Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures

Jan Laukemann*, Julian Hammer†, Johannes Hofmann‡, Georg Hager§ and Gerhard Wellein¶
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
*Department of Computer Science
jan.laukemann@fau.de
†Erlangen Regional Computing Center
julian.hammer@fau.de
‡Chair of Computer Architecture
johannes.hofmann@fau.de
§Erlangen Regional Computing Center
georg.hager@fau.de
¶Erlangen Regional Computing Center
gerhard.wellein@fau.de

***Abstract*—An accurate prediction of scheduling and execution of instruction streams is a necessary prerequisite for predicting the in-core performance behavior of throughput-bound loop kernels on out-of-order processor architectures. Such predictions are an indispensable component of analytical performance models, such as the Roofline and the Execution-Cache-Memory (ECM) model, and allow a deep understanding of the performance-relevant interactions between hardware architecture and loop code.**

We present the Open Source Architecture Code Analyzer (OSACA), a static analysis tool for predicting the execution time of sequential loops comprising x86 instructions under the assumption of an infinite first-level cache and perfect out-of-order scheduling. We show the process of building a machine model from available documentation and semi-automatic benchmarking, and carry it out for the latest Intel Skylake and AMD Zen micro-architectures.

To validate the constructed models, we apply them to several assembly kernels and compare runtime predictions with actual measurements. Finally we give an outlook on how the method may be generalized to new architectures.

***Index Terms*—benchmarking, performance modeling, performance engineering, architecture analysis, static analysis**

I. Introduction

Looking at numerical codes, compute-intensive applications and the resources (time, energy, hardware) they consume, it is vital to optimize them for performance in order to reduce their resource consumption. One of the most fundamental ways of approaching this is performance modeling, where a (simplified) model of the underlying hardware is used to predict the runtime of a computational kernel. The Roofline [1] and ECM [2] performance models are probably the most common tools employed for this task on modern CPUs. When applying them, a performance-aware developer will start to build an understanding of the characteristics of the architecture-code interactions, and the model will pinpoint the constraining bottleneck. Once known, the bottleneck can often be mitigated by changes in the code, the runtime parameters, or the execution environment. When the models' construction is automated [3], [4], compilers and a wider user base can take advantage of them.

In practice, the analysis and modeling process on a given architecture is typically split in two parts: in-core execution and data transfer. For example, in the simplest form of the Roofline model, calculating the chip's maximum performance taking only the floating-point operations into account is the in-core execution analysis while deriving the arithmetic intensity relies on data transfer analysis. In this work we focus on a refined in-core execution analysis, where the essential questions is: How many cycles does it take *at least* to execute a set of assembly instructions that constitute the body of an infinite loop? The resulting cycle count yields an absolute upper performance bound (or roof), and it is valid for all processor models of the same microarchitecture. This is not the same as counting FLOPs, but a similar and more realistic approach, which may also be applied to non-floating-point codes [4].

The in-core analysis makes a number of assumptions, which will be explained later in further detail. Intel already provides the Intel Architecture Code Analyzer (IACA) [5], an in-core static analyzer for their latest architectures. It has proven extremely valuable for analytic performance modeling. Unfortunately, IACA is both closed-source and restricted to Intel CPUs. We want to develop an open version, with which developers can not only see the analysis outcome but also the underlying model. Beyond the tool itself we want to extend our approach to other, non-Intel architectures and platforms.

This paper is organized as follows: In Sections I-A through I-C, we elaborate on the assumptions stated above, give a general overview of the hardware model and describe relevant features of our example architectures and the hardware/software environment. Section I-D covers related work.

This work was partly funded by BMBF through the METACCA project.

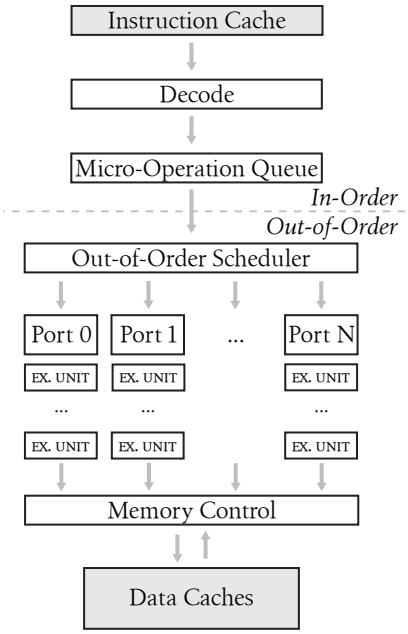

Fig. 1: Assumed generic out-of-order port model for modeling, benchmarking and analysis. Functional units (e.g., ALU, AGU, MUL, DIV) are associated with ports. An out-of-order scheduler assigns μ-ops to ports, which then use their functional units to execute the instructions in an pipelined fashion.

In Section II we explain how to build a detailed machine model for an architecture from available documentation and benchmarking. We exercise this methodology on our example architectures in Section II-C. Section III explains technical details about the static analyzer and compare its predictions with actual measurements in Sections III-A and III-B. Finally, Section IV summarizes the work and gives an outlook to future developments.

The OSACA software is available for download at [6]. Information about how to reproduce the results in this paper can be found in the artifact description [7].

A. Background

When thinking about the performance of a CPU core, we assume what is widely known as the "port model": each instruction is (optionally) split into *micro-ops* (μ-ops), which get executed by functional units. A particular instruction may have multiple functional units that can execute it (e.g., two integer ALUs), or – in case of complex instructions – multiple functional units *must* execute it (e.g., combined load and floating-point add). Functional units are grouped behind ports, with one port serving one or more units. Each port can receive only one instruction per cycle. Figure 1 shows a diagram of such a generic port model.

The following assumptions, already stated in Section I, are assumed for our prediction model:

1) *All data accesses hit the first-level cache.*
   This is where the boundary between in-core and data analysis is drawn. If a dataset fits in the first-level cache, all accesses will behave the same and there is no need to consider the order and pattern of previous accesses or (possibly undisclosed) cache replacement algorithms. Behavior beyond L1 can be modeled separately, but this is beyond the scope of this work (the Kerncraft tool [4], which relies on an in-core analysis from IACA and – in the future – OSACA, combines it with data analysis for a unified Roofline or ECM prediction).
2) *Multiple available ports per instruction are utilized with fixed probabilities.*
   Since the actual scheduling algorithm is unknown, we assume that all suitable ports for the same instruction are used with fixed probabilities. E.g., an add instruction that may use one of two ports may be scheduled half the time on one and half the time on the other, or one-tenth of the time on one and nine-tenth of the time on another. Consideration of actual port pressure is currently not supported yet, but may be considered when it becomes necessary to better mimic measured performance.
3) *Otherwise, out-of-order scheduling by the hardware works perfectly.*
   The previous assumption implies imperfect scheduling if ports are asymmetric. Asymmetry means that multiple ports can handle the same instruction, but other features of those ports differ (e.g., one port supports `add` and `div`, while another supports `add` and `mul`). This may cause load imbalance since, e.g., a code with only `add` and `mul` may be imperfectly scheduled. Since the actual scheduling scheme is unknown and can only be inferred by thorough measurements, reverse engineering the details of the scheduling algorithms is left for future work.
4) *All latencies are hidden via speculative execution.*
   Speculative execution and out-of-order scheduling allows the processor to execute a loop kernel with intra-iteration dependencies as a throughput-bound code (i.e., the pipeline which is the bottleneck is fully utilized). In other words, the critical execution path through the loop iteration can be ignored. Similar to IACA, we focus on throughput modeling at the moment and do not model latency.

