaddpd can only be executed after the other instruction is done and, therefore, its reciprocal throughput must be added to the overall reciprocal throughput. This is the case for both port 0 and 1, thus, we validated `vaddpd xmm, xmm, xmm` needs at least ports 0 and 1 for execution and we can write its port utilization as *1*p01*.

Unfortunately, there does not exist an automatic generation of a port utilization model for instruction forms yet, i.e., it requires a comparably high manual effort.

3.1.3. Instruction Form Data

Having the performance data collected as stated in Sections 3.1.1 and 3.1.2, everything gets collected in the local data files. OSACA differentiates between architecture specific machine files and ISA specific data files. While the ISA specific data file only contains the operand access of extraordinary instruction forms, i.e., the source and destination operands in case they differ from the assumed standard of one destination as last operand (according to the assembly syntax) and otherwise only source operands, most of the information is gathered in the architecture specific machine file. In here, the port model and the system-wide load throughput and latency is written down. Furthermore, besides some meta data, each known instruction form is stored as one entry of a list, containing a unique identifier, i.e., mnemonic and operands of an instruction form, the port utilization, the throughput, and the latency. If an instruction form includes memory addressing in its source or destination operand but is not simply a LOAD or STORE instruction, OSACA can calculate the throughput and latency dynamically out of the register-only variant and the architecture specific performance data for LOAD and STORE. This way, not only a higher variation and, therefore, number of instruction forms can be covered, but it is also necessary for a valid latency prediction to separate the LOAD and the arithmetic part of an instruction form, since the LOAD can be done in parallel with previous instructions, even though the instruction form is part of the critical path.

Furthermore, OSACA does not rely anymore on including all versions of an instruction form with suffixes for determining the operand size, as used in GNU assembler. Instead, it checks the data base for a given mnemonic without suffix and applies its performance data on the investigated instruction form.

3.2. Analysis

With the prerequisites from Section 3.1 fulfilled, OSACA can analyze any given x86 or AArch64 kernel. As a result, it produces a throughput, critical path and loop-carried dependency analysis, which will be described in the following.

3.2.1. Throughput Analysis

The overall block throughput of an analyzed loop body is the maximum of all sums of the reciprocal throughput of the executed instructions per pipeline. Therefore, looking at the analysis report, it is indicated by the largest column sum for the whole block. With version 0.3 of our tool, we introduced an optimal port scheduling as default and provide the scheduling with fixed probabilities if requested by a command line argument. For minimizing the overall reciprocal throughput, we apply a linear step-wise method to decrease and increase the execution of an instruction on suited ports with maximum and minimal port pressure, respectively, to converge to a local minimum. This algorithm is shown and explained by comments in Listing 3.3. For an easier understanding, it iterates the μ-ops of a kernel instead of its instruction forms.

```
STEPSIZE = 0.01
kernel = extract_kernel()    # List of muops, which is a tuple of total cy and ports
TP_table = []                # List of muop contributions to overall port throughputs
TP_sums = {p: 0.0 for p in all_ports}
for muop_cy, muop_ports in kernel:
    muop_contributions = {}
    # Assuming fixed port utilization
    avg_utilization = muop_cy/len(muop_ports)
    for port in muop_ports:
        muop_contributions[port] = avg_utilization
        TP_sums[port] += avg_utilization
    TP_table.append(muop_contributions)

for muop_contributions in TP_table:
    # Unbalance muop's contribution to balance overall throughput

    # Continue balancing until overall throughput on ports used by muop are equal
    # or whole muop was shifted to one port
    while not all_equal(
        [TP_sums[p] for p in muop_contributions if muop_contributions[p] > 0.00]
    ):
        # Find port (i.e., the second element of the tuple) with maximum overall
        # throughput, which has a contribution from current muop:
        p_max = max(
            [(TP, p) for p, TP in TP_sums.items() if p in muop_contributions]
        )[1]
        # Find port (i.e., the second element of the tuple) with minimum overall
        # throughput, which has a contribution from current muop:
        p_min = min(
            [(TP, p) for p, TP in TP_sums.items() if p in muop_contributions]
        )[1]

        # Reduce contribution of p_max by STEPSIZE:
        muop_contributions[p_max] -= STEPSIZE   # Also updates TP_table
        TP_sums[p_max] -= STEPSIZE              # Update overall TP_sum
        # Increase contribution of p_min by STEPSIZE:
        muop_contributions[p_min] += STEPSIZE   # Also updates TP_table
        TP_sums[p_min] += STEPSIZE              # Update overall TP_sum
```

Listing 3.3: OSACA's algorithm for reaching a local minimum of the overall throughput. It is written in valid Python code. For a better understanding, the algorithm iterates the μ-ops of an analyzed kernel.

In the for-loop from line 5 to 12, a fixed port utilization is assigned to the kernel and stored in *TP_table*. This can be compared with one instruction form line in the throughput analysis report of OSACA. The overall throughput sum, which represents the last line of the throughput analysis in OSACA's output, is stored in *TP_sums*. In line 14 to 38, OSACA iterates the kernel and shifts cycles of the previously fixed port utilization with a step size of 0.01 cy to achieve a balanced overall throughput. This way we can simulate the non-fixed issue of instructions during steady-state execution and reach precise predictions for throughput bound kernels with a non-balanced port utilization.

3.2.2. Critical Path Analysis

The basis of the critical path (CP) analysis is represented by a directed acyclic graph (DAG) constructed from inter-instruction register dependencies. The creation follows simple rules:

(i) For each instruction form within the marked kernel, create a vertex containing the gathered information of it.

(ii) For each instruction form, identify all subsequent instruction forms depending on it, i.e., having a source operand reading from a destination operand of the first mentioned instruction. Currently, this

is only done for register operands due to the high complexity of tracking store-to-load dependencies, e.g., due to relative addressing. A dependency chain of a specific register is broken when it is rewritten and now might be dependent of another previous instruction. For all identified dependencies, draw directed edges between the pair of affected vertices representing instruction forms and assign the latency of the start vertex instruction form as weight to the edge.

(iii) If an arithmetic instruction form includes a memory reference as source operand, separate it into the LOAD and the arithmetic part by adding another vertex containing a LOAD instruction with an edge to the original instruction. This edge gets weighted with the pure LOAD latency, while we reassign all outgoing edges from the original instruction form with the latency of the arithmetic part only as weight.

Subsequently, this DAG functions as enhanced representation of the kernel and the CP within it can be determined using a weighted topological sort based on the approach of Manber [34], already implemented in the graph framework *NetworkX*[2] used by OSACA. It is defined as longest consequent subgraph, i.e., path, of the DAG based on the weight of the edges.

3.2.3. Loop-Carried Dependency Analysis

For identifying loop-carried dependencies (LCDs), OSACA creates a DAG based on the rules stated in Section 3.2.2, but uses a code kernel comprising two back-to-back copies of the initial loop body. This way it can analyze dependency paths from vertices of the first kernel section to the second one and identify cyclic dependencies by checking on corresponding instruction forms within one dependency chain. Equally to the CP analysis, OSACA currently only supports the detection of register dependencies, including standard register operands, base and index registers in memory addressing and flag registers. It is not able to identify load-after-store dependencies, like:

```
loop:
  mov     %r9, (%rax)
  mov     (%rax), %r9

  cmp     %r9, %r15
  jne     .loop
```

In this example, a value from register `r9` is written to the memory position stored in `rax` just to get moved back to register `r9` and stored back at the exact same position. So even if one would expect a LCD detected by OSACA, this is currently not supported because of the complexity modeling not only this simple example, but indexing of addresses in memory and sometimes is simply not possible due to the fact we cannot know the content of registers in all cases.

Another weakness of the current LCD analysis can be pinpointed by the next example:

```
.loop:
  add     $1, %rax
  # swap rax, rbx
  mov     %rax, %rcx
  mov     %rbx, %rax
  mov     %rcx, %rbx

  cmp     %rax, %r15
  jne      .loop
```

Besides incrementing `rax`, in each iteration the content of register `rax` gets swapped with the content of register `rbx`. This means, after two iterations, the values in `rax` and `rbx`, respectively, match again and there exists a dependency chain throughout two iterations. This example can be scaled up a arbitrary number of iterations needed for one cyclic LCD. OSACA would not detect any of these currently, but could be extended easily to do so up to a defined number of loop repetitions. For defining a reasonable number of repetitions to find a trade-off between performance and LCD coverage this topic needs more investigation and is part of future work.

[2]The current version of NetworkX is available at https://networkx.github.io/.

4 Evaluation

This chapter will evaluate the quality of the analysis created by OSACA. For this, OSACA is validated by comparing its prediction with actual measurements of the investigated kernels and the runtime analyses by related work, i.e., the performance analyzing tools Intel Architecture Code Analyzer (IACA) and LLVM Machine Code Analyzer (LLVM-MCA). First we will explain how to interpret the results of the related tools and subsequently put these in a context with OSACA's analysis and the measured performance data for various scientific kernels on an Intel Cascade Lake X (CSX), an AMD Zen and an ARM-based Marvell ThunderX2 (TX2) system.

4.1. Data acquisition for IACA and LLVM-MCA

To understand the quality of the prediction of OSACA, it is necessary to compare its results to related work. For this we chose two static analysing tools with similar purpose, Intel's proprietary code analyzer IACA and LLVM's Machine Code Analyzer LLVM-MCA. While not all tools were capable of analyzing all platforms in full scope, e.g., IACA only supports Intel micro-architectures and LLVM-MCA does not identify a critical path (CP) without further investigation, we will show how to retrieve the important data for validation for both tools. None of the two provide a dedicated CSX support but since we do not use any Vector Neural Network Instructions (VNNI) instructions specific to the CSX micro-architecture, we can analyze the code using the Skylake-X (SKX) support of the tools.

For this purpose we will analyse a simple vector update, which adds a scalar double-precision value to each element of a vector. The update is done in-place, i.e., the sum is written back to the same memory address from which the vector element was loaded originally. The high-level C code can be written as follows:

```
double * restrict a;

for(long i=0; i < size; ++i){
    a[i] = a[i] + scalar;
}
```

```
Throughput Analysis Report
--------------------------
Block Throughput: 2.48 Cycles       Throughput Bottleneck: FrontEnd
Loop Count:  30
Port Binding In Cycles Per Iteration:
--------------------------------------------------------------------------------------------------
|  Port  |   0   -  DV   |   1   |   2   -  D    |   3   -  D    |   4   |   5   |   6   |   7   |
--------------------------------------------------------------------------------------------------
| Cycles |  1.0     0.0  |  0.5  |  2.0     1.0  |  2.0     1.0  |  2.0  |  1.0  |  0.5  |  0.0  |
--------------------------------------------------------------------------------------------------

DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3)
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port

| Num Of |                 Ports pressure in cycles
|  Uops  |  0 - DV |  1   |  2 -  D   |  3   - D   |  4   |  5   |  6   | 7 |
---------------------------------------------------------------------------
|   2    | 1.0     |      | 1.0  1.0 |            |      |      |      |   | vaddpd (%r13,%rcx,8), \
                                                                                   %zmm3, %zmm1
|   2    |         |      |          | 1.0  1.0 |      | 1.0 |      |   | vaddpd 64(%r13,%rcx,8), \
                                                                                   %zmm3, %zmm2
|   2    |         |      | 1.0      |            | 1.0 |      |      |   | vmovupd %zmm1, (%r13,%rcx,8)
|   2    |         |      |          | 1.0        | 1.0 |      |      |   | vmovupd %zmm2, 64(%r13,%rcx,8)
|   1    |         | 0.5 |          |            |      |      | 0.5 |   | add $16, %rcx
|   1*   |         |      |          |            |      |      |      |   | cmp %r15, %rcx
|   0*F  |         |      |          |            |      |      |      |   | jb 0xffffffffffffffd9
Total Num Of Uops: 10
```

Listing 4.1: IACA analysis for a vector update kernel for an Intel SKX micro-architecture. For the reader's convenience, the assembly syntax was converted to AT&T, the output shortened and important information for comparison with OSACA is marked in bold red.

We compile the kernel on an Intel CSX system with ICC and the flags "`-xCORE-AVX512 -Ofast -qopenmp-simd -fno-alias -unroll -qopt-zmm-usage=high`," which results in this assembly kernel:

```
..B1.38:
    vaddpd     (%r13,%rcx,8), %zmm3, %zmm1
    vaddpd     64(%r13,%rcx,8), %zmm3, %zmm2
    vmovupd    %zmm1, (%r13,%rcx,8)
    vmovupd    %zmm2, 64(%r13,%rcx,8)
    addq       $16, %rcx
    cmpq       %r15, %rcx
    jb         ..B1.38
```

4.1.1. IACA

For starting an analysis with IACA, one first needs to mark the assembly code. This is done by adding two different byte markers:

```
movl    $111,%ebx        #IACA START MARKER
.byte   100,103,144      #IACA START MARKER
Loop:
  # ...
  jb        .Loop
movl    $222,%ebx        #IACA END MARKER
.byte   100,103,144      #IACA END MARKER
```

Since IACA bases its analysis on the binary opcode, the marked file must be assembled. Thus, it requires the code to be valid. IACA's output for the vector reduction kernel is shown in Listing 4.1.

