Inferring and Analyzing Microarchitectural Performance Models

Dissertation zur Erlangung des Grades des Doktors der Ingenieurwissenschaften der Fakultät für Mathematik und Informatik der Universität des Saarlandes

eingereicht von
Fabian Ritter

Saarbrücken, 2024

Tag des Kolloquiums: 9. Oktober 2024

Dekan der Fakultät: Prof. Dr. Roland Speicher

Prüfungsausschuss:

Vorsitz: Prof. Dr. Sven Apel

Berichterstattende:

- Prof. Dr. Sebastian Hack, Universität des Saarlandes, Saarbrücken
- Prof. Dr. Jan Reineke, Universität des Saarlandes, Saarbrücken
- Dr. Fabrice Rastello, Directeur de Recherche, Inria, Grenoble

Akademischer Mitarbeiter: Dr. Marvin Wyrich

Abstract

Modern processors are complex systems that employ a wide range of techniques to execute programs as fast and efficiently as possible. However, these hardware intricacies make reasoning about the efficiency of code for the processor difficult. Microarchitectural performance models are therefore indispensable for estimating and improving how efficiently software takes advantage of the hardware. This dissertation presents several advancements in the field of microarchitectural performance modeling.

The first part of this thesis proposes techniques to characterize how a processor exploits instruction-level parallelism. Based on a formal model, we explore ways to infer a processor's port mapping from throughput measurements, i.e., how it splits instructions into micro-operations and how these are executed on the processor's functional units. Our techniques enable accurate port mapping inference for processors that prior methods could not reason about.

In the second part, we introduce AnICA, a method to analyze inconsistencies between performance models. AnICA takes inspiration from differential testing and abstract interpretation to systematically characterize differences in the outputs of basic block throughput predictors. It can summarize thousands of inconsistencies in a few dozen descriptions that provide high-level insights into the differing behaviors of such predictors. These results have lead to improvements in the scheduling models of the widely used LLVM compiler infrastructure.

Zusammenfassung

Moderne Prozessoren sind komplexe Systeme, die ein breites Spektrum an Techniken anwenden, um Programme so schnell und effizient wie möglich auszuführen. Diese Komplexität der Hardware erschwert jedoch auch das Abschätzen der Effizienz von Programmen für den jeweiligen Prozessor. Um die Hardware-Ausnutzung von Programmen zu beurteilen und zu verbessern, sind daher Leistungsmodelle nötig. Diese Dissertation präsentiert mehrere Beiträge zum Feld der Leistungsmodellierung von Prozessoren.

Der erste Teil der Dissertation untersucht die Ausnutzung von Parallelität auf Anweisungsebene innerhalb eines Prozessors. Basierend auf einem formalen Modell leiten wir aus Durchsatzmessungen ab, wie der Prozessor Anweisungen in Mikro-Operationen aufteilt und wie diese von den funktionalen Einheiten des Prozessors ausgeführt werden. Die präsentierten Methoden charakterisieren dieses Verhalten erstmals für eine Reihe von Prozessoren.

Der zweite Teil der Arbeit stellt AnICA vor, ein Verfahren zur Untersuchung von Inkonsistenzen zwischen Leistungsmodellen. AnICA vereint differenzielles Testen mit Konzepten der abstrakten Interpretation, um Unterschiede in den Ergebnissen von Durchsatzmodellen für Anweisungssequenzen zu charakterisieren. Tausende von Inkonsistenzen werden so durch wenige kompakte Beschreibungen zusammengefasst, die direkte Einsichten in die Durchsatzmodelle liefern. Durch diese Ergebnisse konnten Code-Generierungs-Modelle der LLVM Compiler-Infrastruktur verbessert werden.

Acknowledgments

First, I would like to thank my advisor, Prof. Sebastian Hack, for providing the opportunity to pursue this research. I am grateful for his encouragement, guidance, and feedback, for the freedom he gave me to explore new ideas, and for the great times we had along the way.

I would also like to thank Prof. Jan Reineke and Dr. Fabrice Rastello for reviewing this dissertation, and for the fruitful discussions I had with each of them over the course of my research. Furthermore, I would like to thank Prof. Sven Apel for leading the examination committee and Dr. Marvin Wyrich for representing the academic staff in the examination committee for this dissertation.

