Friedrich-Alexander-Universität
Erlangen-Nürnberg

Platform-independent Micro-Benchmarking Suite for Automatic Performance Metrics Collection of Assembly Instructions

Bachelor Thesis

Handed in by:	Tobias Rühl
Matriculation number:	23024471
Examiner:	Prof. Dr. Gerhard Wellein
Advisor:	Jan Laukemann
Editing time:	01.04.2025 – 01.09.2025

Professur für Höchstleistungsrechnen Friedrich-Alexander-Universität
Erlangen-Nürnberg

Zusammenfassung

Performancemodelle ermöglichen Entwicklern wichtige Einblicke in die Interaktion des Codes mit der Hardware, auf der er ausgeführt wird, und ermöglichen ihnen zielspezifische Optimierungen vorzunehmen. Um In-Core-Performancemodelle für Out-of-Order-Prozessoren zu entwerfen, werden architekturspezifische Leistungsmetriken wie Latenz, Durchsatz und Portbindung einzelner Instruktionen benötigt. Für x86-Mikroarchitekturen gibt es bereits Tools, mit denen diese Werte automatisch ermittelt werden können. Die Erweiterung von Performancemodellierungstools wie dem Open Source Architecture Code Analyzer (OSACA) für andere Mikroarchitekturen wie RISC-V unterstreicht jedoch die Notwendigkeit eines plattformübergreifendes Mikrobenchmarking-Tools.

Diese Arbeit stellt WINIC (What I Need Is Cycles) vor, ein automatisches plattformübergreifendes Mikrobenchmarking-Tool. Im Gegensatz zu bisherigen Tools kann es Durchsatz- und Latenzmetriken für die meisten unterstützten Instruktionen jeder 64-Bit x86, AArch64 und RISC-V Mikroarchitektur ohne manuellen Aufwand ermitteln. Wir behandeln die Methodik des Mikrobenchmarkings im Allgemeinen sowie die Implementierung von WINIC. Anschließend zeigen wir dessen Verwendung um präzise Leistungsmetriken für x86 zu erhalten, von denen 93% mit vorhandenen Daten übereinstimmen, sowie Leistungsmetriken für AArch64 und RISC-V, und zeigen, wie diese Metriken verwendet werden können, um die In-Core-Performancevorhersagen für ausgewählte Mikrobenchmarks zu verbessern.

Abstract

Performance models allow developers to gain important insights into the interaction of their code with the hardware that executes it, and enable them to apply target-specific optimizations. To create in-core performance models for out-of-order processors, architecture-specific performance metrics like latency, throughput, and port binding of individual instructions are indispensable. For x86 microarchitectures, there already exist tools to automatically obtain those values, however, the extension of performance modeling tools like the Open Source Architecture Code Analyzer (OSACA) to support further architectures like RISC-V emphasizes the need for a cross-platform microbenchmarking tool.

We present WINIC (What I Need Is Cycles), an automatic cross-platform microbenchmarking tool. Unlike previous tools, it can obtain throughput and latency metrics for most supported instructions of any 64-bit x86, AArch64, and RISC-V microarchitecture without the need of manual effort. We cover the methodology of microbenchmarking in general as well as the implementation of WINIC. Then we show that it can be used to obtain accurate results for x86, 93% of which match existing data, as well as performance metrics for AArch64 and RISC-V, and show how these metrics can be used to improve the in-core performance predictions on selected micro-kernels.

Contents

1 Introduction

1.1 Motivation

Optimizing large-scale scientific applications is critical for reducing their cost and energy consumption. Performance modeling plays a vital role in understanding the interaction between the code and the hardware that executes it. Performance modeling methods can be broadly divided into two categories: *black-box* and *white-box* models [1]. Black-box performance prediction tools like Ithemal [2] use a machine learning model trained on instruction sequences and measured throughput values to give an estimate of the performance; they do not, however, reveal if and how potential performance gains can be achieved. White-box models help to gain these insights as they try to provide the reasons for their predictions.

The Open Source Architecture Code Analyzer (OSACA) [3, 4] is a white-box static analysis tool for small loop kernels on specific microarchitectures. It provides a throughput analysis as well as detection of critical path and loop-carried dependencies (LCDs). OSACA supports x86 and AArch64, and support for RISC-V is currently under development. To make accurate predictions, OSACA needs the latency, throughput, and port usage of each instruction on the target architecture. Additionally, this data must be in a unified format, and if new microarchitectures are released, regular efforts must be made to obtain the new data to keep OSACA useful. There are two major sources from which the performance metrics can be obtained: the CPU manufacturer's documentation or microbenchmarking. Documentation is often not machine-readable, in different formats for each CPU manufacturer, or not publicly available at all. Therefore, benchmarking is the more versatile option.

There are already several microbenchmarking tools mentioned in Section 1.3, but none of them are suitable for obtaining all performance metrics needed for OSACA. Table 1 compares the capabilities of some existing tools. Important features are the sup-

	nanoBench	iBench	asmbench	llvm-exegesis	CQA	**WINIC**
x86	✓	✓	✓	✓	✓	✓
AArch64	✗	✓	✓	✓	✓	✓
RISC-V	✗	✗	✗	✓	✗	✓
throughput	✓	✓	✓	✓	✓	✓
latency	✓	✓	✓	✓	✓	✓
per-operand latency	✓	✓	✗	✗	✓	✓
port usage	✓	✗	✗	✓*	✗	✗
automatic	✓	✗	✗	✓	✓	✓
user-space	✓	✓	✓	✓	?‡	✓
memory instructions	✓	✓	✓†	✓	?‡	✗

Table 1: Capabilities of existing microbenchmarking tools compared to WINIC. * indicates only x86 support, † that only simple load/store instructions are supported, and ‡ the lack of documentation for this category.

ported architectures (x86, AArch64, RISC-V) and metrics (latency, per-operand latency, throughput, port usage), as well as automatic benchmark generation and a user-space mode.

1.2 Scope of Work

This work is an attempt to create a versatile microbenchmarking tool, designed to complement OSACA. It should automatically measure throughput, latency, and port usage on x86, AArch64, and RISC-V platforms, and do so for future microarchitectures, possibly featuring new ISA extensions, without needing major adjustments. Furthermore, since it will be used on HPC systems, we want it to run in user-space. This prevents us from measuring privileged instructions; however, since our focus lies on scientific kernels, those are mostly irrelevant.

To keep the scope of work suitable for a bachelor's thesis, we excluded (a) the task of obtaining the port mapping and (b) the benchmarking of instructions accessing memory, however, WINIC is designed in a way that should allow those features to be added in the future.