Similar to OSACA, it differentiates between the ports and pipelines within ports. It recognizes macro fusion (e.g., '0*F' in the last line of the report) and is based on proprietary knowledge we are not capable to reproduce without further insight. This becomes evident when comparing the maximum number of cycles per

port in the upper table with the “Block Throughput” value in the first line of the analysis report. While the tables let us infer that the reciprocal throughput should be 2.0 cy, IACA predicts an overall block throughput of 2.48 cy. When comparing IACA to actual measurements and OSACA, we take both of these numbers into account since we assume IACA—even though the developers dropped CP support in version 2.3—uses more information, e.g., dependency detection, to achieve the “Block Throughput” number. Furthermore, IACA provides the user with information about the number of μ-ops and the estimated bottleneck. For all further tests we used IACA v3.0.

4.1.2. LLVM-MCA

Different to IACA, for an analysis with LLVM-MCA it is not necessary to compile the assembly code. The tool runs its analysis on the whole assembly file if not given any markers. Nevertheless, it is useful to limit the prediction to a specific kernel for the sake of clarity and the readability for the user. Since LLVM-MCA inspects the assembly, the markers can be simple comments:

```
# LLVM-MCA-BEGIN
Loop:
  # ...
  jb        .Loop
# LLVM-MCA-END
```

The output is shown in Listing 4.2. In different views LLVM-MCA prints the number of μ-ops, the throughput and latency of each instruction as well as the overall resource pressure separated by ports. With the “`--timeline`” command line flag, we can enable the “Timeline view” to emulate several iterations of the kernel. It visualizes the expected cycle of dispatching (“D”), execution (“e”/“E”) and retirement (“R”) for each instruction form and models dependencies by doing so. This way, we can use the timeline view as CP analysis and expect the time from the beginning of the first iteration, i.e., the cycle of the dispatching of the first instruction form, to the end, i.e., the retirement of the last instruction form of the loop, as the CP of the kernel and the duration of all further iterations, i.e., from retirement of the first to the retirement of the last instruction form, as the length of the longest loop-carried dependency (LCD). However, this value seems to be inaccurate especially for throughput-bound kernels and should be considered with care or only when the throughput prediction appears to be off the expected runtime by far and long dependency chains can be detected. In case of a non-consistent LCD value, i.e., if multiple further iterations show different durations, we assume the smallest one to be the LCD since other effects could slow the execution.

In our example we can extract the block throughput of 2 cy/it on port “SKXPort4”, as marked in red, and a duration of 16 cy from cycle 0 as the time of detaching instruction form `[0,0]` to cycle 15 as the time of retirement of instruction form `[0,6]` as CP for this kernel. Any further iteration, e.g., the second iteration from instruction form `[1,0]` to `[1,6]`, consistently needs 2 cy. Thus, we assume a LCD of 2 cy per assembly iteration and infer that the kernel is throughput bound due to the fact that the length of the LCDs equals the throughput.

LLVM-MCA allows the user to specify many further options for analysis, however, we confine ourselves to the basic “Resource pressure by instruction” view and additionally the “Timeline view” for comparing it with OSACA. Unfortunately, LLVM-MCA turns out to be sometimes unstable in terms of parsing unknown instruction forms, therefore, we sometimes need to extract the kernel from the overall assembly file to only analyse this part. For all further testing we used LLVM-MCA based on llvm 9.0.1[1].

4.2. Analysis and Comparison of OSACA Results

In the following we analyse the accuracy of OSACA in its current version 0.3.2.dev5[2] for various scientific kernels of different complexity. We gathered results on three different systems:

[1] https://github.com/llvm/llvm-project/releases/tag/llvmorg-9.0.1
[2] https://github.com/RRZE-HPC/OSACA/releases/tag/v0.3.2.dev5

```
[0] Code Region

Iterations:        100      Dispatch Width:    6
Instructions:      700      uOps Per Cycle:    5.12
Total Cycles:      215      IPC:               3.26
Total uOps:        1100     Block RThroughput: 2.0

Instruction Info:
[1]: #uOps           [4]: MayLoad
[2]: Latency         [5]: MayStore
[3]: RThroughput     [6]: HasSideEffects (U)

[1]    [2]    [3]    [4]    [5]    [6]    Instructions:
 2      11     0.50   *                   vaddpd        (%r13,%rcx,8), %zmm3, %zmm1
 2      11     0.50   *                   vaddpd        64(%r13,%rcx,8), %zmm3, %zmm2
 2      1      1.00          *            vmovupd       %zmm1, (%r13,%rcx,8)
 2      1      1.00          *            vmovupd       %zmm2, 64(%r13,%rcx,8)
 1      1      0.25                       addq  $16, %rcx
 1      1      0.25                       cmpq  %r15, %rcx
 1      1      0.50                       jb    ..B1.38


Resources:
[0]   - SKXDivider       [5]   - SKXPort3
[1]   - SKXFPDivider     [6]   - SKXPort4
[2]   - SKXPort0         [7]   - SKXPort5
[3]   - SKXPort1         [8]   - SKXPort6
[4]   - SKXPort2         [9]   - SKXPort7

Resource pressure per iteration:
[0]    [1]    [2]    [3]    [4]    [5]    [6]    [7]    [8]    [9]
 -      -     1.25   1.25   1.34   1.35   2.00   1.25   1.25   1.31

Resource pressure by instruction:
[0]    [1]    [2]    [3]    [4]    [5]    [6]    [7]    [8]    [9]    Instructions:
 -      -     0.49    -     0.85   0.15    -     0.51    -      -     vaddpd    (%r13,%rcx,8), \
                                                                                  %zmm3, %zmm1
 -      -     0.43    -     0.45   0.55    -     0.57    -      -     vaddpd    64(%r13,%rcx,8), \
                                                                                  %zmm3, %zmm2
 -      -      -      -     0.02   0.54   1.00    -      -     0.44   vmovupd   %zmm1, (%r13,%rcx,8)
 -      -      -      -     0.02   0.11   1.00    -      -     0.87   vmovupd   %zmm2, 64(%r13,%rcx,8)
 -      -      -     0.99    -      -      -      -     0.01    -     addq      $16, %rcx
 -      -     0.24   0.26    -      -      -     0.17   0.33    -     cmpq      %r15, %rcx
 -      -     0.09    -      -      -      -      -     0.91    -     jb        ..B1.38


Timeline view:
                    0123456789
Index     0123456789          012345

[0,0]     DeeeeeeeeeeeER .    .    .   vaddpd      (%r13,%rcx,8), %zmm3, %zmm1
[0,1]     DeeeeeeeeeeeER .    .    .   vaddpd   64(%r13,%rcx,8), %zmm3, %zmm2
[0,2]     D===========eER.    .    .   vmovupd  %zmm1, (%r13,%rcx,8)
[0,3]     .D===========eER    .    .   vmovupd  %zmm2, 64(%r13,%rcx,8)
[0,4]     .DeE-----------R    .    .   addq     $16, %rcx
[0,5]     .D=eE----------R    .    .   cmpq     %r15, %rcx
[0,6]     .D==eE---------R    .    .   jb   ..B1.38
[1,0]     . DeeeeeeeeeeeER    .    .   vaddpd (%r13,%rcx,8), %zmm3, %zmm1
[1,1]     . DeeeeeeeeeeeER    .    .   vaddpd   64(%r13,%rcx,8), %zmm3, %zmm2
[1,2]     . D===========eER   .    .   vmovupd  %zmm1, (%r13,%rcx,8)
[1,3]     .  D===========eER  .    .   vmovupd  %zmm2, 64(%r13,%rcx,8)
[1,4]     .  DeE-----------R  .    .   addq     $16, %rcx
[1,5]     .  D=eE----------R  .    .   cmpq     %r15, %rcx
[1,6]     .  D==eE---------R  .    .   jb     ..B1.38
[2,0]     .   DeeeeeeeeeeeER  .    .   vaddpd   (%r13,%rcx,8), %zmm3, %zmm1
[2,1]     .   DeeeeeeeeeeeER  .    .   vaddpd   64(%r13,%rcx,8), %zmm3, %zmm2
[2,2]     .   D===========eER .    .   vmovupd  %zmm1, (%r13,%rcx,8)
[2,3]     .    D===========eER.    .   vmovupd  %zmm2, 64(%r13,%rcx,8)
[2,4]     .    DeE-----------R.    .   addq     $16, %rcx
[2,5]     .    D=eE----------R.    .   cmpq     %r15, %rcx
[2,6]     .    D==eE---------R.    .   jb       ..B1.38
[3,0]     .     .DeeeeeeeeeeeER.   .   vaddpd   (%r13,%rcx,8), %zmm3, %zmm1
...
```

Listing 4.2: LLVM-MCA analysis for the vector reduction kernel including the `--timeline` command line option. For the reader's convenience, the output was shortened and important information for comparison with OSACA is marked in bold red.

- ***ThunderX2 (TX2)***
  ARM-based Marvell ThunderX2 9980 with ThunderX2 micro-architecture (formerly known as "Vulcan" by Cavium). Even though one cannot acquire or fix the frequency as easily as on x86 machines, we can observe a constant frequency of 2.2 GHz during program execution. Therefore, we assume it to behave like a fixed frequency at 2.2 GHz. While comparing the measurements with enabled simultaneous multithreading (SMT) to the measurements with SMT OFF, we could observe an increase of performance of in average more than 10 % for disabled SMT. Therefore, we run all kernels with SMT OFF. To cover multiple compiler vendors, all code examples were compiled with both the Arm C/C++/Fortran Compiler version 19.2[3] and the GNU C/Fortran Compiler version 8.2.0[4]. For all runs we used the compiler flags `-mcpu=thunderx2t99+simd+fp -fopenmp-simd -funroll-loops -Ofast`.

- ***Cascade Lake (CSX)***
  Intel Xeon Gold 6248 with Cascade Lake X micro-architecture running at a fixed frequency of 2.5 GHz. The code was compiled using the Intel C/Fortran Compiler of Intel Parallel Studio version 19.0up05[5] with the optimizing flags `-xCORE-AVX512 -qopenmp-simd -fno-alias -unroll -Ofast -qopt-zmm-usage=high` and the GNU C/Fortran Compiler version 9.1.0[6] using the flags `-march=skylake-avx512 -fopenmp-simd -funroll-loops -Ofast -fargument-noalias`.

- ***Zen***
  AMD EPYC 7451 with Zen micro-architecture with a fixed frequency at 2.3 GHz. For compiling we used the GNU C/Fortran Compiler version 9.1.0 with the flags `-march=znver1 -mavx2 -mfma -fopenmp-simd -fargument-noalias -funroll-loops -Ofast`.

All builds were done with `-fno-builtin` flag to disable inline expansion of intrinsic functions. Each result table in the following sections shows the micro-architecture, the used compiler, and an unroll factor as describing elements next to the measured and predicted data for one benchmark kernel. The unroll factor is represented as the product of the actual unrolling of the assembly kernel and the SIMD width used, e.g., "2×8" for the kernel in the previous section. The performance data consists of the throughput, longest LCD and the critical path extracted from OSACA and LLVM-MCA and two different TP values for IACA. Besides the "normal" TP as sum of the highest port pressure, IACA returns a block throughput that is sometimes significantly higher. We show this value as *BlockTP* in our result tables. All predictions are written as cycles per assembly block iteration, therefore, to calculate the runtime of one high-level iteration, one must divide the result by the unroll and SIMD factor. Additionally, we print an accuracy value as the quotient of the predicted and the measured performance. The TP predictions represent a lower bound of the runtime, thus, we aim to achieve an accuracy slightly below 100 % and consider a prediction with accuracy above 100 % failed. We consider a kernel TP-bound if the TP prediction is equal or greater than the LCD prediction of OSACA and LCD-bound otherwise. Note that we can apply this rule only for OSACA. Based on this result we consequently adapt the decision to take the TP or LCD value as prediction due to the fact LLVM-MCA does not provide a reliable LCD prediction. For IACA we always take the BlockTP value for comparison with measurements and the other tools.

A summary table of all results can be found in Appendix B. All analysis outputs and instructions for reproducing the results can be found in the artifact description [35].

4.2.1. Copy

The copy kernel represents a classic memory copy of the content of one array into another:

[3] https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-compiler-for-linux (accessed January 29, 2020)
[4] https://gcc.gnu.org/gcc-8/ (accessed January 29, 2020)
[5] https://software.intel.com/en-us/compilers (accessed January 29, 2020)
[6] https://gcc.gnu.org/gcc-9/ (accessed January 29, 2020)

```
double * restrict a, * restrict b;

for(long i=0; i < size; ++i){
    a[i] = b[i];
}
```

The resulting assembly kernel contains only MOV or LOAD/STORE instructions, respectively, without any inter-iteration dependencies and we expect the kernel to be throughput-bound.

