2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)

Automatic Throughput and Critical Path Analysis of x86 and ARM Assembly Kernels

Jan Laukemann, Julian Hammer, Georg Hager and Gerhard Wellein
{jan.laukemann, julian.hammer, georg.hager, gerhard.wellein}@fau.de
Erlangen Regional Computing Center
Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany

***Abstract*—Useful models of loop kernel runtimes on out-of-order architectures require an analysis of the in-core performance behavior of instructions and their dependencies. While an instruction throughput prediction sets a lower bound to the kernel runtime, the critical path defines an upper bound. Such predictions are an essential part of analytic (i.e., white-box) performance models like the Roofline and Execution-Cache-Memory (ECM) models. They enable a better understanding of the performance-relevant interactions between hardware architecture and loop code.**

The Open Source Architecture Code Analyzer (OSACA) is a static analysis tool for predicting the execution time of sequential loops. It previously supported only x86 (Intel and AMD) architectures and simple, optimistic full-throughput execution. We have heavily extended OSACA to support ARM instructions and critical path prediction including the detection of loop-carried dependencies, which turns it into a versatile cross-architecture modeling tool. We show runtime predictions 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 actual measurements.

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

I. Introduction

Analytic performance modeling of compute-intensive applications during development or optimization can be a powerful tool and sheds light on how code executes on modern CPU architectures. Such models are hence not only constructed for the sake of prediction but also to study relevant bottlenecks and to assess the compiler's ability to generate optimal code. However, they require a deep understanding of the underlying micro-architecture in order to yield accurate results. Common (simplified) approaches for numerical kernels are the Roofline [1] model or the ECM [2] model, whose construction is supported by the *Kerncraft* open-source performance modeling tool [3]. For Roofline, the Roofline Model Toolkit [4] and Intel's Roofline Advisor[1] are also available.

In general, there are two approaches to predict runtime and performance behavior: simulation and static analysis. Our work implements the latter. Even though simulators may be more thorough and accurate if comprehensive implementations exist, their usage is complicated by obstacles like finding steady states for throughput analysis and pinpointing the runtime-defining hardware bottleneck. In addition, their implementation is much more complex. The analysis and modeling process is split into in-core execution time and data transfer time through the memory hierarchy. See [2], [3] for examples on how this is done. For a long time, the only capable tool for static in-core code analysis was Intel's Architecture Code Analyzer (IACA) [5], which was employed by Kerncraft. Besides being at end-of-life, there are multiple limitations: Intel-only architectures, undisclosed model and later versions restricted to full-throughput assumption. To improve on this, we develop the *Open Source Architecture Code Analyzer* (OSACA) [6], which has, in addition to the features already known from IACA, extended x86 (Intel Cascade Lake and AMD) and AArch64 ARM support and supports critical path (CP) latency analysis and loop-carried dependency detection. All three predictions can be combined to a more accurate performance model, including the throughput as a lower bound and the critical path as an upper bound of the kernel runtime. Like IACA, OSACA assumes that all data originates from the first-level cache (i.e., L1 cache).

With OSACA's semi-automatic benchmarking pipeline, compilers can benefit from an automated model construction [3], [4]. The instruction database is dynamically extendable, which enables users to adapt the tool to other application scenarios beyond numerical kernels found in HPC usecases.

This paper is organized as follows: In Section I-A, we cover related work. Section II details the model assumptions and construction for the underlying architecture and the general methodology of the throughput and critical path analysis as well as the loop-carried dependency detection. In Section III we describe the benchmarking hardware/software environment and validate the methodology against actual measurements and compare with related tools. Section IV summarizes the work and gives an outlook to future developments.

The OSACA software is available for download under AGPLv3 [7]. Information about how to reproduce the results in this paper can be found in the artifact description [8].

A. Related Work

OSACA was inspired by IACA, the Intel Architecture Code Analyzer [5]. Developed by Israel Hirsh and Gideon S. [sic], Intel released the tool in 2012 and announced its end-of-life in April 2019. Therefore, no feature enhancements or new microarchitecture support can be expected. It is closed

This work was in part funded by the BMBF project METACCA.

[1]https://software.intel.com/en-us/advisor-user-guide-roofline-analysis

source and the underlying model has neither been published by the authors, nor peer reviewed. The latest version supports throughput analysis on Intel micro-architectures up to Skylake (including AVX-512), but is not capable of critical path analysis or loop-carried dependency detection.