1.3 Related Work

1.3.1 Performance-Modeling Tools

There are a variety of basic-block throughput prediction tools, which can be broadly classified into ML-based black-box models and white-box models such as analytical models or simulators. For a comparison of efficiency and accuracy of many existing tools using the BHive [5] benchmark suite on Intel CPUs, see [6].

IACA [7] is Intel's proprietary static analysis tool. It provides throughput analysis for Intel Core CPUs up until the Skylake microarchitecture. Its development was discontinued in 2019.

llvm-mca [8] uses LLVM's scheduling models to predict the throughput of a basic block. It simulates the execution and provides an IACA-style report. In contrast to most other tools which are focused on x86, due to its integration in the LLVM ecosystem, it supports a variety of architectures including AArch64 and RISC-V.

uiCA [9] is a simulation-based throughput predictor for Intel CPUs. As it is based on a very complex model constructed from extensive reverse engineering efforts, it yields very accurate results, but supports only Intel microarchitectures.

Facile [6] is an analytical model that evaluates multiple components of the front end and the back end separately, identifying which of these components is the bottleneck for a basic block. It achieves very accurate throughput predictions while still providing useful insights for manual optimization.

CQA (Code Quality Analyzer) [10], in addition to throughput prediction, evaluates loop code quality and reports potential issues such as missed vectorization opportunities, the use of expensive instructions, or data dependencies. It also suggests potential fixes such as changing compiler flags or directives.

Ithemal [2] uses a deep neural network trained on data obtained through measurements to accurately predict the throughput of basic blocks on different microarchitectures. Due to its nature as a black-box model, it is only of limited use for manual performance engineering, as throughput values alone cannot explain the code's interaction with the hardware.

GRANITE [11] is another machine-learning-based throughput predictor using a Graph Neural Network. The paper reports an improvement in accuracy over Ithemal of approximately 1.7 percentage points.

1.3.2 Microbenchmarking Tools

Analytical models such as IACA or OSACA and simulators such as uiCA offer very high interpretability, but they strongly depend on knowledge of the microarchitecture and can lack accuracy if this knowledge is incomplete. This is where microbenchmarking tools play an important role, as they provide the instruction-specific metrics on which these models are based.

iBench [12] is a tool for measuring the latency and throughput of single instructions purely based on runtime and instruction count. It can execute loop kernels manually written by the user and report the number of clock cycles elapsed per instruction. Currently, it only provides templates for x86 and AArch64. WINIC is based on iBench, but adds support for RISC-V and generates the benchmark kernels automatically.

nanoBench [13] is a microbenchmarking tool for x86. It executes benchmarking kernels and then uses the `RDMSR` and `RDPMC` instructions to read core-local performance counters, which store the number of core cycles or of μops dispatched on a specific port. Those values are then used to determine throughput, latency, and port usage of the executed instructions. Since the performance counters used are exclusive to x86, it does not meet our requirements. A database with results for recent Intel and AMD microarchitectures is available at uops.info [14].

llvm-exegesis [15] is a microbenchmarking tool for x86, AArch64, RISC-V, MIPS and PowerPC. It can automatically generate and execute benchmarks for latency, throughput, and port usage; however, it cannot measure latencies per operand, which will be covered in Section 2.3. Furthermore, it has no automatic mitigation of implicit dependencies in throughput benchmarks, covered in Section 3.2.1.

asmbench [16] is a benchmarking toolkit for x86 and AArch64. It uses the LLVM just-in-time (JIT) compilation via the llvmlite Python binding to generate and execute benchmarks.

1.4 Outline

This work is structured as follows: In Chapter 2, we discuss the design and performance metrics of modern processor cores relevant to this work. Chapter 3 goes into detail about the design and usage of WINIC, in particular, the automated benchmarking of per-operand latencies. Chapter 4 evaluates the metrics obtained with WINIC by comparing them to existing data, and Section 4.3 shows how these metrics can be used to improve OSACA's predictions. Finally, in Chapter 5, we summarize the thesis and give an outlook on future work.

2 Background

2.1 Static Performance Analysis

White-box static analysis methods evaluate assembly code snippets (basic blocks) using a given static model of a processor's performance characteristics. A model gives insights into possible optimizations by including a set of performance characteristics of the processor while making assumptions about those outside the scope of the model.

The Roofline model [17], for example, uses the operational intensity, i.e. the floating-point operations carried out per byte of data, of a piece of code to assess whether the performance on a given system is bound by the transfer of data to the cores (the memory bandwidth) or the operations done on this data. It assumes data transfer and execution can happen in parallel and ignores both caching and in-core resource conflicts, potentially limiting the achievable throughput of operations for the given kernel. The Execution-Cache-Memory (ECM) model [18] expands on this by taking caching into account, modeling non-overlapping data transfers, and having a more detailed in-core component.

2.2 Port Model

Figure 1 shows part of the structure of a Sandy Bridge core as an example of an out-of-order processor architecture. The instructions are decoded into one or multiple micro-operations (μops), each of which is assigned to one of the ports by the out-of-order scheduler. A port groups functional units into one logical unit that can receive a maximum of one μop per cycle. It is common for one port to have multiple types of functional unit, and for one type of functional unit to be present on multiple ports. This allows for parallel execution of multiple instructions of the same kind. The Sandy Bridge core

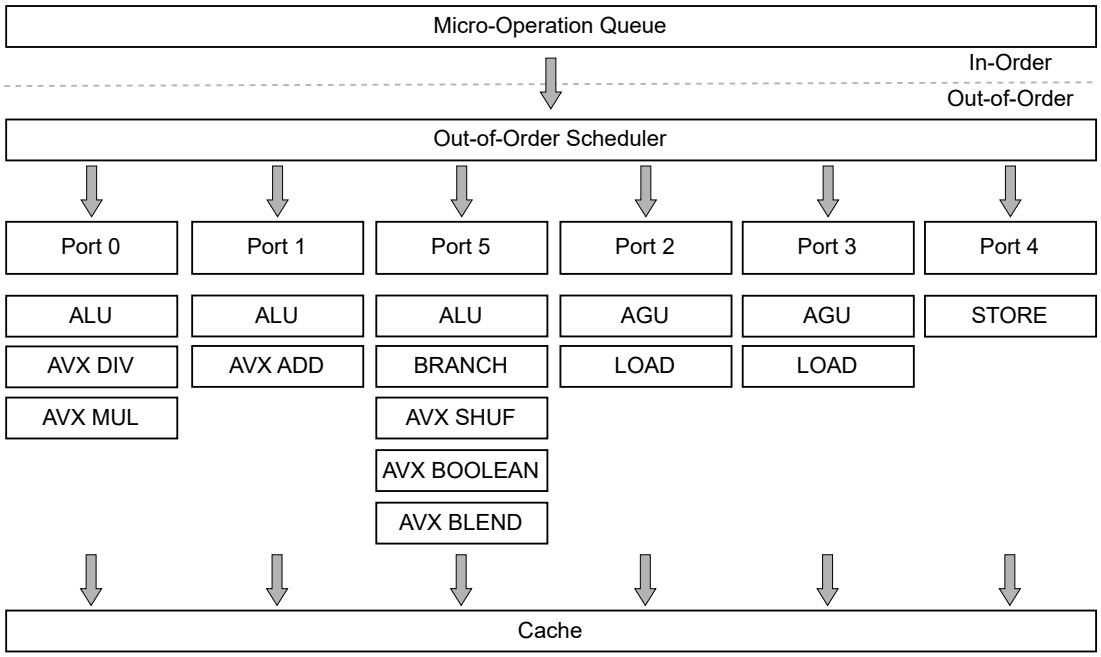

Figure 1: Schematic showing the out-of-order part of a Sandy Bridge core based on [19].