Special thanks go to my dear colleagues Tina Jung and Sebastian Hahn, for the many discussions, collaborations, and distractions we had. Moreover, I am grateful to them and the additional proofreaders of this thesis – Heiko Becker, Joachim Meyer, and Matthias Ritter – for their valuable feedback and suggestions.

I would like to thank all my colleagues from the Compiler Design Lab and the Real-Time and Embedded Systems Lab for providing a supportive and inspiring environment. It was a pleasure to work with them, and I am thankful for the discussions, collaborations, and fun we had. I am particularly grateful to Sandra Neumann for her invaluable help in navigating the administrative challenges and complexities of the university. I am also very thankful for the Wednesday Tea Parties with Sebastian, Florian, Christoph, Jan, and Shrey for providing a regular and welcome break from work. For research collaborations and student projects related to this thesis, I would further like to thank Kallistos Weis, Timo Gros, Luis Paulus, and Lukas Schaller. My gratitude also goes to the International Max Planck Research School for Computer Science for supporting me early on during my doctoral studies.

Last, but not least, I would like to thank my family for their continuous support and encouragement.

Contents

Chapter 1

Introduction

Increasing the performance of computer programs is a fundamental challenge in computing. Faster and more efficient software is beneficial, no matter if it runs at the scale of personal computers, smartphones, supercomputers, or embedded devices. To satisfy the need for improved computing efficiency and performance, hardware is becoming more and more complex. Achieving peak performance on such complex systems requires software optimizations that depend on the hardware's performance characteristics.

Developers of high-performance software therefore rely on performance models of their target hardware. Performance models enable the developers to estimate how code executes on a given system and how to adjust it to run more efficiently. Industry and academia have produced a wide range of performance models for various computing devices – e.g., central processing units (CPUs) (Di Biagio, 2018; Intel, 2012), graphics processing units (GPUs) (Baghsorkhi *et al.*, 2010), specialized accelerators like tensor processing units (TPUs) (Ni *et al.*, 2022), and field-programmable gate arrays (FPGAs) (Hung *et al.*, 2009) – and for a variety of applications. They differ in the kind of input they expect – from short sequences of machine instructions (Intel, 2012) to entire programs (Binkert *et al.*, 2011) – as well as the accuracy they strive for – from cycle-accurate (Böhm *et al.*, 2010) to "back-of-the-envelope calculations" (Ofenbeck *et al.*, 2014; Williams *et al.*, 2009).

The focus of this dissertation lies on microarchitectural CPU models that describe how a processor core executes a stream of machine instructions. Such models are used in a variety of contexts, for example,

- by developers to guide manual code optimizations (Tan *et al.*, 2023),
- by compilers to generate more efficient machine code (GCC, 2023; LLVM, 2022; Pohl *et al.*, 2019), and
- for super-optimization tools (Liu *et al.*, 2023; Phothilimthana *et al.*, 2016) to identify high-performing implementations.

Processor design techniques like out-of-order execution, the decomposition of instructions into smaller micro-operations, and common-case optimizations render performance prediction in this setting challenging. Incomplete documentation by the manufactures often amplifies the challenge further.

This dissertation advances the field of microarchitectural performance modeling for CPUs in two directions: inferring a processor's port mapping and analyzing how available performance models differ.

Port Mapping Inference

The port mapping describes how a processor with an out-of-order design exploits instruction-level parallelism. It models how the processor internally splits instructions into simpler operations, so-called micro-ops or µops, and which execution units, grouped behind so-called ports of the processor, can execute these µops. Figure 1.1 visualizes an example for this relationship: Here, the instructions `add`, `sub`, `mul`, and `store` are implemented with µops u_1, u_2, and u_3 that can be executed on different subsets of the ports p_1, p_2, and p_3. Port mappings are important components of low-level processor performance models, for example for predicting the throughput of instruction sequences (Abel and Reineke, 2022; Di Biagio, 2018; Laukemann *et al.*, 2018) or in the cost models of compiler backends (GCC, 2023; LLVM, 2022). Hardware manufacturers usually do not provide a complete port mapping for their microarchitectures.

Prior work like the instruction tables by Fog (2022), EXEgesis (LLVM, 2023a), and uops.info (Abel and Reineke, 2019) rely on hardware support in the form of performance counters to infer port mappings from microbenchmarks. When the hardware can be configured to count the executed µops per port, determining where each instruction's µops can be executed is straightforward. Suitable performance counters are, however, not always available. For instance, AMD's processors and many ARM designs do not provide all necessary counters and are hence not supported by these approaches.