The measurements with the corresponding prediction results for OSACA, LLVM-MCA and IACA can be found in Table 4.1. For each tool, the column taken for the overall prediction is highlighted in gray.

Arch	Compiler	Unroll × SIMD factor	Measured [cy/asm it]	Prediction [cy/asm it]								Accuracy		
				OSACA			LLVM-MCA			IACA		OSACA	LLVM-MCA	IACA
				TP	CP	LCD	TP	CP	LCD	TP	BlockTP			
TX2	armclang	32 × 2	37.29	32.00	4	1	16.00	42	39	—	—	86 %	43 %	—
	gcc	8 × 2	11.07	8.00	5	1	11.35	28	20	—	—	72 %	103 %	—
CSX	icc	1 × 8	1.17	1.00	4	1	1.01	12	1	1.00	1.23	86 %	86 %	105 %
	gcc	8 × 4	8.11	8.00	4	1	8.00	21	8	8.00	8.00	99 %	99 %	99 %
Zen	gcc	8 × 2	8.09	8.00	4	1	8.00	22	8	—	—	99 %	99 %	—

Table 4.1.: Prediction results of the COPY benchmark in Section 4.2.1.

Since the runtime prediction for OSACA is defined as the maximum of TP and LCD, for throughput-bound kernels we expect the TP value to be always greater than the LCD prediction. Here we can see excellent predictions by OSACA for the gcc-compiled x86-versions of the code and a still accurate prediction for the AVX-512 code on CSX. We can observe a consistent loss of accuracy throughout all AVX-512 code kernels. This is due to little to none unrolling on top of SIMD. We can force icc with `#pragma unroll(N)` to achieve the same factor of total unrolling as in the the gcc-compiled code and observe an increase of accuracy to 98 %. It seems IACA is taking some penalty into consideration, but, therefore, overestimates the kernel and predicts a slower execution than the measured runtime. Otherwise, the related tools predict the gcc-compiled kernels as precisely as OSACA.

For the predictions on ARM we can see a bigger discrepancy in the measurements. While LLVM-MCA slightly overpredicts the gcc-compiled kernel, OSACA predicts only 72 % of the actual measurement. To gain more insight into the execution process, the analyzed kernel is shown in Listing 4.3. We can see the code is clearly bound by port 3, 4 and 5 which contain the LOAD units including the Address Generation Units (AGUs) (port 3, 4) and the STORE unit (port 5). Therefore, we assume there must be a penalty not considered by our hardware model at the time of writing. The LLVM-MCA output, which has a much more accurate prediction is shown in Listing 4.4. Note that the distribution of the functional units across the ports as well as the port model itself is different to OSACA, e.g., no dedicated STORE port, and the reasoning behind LLVM's port model is not publicly documented. One can only guess LLVM-MCA assumes three AGUs to be on port 0, 1 and 2. Furthermore, it shows 3 μ-ops per LOAD and 4 μ-ops per STORE instruction but schedules only one cycle on the data ports and one and two, respectively, across the arithmetic ports. Thus, according to LLVM-MCA the code is bound by the arithmetic ports instead of the data ports. We can see that a different model which is —based on our knowledge about the system and Marvell customer information— not necessarily correct, can achieve higher accuracy in specific cases.

While OSACA returns an accurate prediction for the armclang-compiled version of the kernel, the prediction by LLVM-MCA is far off. This is due to the fact that the compiler uses `ldp` and `stp` instructions for loading and storing a pair of registers. LLVM-MCA expects `stp` to need one μ-op, which can be executed on one of the two data ports. This differs from our assumption of two μ-ops on the data ports as well as two μ-ops on the AGUs port and consequently leads to a wrong prediction.

4.2.2. Vector add

In a vector add kernel, the CPU loads two values from two different arrays and stores the sum into a third array. The C code can be written as follows:

```
Open Source Architecture Code Analyzer (OSACA) - v0.3
Architecture:       tx2

Combined Analysis Report
------------------------
                              Port pressure in cycles
    |  0   - 0DV |  1   - 1DV |  2   |  3   |  4   |  5   ||  CP  | LCD |
-------------------------------------------------------------------------
273 |            |            |      |      |      |      ||      |     |   .L17:
274 | 0.33       | 0.33       | 0.33 |      |      |      ||      |     |   add x16, x15, 16
275 |            |            |      | 0.50 | 0.50 |      ||      |     |   ldr q9, [x19, x15]
276 | 0.33       | 0.33       | 0.33 |      |      |      ||      |     |   add x30, x15, 32
277 | 0.33       | 0.33       | 0.33 |      |      |      ||      |     |   add x17, x15, 48
278 |            |            |      | 0.50 | 0.50 |      ||      |     |   ldr q16, [x19, x16]
279 |            |            |      | 0.50 | 0.50 |      ||      |     |   ldr q18, [x19, x30]
280 | 0.33       | 0.33       | 0.33 |      |      |      ||      |     |   add x18, x15, 64
281 | 0.33       | 0.33       | 0.33 |      |      |      ||      |     |   add x1, x15, 80
282 |            |            |      | 0.50 | 0.50 |      ||      |     |   ldr q17, [x19, x17]
283 |            |            |      | 0.50 | 0.50 |      ||      |     |   ldr q19, [x19, x18]
284 | 0.33       | 0.33       | 0.33 |      |      |      ||      |     |   add x3, x15, 96
285 | 0.33       | 0.33       | 0.33 |      |      |      ||  1.0 |     |   add x2, x15, 112
286 |            |            |      | 0.50 | 0.50 |      ||      |     |   ldr q20, [x19, x1]
287 |            |            |      | 0.50 | 0.50 |      ||      |     |   ldr q21, [x19, x3]
288 |            |            |      | 0.50 | 0.50 | 1.00 ||      |     |   str q9, [x20, x15]
289 |            |            |      | 0.50 | 0.50 |      ||  4.0 |     |   ldr q22, [x19, x2]
290 | 0.33       | 0.33       | 0.33 |      |      |      ||      | 1.0 |   add x15, x15, 128
291 |            |            |      | 0.50 | 0.50 | 1.00 ||      |     |   str q16, [x20, x16]
292 |            |            |      | 0.50 | 0.50 | 1.00 ||      |     |   str q18, [x20, x30]
293 |            |            |      | 0.50 | 0.50 | 1.00 ||      |     |   str q17, [x20, x17]
294 |            |            |      | 0.50 | 0.50 | 1.00 ||      |     |   str q19, [x20, x18]
295 |            |            |      | 0.50 | 0.50 | 1.00 ||      |     |   str q20, [x20, x1]
296 |            |            |      | 0.50 | 0.50 | 1.00 ||      |     |   str q21, [x20, x3]
297 |            |            |      | 0.50 | 0.50 | 1.00 ||  4.0 |     |   str q22, [x20, x2]
298 | 0.33       | 0.33       | 0.33 |      |      |      ||      |     |   cmp x23, x15
299 |            |            |      |      |      |      ||      |     | * bne .L17

      3.00         3.00         3.00   8.00   8.00   8.00     9.0   1.0

Loop-Carried Dependencies Analysis Report
-----------------------------------------
 290 |  1.0 | add       x15, x15, 128                  | [290]
```

Listing 4.3: Condensed OSACA output for the analysis of the gcc-compiled COPY benchmark kernel on a Marvell TX2 micro-architecture.

```
Instruction Info:
[1]: #uOps               [4]: MayLoad
[2]: Latency             [5]: MayStore
[3]: RThroughput         [6]: HasSideEffects (U)

[1]    [2]    [3]    [4]    [5]    [6]    Instructions:
 3      4     0.50    *                   ldr   q9, [x19, x15]
 4      1     0.67           *            str   q16, [x20, x16]

Resource pressure per iteration:
[0]    [1]    [2]    [3]    [4]    [5]
11.32  11.33  11.35   -     8.00   8.00

Resource pressure by instruction:
[0]    [1]    [2]    [3]    [4]    [5]    Instructions:
0.66   0.33   0.01    -      -      -     add   x16, x15, #16
 -     0.67   0.33    -      -     1.00   ldr   q9, [x19, x15]
0.67   0.33    -      -      -      -     add   x30, x15, #32
0.33   0.33   0.34    -      -      -     add   x17, x15, #48
0.33   0.34   0.33    -     1.00    -     ldr   q16, [x19, x16]
0.34   0.33   0.33    -      -     1.00   ldr   q18, [x19, x30]
0.33   0.33   0.34    -      -      -     add   x18, x15, #64
0.33   0.34   0.33    -      -      -     add   x1, x15, #80
0.34   0.33   0.33    -     1.00    -     ldr   q17, [x19, x17]
0.33   0.33   0.34    -      -     1.00   ldr   q19, [x19, x18]
0.33   0.34   0.33    -      -      -     add   x3, x15, #96
0.34   0.33   0.33    -      -      -     add   x2, x15, #112
0.33   0.33   0.34    -     1.00    -     ldr   q20, [x19, x1]
0.33   0.34   0.33    -      -     1.00   ldr   q21, [x19, x3]
0.66   0.68   0.66    -      -     1.00   str   q9, [x20, x15]
0.34   0.33   0.33    -     1.00    -     ldr   q22, [x19, x2]
0.33   0.33   0.34    -      -      -     add   x15, x15, #128
0.68   0.66   0.66    -     1.00    -     str   q16, [x20, x16]
0.66   0.66   0.68    -      -     1.00   str   q18, [x20, x30]
0.66   0.68   0.66    -     1.00    -     str   q17, [x20, x17]
0.68   0.66   0.66    -      -     1.00   str   q19, [x20, x18]
0.66   0.66   0.68    -     1.00    -     str   q20, [x20, x1]
0.66   0.68   0.66    -      -     1.00   str   q21, [x20, x3]
0.66   0.66   0.68    -     1.00    -     str   q22, [x20, x2]
0.34   0.33   0.33    -      -      -     cmp   x23, x15
 -      -     1.00    -      -      -     b.ne  .L17
```

Listing 4.4: Extract of the LLVM-MCA analysis for the gcc-compiled COPY benchmark kernel on a MArvell TX2 micro-architecture.

```
double * restrict a, * restrict b, * restrict c;

for(long i=0; i < size; ++i){
    a[i] = b[i] + c[i];
}
```

The code is again throughput bound and easily vectorizable by the compiler. The results of the analysis are shown in Table 4.2.

Arch	Compiler	Unroll × SIMD factor	Measured [cy/asm it]	Prediction [cy/asm it]								Accuracy		
				OSACA			LLVM-MCA			IACA		OSACA	LLVM-MCA	IACA
				TP	CP	LCD	TP	CP	LCD	TP	BlockTP			
TX2	armclang	32 × 2	49.82	48.00	10	1	39.02	104	99	—	—	96 %	78 %	—
	gcc	8 × 2	19.84	12.00	11	1	17.01	43	39	—	—	60 %	86 %	—
CSX	icc	2 × 8	3.13	3.00	8	1	2.03	17	2	3.00	3.00	96 %	65 %	96 %
	gcc	8 × 4	12.12	12.00	8	1	8.06	29	8	12.00	12.00	99 %	67 %	99 %
Zen	gcc	8 × 4	12.04	12.00	7	1	12.00	32	9	—	—	100 %	100 %	—

Table 4.2.: Prediction results of the VECTOR ADD benchmark in Section 4.2.2.

OSACA predicts a throughput of 96 % to 99.7 % compared to the actual measurements for all x86 and the armclang-built versions of the kernel and is better than or on par with the other tools. Only for the gcc-compiled version on TX2 it mispredicts the runtime by 40 %. This example is especially interesting since the armclang version is predicted almost perfectly and the two code snippets are similar to each other. Both use 128-bit quadwords and the `fadd` instruction for summing up the vectors. Thus, the inaccuracy is likely related to one or more of the following properties::

- *Unroll factor*
  While the armclang version unrolls by the factor of 32 excluding the SIMD unrolling, the gcc-compiled code is only unrolled eight times. Combined with insufficient speculative execution and deep pipelines, this may prevent the code to be executed at the throughput limit.
- *Use of LOAD and STORE instructions*
  The armclang-built code uses `ldp` and `stp` instructions for loading and storing data while gcc uses simple `ldr` and `str` instructions. We cannot measure any difference in the throughput behaviour between two `ldr`/`str` and one `ldp`/`stp` instruction and therefore assume TX2 internally splits a pair-wise LOAD and STORE into two separate instructions.
- *Instruction ordering*
  We can observe two different behaviours in interleaving the high-level iterations to overcome latencies in dependency chains: On the one hand the armclang version interleaves the computation of four quadwords, i.e., eight high-level iterations, before adding a similar code snippet after the previous one to complete one assembly iteration after eight of these snippets. After half of the assembly kernel the behaviour is slightly changed and no repeating pattern in ordering LOAD, ADD and STORE instructions can be found. On the other hand the gcc version only interleaves the LOAD and the ADD instruction of the vectors and puts seven of the overall eight STORE instructions at the very end of the kernel. This would suggest that the out-of-order engine of the TX2 architecture is not able to exploit the kernel's full instruction level parallelism. We therefore assume this to be the most likely reason for the observed runtime behaviour of the CPU; the TX2 is not able to handle many adjacent STORE instructions without a penalty in execution. However, for adding this to our model more investigation is necessary.