To the best of our knowledge, assumptions 1, 3 and 4 apply to IACA as well, but due to the undisclosed machine model behind IACA we are unable to validate this. For assumption 2, IACA shifts probabilities to balance port pressures. In Section II, we will go into detail about how we derive our model parameters from available sources and benchmarking.

Available, but incomplete and sometimes misleading sources are: architecture diagrams and performance numbers found in technical manuals [8] and marketing presentations [9] of vendors, third-party researchers [10] and enthusiasts [11] compiling their own benchmarking results.

B. Intel Skylake and AMD Zen Architectures

Comprehensive information is available on Intel's micro-architectures, and we therefore have a clear understanding of the overall behavior. We will now go into performance-relevant details on Intel Skylake, followed by a discussion of AMD Zen.

On Intel Skylake, each port $(0-7)$ can consume one μ-op per cycle. A μ-op may take any number of cycles to retire. Simple instructions (e.g., `vaddpd %xmm1,%xmm2,%xmm3` or "add values in `xmm1` and `xmm2` and store result to `xmm3`") map to exactly one μ-op, while complex instructions are split into multiple μ-ops (e.g., `vaddpd %xmm1,(%eax),%xmm3` or "load values at memory address `eax`, add with values in `xmm1` and store result to `xmm3`").[1]

In Figure 2 we see the mapping of ports to functional units and thus instructions. Scalar integer instructions need either port 0, 1, 5 or 6. 256 bit wide vector instructions go to port 0 or 1. Divides are always handled by port 0. Loads occupy port 2 or 3, and stores need port 4 as well as 2, 3 or 7 for address calculations.

In addition to ports, there are other potential bottlenecks, in particular instruction cache bandwidth and fetch and decode throughput: The L1 instruction cache is limited to 32 KiB and can serve 16 Bytes per cycle to the fetcher. The decoders can emit a total of five μ-ops per cycle, four from simple instructions and one from a complex instruction. Currently we ignore those limits.

During allocation and renaming, architectural register IDs from the machine code are replaced with physical registers. In combination with move elimination and zeroing idioms (also during the allocation and renaming step), the processor is able to locate and circumvent false data dependencies. All independent instructions can then be scheduled on ports providing the necessary functional units.

One new instruction can be scheduled on each port per cycle; however, some special conditions exist. One prominent example, which we also model, is: Divide instructions are executed scheduled on port 0, and they take four cycles, but the port already becomes available to non-divide instructions on the next cycle. We, as well as IACA, model this using an additional port called 0DV, which only handles divide instructions and is occupied for four cycles, while port 0 is only occupied for 1 cycle.

Loads go through ports 2 and 3. Both ports also lead to the necessary address generation units (AGUs). The "store port" (4) does not come with its own AGU, thus each store requires an AGU from port 2 or 3, or – if the address is simple – from port 7.

[1]Unless otherwise noted, we use the AT&T (destination last) form of the x86 assembly syntax here. IACA uses Intel syntax (destination first) in its output.

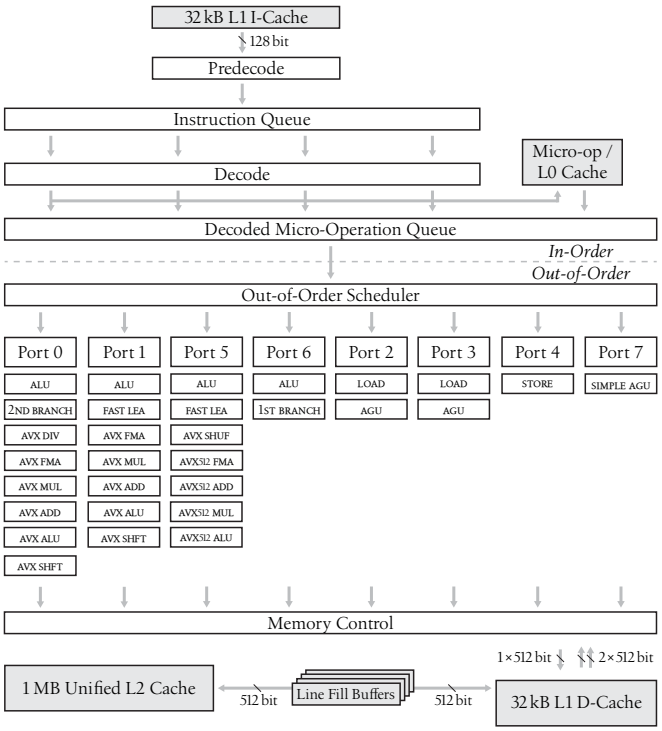

Fig. 2: Intel Skylake core block diagram and port model, compiled from Intel's Optimization Manual [8].

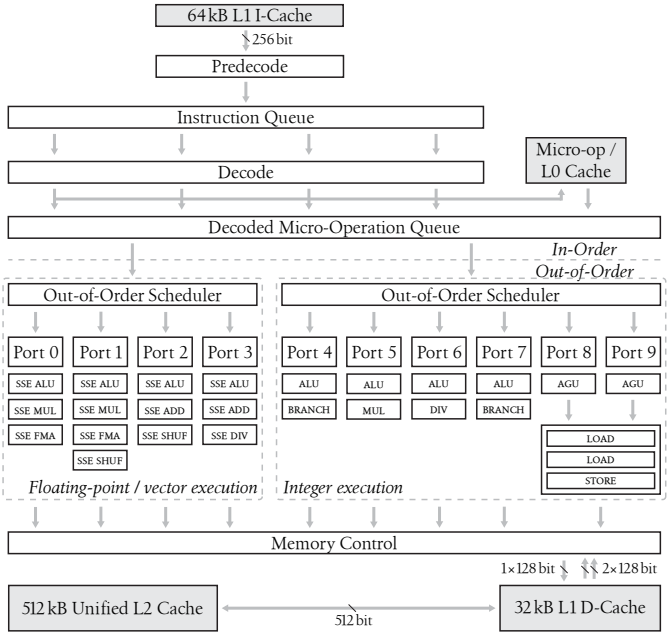

Fig. 3: AMD Zen core block diagram and port model, compiled from AMD's Optimization Guide [12], marketingslides [9] and Agner Fog's Instruction Tables [11].

C. Validation Hardware, Software and Runtime Environment

All results presented were gathered on two machines:

Skylake Intel Xeon i7-6700HQ with Skylake micro-architecture running at fixed 1.8 GHz with turbo disabled

Zen AMD EPYC 7451 with Zen micro-architecture, running at fixed 1.8 GHz with turbo disabled

OSACA (version 0.2.0) was run with Python v3.5.3 and benchmarks were compiled using GCC 7.2.0. When compiling for Intel Skylake we used the flags `-fopenmp-simd -march=broadwell`. Although AVX-512 could be modeled, we deliberately ignored this capability, since we wanted to compare prediction and execution of the same assembly code on both architectures and AVX-512 instructions are not supported on AMD Zen. Compiling for AMD Zen was done with `-fopenmp-simd -march=znver1 -mavx2 -mfma` compiler options. For both platforms we created different versions of the code by using the `-O1`, `-O2` and `-O3` flags, respectively.