has three ports with an ALU, so it can execute three instructions that use that unit in parallel.

2.3 Latency

For the following sections, it is vital to differentiate instructions with the same name but different operand types. Therefore, aligning with [3], we will use the term *instruction form* to refer to an assembly instruction in combination with its operand types, while using the term *instruction* when referencing an instance of an instruction form, e.g. in an assembly file, or a machine instruction the CPU executes. We will denote instruction forms as follows: `Mnemonic (Types...)`.
The minimum time from issue to retirement is referred to as the latency of an instruction form. When an instruction is issued, all operands that get written by the instruction are locked to ensure that no other instructions that may run concurrently can modify them. Once the computation is finished, they are freed again and can be used by subsequent instructions. However, on some instruction forms, some operands may be freed earlier than others. For example, on x86 `BTS (r16, r16)` uses the second operand as an index

to select a bit in the first operand, stores it in the CF flag, and then sets it to 1. On AMD's Zen 4 microarchitecture it has a latency of 2 cycles, but the CF flag is already set and can be used by other instructions after 1 cycle. This is why the latency of the instruction form itself is not sufficient for an accurate performance prediction in some cases. In the following, we will call the latency between one read and one write operand of an instruction form a *sub-latency* of the instruction form. Section 3.3 covers how WINIC measures those sub-latencies.

2.4 Throughput

The number of independent instances of an instruction that can be executed per clock cycle is called the throughput of that instruction. It can be limited by the number of ports that can execute it or the throughput of the sequential part of the pipeline. To maintain the same unit for throughput and latency, it is common to specify reciprocal throughput (cycles/instruction). In the following, when mentioning throughput values, reciprocal throughput is implied unless stated otherwise.

2.5 OSACA

OSACA is an in-core performance modeling tool. This means that it assumes all memory accesses are L1 cache hits and focuses only on the in-core execution of an assembly kernel. Furthermore, it is assumed that the out-of-order scheduling works perfectly and the front end does not limit the throughput. Under those assumptions, there are two main factors that can limit the performance of a loop kernel. The first is the throughput of the port with the highest *port pressure*, so the one with the longest accumulated execution time per loop iteration. Second, the total performance can be limited by data dependencies between instructions. As long as there are only data dependencies between instructions of one loop iteration, their latencies can be hidden as instructions of the next iteration can be executed in parallel if the reorder buffer is large enough to hold the full loop. Dependencies across different loop iterations (*loop-carried dependencies*), however, impose another lower bound on the overall runtime. OSACA additionally provides the critical path—the longest chain of dependent instructions—as a lower bound for the kernel.

line	0	1	5	2	3	2D	3D	4	CP	LCD	Instruction
44											`.LBB0_8:`
45				0.50	0.50	0.50	0.50		4.0		`movsd (%rsi,%rcx,8), %xmm1`
46	1.00			0.50	0.50	0.50	0.50		5.0		`mulsd (%rdx,%rcx,8), %xmm1`
47		1.00							3.0		`addsd %xmm0, %xmm1`
48				0.50	0.50			1.00	0.0		`movsd %xmm1, (%rax,%rcx,8)`
49	0.33	0.33	0.33							1.0	`incq %rcx`
50	0.00	0.00	1.00								`cmpq $rcx, %rdi`
51											`* jne .LBB0_8`
sum	1.33	1.33	1.33	1.50	1.50	1.00	1.00	1.00	12	1.0	

Figure 2: OSACA output example: for each instruction it shows the port pressure, its contribution to the critical path and loop-carried dependencies if present. All throughput values are reciprocal throughput, so lower is better.

Figure 2 shows an example of an OSACA output on the following kernel:

```
1 double a[N], b[N], c[N], d;
2
3 for(int i=0; i<N; ++i) {
4     a[i] =  b[i] + c[i] * d;
5 }
```

Listing 1: The STREAM triad.

It was compiled using Clang 20.1.0 with flags `-O3 -fno-unroll-loops`, and analyzed[1] for the Sandy Bridge architecture. Looking at the output, the port(s) with the highest port pressure (indicated by the sum in the last line) are ports 2 and 3, with 1.5 cycles, and the loop-carried dependency (column "LCD") is 1 cycle. Therefore, the overall performance is bound by the saturation of ports 2 and 3, which, according to Figure 1, are the ports having a `LOAD` unit. The critical path (colkum "CP") is reported to be 12 cycles. Note that the `jne` instruction is marked with an `*`, which means OSACA assumes its throughput to be 0, as it will be fused with the `cmpq`. In summary, this output shows how the code interacts with the hardware of this specific architecture, and helps to identify the bottleneck.

[1] Available at https://godbolt.org/z/oe4YPj6ns

3 Methodology

3.1 Platform-Independent Approach

To obtain reproducible performance metrics of a single instruction, it has to be executed repeatedly until a steady state is reached, and external overhead by the loop code is overcome. In addition, an isolated benchmark environment is important to minimize interference of other processes. There are several options to obtain reliable performance metrics from these measurements. Nanobench [13] queries internal performance counters of the CPU which report the number of instructions retired since the start of the measurement, as well as the instructions retired per port. However, the counters for port occupation are only available on Intel CPUs, and while AArch64 and RISC-V provide performance counters to obtain the total number of instructions retired, to reduce platform-specific code as much as possible, we decided to rely purely on the total runtime of the loop instead. The number of cycles per instruction can be calculated using the total runtime, number of instructions executed, and the clock frequency.

This is the same approach used by iBench [12]. WINIC can be seen as an extension of iBench, generating the loop body for throughput and latency benchmarks for every instruction form supported automatically.