The LLVM-MCA tool has again a higher accuracy of 86 % for the gcc-compiled kernel due to the a bottleneck in the arithmetic ports and, thus, does not help for enhancing our model based on different assumptions.

4.2.3. Vector update

The vector update benchmark is an in-place update of an array by multiplying it with a scalar floating-point:

```
double * restrict a;

for(long i=0; i < size; ++i){
    a[i] = scale * a[i];
}
```

The kernel executes one LOAD, one MUL, and one STORE. Therefore, we assume the code to be bound to the AGUs or the store port for the architectures our testbed. The analysis by OSACA, LLVM-MCA and IACA is presented in Table 4.3.

Arch	Compiler	Unroll × SIMD factor	Measured [cy/asm it]	Prediction [cy/asm it]								Accuracy		
				OSACA			LLVM-MCA			IACA		OSACA	LLVM-MCA	IACA
				TP	CP	LCD	TP	CP	LCD	TP	BlockTP			
TX2	armclang	4 × 2	5.22	4.00	10	1	6.04	19	14	—	—	77 %	116 %	—
	gcc	8 × 2	11.01	8.00	20	1	9.02	28	21	—	—	73 %	82 %	—
CSX	icc	2 × 8	2.15	2.00	8	1	2.00	16	2	2.00	2.48	93 %	93 %	115 %
	gcc	8 × 4	8.02	8.00	8	1	8.00	25	8	8.00	8.00	100 %	100 %	100 %
Zen	gcc	8 × 2	8.03	8.00	8	1	8.00	24	8	—	—	100 %	100 %	—

Table 4.3.: Prediction results of thee VECTOR UPDATE benchmark in Section 4.2.3.

On TX2 OSACA predicts an execution time of only 73 % to 77 % of the measured runtime. For both kernels the bottleneck is predicted to be the data ports, which confirms our expectations. Nonetheless, we were unable to identify the reason for the discrepancy between our model and the actual execution and assume the difference is due to an unsufficient unrolling in case of the armclang-compiled version and a high amount of adjacent STORE instructions in case of the gcc-built kernel. LLVM-MCA expects the bottleneck for both kernels to be the arithmetic ports and achieves a better accuracy of 82 % for the gcc-compiled code because it assumes a reciprocal throughput of 1.0 cy per `fmul` instruction, which is double of what OSACA predicts. Furthermore, LLVM-MCA overpredicts the armclang version of the kernel. Looking at the x86 kernels our tool achieves again highly accurate predictions on the same level as LLVM-MCA. IACA achieves precise results for the gcc version on CSX but overpredicts the icc code by 15 % without giving insight about the additional 0.48 cy, even though the pure TP prediction is accurate. Since the kernel uses only an unrolling by the factor of two together with AVX-512 instructions, it seems this is the reason for the slightly worse accuracy of OSACA and LLVM-MCA compared to the SSE and AVX-2 versions of the other x86 kernels and the immoderate added penalty by IACA.

4.2.4. Sum reduction

The kernel of the sum reduction or *vector reduction* exhibits similar behavior to the vector update but differs in accumulating the sum of all values of a vector into a register instead of storing it in another independent array. This creates a dependency chain while iterating through the vector. The high-level code can be written as:

```
double * restrict a;

for(long i=0; i < size; ++i){
    scale = scale + a[i];
}
```

Consequently, we expect to see a loop-carried dependency in the assembly code, since all the vector elements sum up to one scalar value. By using unrolling with modulo variable expansion (MVE), the compiler can shift the bottleneck from the compute part to the memory. Nevertheless, the kernel should be still LCD-bound. We therefore compare the measurements with the LCD value of the predictions. The results of the analysis is shown in Table 4.4.

```
Open Source Architecture Code Analyzer (OSACA) - v0.3
Architecture:       csx

Combined Analysis Report
-----------------------
                                      Port pressure in cycles
   | 0 - 0DV| 1 | 2 -  2D| 3 -  3D| 4 | 5 | 6 | 7 || CP | LCD |
-----------------------------------------------------------------
305|        |   |        |        |   |   |   |   ||    |     | .L19:
306|0.5     |0.5|0.5  0.5|0.5  0.5|   |   |   |   ||  8 |   4 | vaddpd (%rcx), %ymm3, %ymm4
307|0.0     |0.0|        |        |   |0.5|0.5|   ||    |     | addq $256, %rcx
308|0.5     |0.5|0.5  0.5|0.5  0.5|   |   |   |   ||  4 |   4 | vaddpd -224(%rcx), %ymm4, %ymm5
309|0.5     |0.5|0.5  0.5|0.5  0.5|   |   |   |   ||  4 |   4 | vaddpd -192(%rcx), %ymm5, %ymm6
310|0.5     |0.5|0.5  0.5|0.5  0.5|   |   |   |   ||  4 |   4 | vaddpd -160(%rcx), %ymm6, %ymm8
311|0.5     |0.5|0.5  0.5|0.5  0.5|   |   |   |   ||  4 |   4 | vaddpd -128(%rcx), %ymm8, %ymm9
312|0.5     |0.5|0.5  0.5|0.5  0.5|   |   |   |   ||  4 |   4 | vaddpd -96(%rcx), %ymm9, %ymm10
313|0.5     |0.5|0.5  0.5|0.5  0.5|   |   |   |   ||  4 |   4 | vaddpd  -64(%rcx), %ymm10, %ymm11
314|0.5     |0.5|0.5  0.5|0.5  0.5|   |   |   |   ||  4 |   4 | vaddpd -32(%rcx), %ymm11, %ymm3
315|0.0     |0.0|        |        |   |0.5|0.5|   ||    |     | cmpq %rcx, %r15
316|        |   |        |        |   |   |   |   ||    |     | jne     .L19

    4.0      4.0 4.0  4.0 4.0  4.0      1.0 1.0       36    32


Loop-Carried Dependencies Analysis Report
----------------------------------------
 314 | 32.0 | vaddpd    -32(%rcx), %ymm11, %ymm3    | [306, 308, 309, 310, 311, 312, 313, 314]
 315 |  2.0 | cmpq      %rcx, %r15                  | [307, 315]
```

Listing 4.5: Condensed OSACA output for the analysis of the gcc-compiled SUM REDUCTION benchmark kernel on a CSX micro-architecture using the `-Ofast` optimization flag.

Arch	Compiler	Unroll × SIMD factor	Measured [cy/asm it]	Prediction [cy/asm it]								Accuracy		
				OSACA			LLVM-MCA			IACA		OSACA	LLVM-MCA	IACA
				TP	CP	LCD	TP	CP	LCD	TP	BlockTP			
TX2	armclang	32 × 2	25.41	16.00	28	24	28.00	79	67	—	—	94 %	264 %	—
	gcc	8 × 2	46.76	4.50	52	48	5.98	60	48	—	—	103 %	103 %	—
	gcc (`-O3`)	16 × 1	96.04	16.50	105	96	17.01	115	96	—	—	100 %	100 %	—
CSX	icc	4 × 8	4.09	2.00	8	4	2.14	15	4	2.00	4.00	98 %	98 %	98 %
	gcc	8 × 4	30.29	4.00	36	32	4.05	42	32	4.00	32.00	106 %	106 %	106 %
	gcc (`-O3`)	16 × 1	64.17	8.00	68	64	12.01	74	64	12.00	64.00	100 %	100 %	100 %
Zen	gcc	8 × 2	23.60	4.00	28	24	8.00	27	16	—	—	102 %	68 %	—
	gcc (`-O3`)	16 × 1	48.02	8.00	52	48	8.02	72	64	—	—	100 %	133 %	—

Table 4.4.: Prediction results of the SUM REDUCTION benchmark in Section 4.2.4.

We can see that for the armclang and the icc version OSACA's prediction is highly accurate, but for the standard gcc-compiled versions it overpredicts each kernel. This behavior results from the use of the `-Ofast` flag that allows the compiler to use unsafe math operations to optimize the code. For a better understanding, the OSACA analysis for the gcc-compiled kernel on CSX is shown in Listing 4.5.

Since the use of SIMD instructions in the lines 306 and 308–314 already break the order of operations as given in the high-level code, the gcc compiler neglects the constraints by the dependency chain as detected in the "CP" column by OSACA between two repetitions of the benchmark, i.e., repetitions done to achieve a runtime higher than a reasonable amount of time, and therefore creates an overlap of the single repetitions[7]. Therefore, even though all tools expect a LCD of 32 cy, the code runs slightly faster and OSACA mispredicts the gcc versions by 2 %–6 % with the same accuracy as LLVM-MCA and IACA. Only on Zen LLVM-MCA predicts far below the actual runtime due to the assumption of a latency of 2 cy for the `vaddpd` instruction inside of the LCD. OSACA assumes here 3 cy.

We can overcome this behavior during the measurements by writing the assembly manually, avoiding repetitions of benchmark or forbid the use of "unsafe" instructions, which necessarily rules out SIMD vectorization. To check the accuracy of OSACA in the latter case, we compiled the high-level kernels with gcc using the `-O3` flag instead of `-Ofast`. The results are listed in Table 4.4 with "`-O3`" in parentheses next to the compiler. In the column showing the unroll factor we can now see the compiler only used scalar accumulates, resulting in a highly accurate prediction with 99.73 %–99.95 % accuracy. Both IACA and LLVM-MCA now are also able to predict a correct runtime for the TX2 and CSX system and only

[7] Discovered and explained by Georg Hager in https://blogs.fau.de/hager/archives/7658 (accessed January 29, 2020).

```
Open Source Architecture Code Analyzer (OSACA) - v0.3
Architecture:       csx
Combined Analysis Report
------------------------
                                    Port pressure in cycles
    | 0 - 0DV| 1 | 2 - 2D | 3 - 3D | 4 | 5 | 6 | 7 || CP | LCD |
-----------------------------------------------------------------
392 |        |   |        |        |   |   |   |   ||    |     | ..B1.38:
393 |        |   |        |        |   |   |   |   ||    |     | # Execution count [2.22e+03]
394 |0.5     |   |0.5  0.5|0.5  0.5|   |0.5|   |   ||    |  4  | vaddpd (%r13,%rax,8), %zmm4, %zmm4
395 |0.5     |   |0.5  0.5|0.5  0.5|   |0.5|   |   ||    |     | vaddpd 64(%r13,%rax,8), %zmm3, %zmm3
396 |0.5     |   |0.5  0.5|0.5  0.5|   |0.5|   |   ||    |     | vaddpd 128(%r13,%rax,8), %zmm2, %zmm2
397 |0.5     |   |0.5  0.5|0.5  0.5|   |0.5|   |   || 8  |     | vaddpd 192(%r13,%rax,8), %zmm1, %zmm1
398 |0.0     |0.5|        |        |   |0.0|0.5|   ||    |     | addq $32, %rax
399 |0.0     |0.5|        |        |   |0.0|0.5|   ||    |     | cmpq %r14, %rax
400 |        |   |        |        |   |   |   |   ||    |     | jb     ..B1.38

     2.0      1.0 2.0  2.0 2.0  2.0     2.0 1.0       8     4


Loop-Carried Dependencies Analysis Report
-----------------------------------------
 394 |   4.0 | vaddpd     (%r13,%rax,8), %zmm4, %zmm4
 395 |   4.0 | vaddpd     64(%r13,%rax,8), %zmm3, %zmm3
 396 |   4.0 | vaddpd     128(%r13,%rax,8), %zmm2, %zmm2
 397 |   4.0 | vaddpd     192(%r13,%rax,8), %zmm1, %zmm1
 399 |   2.0 | cmpq       %r14, %rax
```

Listing 4.6: Condensed OSACA output for the analysis of the icc-compiled SUM REDUCTION benchmark kernel on a CSX micro-architecture using the `-Ofast` optimization flag.

```
..B1.39:                          # Preds ..B1.38
                                  # Execution count [4.00e+02]
        vaddpd    %zmm3, %zmm4, %zmm3                         #76.5
        vaddpd    %zmm1, %zmm2, %zmm1                         #76.5
        vaddpd    %zmm1, %zmm3, %zmm4                         #76.5
                                  # LOE r13 r14 ebx r12d r15d xmm0 xmm5 zmm4
..B1.40:                          # Preds ..B1.39 ..B1.36
                                  # Execution count [4.00e+02]
        vshuff32x4 $238, %zmm4, %zmm4, %zmm1                  #76.5
        vaddpd    %zmm4, %zmm1, %zmm2                         #76.5
        vpermpd   $78, %zmm2, %zmm3                           #76.5
        vaddpd    %zmm3, %zmm2, %zmm4                         #76.5
        vpermpd   $177, %zmm4, %zmm6                          #76.5
        vaddpd    %zmm6, %zmm4, %zmm7                         #76.5
        vaddsd    %xmm0, %xmm7, %xmm0
```

Listing 4.7: Code snippet of the final accumulation due to MVE in the icc-compiled SUM REDUCTION kernel on CSX.