LLVM Machine Code Analyzer (llvm-mca) [9] is a performance analysis tool based on LLVM's existing scheduling models. Currently it lacks support for HPC-relevant ARM architectures such as the ThunderX2, and some scheduling models need refinement. Also, llvm-mca cannot analyze CPs, even though one can manually identify a CP by the provided latency analysis. LLVM Machine Instruction Benchmark (llvm-exegesis) [10] is a micro-benchmarking framework for measuring throughput and latency of instruction forms. It could thus be used as a data source to feed the OSACA instruction database. Mendis et al. [11] apply a deep neural network approach to estimate block throughput on Intel x86 architectures from Ivy Bridge to Skylake. It is able to use IACA byte markers for indicating the code block to analyze and is currently not capable of detecting CPs or loop-carried dependencies. Code Quality Analyzer (CQA) [12] is a static performance analysis tool focused on single-core performance of loop-centric x86 code. Unlike OSACA, its goal is not to predict runtime, but rather give the developer a quality estimate of the code based on static binary analysis. Uop Flow Simulation (UFS) [13] extends CQA with a simulator for the out-of-order execution, modeling aspects OSACA assumes to be based on fixed (non-optimal) probabilities.

There are a fair number of simulators available: gem5, developed by Binkert et al. [14], ZSim by Sanches et al. [15] and MARSSx86 by Patel et al. [16]. While gem5 even supports various non-x86 instruction set architectures (ARM, Power, RISC-V among others), all of them are considered as "full-system" simulators, going above and beyond the scope of this work. Therefore, they give a coarse overview on complete (multi- or many-core) systems, rather than detailed insights pinpointing a bottleneck.

II. Methodology

When analyzing loop kernels, we assume for each CPU architecture a corresponding "port model": Each assembly instruction is (optionally) split into *micro-ops* (μ-ops), which get executed by multiple ports. A particular instruction may have multiple ports that can execute it (e.g., two integer ALUs), or – in case of complex instructions – multiple ports that *must* execute it (e.g., combined load and floating-point addition). Shared resources, such as a divider pipeline or a data load unit, are modelled as additional ports.

Each port receives at most one instruction per cycle and may be blocked by an instruction for any number of cycles. To model parallel execution of the same instruction form on multiple ports, the cycles may be spread among multiple ports, also allowing the inverse of integers as acceptable cycle throughput of an instruction per port, but always adding up to at least one cycle per instruction over all ports.

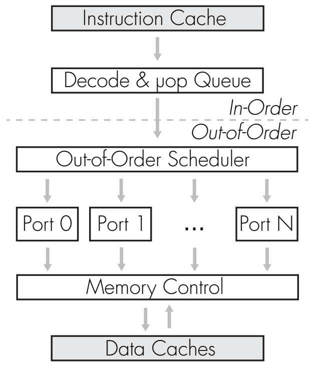


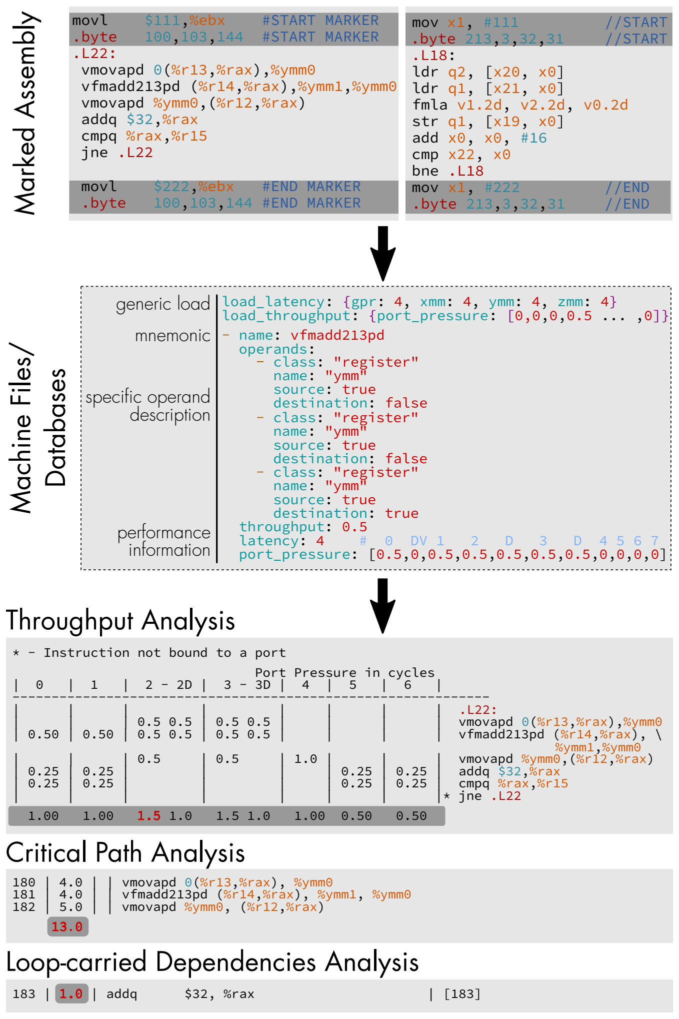
Fig. 1: Assumed generic out-of-order port model. Other shared resources (e.g., DIV pipeline) are modeled as additional ports.