3.2 Measuring Throughput

To measure the throughput of an instruction form, a loop body with a sequence of those instructions has to be generated in a way that allows the CPU to parallelize them as much as possible. Therefore, dependencies between individual instructions must be avoided. The trivial way to do this is to choose different registers for each operand of each instruction. However, since the number of registers available is limited, this

sometimes leads to very short sequences, which can have a negative impact on the accuracy of the measurements as it increases the loop overhead. To generate more independent instructions, WINIC reuses registers for operands that are read but not written by the instruction form. For example, `VADDPD (r128, r128, r128)` on x86 writes to the first register operand and reads from the second and third operands. WINIC generates the following loop body, reusing registers where possible:

```
1  vaddpd  xmm0, xmm1, xmm2
2  vaddpd  xmm3, xmm1, xmm2
3  vaddpd  xmm4, xmm1, xmm2
4  vaddpd  xmm5, xmm1, xmm2
5  vaddpd  xmm6, xmm1, xmm2
6  vaddpd  xmm7, xmm1, xmm2
7  vaddpd  xmm8, xmm1, xmm2
8  vaddpd  xmm9, xmm1, xmm2
9  vaddpd  xmm10, xmm1, xmm2
10 vaddpd  xmm11, xmm1, xmm2
11 vaddpd  xmm12, xmm1, xmm2
12 vaddpd  xmm13, xmm1, xmm2
```

Listing 2: The benchmark kernel generated by WINIC to measure the throughput of `VADDPD (r128, r128, r128)`.

Note that disjoint registers were chosen for the second and third operand within a single instruction. This is because instruction forms can have semantics that allow for optimizations if two operands are the same, e.g. an `VPXOR (XMM, XMM, XMM)` will always have the result 0 if the second and third operand are identical. Modern processors can detect such cases and skip part of the pipeline to speed up execution, which would render the measurements useless. We chose 12 as the target number of instructions to generate, as this proved to yield reliable results while keeping the loop body of manageable size for manual adjustments. However, WINIC will not fail if there are not enough registers to meet this target, but will instead generate as many instructions as possible.

3.2.1 Implicit Dependencies

Since instruction forms can have side effects, such as writes to a flag register, some will still have dependencies on themselves regardless of the registers chosen. In order to measure the throughput anyway, WINIC uses another instruction form to break those

dependencies. The ADC instruction forms on x86 are an example of this, as they read and write the carry-flag. Even if choosing different registers for all instructions in the loop, they would still have an implicit dependency on each other. WINIC detects this and automatically inserts a TEST between each of the ADCs:

```
1   adc    rax, 7
2   test   rcx, rdx
3   adc    rsi, 7
4   test   rcx, rdx
5   adc    r9, 7
6   test   rcx, rdx
7   adc    r10, 7
8   test   rcx, rdx
9   adc    r11, 7
10  test   rcx, rdx
11  adc    rbx, 7
12  test   rcx, rdx
13  adc    r14, 7
14  test   rcx, rdx
15  adc    r15, 7
16  test   rcx, rdx
17  adc    r12, 7
18  test   rcx, rdx
19  adc    r13, 7
20  test   rcx, rdx
```

Listing 3: Benchmark kernel generated to measure the throughput of `ADC(r64, i32)`.

Each TEST writes to the flags but does not depend on any ADC or TEST preceding it, assuming the registers are chosen correctly. Therefore, it breaks the dependency and allows for multiple ADCs to be executed in parallel. The instruction form used for this has to write to the register causing the dependency and not introduce any other explicit or implicit dependencies into the benchmark kernel. Based on those constraints, WINIC dynamically chooses the instruction form to break the dependency.

Adding those breaking instructions can affect the measurements. If the interleaved instruction form uses the same resources as the measured one, its throughput has to be subtracted from the measured result. However, since WINIC currently has no way of determining if this is the case, it instead reports a range within which the throughput

must lie. In case of the `ADC` benchmark, the measured throughput TP_{m} on Zen 4 is 0.5 cycles. If there is a resource conflict, this result is incorrect. In the best-case scenario, both instruction forms use the exact same ports, so the throughput of the `ADC` instruction form can be calculated as $TP_{\text{ADC}} = TP_{\text{m}}$ - TP_{TEST}. Since `TEST` has a throughput of 0.25 cycles, we can infer that TP_{ADC} lies between 0.25 and 0.5 cycles.

3.3 Measuring Latency

As discussed in Section 2.3, we have to measure all sub-latencies of an instruction form individually. For this, we classify the sub-latencies according to the types of the source operand and the destination operand. We call the combination of those the *type* of the sub-latency and denote it as `(type(source)` → `type(destination))`.

3.3.1 Measuring Symmetric Latency

We will call a sub-latency type *symmetric* if the source operand and the destination operand have the same type. The `VADDPD (r128, r128 r128)` instruction form has two symmetric sub-latencies of type `(r128` → `r128)` as it writes to the first operand and reads from both the second and third operands. For the first sub-latency, WINIC generates this kernel:

```
1  vaddpd  xmm0, xmm0, xmm1
2  vaddpd  xmm0, xmm0, xmm1
3  vaddpd  xmm0, xmm0, xmm1
4  vaddpd  xmm0, xmm0, xmm1
5  vaddpd  xmm0, xmm0, xmm1
6  vaddpd  xmm0, xmm0, xmm1
7  vaddpd  xmm0, xmm0, xmm1
8  vaddpd  xmm0, xmm0, xmm1
9  vaddpd  xmm0, xmm0, xmm1
10 vaddpd  xmm0, xmm0, xmm1
11 vaddpd  xmm0, xmm0, xmm1
12 vaddpd  xmm0, xmm0, xmm1
```

Listing 4: The benchmark kernel generated by WINIC to measure the sub-latency of `VADDPD (r128, r128, r128)` from the second operand to the first operand.

It uses the same register for the two operands involved, introducing a read-after-write dependency between the instructions, which limits the throughput of the kernel to the sub-latency of this read-write operand pair. The other registers are chosen in such a way that no additional dependencies are introduced.

3.3.2 Measuring Asymmetric Latency

To measure asymmetric sub-latencies a different approach is needed. For example, a sub-latency of type `(r64 → flags)` cannot be measured alone; however, one can measure a combination of two instruction forms with sub-latencies of type `(r64 → flags)` and `(flags → r64)` by interleaving them and thus creating a latency chain. This leaves one with some latency L_{combined} and no information on which instruction form contributed how much to the combined result. To narrow down the ranges of the individual latencies, we first group all asymmetric sub-latency measurements by their types. For a pair of complementary types `(a → b)` and `(b → a)` all combinations of instructions from those types can be measured. We then systematically measure those combinations until we find the one with the minimal combined latency $L_{\text{combined, min}}$. The two instruction forms determined this way are then used to measure all other instruction forms with sub-latencies of the two types. For type pairs where $L_{\text{combined, min}}$ is 2 cycles, it can be inferred that both instruction forms have a sub-latency of 1 cycle, as we assume every instruction needs at least 1 cycle to be executed, and the other sub-latencies of the two types can be determined exactly. This is the case for around 90% of all latency values on Zen 4, so only around 10% remain ranges. Section 5.2 will give an outlook on how this possibly could be improved.