During execution, we used `likwid-pin` to pin the processes to a physical core. That and fixing the frequency reduced fluctuations during runtime measurements. Leaving turbo mode enabled would lead to unusual results, because the CPU frequency changes during execution and calculation of cycles from a combined runtime becomes impossible. In effect, statistical runtime variations were small enough to be ignored. In all measurements we nevertheless report the "best" value (highest performance, lowest runtime).

D. Related Work

In general, there are two approaches to predicting runtime and performance behavior: static analysis and simulation. Our work is set in the static analysis category, because we expect results to be explanatory in order to guide developers and tools in optimizing performance, and to be available fast in order to allow inclusion in other tools, such as compilers. Simulators on the other hand may be more thorough and accurate *if* comprehensive implementations exist. They can also consider the data side, such as diverging branches or interaction of multiple cores or nodes. These advantages come at a cost: Steady states for throughput analysis need to be found, valid and representative data needs to be available, pinpointing a bottleneck becomes non-trivial and implementation is much more complex.

Being an inspiration for this work, the most prominent example for static analysis tools is IACA itself [5]. Developed by Israel Hirsh and Gideon S. [sic], Intel released the tool in 2012 and has issued the latest version in 2017. It is closed source and the underlying model neither been published by the authors, nor peer reviewed. The latest version supports throughput analysis on Intel micro-architectures Haswell through Skylake (including AVX-512). It has built-in insight on decomposition of instructions into μ-ops, μ-op fusion and the port assignments. It also seems to use a heuristic for scheduling instructions to ports, which we have no knowledge of. The underlying model is bound to be more accurate than anything OSACA can hope for, due to undisclosed information available to the developers and the complete focus on recent Intel architectures. OSACA, on the other hand, can model non-Intel architectures and gives the user information about the underlying model.

Two new projects came up recently in the LLVM community: LLVM-MCA [13] and LLVM-Exegesis [14]. Both of them aim at enhancing and using available out-of-order performance information in LLVM to improve instruction selection during compile time and to support developers. LLVM-Exegesis benchmarks operations and derives latency and port assignment of solitary instructions (i.e., not of assembly basic blocks) through hardware event counting. The gathered information is meant to validate LLVM's TableDef scheduling models. LLVM-MCA is a simulator that uses the available scheduling information from the backend to predict the expected throughput of a basic block, similar to what IACA and OSACA do. Unlike the latter two, LLVM-MCA actually runs a simulation of instructions through LLVM's backend.

Mendis et al. [15] apply a black-box machine learning approach to throughput estimation, while also trying to capture memory hierarchy behavior beyond the first-level cache. The outcome of their prediction is a single-number throughput estimation based on a generic deep neural network. This is helpful to compilers when comparing possible code transformations, but is not sufficient from a performance engineering perspective, where we are interested in the origin of the bottleneck and hints on how to avoid it. It is also very important to us to separate the memory hierarchy from execution effects in order to support performance modeling using the Roofline and ECM models. Ithemal, their software, and the trained neural network were not publicly available at the time of writing.

Another simulator covering instruction execution is gem5 [16], developed by Binkert et al. It supports many instruction set architectures (x86, ARM, Power and SPARC, among others), including a complete memory system, multi-core, cache coherency, DMA, PCI, networks and more. It is considered a "full-system" simulator, which goes above and beyond what the scope of this work is, but is is rooted in the simulation domain. Gem5 also lacks support for important ISA extensions, such as AVX.

ZSim by Sanches et al. [17] and MARSSx86 by Patel et al. [18] are also full-system simulators which give a coarse overview on complete systems (with thousands of cores or machines), rather than detailed insights pinpointing at a bottleneck.

Charif-Rubial et al. introduced CQA [19], a performance static analysis tool focused on single-core performance of loop-centric code. It is not their goal to predict runtime, but rather give the developer a quality estimate of the code based on static binary analysis. While they also use benchmarks to determine instruction throughput and latency, they have opted for not modeling out-of-order execution.

II. Model-Construction Methodology

To construct a suitable port model for a given CPU architecture, we need to identify the relevant ports for throughput and latency during execution, as well as any other functional units occupied. Additional non-bottleneck units do not influence the runtime of an instruction (the latency is hidden by the bottleneck), but they may become a bottleneck when used in combination with other instructions simultaneously. Identification of hidden non-bottleneck ports can be achieved by combined benchmarking of multiple instructions. In the following sections, we will explain this approach in detail for the latest AMD Zen and Intel Skylake architectures. Further on, we show how to integrate the gained knowledge into OSACA's database [6] for a throughput prediction model.

Since the definition of "instruction" is ambiguous, we introduce the term *instruction form* [20], which refers to an assembly instruction together with their operand types. E.g., `vaddpd` may be used with 128 bit, 256 bit or 512 bit registers, and memory operands and an optional masking register. The types of operands have an impact on the resulting performance and therefore need to be considered. `vaddpd mem,xmm,xmm` is the instruction form of `vaddpd` with a source memory reference, a 128 bit source register and a 128 bit target register.

Although OSACA is capable of distinguishing between different ways of addressing the memory (detecting base, offset, index, scale factor and segment registers), in the current stage of development a separation regarding the benchmark measurement and therefore the port distribution is not provided. Hence, we assume that the maximum throughput of an instruction is independent of its memory addressing mode.

A. Benchmarking Latency and Throughput

To obtain the latency and throughput of an instruction, we automatically create assembly benchmarks for use with ibench [21]. It offers the infrastructure to initialize, run and accurately measure the desired parameters.

For latency benchmarking we create a dependency chain by using the destination register of one instruction as a source register for the next and embedding a suitable number of back-to-back instructions into a loop. A benchmark code for the latency of `vaddpd` may look as follows:

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

The above code yields a latency of 4 cy on Intel Skylake and 3 cycles on AMD Zen.

For throughput measurement, instructions with independent source and destination operands must be issued. This could be achieved by not reusing any destination registers, but will easily exhaust all available registers. Since we do not want to rely on the register renaming capabilities of the core to compensate for that, multiple independent dependency chains are created to ensure that enough independent instructions are available to utilize all functional units. The inner loop body is long enough to compensate loop overheads:

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

This benchmark yields a throughput of 2 instructions per cycle on Intel Skylake and AMD Zen. From this we can infer that two independent ports (and thus pipelines) are available for `vaddpd xmm,xmm,xmm`.

B. Benchmarking Port Occupation

The port model, in which each port may feed multiple execution units, creates a peculiar bottleneck when a code comprises a mixture of different instruction forms that must go through the same port. Even though ample execution resources are available, the performance may be impeded by the limit of one instruction per cycle and port. This "port conflict" can be measured: By adding another instruction form into the already throughput bound benchmark, either an increase or no change in runtime is expected. If the runtime increased, both instruction forms utilize at least one common port, which needs to be considered when mapping instruction forms to ports. This method is currently used to validate known information, but can be extended to derive a complete, previously unknown, port model.