Both x86 and ARM allow memory references to be used in combination with arithmetic instructions. This is modelled by splitting the instruction in the load and the arithmetic part, and accounting for their respective port pressures and dependencies separately (see below).

Figure 1 shows a diagram of the generic port model. Cascade Lake would be modeled with eight ports, plus one divider pipeline port and two data ports. A floating-point divide instruction would occupy port 0 for one cycle and the DIV port (i.e., pipeline) for four cycles, while an add instruction would use ports 0 and 1 for each half a cycle, because it may be executed on both.

We repeat here the assumptions behind our prediction model [6]:

- *All data accesses hit the first-level cache.* This is where the boundary between in-core and data transfer analysis is drawn. If a dataset fits in the first-level cache, no cache misses occur. Replacement strategies, prefetching, line buffering, etc., are insignificant on this level. Behavior beyond L1 can be modeled with Kerncraft [3], which relies on an in-core analysis from OSACA and combines it with data analysis to arrive at a unified model prediction.
- *Multiple available ports per instruction are utilized with fixed probabilities.* If the exact amount of μ-ops per port per instruction form 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 out of four possible ports and has a maximum throughput of 1 instr./cy on any unit will be assigned 0.25 cy on each of the four ports. This 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. The consideration of the full kernel for a more realistic port pressure model is currently not supported, but is taken



```
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
.byte 213,3,32,31         //START
.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
.byte 213,3,32,31         //END
```

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

```
* - Instruction not bound to a port

                                   Port Pressure in cycles
|  0   |  1   |  2 - 2D |  3 - 3D |  4  |  5   |  6   |
-------------------------------------------------------------------
|      |      |         |         |     |      |      |  .L22:
|      |      | 0.5 0.5 | 0.5 0.5 |     |      |      |  vmovapd 0(%r13,%rax),%ymm0
| 0.50 | 0.50 | 0.5 0.5 | 0.5 0.5 |     |      |      |  vfmadd213pd (%r14,%rax), \
                                                              %ymm1,%ymm0
|      |      | 0.5     | 0.5     | 1.0 |      |      |  vmovapd %ymm0,(%r12,%rax)
| 0.25 | 0.25 |         |         |     | 0.25 | 0.25 |  addq $32,%rax
| 0.25 | 0.25 |         |         |     | 0.25 | 0.25 |  cmpq %rax,%r15
|      |      |         |         |     |      |      |* jne .L22
  1.00   1.00   1.5 1.0   1.5 1.0   1.00  0.50   0.50
```

```
180 | 4.0 | | vmovapd 0(%r13,%rax), %ymm0
181 | 4.0 | | vfmadd213pd (%r14,%rax), %ymm1, %ymm0
182 | 5.0 | | vmovapd %ymm0, (%r12,%rax)
     13.0
```

```
183 | 1.0 | addq      $32, %rax                        | [183]
```

Fig. 2: Structural design of OSACA and its workflow, for STREAM triad (`A(:)=B(:)+s*C(:)`) loop.

into account for future versions.

A. Port Model Construction

The overall methodology of OSACA is exemplified using the STREAM triad `A(:)=B(:)+s*C(:)` loop in Figure 2. The x86 or AArch64 ARM assembly is parsed and the kernel in between the byte markers is extracted. For convenience, OSACA supports IACA's byte markers for x86 and uses the same instruction pattern for ARM assembly. For each parsed instruction form within the kernel, OSACA obtains the maximum inverse throughput and latency in cycles and the ports it can be scheduled to from its instruction database. Furthermore, it keeps track of source and destination operands for identifying register dependencies.