An example of an instruction form with asymmetric sub-latencies is `BTS (r16, r16)` on Zen 4. It reads and writes the first register operand, reads the second register operand and writes the CF flag, therefore, it has four sub-latencies of types `(r16 → r16)` and `(r16 → flags)`. To measure the sub-latencies of type `(r16 → flags)`, another instruction form is needed. When doing a full run on Zen 4, WINIC finds the pair `ADC (r16, r16)` and `ADC (r16, i16)` with sub-latencies of types `(r16 → flags)` and `(flags → r16)` that have $L_{\text{combined}} = 2$ cycles. WINIC infers that both have a latency of 1 cycle, and uses them as helpers for other instruction forms, such as `BTS (r16, r16)`. It then generates the following loop to measure the sub-latency of `BTS (r16, r16)` from the second register operand to the flags:

```
1  bts    dx, ax   # read ax, write flags
2  adc    ax, cx   # read flags, write ax
3  bts    dx, ax
4  adc    ax, cx
5  ...
6  bts    dx, ax
7  adc    ax, cx
```

Listing 5: Shortened benchmark kernel generated for `BTS (r16, r16)` to measure the sub-latency from the second register operand to the flags.

There is a read-after-write dependency on the `ax` register between each `ADC` and the next `BTS`. The latency this dependency introduces is known to be 1 cycle, as discussed before. Between each `BTS` and the next `ADC`, there is an implicit read-after-write dependency, as `BTS` writes to the flags and `ADC` reads them. The latency this dependency introduces is the sub-latency we are interested in. The measurements reveal that, on Zen 4, it is only 1 cycle, whereas the sub-latencies of type `(r16` $\rightarrow$ `r16)` are 2 cycles.

3.4 LLVM

To automatically generate the benchmarking kernels with all correction mechanisms described above, a lot of meta-information is needed:

- A list of all supported instruction forms
- An operand list and side effects per instruction form
- Type and applicable values for each operand
- Information on how to generate syntactically correct assembly code from the above

LLVM [20] was chosen for this, as it can serve as a single source for all of this information. LLVM is a ”collection of modular and reusable compiler and toolchain technologies” [20], including the Clang compiler, the LLDB debugger, implementations of the C and C++ standard libraries, and many more. It supports a wide variety of architectures, making it ideal for this purpose. LLVM’s internal data structures for storing this information are the same for all architectures, so it was possible to implement the loop generation

logic in a fully architecture-independent way. Furthermore, LLVM is open source, well maintained and is continuously updated to support new microarchitectures.

To generate the kernel in Listing 3, for example, WINIC interacts with LLVM like this:

- It looks up the operand list.
- For each register operand, it looks up all registers of the correct type for that operand.
- It generates the desired number of instructions, using LLVM's `MCInst` data structure and populates register and immediate operands, keeping track of the registers used to avoid dependencies.
- It detects the implicit dependency on the flags based on the list of implicit register usages LLVM provides for the instruction form and searches for a dependency-breaking instruction form.
- It generates suitable breaking `MCInst`s and interleaves them.
- It uses LLVM's `MCInstrPrinter` to generate valid assembly from the `MCInst`s.

3.5 Executing Benchmarks

The generated kernels are embedded in a target-specific template that provides a function label, the loop logic, and code to save and restore callee-saved registers. The result is assembled to a shared library, which is then loaded and provides the generated function with the loop count as its only argument. Finally, the function is executed and its runtime measured. Every benchmark is run in a subprocess to recover from faults or unwanted side effects executing arbitrary instructions can result in. To ensure reproducibility when assembling the benchmarks, a fixed version of Clang is used, which is built alongside LLVM during the setup process.

3.6 Improving Accuracy

The total execution time of the benchmark function contains overhead from the loop itself, as well as the saving and restoring of registers. To improve the accuracy of the measurements, WINIC does a second benchmark with the loop body unrolled twice. Using the two results, the overhead can be removed:

$$\begin{aligned} T_{\text{Unrolled}} - T_{\text{Single}} &= (2 * T_{\text{Loop}} + T_{\text{Overhead}}) - (T_{\text{Loop}} + T_{\text{Overhead}}) \\ &= T_{\text{Loop}} \end{aligned} \tag{3.1}$$

Some instructions can be executed efficiently if the operands meet certain criteria. The CPU may, for example, diverge from the normal execution path for a multiplication if one of the operands is zero. To avoid such optimizations, all registers used get initialized to "uncritical" values (neither zero or one). This adds to the overhead of the function but, since this overhead is identical for the normal and the unrolled variant, it is corrected by the method described above.

On some x86 architectures, there is a performance penalty when transitioning between AVX and SSE instructions [21]. The penalty occurs if a register previously used by an SSE instruction is used by an AVX instruction, and vice versa. Since it is not guaranteed to occur both when running the normal and the unrolled benchmark, it cannot be corrected using Equation (3.1) reliably. To prevent a potential transition at the first loop iteration of the benchmark, an additional init function gets called just before the benchmark function. It executes all instructions in the kernel to be benchmarked once, thereby setting the AVX/SSE state of all vector registers used.

3.7 Usage

The command line interface of WINIC has the following format:

```
winic -f <frequency> [options] MODE [mode options]
```

Before running WINIC, the clock frequency must be set to a fixed value and supplied via the `-f` or `--frequency` option. The `winic` command is always used with one of three

modes (`TP`, `LAT`, `MAN`), representing the throughput mode, latency mode, and manual mode, respectively. The following options are available for all modes:

- `-h` or `--help` prints the help message.
- `-d` and `--debug` enables debug messages.
- `-c`/`--cpu` and `-m`/`--march` only have to be used if LLVM cannot detect the CPU model or the microarchitecture on its own.

3.7.1 Throughput and Latency Mode

TP and LAT modes are used to automatically generate and execute throughput or latency benchmarks, respectively. Without further arguments, they will go through all instruction forms LLVM knows for the architecture and generate a timestamped YAML database with the results as well as a report file. To alter WINICs behavior, both modes share the same set of options:

- `-i` or `--instruction` executes benchmarks only for the specified list of LLVM instruction forms. The WINIC repository contains reference files to find the LLVM name for an instruction form.
- `--minOpcode` and `--maxOpcode` can be used to limit the range of opcodes to measure. This is mainly useful for benchmarking and development.
- `--noReport` disables the generation of report files.
- `-o` or `--output` changes the path of the YAML output file. If it already exists, all results obtained in the run will overwrite the existing ones in the file, and all other entries will be left untouched. If set to */dev/null* no output is generated.
- `--x87FP` enables x87 floating point instruction forms. They are disabled by default, as they are deprecated and emulated on some platforms, which can significantly increase the total runtime.