In this dissertation, we explore port mapping inference strategies that reduce or eliminate the need for hardware performance counters. Instead, our techniques measure the processor's instruction throughput for specific microbenchmarks. By benchmarking various combinations of instructions, we determine conflicting port requirements among the instructions. These observations characterize the processor's port mapping, albeit more indirectly than the performance counter readings used in previous approaches. The key challenges in throughput-based port mapping inference are to select suitable microbenchmarks and to find a port mapping that fits this indirect characterization.

The port mapping inference algorithms presented in this thesis are based on a formal port mapping model. The core of this model is a linear program that describes how a processor's port mapping determines the instruction throughput of dependency-free instruction sequences.

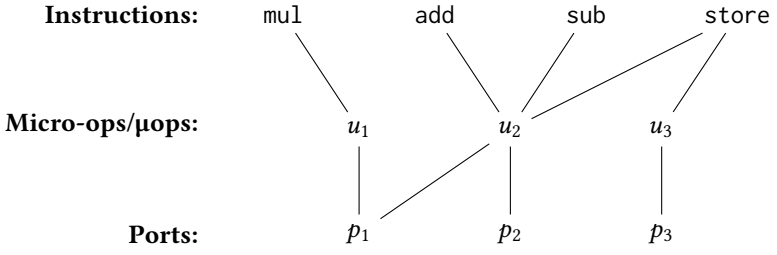


Figure 1.1. Example of a port mapping. Instructions are connected to the µops that implement them and µops are connected to the ports that can execute them.

In contrast to prior presentations of this linear program, this dissertation substantiates the model with a formal derivation from first principles and a correctness proof. We investigate the accuracy and the limitations of the port mapping model with a study that compares its throughput estimates to measurements on a microarchitecture with a known port mapping. Based on the port mapping model, we develop three port mapping inference algorithms that represent different trade-offs between applicability and accuracy.

The first port mapping inference algorithm uses a satisfiability modulo theories (SMT) solver to leverage the port mapping formalization directly. It incrementally constructs a set of microbenchmarks that determine the processor's port mapping. In each step, the algorithm finds a port mapping candidate that explains the measured throughputs for the current benchmark set. Then, it searches for a second port mapping explaining the collected measurements and a new microbenchmark where the candidates would yield different throughputs. If one is found, the new microbenchmark is carried out and added to the set. The algorithm terminates once it finds a fitting port mapping for the collected benchmarks for which no second candidate with a distinguishing microbenchmark exists.

Of the presented inference methods, this counter-example-guided algorithm provides the strongest guarantees. If the processor under investigation follows the port mapping model, the algorithm finds a port mapping that cannot be distinguished from the correct one via throughput measurements. However, applying the algorithm to inference problems at a practically relevant scale leads to prohibitively slow inference times.

The second approach, PMEvo, is more practical. Here, we address the key challenges of throughput-based port mapping inference approximatively. The microbenchmarks follow a fixed and potentially incomplete strategy that investigates instructions only individually and in pairs. We employ a randomized evolutionary optimization algorithm to obtain a port mapping that minimizes the difference between modeled and observed throughputs in these benchmarks.

These approximations allow PMEvo to infer port mappings for real-world microarchitectures like Intel's Skylake, AMD's Zen+, and ARM's A72 without relying on dedicated performance counters. The resulting port mappings match the observed throughputs, but they do not necessarily follow the actual structure of the hardware. They are therefore suitable for analytical throughput modeling, but may be lacking when the goal is to analyze, e.g., the utilization of specific processor resources.

The final inference algorithm finds port mappings that follow the structure of the hardware – as far as it is documented – more closely. To establish confidence in the resulting port mappings, the algorithm justifies each inferred µop decomposition with microbenchmarks that can be validated manually. This algorithm combines our SMT-based counter-example-guided inference algorithm with the high-level structure of the performance-counter-based strategy by Abel and Reineke (2019). Compared to the other two inference algorithms, this approach has stronger requirements on the processor under investigation: For each µop in the processor's port mapping, there has to be an instruction that is implemented only with this µop. Furthermore, there needs to be a hardware performance counter for the total number of executed µops in a microbenchmark.

We demonstrate that this inference method scales to practical problem sizes with an application to AMD's Zen+ architecture. The result is, to the best of our knowledge, the most comprehensive and accurate port mapping available for Zen+.