Possible sources for OSACA's database are microbenchmark databases like uops.info [17], Agner Fog's "Instruction Table" [18], or specific microbenchmarks using our own frameworks asmbench [19] and ibench [20]. For the latter, OSACA can automatically create benchmark files and import the output into its database, resulting in a semi-automatic benchmark pipeline.

B. Instruction Throughput and Latency Analysis

To obtain the latency and throughput of an instruction, we automatically create assembly benchmarks for use with ibench. It offers the infrastructure to initialize, run and accurately measure the desired parameters. It is also intended to support a python-based approach to micro-benchmarking, using the asmbench framework, which is not yet implemented at the time of writing.

Synthetic dependency chain generation within the assembly kernel allows measurement of throughput and latency of an instruction form and has been described in our previous work [6]. As stated in Section II-A, in addition to directly measuring throughput and latency of instruction forms including memory references in combination with register operands, which currently requires manual effort, OSACA is able to dynamically calculate the throughput by taking the maximum of both the load and arithmetic part and the latency by taking the sum of both parts. The throughput prediction assumes a fixed and balanced utilization of all suitable ports for any instruction form and perfect out-of-order scheduling without loop-carried dependencies. It thus yields a lower bound for execution time.

C. Critical Path Analysis

The critical path analysis is based on a directed acyclic graph (DAG) constructed from inter-instruction register dependencies following these rules:

1) A vertex is created for every instruction form in the marked piece of code.
2) From each instruction form's destination operands, edges are drawn to all instruction forms "further down" relying on these outputs, unless a break of dependency is found in between (e.g., by zeroing the register).
3) All edges are weighted with their source instruction's latency.
4) If a source memory reference has a dependency, an intermediate load-vertex is added along this edge and the additional edge weighted with the load latency.

After creating the DAG, the longest path within it is determined by using a weighted topological sort based on the approach of Manber [21]. The CP is thus an upper bound for the execution time of a single instance of the loop body.

D. Loop-Carried Dependency Detection

Dependencies in between iterations, i.e., loop-carried dependencies (LCDs), can drastically influence the runtime prediction of loop kernels: Even with sufficient out-of-order execution resources, overlap of successive iterations is only possible up to the limit set by the LCD. The actual runtime is thus limited from below by the length of the LCD chain. OSACA can detect LCDs by creating a DAG of a code comprising two back-to-back copies of the loop body. It can thus analyze paths from each vertex of the first kernel section and detect most cyclic LCDs if there exists a dependency chain observable by register dependencies from one instruction form to its corresponding duplicate in the next iteration.

III. Results

The CP and LCD detection described in Section II are included in OSACA's analysis of loop code and presented together with the "classic" throughput results. For validation we will use assembly representations generated by the Intel Fortran Compiler for x86 and the GNU Fortran Compiler for ARM, respectively. In case of CLX we also compare to the IACA and LLVM-MCA predictions for Skylake-X, which does not differ in terms of the port model. Due to the proprietary nature of IACA, we cannot use it on any AMD- or ARM-based system; hence, we compare against LLVM-MCA on AMD Zen. For lack of other tools, on TX2 OSACA's prediction can only be compared to measurements.

A. Example: Gauss-Seidel method on CSX, ZEN and TX2

An interesting floating-point benchmark for comparing predictions with the measured runtime is a 2D version of the "Gauss-Seidel" sweep [22]:

```
do it=1,itmax
  do k=1,kmax-1
    do i=1,imax-1
      phi(i,k,t0) = 0.25 * (
        phi(i,k-1,t0) + phi(i+1,k,t0) +
        phi(i,k+1,t0) + phi(i-1,k,t0))
    enddo
  enddo
enddo
```

It has one multiplication and three additions per iteration. As the update of the matrix happens in-place, each iteration is dependent on the previously calculated value of its "left" (`i-1`) and "bottom" (`k-1`) neighbor. This is the basic LCD that should govern the code's runtime; the CP may be longer since it may contain instructions that are not part of the LCD. If the hardware has sufficient out-of-order capabilities, it should be able to overlap that "extra" part across successive loop iterations. And finally, the pure throughput prediction (TP) should be much too optimistic since it ignores all dependencies.