3.7.2 Manual Mode

The manual mode can be used to execute arbitrary custom benchmark kernels.

The following mode options are required to execute the correct function and calculate the cycles per instruction:

- `-p` or `--path` specifies the assembly file.
- `--funcName` specifies the name of the function to be executed in that file.
- `-n` or `--nInst` specifies the number of instructions in the loop body.

Optionally, `--initName` can be used to specify a function to be executed before the benchmark function, as described in Section 3.6.

By default, when measuring a single instruction form, WINIC dumps the assembly file generated. A common workflow therefore is to do a single instruction from run, then edit the dumped file and execute it again using the manual mode.

3.7.3 Output

In latency and throughput mode, WINIC produces a YAML file with the results. For the `BTS (r16, r16)` instruction form covered in Section 3.3.2, the output looks like this:

```yaml
- llvmName:         BTS16rr
  name:             bts
  operands:
    - class:            register
      name:             GR16
      read:             true
      write:            true
    - class:            register
      name:             GR16
      read:             true
      write:            false
  latency:          2
  operandLatencies:
    - sourceOperand:    '0'
      targetOperand:    '0'
      latencyMax:       2
      latencyMin:       2
    - sourceOperand:    '1'
      targetOperand:    '0'
      latencyMax:       2
      latencyMin:       2
    - sourceOperand:    '0'
      targetOperand:    EFLAGS
      latencyMax:       1
      latencyMin:       1
    - sourceOperand:    '1'
      targetOperand:    EFLAGS
      latencyMax:       1
      latencyMin:       1
  throughput:       1
  throughputMin:    1
  throughputMax:    1
```

Listing 6: YAML output for `BTS(r16, r16)`.

As discussed before, it has four sub-latencies. Each of those has a source and target operand field, which are indices that point to an entry in the operand list or register names for implicit operands such as the `EFLAGS` register. WINIC was able to determine an exact throughput value, as `throughputMin` is equal to `throughputMax`. The output format is similar to the one that OSACA uses to allow easy integration into its database.

During the benchmarking process, WINIC dynamically selects helper instruction forms, as discussed in Sections 3.2.1 and 3.3.2. This is logged in a report file as follows:

```
-----BTS16rr-----
    BTS16rr(1(Class<GR16>) -> 0(Class<GR16>))  [2.01;2.01]
        Successful, latency: 2.01
    BTS16rr(2(Class<GR16>) -> 0(Class<GR16>))  [2.01;2.01]
        Successful, latency: 2.01
    BTS16rr(1(Class<GR16>) -> impl(Reg<EFLAGS>))  [1;1.01]
        Dependencies:
            ADC16ri(impl(Reg<EFLAGS>) -> 0(Class<GR16>))
            ADC16ri8(1(Class<GR16>) -> impl(Reg<EFLAGS>))
        Combined result: 2.01 cycles
    BTS16rr(2(Class<GR16>) -> impl(Reg<EFLAGS>))  [1;1.01]
        Dependencies:
            ADC16ri(impl(Reg<EFLAGS>) -> 0(Class<GR16>))
            ADC16ri8(1(Class<GR16>) -> impl(Reg<EFLAGS>))
        Combined result: 2.01 cycles
```

Listing 7: Report for latency benchmarks of `BTS (r16, r16)`.

The third and fourth measurements have two dependencies, `ADC16ri` and `ADC16ri8`, which were selected as helpers for the types `(EFLAGS` $\rightarrow$ `r16)` and `(r16` $\rightarrow$ `EFLAGS)`. The measurement can now be reproduced by supplying those along with the `BTS` instruction form:

```
winic -f <frequency> LAT -i BTS16rr ADC16ri ADC16ri8
```

4 Results

4.1 Testing Environment

The results covered in this chapter were obtained on three systems listed in Table 2.

		x86-64	AArch64	RISC-V
HW	CPU Model	AMD EPYC 9654	Nvidia Grace	Banana Pi F3
	Microarchitecture	Zen 4	Neoverse V2	Spacemit X60/K1
	Base Frequency	2.4 GHz	3.1 GHz	1.6 GHz
	Fixed Frequency	1.5 GHz	3.2 GHz	1.6 GHz
SW	Compiler	GCC 13.3.0	GCC 13.3.0	GCC 15.1.1
	OS	Ubuntu 24.04	Ubuntu 24.04	ArchLinux (rolling)
	Linux kernel	6.8.0-63-generic	6.8.0-78-generic	6.1.15
RES	Obtained TP values	6087	3540	453
	Obtained LAT values	12372	5967	729

Table 2: Hardware, software, and number of results for our three test systems.

WINIC[1] currently uses LLVM and Clang at version 20.1.5 on all platforms. For every instruction form, multiple (sub-latency) values can be obtained, but no more than one throughput value. This means that the number of throughput values also indicates the total number of instruction forms WINIC was able to measure on the given platform. At the time of writing, there was no single version of the GCC compiler available on all three systems, which is why we used a different version on the RISC-V system. The measurements for this chapter were done using the WINIC repository at commit 6341e1a.

[1]Available at `https://github.com/RRZE-HPC/WINIC`

4.2 Comparison to Existing Data

WINIC produces a large number of measurements, as shown in Table 2, which makes it difficult to evaluate their quality. Only for x86 processors, uops.info [14] provides a comprehensive database of latency and throughput values obtained using Nanobench [13]. We therefore compare WINIC and uops.info to obtain an estimate of the quality of the results.

LLVM and uops.info store instruction forms in different formats, so matching an LLVM instruction form to an entry in the uops.info database is not trivial. The main two difficulties are:

- There are some cases where the databases disagree; for example, for `IMUL (R8)` LLVM marks `AL` as being read and written to, whereas uops.info marks it as read-only. In those cases, the script to compare the results cannot match the instruction forms.
- The script can find multiple matches for one LLVM instruction form, e.g. because uops.info differentiates between `R8l` and `R8h` registers, but LLVM does not. In the following, the results are considered the same if all matching uops results have the same throughput or latency values as the WINIC result.

Figure 3 visualizes the results on Zen 4 taking all of this into account. It shows that for both throughput and latency, around 95% of the values can be matched to one or more uops results. Around 93% of those are identical to the values from uops.info. As explained in Sections 3.2.1 and 3.3.2, WINIC can sometimes only determine ranges instead of exact results. On Zen 4, this occurred for 2% of the throughput values and 11% of the latency values.