C. Example: Fused Multiply-Add on Skylake and Zen

To illustrate our model construction method, we carry out the analysis of the instruction form `vfmadd132pd m128,xmm2,xmm1` (i.e., multiplying a packed double-precision value from memory and `xmm1`, adding this to `xmm2` and storing the result in `xmm1`) for the latest Intel and AMD architectures.

We use the port model for Skylake, shown in Figure 2, and Zen, as presented in Figure 3. The benchmark files for latency and throughput are generated automatically as shown in the previous section. E.g., the basic repetitive instruction form for the latency measurement is `vfmadd132pd (%rax), %xmm0, %xmm0`. All these instruction forms must be executed separately due to the read-after-write hazard between the current target register and the future source register xmm0. The throughput benchmark is generated analogously with independent registers as operands.

Based on these files, we configure and run benchmarks for various levels of parallelism. On AMD Zen the output will look like this (note that we are using Intel operand ordering here since ibench works with Intel assembly syntax internally):

```
    Using frequency 1.80GHz.
2   vfmadd132pd-xmm_xmm_mem-1:          5.011 (clk cy)
    vfmadd132pd-xmm_xmm_mem-2:          2.506 (clk cy)
4   vfmadd132pd-xmm_xmm_mem-4:          1.251 (clk cy)
    vfmadd132pd-xmm_xmm_mem-5:          1.003 (clk cy)
6   vfmadd132pd-xmm_xmm_mem-8:          0.679 (clk cy)
    vfmadd132pd-xmm_xmm_mem-10:         0.503 (clk cy)
8   vfmadd132pd-xmm_xmm_mem-12:         0.502 (clk cy)
    vfmadd132pd-xmm_xmm_mem-TP:         0.500 (clk cy)
10  vfmadd132pd-xmm_xmm_xmm-TP:         0.502 (clk cy)
```

The number behind every instruction form is the amount of independent parallel instructions in one loop iteration given the dependencies in every benchmark. "TP" marks throughput benchmarks, without dependencies. On line 2, we can see that the latency of this instruction form is 5 cy. The reciprocal throughput shown on line 9 is 0.5 cy/instr. The measured throughput is unaffected for benchmarks with ten or more independent instruction forms, which corroborates our general assumptions about multi-port code execution: The instruction form can be spread among two separate ports, because its throughput is one half and we expect each port to handle one instruction per cycle. Given that Zen can do two loads per cycle and the instruction form without a memory operand has a reciprocal throughput of 0.5 cy/instr. as well, we need to find which floating point ports (0, 1, 2 or 3) are needed for fused-multiply-add (FMA). We therefore create benchmarks with `vmulpd %xmm1,%xmm2,%xmm3` and `vaddpd %xmm1,%xmm2,%xmm3` instruction forms interleaved with the prior `vfmadd132pd`. At this point, we already know that `vmulpd` is executed on floating point port 0 or 1, `vaddpd` goes to port 2 or 3 and both instruction forms have a reciprocal throughput of 0.5 cy/instr. The chosen operands must be independent of the target register to prevent hazards and therefore affect dependencies. The result is the following:

```
Using frequency 1.80GHz.
vfmadd132pd-xmm_xmm_mem-TP-vaddpd:    0.522 (clk cy)
vfmadd132pd-xmm_xmm_mem-TP-vmulpd:    1.024 (clk cy)
```

From the combined measurement we see that `vmulpd` – unlike `vaddpd` – can not be hidden behind the execution of `vfmadd132pd`, so `vfmadd132pd` must be scheduled to the same ports as `vmulpd`, i.e., 0 or 1. To add the instruction form to the Zen port model of OSACA, we create a new entry with a reciprocal throughput of 0.5 cy/instr. on port 0, 1, 8 and 9 to the database:

```
vfmadd132pd-xmm_xmm_mem, 0.5, 5.0, \
                  "(0.5,0.5,0,0,0,0,0,0,0.5,0.5)"
```

Note that for floating point division we assume that there is an additional divider pipe on port 3, which is included in the port occupation notation of the database. For doing the same workflow on Skylake, we can reuse all priorly created benchmark codes and get the following results:

```
    Using frequency 1.80GHz.
2   vfmadd132pd-xmm_xmm_mem-1:          4.009 (clk cy)
    vfmadd132pd-xmm_xmm_mem-2:          2.006 (clk cy)
4   vfmadd132pd-xmm_xmm_mem-4:          1.011 (clk cy)
    vfmadd132pd-xmm_xmm_mem-5:          0.805 (clk cy)
6   vfmadd132pd-xmm_xmm_mem-8:          0.556 (clk cy)
    vfmadd132pd-xmm_xmm_mem-10:         0.554 (clk cy)
8   vfmadd132pd-xmm_xmm_mem-12:         0.551 (clk cy)
    vfmadd132pd-xmm_xmm_mem-TP:         0.553 (clk cy)
10  vfmadd132pd-xmm_xmm_xmm-TP:         0.502 (clk cy)
    vfmadd132pd-xmm_xmm_mem-TP-vaddpd: 1.010 (clk cy)
12  vfmadd132pd-xmm_xmm_mem-TP-vmulpd: 1.004 (clk cy)
```

Here we get the same expected amount of cycles for the instruction form in combination with `vaddpd` and `vmulpd` because both functional units are assigned to port 0 and 1, increasing the overall throughput to 1 cy. This leads to the assumption of a latency of 4 cy and a reciprocal throughput of 0.5 cy/instr. The port distribution is 0, 1 for FMA, and 2, 3 for Load. This is represented in the database in the following way:

```
vfmadd132pd-xmm_xmm_mem, 0.5, 4.0, \
                  "(0.5,0,0.5,0.5,0.5,0,0,0,0)"
```

Note that – similar to Zen – the Skylake architecture has an additional divider pipe on port 0.

Doing this for every instruction will give a validated port model, which follows the general structure as seen in Figure 2 and Figure 3.

III. Static Analyzer Implementation

After collecting the performance and scheduling information about specific instruction forms for a given architecture, as done in Section II, OSACA can use it to predict the throughput of kernels.

OSACA extracts a marked kernel section out of an assembly or object file. For convenience OSACA supports the same byte markers as IACA, i.e.:

```
movl $111, %ebx
.byte 100,103,144
    # ..LABEL:
        # Some code
        # ...
        # conditional jump to ..LABEL
movl $222, %ebx
.byte 100,103,144
```

These markers can be inserted in the source code, but we have found that this strongly influences the code generated by the compiler. We therefore recommend to insert the marker instructions directly into assembly code, to guarantee preservation of the original instructions.

Extracting the inner kernel is done using regular expressions. IACA's analysis is based on compiled binary object files, which is an unnecessary step with OSACA. Each instruction form is analyzed regarding its operands and matched to entries in the database. If no match was found, corresponding benchmark files, as described in Section II-A, are generated automatically. If every instruction form was found, OSACA performs a throughput analysis based on earlier measured

Compiled for	Flag	unroll factor	OSACA pred. [cy] Zen	OSACA pred. [cy] SKL	IACA pred. [cy] SKL
Skylake	-O1	1	2.00	2.00	2.24
Skylake	-O2	1	2.00	2.00	2.00
Skylake	-O3	4	4.00	2.00	2.21
Zen	-O1	1	2.00	2.00	2.24
Zen	-O2	1	2.00	2.00	2.00
Zen	-O3	2	2.00	2.00	2.21

TABLE I: OSACA and IACA throughput analyses for the Schönauer triad kernel. Note that the cycle counts pertain to one assembly loop iteration, which may comprise several source code iterations (according to the unroll factor).