Since we have demonstrated OSACA's TP analysis in previous work [6], we will focus here on the refinement of runtime predictions via CP and LCD analysis. The total runtime is measured and combined with the number of iterations to get lattice site updates per second [LUP/s] and cycles per iteration [cy/it] in columns 3–4 of Table I.

Unrolling by the compiler must be considered when interpreting OSACA predictions since they strictly pertain to the assembly level. E.g., if a loop was unrolled four times, as it is the case for our Gauss-Seidel examples, the prediction by OSACA will be for four original (high-level) iterations. This also applies to unrolling for SIMD vectorization, which is not possible here, however. In this paper, OSACA and IACA predictions in cycles are given for one assembly code iteration, whereas the unit "cy/it" always refers to high-level source code iterations. The total unrolling factor chosen by the compilers has been 4x for all architectures. Table II shows the condensed OSACA output for the TX2. Predictions by OSACA, IACA, and LLVM-MCA can be found in Table I.

The predicted block throughput of all three analysis tools is far from the measurements, as expected. Even though IACA is not capable of detecting CPs and analysing the latency of kernels anymore, its block throughput in the analysis report states 14 cy/it, contrary to the pure port binding of 2 cy/it. No explanation for this behavior can be found in the output although it matches exactly the LCD and the measurement.

Using the additional `-timeline` flag, LLVM-MCA provides a timeline view showing for a various number of cycles or iterations the expected cycle of dispatching, execution and retirement. Since it models register dependencies, we assume this to be its CP analysis and expect the time from the beginning of the first iteration to the retirement of its jump instruction to be the CP length, while all further executions have the length of the LCD. Both numbers can be found in the last column of Tab. I. While we can observe that LLVM-MCA overestimates the execution on ZEN by almost 50%, it predicts the runtime on CLX nearly exactly. For the ThunderX2, LLVM-MCA is neither capable of analyzing throughput nor latency at the time of writing.

OSACA provides a runtime bracket determined by the CP (upper bound) and the length of the longest cyclic LCD path (lower bound). The measured execution time should usually lie between these limits unless bottlenecks apply that are beyond our model (e.g., instruction cache misses, bank conflicts, etc.). As seen in column 6 of Table I, the actual measurement lies within the prediction frame in every analysis case, and the measurement is very close to the longest LCD path for this kernel. As expected, the runtime is faster than the pure CP length, since instructions that are not part of the LCD path can overlap across iterations.

The detailed OSACA analysis for ThunderX2 can be found in Table II. The LCD and CP columns show latency values for instruction forms along the CP and the longest cyclic LCD path, respectively. Fig. 3 depicts the graph generated by OSACA from the assembly.

Note that in cases where the LCD is very short or zero, the throughput prediction applies, and a deviation of the measurement from this lower limit points to either a shortage of OoO resources (physical registers, reorder buffer) or an architectural effect not covered by the machine model.

B. Validation Hardware, Software, and Runtime Environment

OSACA (version 0.3.1.dev0) was run with Python v3.6.8 and benchmarks were compiled using Intel ifort v19.0.2 and GNU Fortran (ARM-build-8) 8.2.0, respectively. All results presented were gathered on three machines, with fixed clock frequency and disabled turbo mode:

Architecture	Unroll	Measured		Prediction [cy/it]									
	factor			OSACA			IACA			LLVM-MCA			
		MLUP/s	cy/it	TP	LCD	CP	TP	LCD	CP	TP	LCD	CP	
Marvel ThunderX2	4x	118.9	18.50	2.46	18.00	25.00	—	—	—	—	—	—	
Intel Cascade Lake X	4x	178.3	14.02	2.19	14.00	18.00	14.00	—	—	2.00	14.75	19.00	
AMD Zen	4x	194.4	11.83	2.00	11.50	15.00	—	—	—	3.00	18.00	24.00	

TABLE I: Analysis and measurement of the Gauss-Seidel code on three architectures with OSACA, IACA, and LLVM-MCA predictions. Dashes denote unsupported analysis types or architectures. TP is the throughput prediction, a lower runtime bound. LCD is the loop carried dependency prediction, an expected runtime. CP is the critical path prediction, an upper runtime bound.