LLVM-MCA overpredicts the kernel on Zen due to a different assumption of latency for `vaddsd` of 3 cy inside of the LCD while in comparison OSACA assumes 4 cy (which is the correct value on CSX).

But why does OSACA achieve accurate results for the armclang- and icc-compiled versions? To answer this question, we show the OSACA output of the CSX analysis in Listing 4.6.

The Intel compiler uses MVE on top of SIMD and therefore creates four parallel executable LCDs as shown in the bottom in the Loop-Carried Dependencies Analysis Report (instructions 394–397). This way, it bounds the code to the ADD peak of 4 cy per assembly iteration. By all means, this requires extra work after all iterations to accumulate the 32 results which we can find in the assembly file after the loop body and is shown in Listing 4.7. Since this is only done once at the end of all loop iterations, we can neglect the overhead of this code snippet. The armclang-compiled code uses MVE as well but unrolls additionally by a factor of four. Thus, the LCD within this kernel has a length of four instructions instead of one in icc's version. It is not completely clear why LLVM-MCA assumes a runtime of more than 1.5 times the actual measurement in that case but it seems it can not distinguish precisely between the independent MVE loops or struggles with `ldp` instructions since the block TP is already higher than the measured runtime.

4.2.5. DAXPY

The DAXPY benchmark adds a scalar multiple of a floating-point vector to another floating-point vector and is therefore well suited to analyze the fused multiply–add (FMA) units in a micro-architecture. The high-level loop is written as follows:

```c
double * restrict a, * restrict b;

for(long i=0; i < size; ++i){
    a[i] = a[i] + scale * b[i];
}
```

The analysis of OSACA, LLVM-MCA and IACA for the DAXPY kernel can be found in Table 4.5.

Arch	**Compiler**	**Unroll × SIMD factor**	**Measured [cy/asm it]**	**Prediction [cy/asm it]** OSACA			LLVM-MCA			IACA		**Accuracy** OSACA	LLVM-MCA	IACA
				TP	LCD	CP	TP	LCD	CP	TP	BlockTP			
TX2	armclang	32 × 2	48.40	48.00	10	1	41.02	106	99	—	—	99 %	85 %	—
	gcc	8 × 2	13.12	12.00	20	1	12.00	37	31	—	—	91 %	91 %	—
CSX	icc	1 × 8	1.82	1.50	8	1	1.25	16	1	1.50	1.74	83 %	69 %	96 %
	gcc	8 × 4	12.12	12.00	8	1	8.06	29	8	12.00	12.00	99 %	67 %	99 %
Zen	gcc	8 × 2	12.02	12.00	8	1	12.00	30	8	—	—	100 %	100 %	—

Table 4.5.: Prediction results of the DAXPY benchmark in Section 4.2.5.

In all versions OSACA expects the code to be throughput bound and identifies the bottleneck at the data ports. It achieves for all cases a higher or equal accuracy than LLVM-MCA and predicts the same throughput for the Intel architectures as IACA. By adding an unknown penalty to the throughput prediction in the icc-compiled case, IACA predicts the iteration runtime with 96 % accuracy more precisely than OSACA. Even though we can find five consecutive STORE instructions at the end of the gcc-built loop body for the TX2 system and have the same unroll factor of 16, we do not experience a big loss in accuracy as seen for the VECTOR ADD benchmark in Section 4.2.2, which might be related to the fact that the computation is done in-place for the array in case of DAXPY but is completely independent in case of the VECTOR ADD. This cannot be verified at the time of writing.

4.2.6. STREAM triad

In comparison to the previously covered STREAM kernels, i.e., copy and add, the triad benchmark shows a more complex scenario. Its kernel accumulates and multiplies two vectors and a scalar floating-point and stores the result in a third vector:

```c
double * restrict a, * restrict b, * restrict c;

for(long i=0; i < size; ++i){
    a[i] = b[i] + scale * c[i];
}
```

The analysis results are shown in Table 4.6.

Arch	**Compiler**	**Unroll × SIMD factor**	**Measured [cy/asm it]**	**Prediction [cy/asm it]** OSACA			LLVM-MCA			IACA		**Accuracy** OSACA	LLVM-MCA	IACA
				TP	CP	LCD	TP	CP	LCD	TP	BlockTP			
TX2	armclang	32 × 2	64.61	61.50	10	1	42.02	133	127	—	—	95 %	65 %	—
	gcc	8 × 2	19.73	12.00	11	1	17.01	43	39	—	—	61 %	86 %	—
CSX	icc	1 × 8	1.74	1.50	8	1	1.25	16	1	1.50	1.74	86 %	72 %	100 %
	gcc	8 × 4	12.12	12.00	8	1	8.06	29	8	12.00	12.00	99 %	67 %	99 %
Zen	gcc	8 × 2	12.05	12.00	9	1	12.00	30	8	—	—	100 %	100 %	—

Table 4.6.: Prediction results of the TRIAD benchmark in Section 4.2.6.

Once again OSACA accurately predicts the runtime for the armclang-compiled code on TX2, the icc-built

kernel on CSX and the gcc version on Zen and shows a small deviation for the icc-built code using AVX-512 instruction with no additional unrolling, for which IACA adds a penalty to predict the throughput precisely but does not give any deeper insight. For all versions OSACA assumes the AGU ports as the bottleneck, which meets our expectations. We can see LLVM-MCA returns relatively inaccurate predictions for the armclang version and the kernels on CSX, which is due to `stp` instructions on ARM and the utilization of AGU port 7 on CSX for complex memory addressing, respectively. Nevertheless, it assumes a better runtime for the gcc-compiled kernel on the TX2 than OSACA. This results from the assumption that the bottleneck are the arithmetic ports since it predicts a reciprocal throughput of 12 cy for both data ports and 16.49 cy, 16.50 cy and 17.01 cy for the three ALU ports. We assume the reason for OSACA's accuracy of only 61 % for that kernel is the high amount of adjacent STOREs at the end of the kernel.

4.2.7. Schönauer triad

The Schönauer triad represents a variation of the classic STREAM triad in which the scalar value is replaced by another vector, creating a multiply-add of three vectors, and stores the result in a fourth vector. The high-level code can be written as:

```
double * restrict a, * restrict b, * restrict c, * restrict d;

for(long i=0; i < size; ++i){
    a[i] = b[i] + c[i] * d[i];
}
```

Since the STREAM triad was already AGU-bound and the processor needs to generate one additional address in the present case, we expect the bottleneck at the AGUs. The compiler can reduce this bottleneck on modern Intel architectures by using simple addressing for STOREs to make use of the AGU on port 7. However, we observe very few situations in which compilers make use of the simple AGU on port 7. All measurements and analysis results can be found in Table 4.7.

Arch	Compiler	Unroll × SIMD factor	Measured [cy/asm it]	Prediction [cy/asm it]								Accuracy		
				OSACA			LLVM-MCA			IACA		OSACA	LLVM-MCA	IACA
				TP	CP	LCD	TP	CP	LCD	TP	BlockTP			
TX2	armclang	32 × 2	67.54	66.00	10	1	50.02	139	134	—	—	98 %	74 %	—
	gcc	8 × 2	22.81	16.00	11	1	20.01	52	47	—	—	70 %	88 %	—
CSX	icc	2 × 8	4.34	4.00	8	1	3.03	25	3	4.00	4.00	92 %	70 %	92 %
	gcc	8 × 4	16.08	16.00	8	1	12.04	33	12	16.00	16.00	100 %	75 %	100 %
Zen	gcc	8 × 2	16.05	16.00	8	1	16.00	34	13	—	—	100 %	100 %	—

Table 4.7.: Prediction results for the SCHÖNAUER TRIAD benchmark in Section 4.2.6.

Unfortunately neither icc nor gcc use simple addressing on CSX for this code. Nevertheless, OSACA predicts x86 code compiled by gcc with 100 % accuracy and is as precise as IACA for both kernels on Intel CSX. The prediction of LLVM-MCA for CSX shows only 70 % and 75 % accuracy, respectively, due to the assumption that the processor in fact uses port 7 for generating the store address.

On the ARM-based TX2, OSACA returns a precise prediction for the armclang-compiled kernel, but reveals weaknesses for the gcc version. We can identify five consecutive STORE instructions at the end of the kernel but still cannot verify this to be the reason for a possible penalty. Different to OSACA, LLVM-MCA assumes the bottleneck for both TX2 kernels in the arithmetic ports and achieves an accuracy of 74 % for the armclang version and 88 % for the gcc-built code.

4.2.8. Gauss-Seidel method

Almost all kernels investigated so far do not show more than the "trivial" LCD due to the loop counter update. A relevant kernel that is more interesting in this respect is the two-dimensional Gauss-Seidel method [36]. It iterates over a grid, accumulates the neighbors of a cell, multiplies the result with a scalar value and updates the current cell with the new value:

```
Open Source Architecture Code Analyzer (OSACA) - v0.3
Architecture:       tx2

Combined Analysis Report
------------------------
                          Port pressure in cycles
    |  0   - 0DV  |  1   - 1DV  |  2   |  3   |  4   |  5   ||  CP  | LCD  |
------------------------------------------------------------------------------
740 |             |             |      |      |      |      ||      |      | .LBB0_62:
745 | 0.33        | 0.33        | 0.33 | 0.50 | 0.50 |      ||  4.0 |      | ldr   d1, [x7], #8
746 | 0.50        | 0.50        |      |      |      |      ||  6.0 |  6.0 | fadd  d0, d1, d0
747 |             |             |      | 0.50 | 0.50 |      ||      |      | ldr   d2, [x22]
748 | 0.33        | 0.33        | 0.33 | 0.50 | 0.50 |      ||      |      | ldr   d3, [x23], #8
749 | 0.50        | 0.50        |      |      |      |      ||      |      | fadd  d2, d2, d3
750 | 0.50        | 0.50        |      |      |      |      ||  6.0 |  6.0 | fadd  d0, d0, d2
751 | 0.33        | 0.33        | 0.33 |      |      |      ||      |      | sub   w26, w26, #1
752 | 0.50        | 0.50        |      |      |      |      ||  6.0 |  6.0 | fmul  d0, d0, d9
753 |             |             |      | 0.50 | 0.50 | 1.00 ||  4.0 |      | stur  d0, [x22, #-8]
754 | 0.00        | 0.00        | 1.00 |      |      |      ||      |      | add   x22, x22, #8
755 | 0.00        | 0.00        | 1.00 |      |      |      ||      |      | cmp   w26, #2
756 |             |             |      |      |      |      ||      |      | b.gt  .LBB0_62

      3.00          3.00          3.00   2.00   2.00   1.00     26.0   18.0

Loop-Carried Dependencies Analysis Report
-----------------------------------------
 745 |  0.0 | ldr       d1, [x7], #8         | [745]
 748 |  0.0 | ldr       d3, [x23], #8        | [748]
 752 | 18.0 | fmul      d0, d0, d9           | [746, 750, 752]
 754 |  1.0 | add       x22, x22, #8         | [754]
```

Listing 4.8: Condensed OSACA output for the analysis of the armflang-compiled GAUSS-SEIDEL method kernel for the TX2 micro-architecture.

```
double ** restrict a;

for(long k=1; k < size_k-1; ++k){
  for(long i=1; i < size_i-1; ++i){
    a[k][i] = scale * (
      a[k][i-1] + a[k+1][i]
      + a[k][i+1] + a[k-1][i]
    );
  }
}
```

As this update happens in-place, each iteration depends on the previously calculated value of its “left”, i.e., `a[k][i-1]`, and “bottom”, i.e., `a[k-1][i]`, neighbor. Therefore, we expect this dependency chain to be the bottleneck. The result of the predictions and measurements are presented in Table 4.8.

Arch	Compiler	Unroll × SIMD factor	Measured [cy/asm it]	Prediction [cy/asm it]								Accuracy		
				OSACA			LLVM-MCA			IACA		OSACA	LLVM-MCA	IACA
				TP	CP	LCD	TP	CP	LCD	TP	BlockTP			
TX2	armflang	1 × 1	18.37	3.00	26	18	5.00	27	18	—	—	98 %	98 %	—
	gfortran	4 × 1	74.77	8.50	92	72	14.00	97	83	—	—	96 %	111 %	—
CSX	ifort	4 × 1	56.23	8.00	68	56	8.01	76	59	8.00	56.00	100 %	105 %	100 %
	gfortran	8 × 1	102.94	16.00	100	96	16.02	113	104	16.00	96.00	93 %	101 %	93 %
Zen	gfortran	8 × 1	83.51	19.50	84	80	24.00	112	104	—	—	96 %	125 %	—

Table 4.8.: Prediction results of the GAUSS-SEIDEL sweep in Section 4.2.8.