4.3 Case Study

Now we will look at a loop kernel for which the new sub-latency values can improve the accuracy of OSACA's prediction.

Fused-multiply-add operations in a streaming 1D-fasion as well as in multiple dimensions are common in a variety of scientific applications; among others, they can be found in stencil codes and linear algebra kernels (e.g. AXPY, GEMM). For simplicity, we look

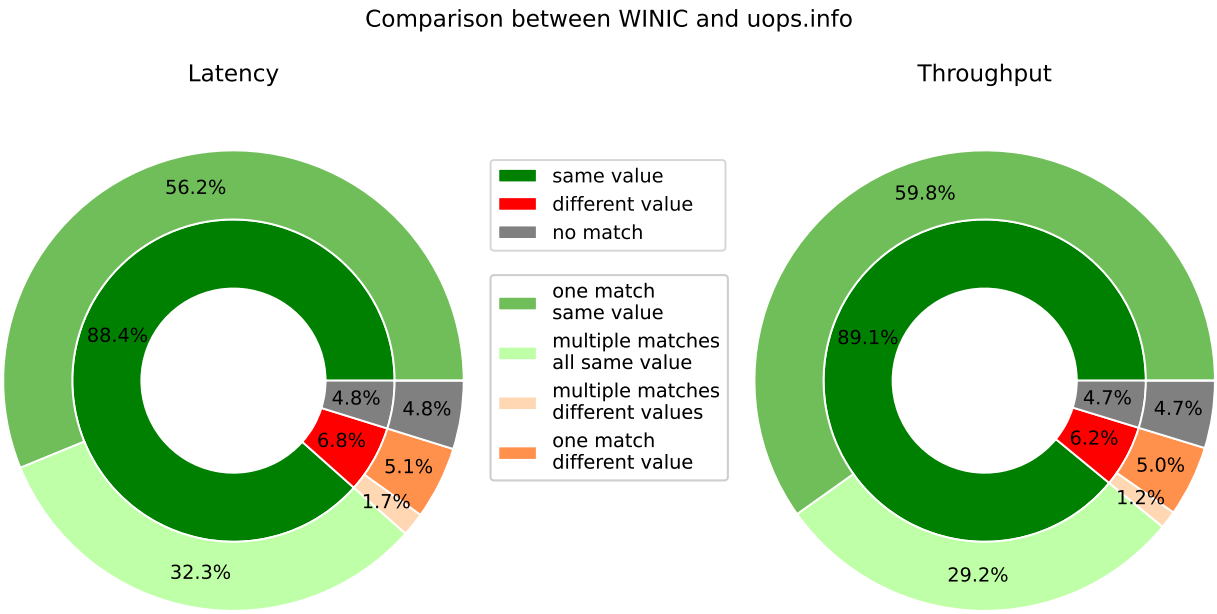

Figure 3: Comparison of the results with uops.info: Values are considered to be the same if the uops value lies in the determined range with an additional tolerance of 10%.

at a slightly modified Schönauer-DAXPY-kernel. To observe the effect of sub-latency values, we adjust it so that the fused-multiply-add cannot be vectorized because of a read-after-write dependency. The full example including the OSACA output is available on the Compiler Explorer[2].

```
1 double a[N], b[N], c[N];
2
3 for(int i=1; i<N; ++i) {
4     a[i] =  a[i-1] + b[i] * c[i];
5 }
```

Listing 8: A variant of the Schönauer-DAXPY-kernel with a read-after-write dependency.

The above kernel was compiled for Neoverse-V2 using the Clang compiler at version 20.1.0 with flags `-mcpu=grace`, `-O3`, and `-fno-unroll-loops`. The resulting assembly code looks like this:

[2]https://godbolt.org/z/bzrr99fK8

```
1  ldr     d1, [x20, x8]
2  ldr     d2, [x0, x8]
3  fmadd   d0, d1, d2, d0
4  str     d0, [x19, x8]
5  add     x8, x8, #8
6  cmp     x8, #1, lsl #12
7  b.ne    .LBB0_3
```

Listing 9: The compiled assembly code for the kernel in Listing 8.

Figure 4 shows OSACA's report on this kernel. The throughput prediction would be 1 cycle, as that is the throughput of the ports with the highest port pressure (ports 12, 13 and 14). However, the LCD report reveals that the loop is expected to be bound by the loop-carried dependency of the `FMADD` instruction, which reads and writes `d0` every iteration. `FMADD(FPR64,FPR64,FPR64)` has a latency of 4 cycles, so a minimum of 4 cycles per loop iteration is needed. However, when benchmarking this kernel on an NVIDIA GH200 chip, we measure only 2 cycles per iteration. To investigate why, we use WINIC to measure the sub-latencies of the `FMADD` as follows:

```
winic -f 3.2 LAT -i FMADDDrrr
    FMADDDrrr(1(Class<FPR64>) -> 0(Class<FPR64>))  [4.01;4.01]
    FMADDDrrr(2(Class<FPR64>) -> 0(Class<FPR64>))  [4.01;4.01]
    FMADDDrrr(3(Class<FPR64>) -> 0(Class<FPR64>))  [2.01;2.01]
    FMADDDrrr(impl(Reg<FPCR>) -> 0(Class<FPR64>))  [ERROR_NO_HELPER]
```

Listing 10: WINIC output for measuring the latency of FMADDrrr on AArch64.

The output shows that `FMADD` has four sub-latencies. Measuring the last one fails, as it is of asymmetric type and therefore would need a helper instruction form[3] as described in Section 3.3.2. The other measurements reveal the reason for the mismatch between the OSACA prediction and the empirical result. While the other sub-latencies are 4 cycles, the one relevant to the loop-carried dependency is only 2 cycles. Once OSACA

[3]WINIC can, in fact, not measure this specific sub-latency as no instruction form of the AArch64 instruction set has a sub-latency of type `(FPR64 → FPCR)`. This is because `FPCR` (floating point control register) controls floating point rounding modes, overflow behavior, etc., and can only be set by the `MSR(FPCR,r64)` (Move to Special Register) instruction form. This being said, for the missing sub-latency value to affect performance, one would need a loop kernel where the floating point behavior is altered every iteration, which is highly unlikely. Therefore, this value can be considered irrelevant.