P0	P1	P2	P3	P4	P5	LCD	CP	LN	Assembly Instructions
								519	`.L20:`
			0.50	0.50			4.0	520	`ldr d31, [x15, x18, lsl 3]`
			0.50	0.50				521	`ldr d0, [x15, 8]`
0.50	0.50							522	`mov x14, x15`
0.33	0.33	0.33						523	`add x16, x15, 24`
			0.50	0.50				524	`ldr d2, [x15, x30, lsl 3]`
0.33	0.33	0.33						525	`add x15, x15, 32`
0.50	0.50						6.0	526	`fadd d1, d31, d0`
0.50	0.50					6.0	6.0	527	`fadd d3, d1, d30`
0.50	0.50					6.0	6.0	528	`fadd d4, d3, d2`
0.50	0.50					6.0	6.0	529	`fmul d5, d4, d9`
			0.50	0.50	1.00		4.0	530	`str d5, [x14], 8`
			0.50	0.50			4.0	531	`ldr d6, [x14, x18, lsl 3]`
			0.50	0.50				532	`ldr d16, [x14, 8]`
0.33	0.33	0.33						533	`add x13, x14, 8`
			0.50	0.50				534	`ldr d7, [x14, x30, lsl 3]`
0.50	0.50						6.0	535	`fadd d17, d6, d16`
0.50	0.50					6.0	6.0	536	`fadd d18, d17, d5`
0.50	0.50					6.0	6.0	537	`fadd d19, d18, d7`
0.50	0.50					6.0	6.0	538	`fmul d20, d19, d9`
			0.50	0.50	1.00			539	`str d20, [x15, -24]`
			0.50	0.50				540	`ldr d21, [x13, x18, lsl 3]`
			0.50	0.50				541	`ldr d23, [x14, 16]`
			0.50	0.50				542	`ldr d22, [x13, x30, lsl 3]`
0.50	0.50							543	`fadd d24, d21, d23`
0.50	0.50					6.0	6.0	544	`fadd d25, d24, d20`
0.50	0.50					6.0	6.0	545	`fadd d26, d25, d22`
0.50	0.50					6.0	6.0	546	`fmul d27, d26, d9`
			0.50	0.50	1.00			547	`str d27, [x14, 8]`
			0.50	0.50				548	`ldr d30, [x15]`
			0.50	0.50				549	`ldr d28, [x16, x18, lsl 3]`
			0.50	0.50				550	`ldr d29, [x16, x30, lsl 3]`
0.50	0.50							551	`fadd d31, d28, d30`
0.50	0.50					6.0	6.0	552	`fadd d2, d31, d27`
0.50	0.50					6.0	6.0	553	`fadd d0, d2, d29`
0.50	0.50					6.0	6.0	554	`fmul d30, d0, d9`
			0.50	0.50	1.00		4.0	555	`str d30, [x15, -8]`
0.33	0.33	0.33						556	`cmp x7, x15`
								557	`bne .L20`
9.83	9.83	1.33	8.00	8.00	4.00	72.0	100.0		sum (4x unrolled)
2.46	2.46	0.33	2.00	2.00	1.00	**18.0**	**25.0**		per high-level iteration

TABLE II: (Condensed) OSACA analysis of Gauss-Seidel assembly code for ARM-based ThunderX2 architecture. The LN column are line numbers.

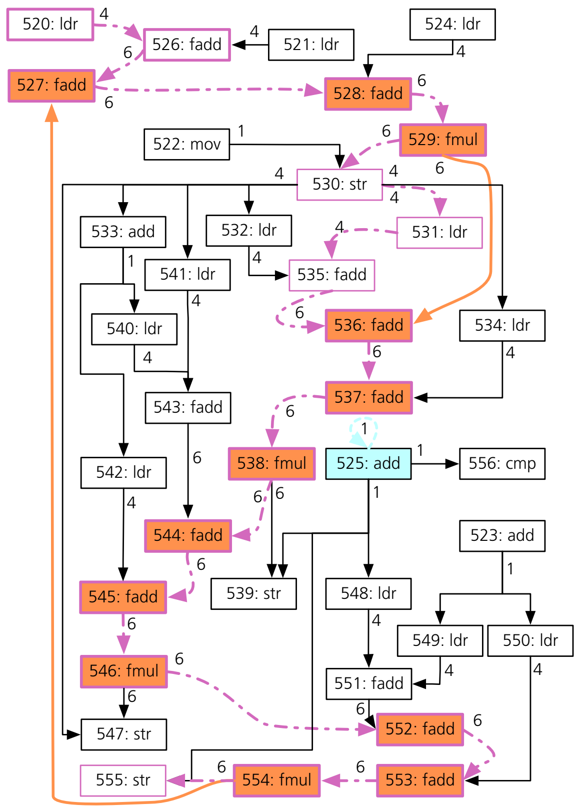


Fig. 3: (Compressed) dependency graph of the Gauss-Seidel code on TX2, created by OSACA. Orange nodes are on the longest LCD, including the backedge. Pink dashed lines and outlined nodes make up the CP. Numbers in nodes are line numbers, as found in Table II, and weights along the edges are latency cycles.