Note that the code for this benchmark is written in Fortran, therefore we used the corresponding Fortran compilers armflang, ifort, and gfortran, respectively. As expected, the actual runtime differs tremendously from the plain TP prediction of all tools. Furthermore, all kernels have a SIMD factor of 1 because of the dependency constraint to process one cell after another. OSACA returns an accurate prediction throughout all kernels on all systems and provides an equally good or better prediction than the related tools LLVM-MCA and IACA.

To gain more insight into the kernel by OSACA’s analysis, its output of the armflang-compiled code for TX2 is shown in Listing 4.8. The assembly kernel consists of three LOADs since the fourth element representing the previous cell in the grid is already stored in a register, in this particular case in `d0`. The kernel accumulates (`fadd`) all four cells and multiplies (`fmul`) the scalar floating-point. Subsequently, the

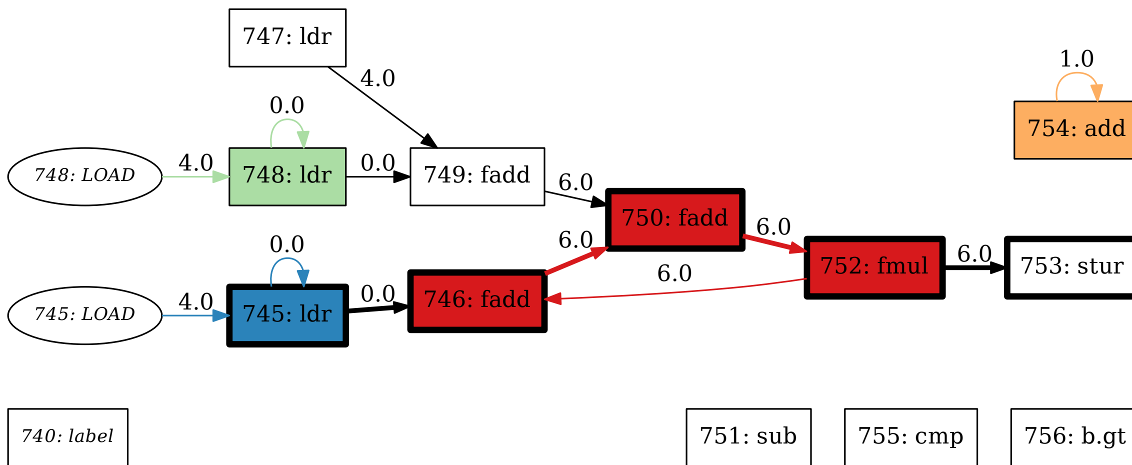


Figure 4.1.: Depdendency graph of the GAUSS SEIDEL method kernel compiled by armflang for TX2. Vertices represent individual instruction forms while the edges indicate dependencies in between the instruction forms, weighted with the latency of the start vertex of the edge. Each LCD is symbolized by a different color and the CP is marked with bold frames. The graph is created by OSACA using the `--export-graph` command line option and adjusted for printing.

result gets stored into the memory. As we can see in the LCD-column, only two ADDs and the MUL are part of the LCD since the load and store, which are part of the CP, can be done independently and therefore overlap with previous and following iterations. To visualize CP and LCDs as well as the data flow of the kernel, OSACA can produce a dependency graph as a DOT file when given the `--export-graph` option to the command line. This is shown in Figure 4.1. While the CP is marked in bold for the whole path, each LCD is represented in a different color. In Figure 4.1, the LCD defining the bottleneck is marked in red. We can see the compiler creates suboptimal code (as do the other compilers) since a rearrangement of the ADDs can shorten the LCD by one instruction: If the kernel accumulates all three memory values of the neighbor cells first, only one `fadd`, i.e., the ADD containing the previously updated cell, and the MUL are part of the LCD and the dependency chain. Consequently, the LCD would have a length of 12 cy instead of 18 cy for the shown code.

Furthermore, besides the LCD consisting of the loop pointer increment in line 754, marked in orange, we can identify two more LCDs in the load instructions at 745 and 748. These two LOADs use post-indexing, i.e., the base register is stored with its new value after calculating the memory address using the immediate index. We assume that these LCDs take 1 cy to execute, but they are currently not considered for CP and LCD analysis. We therefore assume the latency of the additional accumulate of the base register as 0 for now and need to investigate the validity of assigning it a latency of 1 cy since the processor might overcome this computation time by using shadow registers or overlap with the LOAD.

All other kernels for the Gauss-Seidel method are based on the same structure but use a higher unroll factor of 4 and 8, respectively. LLVM-MCA overpredicts four out of the five kernels as it cannot identify the LCD precisely.

4.2.9. 2D-5pt stencil

A two-dimensional five-point stencil code, also called *2D Jacobi* kernel, is a well known method of iterative kernel updates as such structured kernels emerge from finite difference discretizations.. It computes the arithmetic mean of a cell out of its four neighbors in a 2D grid and, other than the Gauss-Seidel method, writes the result into a second matrix:

```c
double ** restrict a, ** restrict b;

for(long k=1; k < size_k-1; ++k){
  for(long i=1; i < size_i-1; ++i){
    a[k][i] = 0.25 * (
      b[k][i-1] + b[k+1][i]
      + b[k][i+1] + b[k-1][i]
    );
  }
}
```

Since the kernel computes each cell independently and needs four LOADS, three ADDS, one MUL and one STORE per SIMD iteration, we expect it to be throughput bound by the AGU ports. The analysis results are shown in Table 4.9.

Arch	Compiler	Unroll × SIMD factor	Measured [cy/asm it]	Prediction [cy/asm it]								Accuracy		
				OSACA			LLVM-MCA			IACA		OSACA	LLVM-MCA	IACA
				TP	CP	LCD	TP	CP	LCD	TP	BlockTP			
TX2	armclang	16 × 2	49.31	34.00	28	1	43.49	130	122	—	—	69 %	88 %	—
	gcc	4 × 2	22.73	8.50	23	1	4.98	38	14	—	—	37 %	22 %	—
CSX	icc	4 × 8	18.10	10.00	21	1	9.43	36	9	10.00	11.53	55 %	52 %	64 %
	icc (AVX)	4 × 4	11.78	10.00	20	1	8.03	31	7	10.00	10.53	85 %	68 %	89 %
	icc (SSE)	4 × 2	9.51	8.50	12	1	8.00	31	7	8.50	10.53	89 %	84 %	102 %
	gcc	4 × 4	14.00	10.00	16	1	8.03	30	8	10.00	10.47	71 %	57 %	75 %
	gcc (SSE)	4 × 2	10.82	8.00	17	1	8.01	28	7	8.00	9.79	74 %	74 %	91 %
Zen	gcc	4 × 2	10.67	8.50	15	1	12.00	28	11	—	—	80 %	112 %	—

Table 4.9.: Prediction results of the 2D5PT STENCIL or 2D JACOBI kernel in Section 4.2.9.

In addition to the code versions used in previous examples, we also created specific versions for CSX forcing the compiler to use only AVX or SSE instructions. These kernels are marked with their instruction set limitation mentioned in the Compiler column. We can observe the prediction done by OSACA for the regular kernels is highly imprecise as it ranges from an accuracy of 37 % for the gcc-compiled code on TX2 up to 80 % for Zen. However, comparing these results to the predictions of LLVM-MCA and IACA, we see both tools predict a runtime similar to OSACA. While IACA achieves an accuracy of 64 % and 75 % for the icc-built and the gcc-compiled version, respectively, LLVM-MCA shows no detectable pattern as it overpredicts the kernel on Zen, reaches an accuracy of 52 % and 57 % on CSX and predicts a runtime less than a quarter of the measured runtime for the gcc-compiled kernel on TX2. Only for the armclang version it achieves an accuracy of 88 % and predicts a bottleneck in the two NEON ports. A manual analysis of the assembly did not show any noticeable problems, thus, we decided to additionally run an analysis by all tools for less vectorized versions of the kernel on CSX to be able to comprehend the process. While OSACA achieves a slightly better accuracy of 74 % (previously 71 %) for the gcc-compiled version using only SSE instructions, we can notice a conspicuously rise of accuracy with icc from 55 % to 85 % and further 89 % for the AVX-only and SSE-only variants, respectively. The analysis with LLVM-MCA shows similar behavior as the accuracy rises from 57 % to 74 % for the gcc-built kernel using only SSE instructions and from 52 % to 68% and 84 %, respectively, using the icc compiler. IACA adds for all kernels an unknown penalty to its BlockTP value and reaches 89 % and 91% accuracy for the AVX-only icc code and the SSE-only gcc code also overpredicts the SSE-only icc-built kernel. These results let us assume an unknown factor in all three models for CSX whose impact gets stronger as the SIMD width grows. Likewise, the code on TX2 must hit some unknown limitations not encompassed by our architecture-specific model yet.

This unexpected behavior needs to be taken into consideration for future enhancements of the model and reveals weaknesses in all the investigated tools.

5 Conclusion and Future Work

5.1. Summary

We have shown that the automatic detection of dependencies within an assembly loop kernel to detect the critical path (CP) and loop-carried dependencies (LCDs) is possible and its analysis allows a detailed insight into in-core loop performance. This is not only feasible for x86-based micro-architectures but has been applied to an ARM-based hardware architecture as well. The OSACA tool can extract loop kernels out of a marked x86 or AArch64 assembly files and applies a machine model on the instruction forms of the selected code to return an accurate runtime prediction based on lower-bound optimally scheduled throughput prediction, loop-carried dependencies and a worst-case critical path estimation. For the supported platforms, ISA- and instruction-specific performance data was heavily extended and OSACA's database allows both the out-of-the-box analysis of simple kernels and an extension of the machine model by the user to analyze complex, user-specific codes of interest. OSACA can differentiate between various memory addressing formats and dynamically combines memory addressing within instruction forms to provide a broader support of instructions and simplify the database handling for the user.

While especially the ARM-based ThunderX2 (TX2) and modern AVX-512 kernels show performance limitations not fully covered by our current model, OSACA provides an overall accurate prediction for various scientific kernels and, thus, stands out against related tools with limited functionality in terms of support of multiple micro-architectures, dependency detection within and in between loops, user insight and adaptability. By using OSACA, an experienced user can gain insight into the investigated hardware by spotting abnormalities that otherwise might have stayed undetected, such as inefficient code generation of the compiler, loop overlapping or hidden hardware limitations. With its current scope of functionality OSACA may be used as a replacement for Intel's proprietary IACA tool, which reached its end-of-life, and represents an alternative to the LLVM Machine Code Analyzer (LLVM-MCA) tool provided by the LLVM project. OSACA is also integrated in Kerncraft to be used as in-core performance predictor for a complete analysis, e.g., with the ECM model.

5.2. Future Work

In the future we intend to extend OSACA in several ways. It is planned to support further micro-architectures like AMD's Zen 2, ARM Neoverse N1 and eventually IBM's Power9. For the support of IBM Power architectures it is inevitable to implement a new assembly parser and to address vendor peculiarities and therefore it is an open question how challenging a support of all functionalities will be.

The hardware model of the most recent generations of ARM-based and x86 micro-architectures require more insight to let OSACA return precise runtime predictions under all circumstances. We therefore want to explore the behavior of Intel Cascade Lake X (CSX) as well as Intel Skylake-X (SKX) systems to identify possible penalties occurring during the execution of AVX-512 kernels without additional unrolling. Furthermore, we intend on gaining more insight on the Marvell TX2 architecture. We noticed an increase of performance for single-core execution when disabling simultaneous multithreading (SMT). We therefore assume a static distribution of resources among the logical cores and want to model this in OSACA if this premise applies to this specific machine. Consequently, we want to investigate the behavior of consecutive STORE instructions on TX2 to validate our assumptions made in Section 4.2 and identify if this behavior is system or instruction set architecture (ISA)-specific to apply the knowledge to our current and future hardware models. As seen in Section 4.2.9, some kernels experience penalties during execution. This could neither be detected by OSACA nor by any other tested tool. Thus, we want to investigate more complex kernels to enhance the robustness and versatility of OSACA.

Furthermore, we want to overcome some known issues in the LCD analysis of OSACA. Due to the lack of runtime information, we cannot build fully accurate dependency chains with static analysis. Nonetheless, we intend on modeling static read-after-write dependencies within and across loops. A dependency analysis on kernels unrolled by a higher factor than two will be able to detect longer LCDs occurring only after several iterations. Even though we stored the information in the data files, we currently do not make use of the number of μ-ops of an instruction form. Considering this we additionally could pinpoint the instruction throughput of the kernel as bottleneck. Pre- and post-indexing on ARM as well as flag register modifications can be modeled as separate sub-instructions to enable a more precise LCD detection and simplify the representation for the user. In addition, we want to enhance the modeling of latencies for macro-operations that do not need an operand in the beginning of the execution, e.g., a fused-multiply-add instruction might access the register for adding the product out of the other two registers later than the rest. The fusion of μ-ops is currently not supported, but can be achieved by changes in the hardware model in future releases.

The consideration of performance characteristics done both out-of-order and in-order before the execution, e.g., the decode stage, the reorder buffer or shadow registers, is currently out of scope, even though they all may limit the execution throughput and impose latency penalties.