P0	P1	P2	P3	P4	P5	P6	P7	Assembly Instructions
								.L10:
		0.50	0.50					vmovapd (%r15,%rax), %ymm0
		0.50	0.50					vmovapd (%r12,%rax), %ymm3
0.25	0.25				0.25	0.25		addl $1, %ecx
0.50	0.50	0.50	0.50					vfmadd132pd 0(%r13,%rax),%ymm3,%ymm0
		0.50	0.50	1.00				vmovapd %ymm0, (%r14,%rax)
0.25	0.25				0.25	0.25		addq $32, %rax
0.25	0.25				0.25	0.25		cmpl %ecx, %r10d
								ja .L10
1.25	1.25	**2.00**	**2.00**	1.00	0.75	0.75	0.00	

TABLE II: OSACA prediction (shortened) of -O3 Schönauer triad benchmark for Skylake with code compiled for Skylake. See Section I-C for system configuration.

data and the port distribution from its database. The workflow of OSACA is depicted in Figure 4. For validation we will use different assembly representations, which are generated by the GNU C Compiler with different optimization levels: -O1, -O2 or -O3. The predictions by OSACA are validated by comparing predicted runtime to measured execution time on the systems described in Section I-C. In case of Skylake we also compare OSACA and IACA predictions, which is impossible for Zen due to the proprietary nature of IACA.

A. Example: Triad on Skylake and Zen

A typical benchmark for measuring data throughput in combination with floating-point operations is the "Schönauer" triad benchmark [22]:

```
for(int j=0; j<size; ++j)
        a[j] = b[j] + c[j]*d[j];
```

First, we analyze the kernel compiled with Skylake-specific optimization flags on both architectures. Later, we will do the same analysis on both architectures with code compiled for Zen. The resulting maximum measured number of floating operations (FLOP) per second, the maximum number of high-level, i.e., source code loop iterations (it) per second, and the minimum number of cycles per iteration are stated in columns 5–7 of Table III.[2]

The FLOP/s metric is calculated from the total runtime and total number of FLOPs: $2\frac{\text{FLOP}}{\text{iteration}} \times \texttt{size} \times \text{repetitions}/\text{runtime}$. It/s is calculated from $\texttt{size} \times \text{repetitions}/\text{runtime}$. Finally, the number of cycles (cy/it) is calculated by dividing the clock speed (cy/s) by the performance (it/s).

The compiler unrolls the kernel four times at -O3 for AVX SIMD vectorization (see Figure 4). Unrolling must be observed when interpreting IACA or OSACA predictions, as they disregard the source code and only predict for assembly-level iterations. E.g., if a loop was unrolled twice, the prediction by IACA and OSACA will be for two original iterations instead of one. This also applies to any additional unrolling on top of SIMD. In this paper, OSACA and IACA predictions given in cycles are for one assembly code iteration, whereas the unit "cy/it" always refers to source code iterations.

All predictions by OSACA and IACA for "Skylake-optimized" code can be found in the first three rows of Table I. OSACA's throughput analysis via `osaca --arch skl --iaca asmfile.s`, i.e., for Skylake, predicts 2 cycles independent of the optimization level. As mentioned above, OSACA predicts the throughput for one iteration of the marked kernel code, which corresponds to one iteration in case of the -O1 and -O2 code and four iterations in case of -O3. The OSACA prediction for the -O3 code is shown in somewhat condensed form in Table II. Our measurement for the -O3 code is 0.53 cy/it (see last row of Table III), which matches both the OSACA and IACA predictions well since $4\,\text{it} \cdot 0.53\,\text{cy/it} = 2.12\,\text{cy}$.

Unlike OSACA, IACA does not schedule instruction forms with an average probability but weighs specific ports. The reason for this is not disclosed and may be based on internal information. However, this does not affect the overall throughput and bottleneck prediction for the triad benchmark. For the benchmark versions compiled with -O1, -O2 and -O3, IACA predicts between 2.00 cy/it and 2.24 cy/it for each kernel, but all with a pure port binding of 2.0 cy in the bottleneck. Running this code on Zen results in the same runtime as on Skylake for the -O1 and -O2 versions, but shows worse performance with -O3 (see rows 7–9 in Table III). OSACA's throughput prediction for this version can be found in the structural design of Figure 4. The lower performance is due to the Zen architecture executing AVX instructions as two successive 128-bit chunks. This leads to an expected total runtime of 4 cycles per (assembly) iteration instead of Skylake's 2 (i.e., 1 cy/it instead of 0.5), which is confirmed by the measurements in column 7 of Table III.

The performance results for the triad benchmark compiled for the Zen architecture are shown in the first six rows of Table III. While we can observe similar behavior for the -O1 and -O2 versions compared to the previous example, the com-

[2]The relation between FLOPs and iterations is trivial in this example; for more complicated codes it is often useful to think in terms of iterations instead of FLOPs, so we keep both metrics.

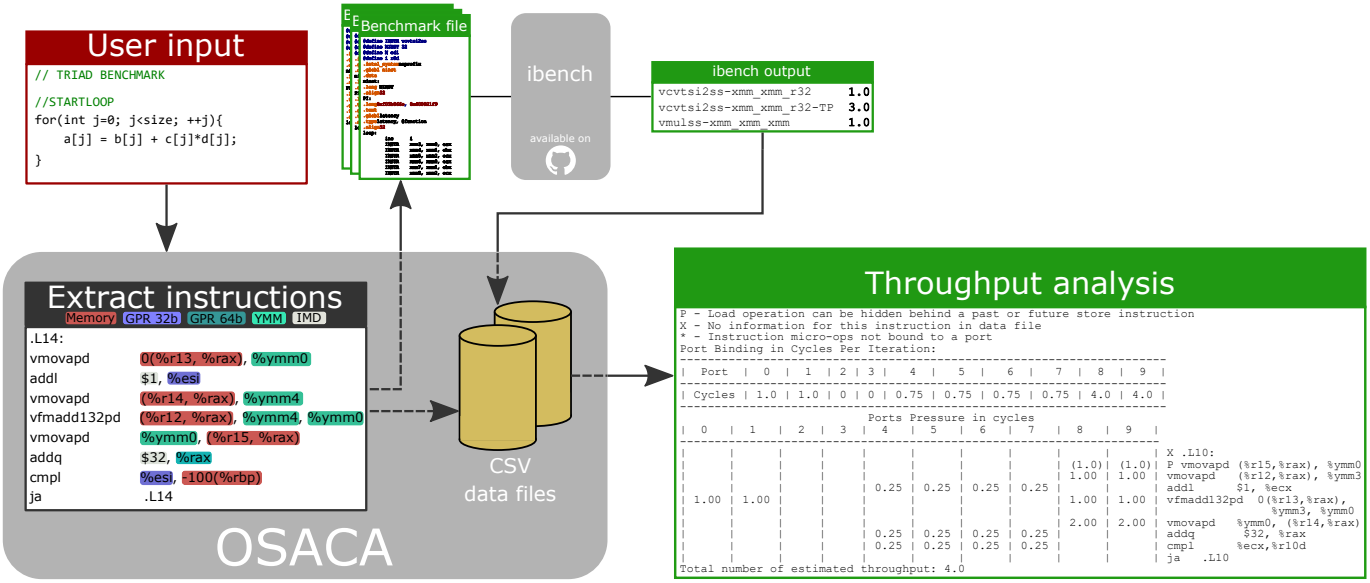

Fig. 4: Structural design of OSACA and its workflow. The code example shown here is the Schönauer triad benchmark compiled with `-O3` and analyzed with OSACA assuming the Zen architecture.