line	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	CP	LCD	Instruction
35													0.33	0.33	0.33			4.0		`ldr d1, [x20, x8]`
36													0.17	0.16	0.66					`ldr d2, [x0, x8]`
37									0.25	0.25	0.25	0.25						4.0	4.0	`fmadd d0, d1, d2, d0`
38													0.50	0.50		0.50	0.50	0.0		`str d0, [x19, x8]`
39			0.17	0.49			0.17	0.17												`add x8, x8, #8`
40			0.33				0.33	0.33												`cmp x8, #1, lsl #12`
41	0.50	0.50																		`b.ne .LBB0_3`
sum	0.50	0.50	0.50	0.49			0.50	0.50	0.25	0.25	0.25	0.25	1.01	1.00	1.00	0.50	0.50	8.0	4.0	

Figure 4: OSACA output for the kernel in Listing 8: Marked in red, the performance is expected to be limited to 4 cycles/iteration by the loop-carried dependency on the `fmadd` instruction.

line	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	CP	LCD	Instruction
35													0.33	0.33	0.33			4.0		`ldr d1, [x20, x8]`
36													0.17	0.16	0.66					`ldr d2, [x0, x8]`
37									0.25	0.25	0.25	0.25						4.0	2.0	`fmadd d0, d1, d2, d0`
38													0.50	0.50		0.50	0.50	0.0		`str d0, [x19, x8]`
39			0.17	0.49			0.17	0.17												`add x8, x8, #8`
40			0.33				0.33	0.33												`cmp x8, #1, lsl #12`
41	0.50	0.50																		`b.ne .LBB0_3`
sum	0.50	0.50	0.50	0.49			0.50	0.50	0.25	0.25	0.25	0.25	1.01	1.00	1.00	0.50	0.50	8.0	2.0	

Figure 5: Updated OSACA output for the kernel in Listing 8: Marked in red, the performance is now correctly predicted to be 2 cycles/iteration due to the new sub-latency values.

can handle the new sub-latency values, it will be able to provide an accurate prediction for this and other similar cases. Figure 5 shows the output with the new value.

5 Conclusion

5.1 Summary

This work provides a platform-independent microbenchmarking tool running in user-space. WINIC can automatically benchmark throughput and latency of all instruction forms available in LLVM on x86, AArch64, and RISC-V. Furthermore, it can break dependencies for throughput benchmarks if needed. We showed that WINIC can be used to automatically obtain sub-latency values for AArch64 instruction forms, which, to our knowledge, no other tool was capable of before. For x86, the results align with existing data by 93%, which gives confidence in the approach and the accuracy of the measurements.

5.2 Future Work

The next step for WINIC's development is to implement support for instruction forms with memory access, followed by an algorithm to measure port usage. Due to its integration with LLVM, it is technically possible to expand WINIC to architectures beyond the ones currently supported without major changes to the codebase.

The current algorithm used to measure latencies between asymmetric operand pairs described in Section 3.3.2 leaves room for improvement, as it allows for cases where only a latency range can be determined. Currently, only latency chains of length 2 are constructed and measured. Building longer chains and representing the results in a system of linear equations in the form of $L_1 + L_2 + \ldots + L_\mathrm{n} = L_\mathrm{measured}$ might yield exact results for all instruction forms involved, assuming that the system has a unique solution after including enough measurements. Future work might investigate this or other approaches to eliminate latency ranges.

Bibliography

[1] L. Eeckhout. *Computer Architecture Performance Evaluation Methods*. Cham: Springer International Publishing, 2010. DOI: 10.1007/978-3-031-01727-8.

[2] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. *Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks*. 2019. arXiv: 1808.07412 [cs.DC].

[3] 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)*. 2018, pp. 121–131. DOI: 10.1109/PMBS.2018.8641578.

[4] 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)*. 2019, pp. 1–6. DOI: 10.1109/PMBS49563.2019.00006.

[5] Y. Chen, A. Brahmakshatriya, C. Mendis, A. Renda, E. Atkinson, O. Sýkora, S. Amarasinghe, and M. 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)*. 2019, pp. 167–177. DOI: 10.1109/IISWC47752.2019.9042166.

[6] A. Abel, S. Sharma, and J. Reineke. "Facile: Fast, Accurate, and Interpretable Basic-Block Throughput Prediction". In: *2023 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2023, pp. 87–99. DOI: 10.1109/iiswc59245.2023.00023.

[7] *Intel Architecture Code Analyzer*. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/intel-architecture-code-analyzer-3-0-users-guide-157552.pdf (visited on Aug. 22, 2025).

[8] *LLVM Machine Code Analyzer*. URL: https://llvm.org/docs/CommandGuide/llvm-mca.html (visited on Aug. 4, 2025).

[9] A. Abel and J. Reineke. "uiCA: accurate throughput prediction of basic blocks on recent intel microarchitectures". In: *Proceedings of the 36th ACM International Conference on Supercomputing*. ICS '22. Virtual Event: Association for Computing Machinery, 2022. DOI: 10.1145/3524059.3532396.

[10] A. C. 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)* (2014). DOI: 10.1109/hipc.2014.7116904.

[11] O. Sykora, P. M. Phothilimthana, C. Mendis, and A. Yazdanbakhsh. *GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation*. 2022. arXiv: 2210.03894 [cs.LG].

[12] J. Hofmann. *iBench*. URL: https://github.com/RRZE-HPC/ibench (visited on July 1, 2025).

[13] A. Abel and J. Reineke. "nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems". In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 34–46. DOI: 10.1109/ispass48437.2020.00014.

[14] S. University. *uops.info*. URL: https://www.uops.info (visited on Aug. 25, 2025).

[15] *llvm-exegesis - LLVM Machine Instruction Benchmark*. URL: https://llvm.org/docs/CommandGuide/llvm-exegesis.html (visited on July 1, 2025).

[16] J. Hammer, G. Hager, and G. 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.

[17] S. Williams, A. Waterman, and D. Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Commun. ACM* 52.4 (2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

[18] 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 on International Conference on Supercomputing*. ICS’15. ACM, 2015, pp. 207–216. DOI: 10.1145/2751205.2751240.

[19] J. Hofmann. *A first-principles approach to performance, power, and energy models for contemporary multi-and many-core processors*. Verlag Dr. Hut, 2019. URL: https://www.dr.hut-verlag.de/9783843941877.html.

[20] *The LLVM Compiler Infrastructure*. URL: https://llvm.org (visited on July 1, 2025).

[21] Intel. *Avoiding AVX-SSE Transition Penalties*. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/11mc12-avoiding-2bavx-sse-2btransition-2bpenalties-2brh-2bfinal-809104.pdf (visited on Feb. 10, 2025).

Declaration of authorship

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Für das Erstellen des Sourcecodes wurde in Teilen KI-gestützte Autovervollständigung verwendet.

Der Friedrich-Alexander-Universität, vertreten durch die Professur für Höchstleistungsrechnen, wird für die 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, 01.09.2025 Tobias Rühl