Finally, to simplify the access to OSACA for non-experienced users, we want to provide a web interface to interactively run OSACA, similar to the LLVM-MCA support in M. Godbolt's Compiler Explorer[1].

[1] See https://godbolt.org/ (accessed January 29, 2020).

A
OSACA MANUAL

In the following the usage of OSACA as command line tool will be explained. All functionalities and options are based on OSACA version 0.3.2dev5[1].

For an overview, Listing A.1 shows the output of the command `osaca --help`. The functionality of OSACA can be separated in three parts: (i) the main analysis of a assembly file based on a micro-architecture, (ii) the import of micro-benchmark results done by asmbench or ibench, and (iii) a database check to inspect the sanity of the architecture database (DB) and ISA DB. We will inspect all three parts by themselves.

Assembly file analysis

To analyse an assembly file, one must first mark the kernel to be investigated so that OSACA can extract it during parsing. The Intel Architecture Code Analyzer (IACA) tool, which provides similar functionality as OSACA, requires byte markers since it operates on opcode-level. To provide a trade-off between reusability for such tool and convenient usability, OSACA supports both byte markers and comment line markers. While the byte markers for x86 are equivalent to IACA byte markers, the comment keywords “OSACA-BEGIN” and “OSACA-END” are based on LLVM-MCA’s markers. All options for marking the assembly of the currently supported ISAs are presented in Listing A.2.

OSACA in combination with Kerncraft provides a functionality for the automatic detection of possible loop kernels and inserting markers. This can be done by using the `--insert-marker` flag together with the path to the target assembly file. Currently only x86 byte marker insertion is provided but ARM support is under development.

[1] https://github.com/RRZE-HPC/OSACA/releases/tag/v0.3.2.dev5, commit hash: 77aa7f8

```
usage: osaca [-h] [-V] [--arch ARCH] [--fixed] [--db-check]
             [--import MICROBENCH] [--insert-marker]
             [--export-graph EXPORT_PATH] [--ignore-unknown] [--verbose]
             file

Analyzes a marked innermost loop snippet for a given architecture type.

positional arguments:
  file                  Path to object (ASM or instruction file).

optional arguments:
  -h, -help            show this help message and exit
  -V, -version         show program's version number and exit
  -arch ARCH           Define architecture (SNB, IVB, HSW, BDW, SKX, CSX, ZEN1, TX2).
  -fixed               Run the throughput analysis with fixed probabilities or all
                       suitable ports per instruction. Otherwise, OSACA will print out
                       the optimal port utilization for the kernel.
  -db-check            Run a sanity check on the by "--arch" specified database.
                       The output depends on the verbosity level.
  -import MICROBENCH   Import a given microbenchmark output file into the corresponding
                       architecture instruction database. Define the type of
                       microbenchmark either as "ibench" or "asmbench".
  -insert-marker       Try to find assembly block containing the loop to analyse and
                       insert byte marker by using Kerncraft.
  -export-graph EXPORT_PATH
                       Output path for .dot file export. If "." is given, the file will
                       be stored as "./osaca_dg.dot"
  -ignore-unknown      Ignore if instructions cannot be found in the data file
                       and print analysis anyway.
  -verbose, -v         Increases verbosity level.

For help, examples, documentation and bug reports go to:
https://github.com/RRZE-HPC/OSACA/ | License: AGPLv3
```

Listing A.1: OSACA output for running the `--help` command. The listing is adjusted for the reader's convenience.

OSACA starts the analysis on a marked assembly file by running the following command with one or more of the optional parameters:

```
osaca --arch ARCH [--fixed] [--ignore-unknown]
                  [--export-graph EXPORT_PATH]
        file
```

The `file` parameter specifies the target assembly file and is always mandatory. The parameter `ARCH` is positional for the analysis and must be replaced by the target architecture abbreviation. Currently OSACA supports all Intel architectures from Ivy Bridge to Cascade Lake X, AMD Zen and Marvell TX2.

As explained in Section 3.2.1, OSACA assumes an optimal scheduling for all instructions and assumes the processor to be able to schedule instructions in a way that it achieves a minimal reciprocal throughput. However, in version up to 0.2.2 of OSACA, a fixed probability for port utilization was assumed. This means, instructions with N available ports for execution were scheduled with a probability of $1/N$ to each of the ports. This behavior can be enforced by using the `--fixed` flag.

If one or more instruction forms are unknown to OSACA, it refuses to print an overall throughput, CP and LCD analysis and marks all unknown instruction forms with "`X`" next to the mnemonic. This is done so the user does not miss out on this unrecognized instruction and might assume an incorrect runtime prediction. To force OSACA to apply a throughput and latency of 0.0 cy for all unknown instruction forms, the flag `--ignore-unknown` can be specified.

To get a visualization of the analyzed kernel and its dependency chains, OSACA provides the option to additionally produce a graph as DOT file [37], which represents the kernel and all register dependencies inside of it. The tool highlights all LCDs and the CP. The graph generation is done by running OSACA with the `--export-graph EXPORT_GRAPH` flag. OSACA stores the DOT file either at the by

```
    mov $111, %ebx     # START MARKER
    .byte 100,103,144 # START MARKER
.loop:
    # loop body ...
    jb       .loop
    mov $222, %ebx     # END MARKER
    .byte 100,103,144 # END MARKER
```

a: Byte markers for x86 AT&T syntax.

```
    mov x1, #111       // START MARKER
    .byte 213,3,32,31 // START MARKER
.loop:
    // loop body...
    b.ne     .loop
    mov x1, #222       // END MARKER
    .byte 213,3,32,31 // END MARKER
```

b: Byte markers for AArch64 syntax.

```
    # OSACA-BEGIN
.loop:
    # loop body ...
    jb       .loop
    # OSACA-END
```

c: Comment line markers for x86 AT&T syntax.

```
    // OSACA-BEGIN
.loop:
    // loop body ...
    b.ne     .loop
    // OSACA-END
```

d: Comment line markers for AArch64 syntax.

Listing A.2.: Possible OSACA markers (indicated through bold font) for inserting into the assembly file. Listing A.2a and Listing A.2b show the byte markers for x86 and AArch64, respectively. The comment line markers are shown in Listing A.2c (x86) and Listing A.2d (AArch64).

`EXPORT_GRAPH` specified filepath or uses the default filename "`osaca_dg.dot`" in the current working directory. Subsequently, the DOT-graph can be adjusted in its appearance and converted to various output formats such as PDF, SVG, or PNG using the dot[2] command, e.g., `dot -Tpdf osaca_dg.dot -o graph.pdf` to generate a PDF document.

To illustrate this process, both the analysis report and a dependency graph are shown for the STREAM triad on CSX in Figure A.1.

Benchmark import

OSACA supports the automatic integration of new instruction forms by parsing the output of the micro-benchmark tools asmbench[3] and ibench[4]. This can be achieved by running OSACA with the command line option `--import MICROBENCH`:

```
osaca --arch ARCH --import MICROBENCH file
```

`MICROBENCH` specifies one of the currently supported benchmark tools, i.e., "asmbench" or "ibench". `ARCH` defines the abbreviation of the target architecture for which the instructions will be added and `file` must be the path to the generated output file of the benchmark. The format of this file has to match either the basic command line output of ibench, e.g.,

```
[INSTRUCTION FORM]-TP:      0.500 (clock cycles)    [DEBUG - result: 1.000000]
[INSTRUCTION FORM]-LT:      4.000 (clock cycles)    [DEBUG - result: 1.000000]
```

or the command line output of asmbench including the name of the instruction form in a separate line at the beginning, e.g.:

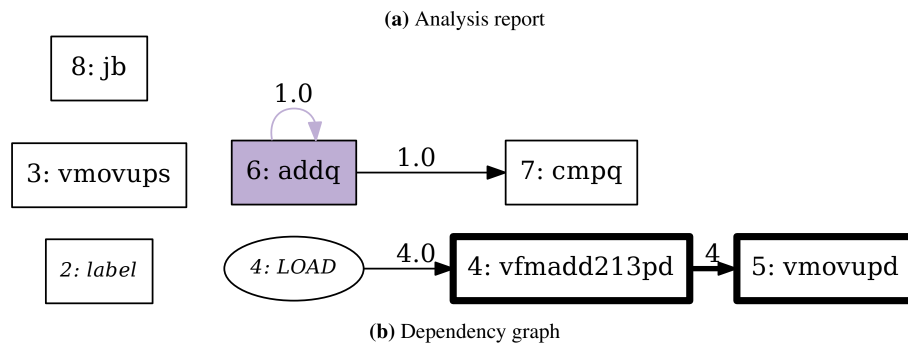
```
[INSTRUCTION FORM]
Latency: 4.00 cycle
Throughput: 0.50 cycle
```

Not that there is an empty line after the throughput measurement as part of the output. For the ISA-specific naming conventions of the instruction form name to be parsed correctly, please check the OSACA

[2]For more information see the DOT guide by Graphviz: https://graphviz.gitlab.io/_pages/pdf/dotguide.pdf
[3]https://github.com/RRZE-HPC/asmbench
[4]https://github.com/RRZE-HPC/ibench

```
Open Source Architecture Code Analyzer (OSACA) - v0.3
Analyzed file:      manual-triad.s
Architecture:       csx

 * - Instruction micro-ops not bound to a port
 X - No throughput/latency information for this instruction in data file


Combined Analysis Report
------------------------
                         Port pressure in cycles
  | 0  - 0DV | 1  | 2  - 2D | 3  - 3D | 4  | 5  | 6  | 7 || CP | LCD|
----------------------------------------------------------------------
2 |          |    |         |         |    |    |    |   ||    |    |  ..TRIAD:
3 |          |    |0.50 0.50|0.50 0.50|    |    |    |   ||    |    |  vmovups (%r13,%rax,8), %zmm1
4 |0.50      |    |0.50 0.50|0.50 0.50|    |0.50|    |   ||  8 |    |  vfmadd213pd (%rcx,%rax,8), \
                                                                           %zmm2, %zmm1
5 |          |    |0.50     |0.50     |1.00|    |    |   ||  0 |    |  vmovupd %zmm1, (%r14,%rax,8)
6 |0.25      |0.25|         |         |    |0.25|0.25|   ||    |  1 |  addq $8, %rax
7 |0.00      |0.50|         |         |    |0.00|0.50|   ||    |    |  cmpq %r12, %rax
8 |          |    |         |         |    |    |    |   ||    |    |* jb ..TRIAD

   0.75       0.75 1.50 1.00 1.50 1.00 1.00 0.75 0.75        8    1


Loop-Carried Dependencies Analysis Report
-----------------------------------------
   6 |  1.0 | addq      $8, %rax                  | [6]
```

(a) Analysis report



(b) Dependency graph

Figure A.1.: Sample output of an OSACA analysis for the STREAM triad (`a[i] = b[i] + scale * c[i]`) on CSX. Listing A.1a shows the analysis report with the predicted port utilization, the CP and the LCDs. Figure A.1b shows the generated DOT dependency graph. The LCD is colored in purple while the CP is marked with bold frames. The edge weights represent the latency of the instruction forms of the start vertex. The output format was adjusted for the convenience of the reader.

git repository[5]. OSACA parses the output for an arbitrary number of instruction forms and adds them as entries to the architecture DB. The user must edit the ISA DB in case the instruction form shows irregular source and destination operands for its ISA syntax. OSACA applies the following rules by default:

- If there is only one operand, it is considered as source operand
- In case of multiple operands the target operand (depending on the ISA syntax the last or first one) is considered to be the destination operand, all others are considered as source operands.

Database check

Since a manual adjustment of the ISA DB is currently indispensable when adding new instruction forms, OSACA provides a database sanity check using the `--db-check` flag. It can be executed via:

```
osaca --arch ARCH --db-check [-v] file
```

[5]https://github.com/RRZE-HPC/OSACA/

`ARCH` defines the abbreviation of the target architecture of the database to check. The `file` argument needs to be specified as it is positional but may be any existing dummy path. When called, OSACA prints a summary of database information containing the amount of missing throughput values, latency values or μ-ops assignments for an instruction form. Furthermore, it shows the amount of duplicate instruction forms in both the architecture DB and the ISA DB and checks how many instruction forms in the ISA DB are non-existent in the architecture DB. Finally, it checks via simple heuristics how many of the instruction forms contained in the architecture DB might miss an ISA DB entry. Running the database check including the `-v` verbosity flag, OSACA prints in addition the specific name of the identified instruction forms so that the user can check the mentioned incidents.

B

ASSEMBLY CODE SUMMARY TABLE

In the following, all results shown during evaluation in Section 4.2, i.e., Table 4.1 to Table 4.9, are summarized in Table B.1. To calculate the runtime of one high level iteration, one must divide the result by the unroll and SIMD factor. The accuracy represents the quotient of the predicted and the measured performance. Since OSACA's prediction is a lower bound of the runtime, we aim to achieve an accuracy slightly below 100 % and consider a prediction with accuracy above 100 % failed. For convenience, the accuracy cells are accordingly colored using a gradient scale from red representing an inaccurate prediction to green representing an accurate prediction. A dark red cell with an accuracy written in bold white represents an overprediction. All analysis outputs and instructions for reproducing the results can be found in the artifact description [35].