Architecture		Optimization	Unroll	Measured			Prediction [cy/it]	
executed on	compiled for	flag	factor	MFLOP/s	Mit/s	cy/it	OSACA	IACA
Zen	Zen	-O1	1x	1797	898	2.00	2.00	–
Zen	Zen	-O2	1x	1797	898	2.00	2.00	–
Zen	Zen	-O3	2x	3531	1754	1.02	2.00/2	–
Skylake	Zen	-O1	1x	1770	885	2.03	2.00	2.24
Skylake	Zen	-O2	1x	1768	884	2.04	2.00	2.00
Skylake	Zen	-O3	2x	3505	1753	1.03	2.00/2	2.21/2
Zen	Skylake	-O1	1x	1792	896	2.01	2.00	–
Zen	Skylake	-O2	1x	1797	898	2.01	2.00	–
Zen	Skylake	-O3	4x	3166	1589	1.01	4.00/4	–
Skylake	Skylake	-O1	1x	1767	884	2.04	2.00	2.24
Skylake	Skylake	-O2	1x	1776	888	2.03	2.00	2.00
Skylake	Skylake	-O3	4x	6808	2738	0.53	2.00/4	2.21/4

TABLE III: Measurements of the Schönauer triad benchmark compiled for Intel Skylake and AMD Zen together with the corresponding predictions by OSACA and Intel IACA.

piler only unrolls twice for the `-O3` version, i.e., it only uses 128-bit wide registers. For all six versions of the benchmark OSACA predicts 2 cy per assembly iteration, which matches the measured performance. Since both architectures have the same throughput limits for 128-bit wide data movement we do not see a performance difference between Zen and Skylake.

The OSACA output for the `-O3` version compiled for Zen can be found in Table IV. Although Zen has two load units and one store unit on ports 8 and 9, it has only two AGUs on the very same ports, so it is only capable of executing either up to two loads or one load and one store per cycle. OSACA models this by hiding one load instruction behind a given store instruction, as seen on the second row in Table IV (`vmovaps 0(%r13,%rax),%xmm0`).

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	Assembly Instructions
										.L10:
0.25	0.25	0.25	0.25					(0.5)	(0.5)	vmovaps 0(%r13,%rax),%xmm0
0.25	0.25	0.25	0.25					0.50	0.50	vmovaps %r15,%rax),%xmm3
				0.25	0.25	0.25	0.25			incl %esi
0.50	0.50							0.50	0.50	vfmadd132pd (%r14,%rax),%xmm3,%xmm0
0.25	0.25	0.25	0.25					1.00	1.00	vmovaps %xmm0,(%r12,%rax)
				0.25	0.25	0.25	0.25			addq $16,%rax
				0.25	0.25	0.25	0.25			cmpl %esi, %ebx
										ja .L10
1.25	1.25	0.75	0.75	0.75	0.75	0.75	0.75	**2.0**	**2.0**	

TABLE IV: OSACA prediction of `-O3` Schönauer triad benchmark for Zen with code compiled for Zen. Parentheses indicate a hide-able load μ-op.

Arch.	Opt.	IACA	OSACA	Measurement
Skylake	-O1	3.91 cy/it	4.75 cy/it	9.02 cy/it
Skylake	-O2	4.00 cy/it	4.25 cy/it	4.00 cy/it
Skylake	-O3	2.00 cy/it	2.00 cy/it	2.06 cy/it
Zen	-O1		4.00 cy/it	11.48 cy/it
Zen	-O2		4.00 cy/it	4.96 cy/it
Zen	-O3		2.00 cy/it	2.44 cy/it

TABLE V: Predictions and measurements of π benchmark on Skylake and Zen.

B. Example: π Benchmark on Skylake and Zen

The Schönauer triad benchmark is bound by load throughput. In the following we will look at an arithmetic instruction bound benchmark that calculates $\pi = \int_0^1 4/(1+x^2)\,\mathrm{dx}$ by simple rectangular integration:

```
int SLICES = 1000000000;
double sum = 0., delta_x = 1./SLICES;
for(int i=0; i<SLICES; ++i) {
   double x = (i+0.5)*delta_x;
   sum = sum + 4.0 / ( 1.0 + x * x);
}
double Pi = sum * delta_x;
```

As in the previous example, we compiled the benchmark code with -O1, -O2 and -O3 for Intel Skylake and AMD Zen with the flags described in Section I-C, but we only analyze and run on the architecture we compile for. In order to convince GCC to vectorize this code with -O3, the flag -funsafe-math-optimizations was required. In Table V we have compiled IACA and OSACA predictions, as well as the measured reciprocal throughput.

Predictions for -O1 failed to describe the measured runtime by more than a factor of two on both Skylake and Zen. Manual inspection of the code confirmed the validity of the predictions under the model assumptions. To investigate the discrepancy we checked the UOPS_EXECUTED_STALL_CYCLES hardware event using likwid-perfctr on Intel Skylake, and found that almost 17 times as many stall cycles were counted with -O1 compared to -O2. Measuring the average stall duration (part of likwid's UOPS_ISSUE group) also yields 5.5 cy, which is roughly the discrepancy we measured. Looking into the code again, the relevant difference is that at -O1, the value of sum is read from the stack, updated, and written back in every iteration:

```
.L2:
  vxorpd        %xmm0, %xmm0, %xmm0
  vcvtsi2sd     %eax, %xmm0, %xmm0
  vaddsd        %xmm4, %xmm0, %xmm0
  vmulsd        %xmm3, %xmm0, %xmm0
  vmulsd        %xmm0, %xmm0, %xmm0
  vaddsd        %xmm2, %xmm0, %xmm0
  vdivsd        %xmm0, %xmm1, %xmm0
  vaddsd        (%rsp), %xmm0, %xmm5
  vmovsd        %xmm5, (%rsp)
  addl  $1, %eax
  cmpl  $1000000000, %eax
  jne .L2
```