ThunderX2: ARM-based Marvell ThunderX2 9980 with ThunderX2 micro-architecture (formerly known as Cavium Vulcan) at 2.2 GHz (TX2), gfortran, options `-mcpu=thunderx2t99+simd+fp -fopenmp-simd -funroll-loops -Ofast`

Cascade Lake: Intel Xeon Gold 6248 with Cascade Lake X micro-architecture at 2.5 GHz (CLX), ifort, options `-funroll-loops -xCASCADELAKE -Ofast`

Zen: AMD EPYC 7451 with Zen micro-architecture at 2.3 GHz (ZEN), gfortran, options `-funroll-loops -mavx2 -mfma -Ofast`

The process was always bound to a physical core. In effect, statistical runtime variations were small enough to be ignored.

IV. Conclusion

A. Summary

We have shown that automatic extraction, throughput, and critical path analysis of assembly loop kernels is feasible using our cross-platform tool OSACA. OSACAs results are accurate and sometimes even more precise and versatile than predictions of comparable tools like IACA and LLVM-MCA. Additionally, direct critical path analysis including loop-carried dependencies is not supported by any other tool to date, although it can be inferred manually from LLVM-MCA's timeline information.

B. Future Work

In the future we intend to extend OSACA to support hidden dependencies, i.e., instructions accessing resources not named specifically in the assembly, such as status flags and load-after-store dependencies, including stack operations. The LCD analysis is not perfect and may miss dependencies in some special cases, which can be improved by taking more than two iterations into account. Furthermore, we plan to increase the number of micro-benchmark interfaces and to support the semi-automatic usage of asmbench in the OSACA toolchain. Beyond the even distribution of μ-ops across multiple ports, we want to implement a more realistic scheduling scheme that takes port utilization into account. Support for new micro-architectures like AMD's Zen 2 and eventually IBM's Power9 is also planned. Another topic is the overlap of latency in complex instructions, which can change the outcome of the analysis slightly but may be significant in pathological cases (e.g., in $a = a + b \times c$ with an FMA instruction, the multiplication may already execute before a becomes available). The split, as well as the fusion, of μ-ops is currently not considered, but can be achieved with replacement rules in the architecture model description.

Accurately modeling the performance characteristics of the decode, reorder buffer, register allocation/renaming, retirement and other stages, which all may limit the execution throughput and impose latency penalties, is currently out of scope for OSACA.

References

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[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] 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.

[4] 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.

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

[6] 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)*, Nov 2018, pp. 121–131.

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

[8] "Artifact description: Automatic throughput and critical path analysis of x86 and arm assembly kernels." [Online]. Available: https://github.com/RRZE-HPC/OSACA-CP-2019

[9] 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

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

[11] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, Jun 2019, pp. 4505–4515. [Online]. Available: http://proceedings.mlr.press/v97/mendis19a.html

[12] 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, doi: 10.1109/HiPC.2014.7116904.

[13] V. Palomares, D. C. Wong, D. J. Kuck, and W. Jalby, "Evaluating out-of-order engine limitations using uop flow simulation," in *Tools for High Performance Computing 2015*, A. Knüpfer, T. Hilbrich, C. Niethammer, J. Gracia, W. E. Nagel, and M. M. Resch, Eds. Cham: Springer International Publishing, 2016, pp. 161–181, doi: 10.1007/978-3-319-39589-0_13.

[14] 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

[15] 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

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

[17] A. Abel and J. 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*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 673–686. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304062

[18] (2018, 4) Instruction tables. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf

[19] J. Hammer, G. Hager, and G. Wellein, "OoO Instruction Benchmarking Framework on the Back of Dragons," 2018, SC18 ACM SRC Poster. [Online]. Available: https://sc18.supercomputing.org/proceedings/src_poster/src_poster_pages/spost115.html

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

[21] U. Manber, "Introduction to algorithms - a creative approach," 1989.

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