Kernel	Arch	Compiler	Unroll × SIMD factor	Measured [cy/asm it]	Prediction [cy/asm it] OSACA TP	OSACA CP	OSACA LCD	LLVM-MCA TP	LLVM-MCA CP	LLVM-MCA LCD	IACA TP	IACA BlockTP	Accuracy OSACA	Accuracy LLVM-MCA	Accuracy IACA
COPY	TX2	armclang	32 × 2	37.29	32.00	4	1	16.00	42	39	—	—	86 %	**43 %**	—
		gcc	8 × 2	11.07	8.00	5	1	11.35	28	20	—	—	72 %	**103 %**	—
	CSX	icc	1 × 8	1.17	1.00	4	1	1.01	12	1	1.00	1.23	86 %	86 %	**105 %**
		gcc	8 × 4	8.11	8.00	4	1	8.00	21	8	8.00	8.00	99 %	99 %	99 %
	Zen	gcc	8 × 2	8.09	8.00	4	1	8.00	22	8	—	—	99 %	99 %	—
ADD	TX2	armclang	32 × 2	49.82	48.00	10	1	39.02	104	99	—	—	96 %	78 %	—
		gcc	8 × 2	19.84	12.00	11	1	17.01	43	39	—	—	60 %	86 %	—
	CSX	icc	2 × 8	3.13	3.00	8	1	2.03	17	2	3.00	3.00	96 %	65 %	96 %
		gcc	8 × 4	12.12	12.00	8	1	8.06	29	8	12.00	12.00	99 %	67 %	99 %
	Zen	gcc	8 × 4	12.04	12.00	7	1	12.00	32	9	—	—	100 %	100 %	—
UPDATE	TX2	armclang	4 × 2	5.22	4.00	10	1	6.04	19	14	—	—	77 %	**116 %**	—
		gcc	8 × 2	11.01	8.00	20	1	9.02	28	21	—	—	73 %	82 %	—
	CSX	icc	2 × 8	2.15	2.00	8	1	2.00	16	2	2.00	2.48	93 %	93 %	115 %
		gcc	8 × 4	8.02	8.00	8	1	8.00	25	8	8.00	8.00	100 %	100 %	100 %
	Zen	gcc	8 × 2	8.03	8.00	8	1	8.00	24	8	—	—	100 %	100 %	—
SUM REDUCTION	TX2	armclang	32 × 2	25.41	16.00	28	24	28.00	79	67	—	—	94 %	**264 %**	—
		gcc	8 × 2	46.76	4.50	52	48	5.98	60	48	—	—	**103 %**	**103 %**	—
		gcc (`-O3`)	16 × 1	96.04	16.50	105	96	17.01	115	96	—	—	100 %	100 %	—
	CSX	icc	4 × 8	4.09	2.00	8	4	2.14	15	4	2.00	4.00	98 %	98 %	98 %
		gcc	8 × 4	30.29	4.00	36	32	4.05	42	32	4.00	32.00	**106 %**	**106 %**	**106 %**
		gcc (`-O3`)	16 × 1	64.17	8.00	68	64	12.01	74	64	12.00	64.00	100 %	100 %	100 %
	Zen	gcc	8 × 2	23.60	4.00	28	24	8.00	27	16	—	—	**102 %**	68 %	—
		gcc (`-O3`)	16 × 1	48.02	8.00	52	48	8.02	72	64	—	—	100 %	**133 %**	—
DAXPY	TX2	armclang	32 × 2	48.40	48.00	10	1	41.02	106	99	—	—	99 %	85 %	—
		gcc	8 × 2	13.12	12.00	20	1	12.00	37	31	—	—	91 %	91 %	—
	CSX	icc	1 × 8	1.82	1.50	8	1	1.25	16	1	1.50	1.74	83 %	69 %	96 %
		gcc	8 × 4	12.12	12.00	8	1	8.06	29	8	12.00	12.00	99 %	67 %	99 %
	Zen	gcc	8 × 2	12.02	12.00	8	1	12.00	30	8	—	—	100 %	100 %	—
TRIAD	TX2	armclang	32 × 2	64.61	61.50	10	1	42.02	133	127	—	—	95 %	65 %	—
		gcc	8 × 2	19.73	12.00	11	1	17.01	43	39	—	—	61 %	86 %	—
	CSX	icc	1 × 8	1.74	1.50	8	1	1.25	16	1	1.50	1.74	86 %	72 %	100 %
		gcc	8 × 4	12.12	12.00	8	1	8.06	29	8	12.00	12.00	99 %	67 %	99 %
	Zen	gcc	8 × 2	12.05	12.00	9	1	12.00	30	8	—	—	100 %	100 %	—
SCH. TRIAD	TX2	armclang	32 × 2	67.54	66.00	10	1	50.02	139	134	—	—	98 %	74 %	—
		gcc	8 × 2	22.81	16.00	11	1	20.01	52	47	—	—	70 %	88 %	—
	CSX	icc	2 × 8	4.34	4.00	8	1	3.03	25	3	4.00	4.00	92 %	70 %	92 %
		gcc	8 × 4	16.08	16.00	8	1	12.04	33	12	16.00	16.00	100 %	75 %	100 %
	Zen	gcc	8 × 2	16.05	16.00	8	1	16.00	34	13	—	—	100 %	100 %	—
GAUSS-S.	TX2	armflang	1 × 1	18.37	3.00	26	18	5.00	27	18	—	—	98 %	98 %	—
		gfortran	4 × 1	74.77	8.50	92	72	14.00	97	83	—	—	96 %	**111 %**	—
	CSX	ifort	4 × 1	56.23	8.00	68	56	8.01	76	59	8.00	56.00	100 %	**105 %**	100 %
		gfortran	8 × 1	102.94	16.00	100	96	16.02	113	104	16.00	96.00	93 %	**101 %**	93 %
	Zen	gfortran	8 × 1	83.51	19.50	84	80	24.00	112	104	—	—	96 %	**125 %**	—
JACOBI	TX2	armclang	16 × 2	49.31	34.00	28	1	43.49	130	122	—	—	69 %	88 %	—
		gcc	4 × 2	22.73	8.50	23	1	4.98	38	14	—	—	37 %	22 %	—
	CSX	icc	4 × 8	18.10	10.00	21	1	9.43	36	9	10.00	11.53	55 %	52 %	64 %
		icc (AVX)	4 × 4	11.78	10.00	20	1	8.03	31	7	10.00	10.53	85 %	68 %	89 %
		icc (SSE)	4 × 2	9.51	8.50	12	1	8.00	31	7	8.50	10.53	89 %	84 %	**102 %**
		gcc	4 × 4	14.00	10.00	16	1	8.03	30	8	10.00	10.47	71 %	57 %	75 %
		gcc (SSE)	4 × 2	10.82	8.00	17	1	8.01	28	7	8.00	9.79	74 %	74 %	91 %
	Zen	gcc	4 × 2	10.67	8.50	15	1	12.00	28	11	—	—	80 %	**112 %**	—

Table B.1.: Summary of all prediction result tables in Section 4.2.

B
BIBLIOGRAPHY

[1] J. Laukemann, J. Hammer, G. Hager, and G. Wellein. Automatic Throughput and Critical Path Analysis of x86 and ARM Assembly Kernels. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, page TBA, Nov 2019. doi: 10.1109/PMBS49563.2019.00006.

[2] Jan Laukemann. Design and Implementation of a Framework for Performance Prediction. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 01 2018. URL `https://github.com/RRZE-HPC/OSACA/blob/master/doc/Design_and_Implementation_For_a_Framework_Predicting_Instruction_Throughput.pdf`.

[3] Lieven Eeckhout. *Computer Architecture Performance Evaluation Methods*. Morgan & Claypool Publishers, 1st edition, 2010. ISBN 1608454673, 9781608454679. doi: 10.2200/s00273ed1v01y201006cac010.

[4] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785.

[5] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware Roofline Model: Upgrading the Loft. *IEEE Comput. Archit. Lett.*, 13(1):21–24, January 2014. ISSN 1556-6056. doi: 10.1109/L-CA.2013.6.

[6] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995. URL `http://cs.virginia.edu/stream`.

[7] Samuel Webb Williams. *Auto-Tuning Performance on Multicore Computers*. PhD thesis, University of California at Berkeley, USA, 2008. doi: 10.5555/1713792.

[8] Georg Hager and Gerhard Wellein. "Simple" performance modeling: The Roofline Model, 2019. URL `https://moodle.rrze.uni-erlangen.de/pluginfile.php/12919/mod_resource/content/8/04_Roofline_Model.pdf`.

[9] Jan Treibig and Georg Hager. Introducing a Performance Model for Bandwidth-Limited Loop Kernels. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, pages 615–624, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14390-8. doi: 10.1007/978-3-642-14390-8_64.

[10] Johannes Hofmann. *A first-principles approach to performance, power, and energy models for contemporary multi- and many-core processors*. PhD thesis, Faculty of Engineering, Friedrich-Alexander-University of Erlangen-Nürnberg, München, 2019. ISBN: 978-3-8439-4187-7.

[11] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 207–216, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3559-1. doi: 10.1145/2751205.2751240.

[12] *Intel® 64 and IA-32 Architecture Optimization Reference Manual*. Intel Corporation, 9 2019. URL `https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual`.

[13] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Computation: Practice and Experience*, 28(2):189–210, 2016. doi: 10.1002/cpe.3180.

[14] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels. In Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2016*, pages 1–22, Cham, 2017. Springer International Publishing. ISBN 978-3-319-56702-0. doi: 10.1007/978-3-319-56702-0_1.

[15] Israel Hirsh and Gideon S. Intel® Architecture Code Analyzer, 2012. URL `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer`.

[16] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 121–131, Nov 2018. doi: 10.1109/PMBS.2018.8641578.

[17] Johannes Hofmann. ibench - Instruction Benchmarks, 2017. URL `https://github.com/RRZE-HPC/ibench`.

[18] Julian Hammer, Georg Hager, and Gerhard Wellein. OoO Instruction Benchmarking Framework on the Back of Dragons. SC18 ACM SRC Poster, 2018. URL `https://sc18.supercomputing.org/proceedings/src_poster/src_poster_pages/spost115.html`.

[19] Dimitry Andric. [RFC] llvm-mca: a static performance analysis tool, 2018. URL `http://llvm.1065342.n5.nabble.com/llvm-dev-RFC-llvm-mca-a-static-performance-analysis-tool-td117477.html`.

[20] Google. EXEgesis, 12 2016. URL `https://github.com/google/EXEgesis`.

[21] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, California, USA, Jun 2019. PMLR. URL `http://proceedings.mlr.press/v97/mendis19a.html`.

[22] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondrej Sykora, Saman Amarasinghe, and Michael Carbin. BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. In *2019 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2019.

[23] A. S. Charif-Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue. CQA: A code quality analyzer tool at binary level. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014. doi: 10.1109/HiPC.2014.7116904.

[24] Vincent Palomares, David C. Wong, David J. Kuck, and William Jalby. Evaluating Out-of-Order Engine Limitations Using Uop Flow Simulation. In Andreas Knüpfer, Tobias Hilbrich, Christoph Niethammer, José Gracia, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2015*, pages 161–181, Cham, 2016. Springer International Publishing. doi: 10.1007/978-3-319-39589-0_13.

[25] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, Bradford Beckmann, Gabriel Black, and et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1, 8 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. doi: 10.1145/2024716.2024718.

[26] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, 2013. doi: 10.1145/2485922.2485963.

[27] Avadh Patel, Furat Afram, and Kanad Ghose. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *1st International Qemu Users' Forum*, pages 29–30, 2011.

[28] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304062. URL `http://doi.acm.org/10.1145/3297858.3304062`.

[29] Agner Fog. 4. Instruction Tables, 1997-2019. URL `http://www.agner.org/optimize/instruction_tables.pdf`.

[30] J. von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15 (4):27–75, 10 1993. ISSN 1934-1547. doi: 10.1109/85.238389.

[31] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017. ISBN 0128119055, 9780128119051.

[32] Agner Fog. *3. The Microarchitecture of Intel, AMD and VIA CPUs*. Technical University of Denmark, http://www.agner.org/optimize/microarchitecture.pdf, 1997-2018.

[33] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sep. 1972. ISSN 2326-3814. doi: 10.1109/TC.1972.5009071.

[34] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc., USA, 1989. ISBN 0201120372.

[35] Artifact Description: Cross-Architecture Automatic Critical Path Detection For In-Core Performance Analysis. URL `https://github.com/RRZE-HPC/OSACA-Artifact-Appendix`.

[36] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[37] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem phong Vo. A technique for drawing directed graphs. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 19(3):214–230, 1993. doi: 10.1109/32.221135.