P0	– DV	P1	P2	P3	P4	P5	P6	P7	Assembly Instructions
									X .L2:
						1.00			vextracti128 $0x1, %ymm2, %xmm1
1.00						1.00			vcvtdq2pd %xmm2, %ymm0
0.50		0.50							vaddpd %ymm7, %ymm0, %ymm0
0.25		0.25				0.25	0.25		addl $1, %eax
1.00						1.00			vcvtdq2pd %xmm1, %ymm1
0.50		0.50							vaddpd %ymm7, %ymm1, %ymm1
0.33		0.33				0.33			vpaddd %ymm8, %ymm2, %ymm2
0.50		0.50							vmulpd %ymm6, %ymm0, %ymm0
0.50		0.50							vmulpd %ymm6, %ymm1, %ymm1
0.50		0.50							vfmadd132pd %ymm0, %ymm5, %ymm0
0.50		0.50							vfmadd132pd %ymm1, %ymm5, %ymm1
1.00	8.00								vdivpd %ymm0, %ymm4, %ymm0
1.00	8.00								vdivpd %ymm1, %ymm4, %ymm1
0.50		0.50							vaddpd %ymm1, %ymm0, %ymm0
0.50		0.50							vaddpd %ymm0, %ymm3, %ymm3
0.25		0.25				0.25	0.25		cmpl $125000000, %eax
									jne .L2
8.83	**16.0**	4.83	0.00	0.00	0.00	3.83	0.50	0.00	

TABLE VI: OSACA prediction of -O3 π–benchmark for Skylake, compiled for Skylake.

It is kept in a register with -O2 and only written back after the loop.[3] We therefore conclude that on Skylake the write-after-read dependency invalidates the full throughput assumption because of problems with the out-of-order scheduler or speculative execution. On AMD Zen the DYN_TOKENS_DISP_STALL_CYCLES_RETIRE_TOKEN_STALL hardware event points into the same direction, increasing to 7× to that of -O2, and thus we suspect that the same problem exists there as well.

For -O2 and -O3, predictions and measurements match rather well, in particular on Intel Skylake. The throughput analysis for the π benchmark compiled for Skylake with -O3 and predicted for execution on Skylake can be found in Table VI. The compiler unrolled the kernel eight times, so that OSACA reports the model for eight iterations of the loop. Note that for a precise prediction it is necessary to take a realistic execution of division μ-ops into account. Both Skylake and Zen use a different pipeline for their divisions. Therefore, the "main" port is allocated only during one cycle of the execution, while the remaining cycles leave the port free for other instructions. OSACA supports division pipelines and marks them in the output as "DV".

The fact that OSACA models the instruction throughput in average port occupation, as mentioned before in Section I-A, leads to a prediction of 4.25 cy instead of 4 cy for the execution of the benchmark compiled with -O2 on Skylake (see Tables V and VII). According to IACA, μ-ops for instructions such as vxorpd or cmp are not bound to a port, indicating that they take "shortcuts" through the architecture, avoiding port contention. This knowledge is (still) lacking in OSACA, which leads to the OSACA in-core throughput model not being

[3]All assembly kernels and the corresponding IACA and OSACA analysis can be found in the artifacts repository [7].

P0	– DV	P1	P2	P3	P4	P5	P6	P7	Assembly Instructions
									`X .L2:`
0.25		0.25				0.25	0.25		`vxorpd %xmm0, %xmm0, %xmm0`
0.50		0.50				1.00			`vcvtsi2sd %eax, %xmm0, %xmm0`
0.25		0.25				0.25	0.25		`addl $1, %eax`
0.50		0.50							`vaddsd %xmm5, %xmm0, %xmm0`
0.50		0.50							`vmulsd %xmm3, %xmm0, %xmm0`
0.50		0.50							`vfmadd132sd %xmm0, %xmm4, %xmm0`
1.00	4.00								`vdivsd %xmm0, %xmm2, %xmm0`
0.50		0.50							`vaddsd %xmm0, %xmm1, %xmm1`
0.25		0.25				0.25	0.25		`cmpl $1000000000, %eax`
									`jne .L2`
4.25	4.00	3.25	0.00	0.00	0.00	1.75	0.75	0.00	

TABLE VII: OSACA prediction of `-O2` π–benchmark for Skylake, compiled for Skylake.

a strictly lower bound for the execution time in all cases. This error is generally small, however.

In all other `-O2` and `-O3` predictions the division pipeline on port 0 for Skylake (and port 3 for Zen, respectively) is the main throughput bottleneck of the code. With AMD Zen, the execution is about 20% slower than the prediction. Just as on Skylake, we can observe the bottleneck in the division pipeline for both the `-O2` and `-O3` version.

IV. Conclusion

A. Summary

Using our Open-Source Architecture Code Analyzer (OSACA) we have shown that a partially automatic machine model construction and fully automatic throughput analysis of loop kernels based on benchmarking and known hardware features is possible and yields accurate results. Benchmarks are necessary to build a port model and gather throughput and latency numbers of specific instruction forms. This approach yields deep insight into the in-core performance limitations of a core micro-architecture. We verified our model, performance data and predictions on Intel Skylake and AMD Zen CPU architectures using two kernels that show different bottlenecks, and compared it with measured runtimes as well as predictions from Intel's Architecture Code Analyzer (IACA).

OSACA can extract loop kernels and analyze their instruction forms out of marked assembly code. Using techniques shown in this work, one can refine the port models and create realistic best-case throughput predictions for in-core execution. OSACA is intended as an alternative to IACA, with the ability to go beyond Intel hardware.

B. Future Work

OSACA in its current state is a first draft of what we envision for the future. We intend to extend it with various new core features, the most relevant one being latency modeling (which has been dropped by IACA some years ago). This requires support for critical path analysis, tracking dependencies between sources and destinations as well as a model for output forwarding. Differentiation between memory addressing modes is already part of the design of OSACA, but not completely implemented. This is crucial for modeling the peculiar AGU behind port 7 on Haswell and beyond, and more generally for any architecture where different addressing modes can have varying performance impact.

Once there is a solid model of all the disclosed and known features, we will also start including less well understood behavior, such as heuristics of the out-of-order scheduler and "shortcuts" that bypass the port scheduler. This will involve very detailed and precise benchmarking to identify the underlying rules. Furthermore, we intend to add information about the critical path and loop carried dependencies to enhance the lower bound model in a realistic way.

Support for non-x86 architectures is on the horizon but will require a lot of manual model building and validation, since they are not as well understood from a performance perspective. In order to support these efforts, we are pursuing an automatic deduction approach, which is in early development.

A framework for easier and more flexible generation of benchmarks is currently in development. [23].

References

[1] H. T. Kung, "Memory Requirements for Balanced Computer Architectures," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 49–54, doi: 10.1145/17356.17362.

[2] H. Stengel, J. Treibig, G. Hager, and G. Wellein, "Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model," in *Proceedings of the 29th ACM International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 207–216, doi: 10.1145/2751205.2751240.

[3] Y. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, ser. Lecture Notes in Computer Science, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds., vol. 8966. Springer International Publishing, 2015, pp. 129–148, doi: 10.1007/978-3-319-17248-4_7.

[4] J. Hammer, J. Eitzinger, G. Hager, and G. Wellein, "Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels," in *Tools for High Performance Computing 2016*, C. Niethammer, J. Gracia, T. Hilbrich, A. Knüpfer, M. M. Resch, and W. E. Nagel, Eds. Cham: Springer International Publishing, 2017, pp. 1–22, doi: 10.1007/978-3-319-56702-0_1.

[5] (2017, 11) Intel Architecture Code Analyzer. [Online]. Available: https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

[6] J. Laukemann. (2017, 12) OSACA – Open Source Architecture Code Analyzer. [Online]. Available: https://github.com/RRZE-HPC/OSACA

[7] "Artifact description: Automated instruction stream throughput prediction for intel and amd microarchitectures." [Online]. Available: https://github.com/RRZE-HPC/pmbs2018-paper-artifact/

[8] Intel 64 and IA-32 Architectures Optimization Reference Manual. [Online]. Available: https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual

[9] M. Clark. A New X86 Core Architecture for the Next Generation of Computing. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.930-X86-core-MikeClark-AMD-final_v2-28.pdf

[10] J. Diamond, M. Burtscher, J. D. McCalpin, B. Kim, S. W. Keckler, and J. C. Browne, "Evaluation and optimization of multicore performance bottlenecks in supercomputing applications," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011, pp. 32–43.

[11] (2018, 4) Instruction tables. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf

[12] (2017, 8) Software Optimization Guide for AMD Family 17h Processors. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf

[13] D. Andric. [RFC] llvm-mca: a static performance analysis tool. [Online]. Available: http://llvm.1065342.n5.nabble.com/llvm-dev-RFC-llvm-mca-a-static-performance-analysis-tool-td117477.html

[14] llvm-exegesis – LLVM Machine Instruction Benchmark. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-exegesis.html

[15] C. Mendis, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks," *ArXiv e-prints*, Aug. 2018, arXiv:1808.07412 [cs.DC].

[16] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, and et al., "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, p. 1, 8 2011, doi: 10.1145/2024716.2024718. [Online]. Available: http://dx.doi.org/10.1145/2024716.2024718

[17] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, 2013. [Online]. Available: http://dx.doi.org/10.1145/2485922.2485963

[18] A. Patel, F. Afram, and K. Ghose, "Marss-x86: A qemu-based microarchitectural and systems simulator for x86 multicore processors," in *1st International Qemu Users' Forum*, 2011, pp. 29–30.

[19] 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)*, Dec 2014, pp. 1–10.

[20] J. Laukemann, "Design and Implemention of a Framework for Predicting Instruction Throughput," Bachelor's Thesis, 2017. [Online]. Available: https://hpc.fau.de/files/2018/08/Laukemann_Jan_Design_and_Implementation_For_a_Framework_Predicting_Instruction_Throughput.pdf

[21] J. Hofmann. (2018, 1) ibench – Measure Instruction Latency and Throughput. [Online]. Available: https://github.com/hofm/ibench

[22] W. Schönauer, *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers*. Self-edition, 2000. [Online]. Available: http://www.rz.uni-karlsruhe.de/~rx03/book

[23] J. Hammer, G. Hager, and G. Wellein, "OoO Instruction Benchmarking Framework on the Back of Dragons," SC18 SRC Poster (in review).

Artifact Description: Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures

- Jan Laukemann, Computer Science University of Erlangen-Nürnberg, jan.laukemann@fau.de
- Julian Hammer, RRZE University of Erlangen-Nürnberg, julian.hammer@fau.de, +49 9131 85 20101
- Johannes Hofmann, Chair of Computer Architecture, University of Erlangen-Nürnberg, johannes.hofmann@fau.de
- Georg Hager (advisor), RRZE University of Erlangen-Nürnberg, georg.hager@fau.de
- Gerhard Wellein (advisor), RRZE University of Erlangen-Nürnberg, gerhard.wellein@fau.de

A.1 Abstract

To model in-core performance behavior of throughput bound kernels, accurate prediction of scheduling and execution of instruction streams is unavoidable. Such in-core models are indispensable for analytical performance modeling, as done in the Roofline and ECM model, and allow a deep understanding of the performance-relevant interactions between hardware architecture and kernel code.

We present a static analysis tool for predicting sequential x86 instruction stream execution time under the assumption of an infinite first level cache and perfect on out-of-order scheduling. We show the process of building a model from available documentation and manual benchmarking, carried out for the latest Intel Skylake and AMD Zen micro-architectures.

To validate the gained model, we apply the methodology to several assembly kernels and compare runtime predictions with actual measurements on two distinct microarchitectures. In the end we outline the methodology to extend such models to new architectures.

A.2 Description

A.2.1 Check-list (artifact meta information)

- Compilation: gcc
- Binary: x86
- Hardware: Intel skylake, AMD zen
- Publicly available?: yes

A.2.2 How software can be obtained (if available)

Check out https://github.com/RRZE-HPC/OSACA

A.2.3 Hardware dependencies

We ran on an AMD EPYC 7451 (Zen architecture) at 1.8 GHz (fixed, turbo disabled) and Intel I7-6700HQ (Skylake SP architecture) at 2.4 GHz (fixed, turbo disabled). The results should be reproducible on any Zen and Skylake SP processors. Fixing the frequency and disabling turbo is vital for experiment reproduction.

A.2.4 Software dependencies

- Python >= 3.5, with the following packages installed: numpy, pandas, kerncraft
- likwid
- GNU GCC 7.2.0

On Ubuntu 17.10 install with:

```
apt install gcc-7 python3 python3-pip likwid
pip3 install numpy pandas kerncraft
```

A.2.5 Datasets

None necessary, everything is part of the code.

A.3 Installation

Please install OSACA using pip:

```
pip3 install osaca==0.2
```

A.4 Experiment workflow

To validate our results use the following commands.

Download script and benchmark codes:

```
git clone https://github.com/RRZE-HPC/pmbs2018-paper-artifact
cd pmbs18-paper-appendix/
```

Fix frequencies and disable turbo mode on CPU (for 1.8 GHz, or which ever frequency your CPU will be stable on):

`likwid-setFrequencies -t 0 -f 1.8`

Generate predictions (can be gained on any architecture) results with

`./run_predictions.sh`

Generate performance (must be done on AMD Zen and Intel Skylake SP machines) results with

`./run_measurements.sh`

A.5 Evaluation and expected result

Fixing the frequency and disabling turbo is very important to verify our results.

`./run_predictions.sh`

We expect these numbers to exactly match those in the paper. If your numbers deviate you will mostlikly have used a different compiler. Please compare the generated assembly of your compiler (found in `pi/*.s` and triad/*.s, respectively) with those we have generated for the paper (found in `orig/pi/*.s` and 'orig/triad/*.s).

The OSACA and IACA output of your measurements can be found in the `results` directory of the corresponding benchmark in the structure `osaca.[architecture].[optimization flag].out`.

Compare numbers to Table II, IV and V.

`./run_measurements.sh`

It outputs performance measurements in Time [s], MFlop/s and MLUP/s. MLUP/s can be easily translated to cy/it, as used in the paper: 1 / MLUP/s * Frequency. E.g., 1 / (362.6 MLUP/s) * 1.8 GHz = 4.96 cy/it.

We expect these numbers to lie within 10% of those in the paper, if run on the same micro architectures as mentioned. If your numbers are significantly faster, turbo mode or frequency scaling might be the reson. If they are slower, while running on a laptop or desktop machine, energy saving features may have interfered.

The measured results will be stored as `measurements.[benchmark].txt`.

Compare numbers to Table I, III and V.

A.6 Experiment customization

None

A.7 Notes

None