Analyzing Microarchitectural Performance Models

The performance models of microarchitectural code analyzers like llvm-mca (Di Biagio, 2018), OSACA (Laukemann *et al.*, 2018), and uiCA (Abel and Reineke, 2022) are one of the main applications of inferred port mappings. These tools model how a processor executes a given machine instruction sequence and estimate the achieved instruction throughput. When they are configured to model the same microarchitecture, one might assume that they produce similar throughput estimations. However, we found in experiments that substantial inconsistencies in their predictions are very common.

Inconsistently estimated throughputs can have a variety of reasons: The individual performance models may not capture relevant parts of the execution, the tools might rely on different, possibly implicit assumptions, or they might contain bugs. Understanding the inconsistencies is vital for improving the tools and for determining their practical limitations. Gaining such an understanding by investigating individual input instruction sequences is arduous: Any subset of the instructions, their interactions, and their individual features, e.g., if they access memory, use a particular functional unit, or belong to a special instruction set extension, may trigger the inconsistency.

We therefore develop AnICA to find inputs that exhibit inconsistencies in throughput predictors and to automatically determine responsible input features. AnICA provides a general framework to find interesting inputs through randomized differential testing (McKeeman, 1998) and to generalize them with a method based on abstract interpretation (Cousot and Cousot, 1977). We instantiate this framework for inputs in the form of x86-64 instruction sequences. Such instruction sequences are considered interesting if they exhibit inconsistent results in a pair of throughput predictors. AnICA's results are hence compact characterizations of classes of inconsistently predicted instruction sequences. These characterizations give high-level insights into differences between the tools.

Since our implementation treats the throughput predictors as black boxes, it is not limited to analyzing traditional microarchitectural code analyzers. It further supports machine-learning-based throughput predictors like Ithemal (Mendis *et al.*, 2019) and DiffTune (Renda *et al.*, 2020) and throughput measurements on the actual hardware. We explore a series of case studies where AnICA exposed subtle modeling differences between the tools and identified underrepresented constructs in the training sets of learned predictors. Furthermore, it pinpointed a long-standing crash in llvm-mca with a two-instruction test case and characterized several inaccuracies in llvm-mca's model for AMD's Zen+ microarchitecture. In this process, AnICA even found a quirk in the Zen+ microarchitecture itself.

Structure of the Thesis

Following this introduction, Chapter 2 reviews relevant aspects of computer architecture and performance modeling. The remaining dissertation relies on these concepts.

Our discussion of port mapping inference strategies begins in Chapter 3 with an introduction and experimental evaluation of the formal port mapping model. This formal model is integral for the following Chapters 4 to 6, which describe our three throughput-based port mapping inference algorithms. We discuss related work regarding the use and inference of port mappings in Chapter 7.

Chapter 8 presents AnICA, our approach to analyze inconsistencies in microarchitectural code analyzers. This chapter is independent of the previous chapters on port mapping inference.

Finally, we draw conclusions from the presented work and outline future research directions in Chapter 9.

Most of the chapters include formal statements about the presented constraint systems and algorithms. We present proofs for these statements in Appendix A. Appendix B summarizes pertinent terminology regarding satisfiability modulo theories and (integer) linear programming. In Appendix C, we provide a short reference of the registers of the x86-64 instruction set architecture to give additional background for the assembly examples in the thesis. Appendix D provides technical details of the microbenchmarking infrastructure used in evaluations for the thesis. Lastly, Appendix E discusses supplementary examples that show limitations of the related work. We refer to these appendices throughout the thesis when they are relevant.

Publications

This dissertation includes and expands upon results of the following articles:

- PMEvo: Portable Inference of Port Mappings for Out-of-Order Processors by Evolutionary Optimization, by Fabian Ritter and Sebastian Hack, published in the Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020. (Ritter and Hack, 2020)
- AnICA: Analyzing Inconsistencies in Microarchitectural Code Analyzers, by Fabian Ritter and Sebastian Hack, published in the Proceedings of the ACM on Programming Languages, OOPSLA 2022. (Ritter and Hack, 2022)
- Explainable Port Mapping Inference with Sparse Performance Counters for AMD's Zen Architectures, by Fabian Ritter and Sebastian Hack, published in the Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2024. (Ritter and Hack, 2024)

Chapter 2

Background on CPU Performance

This chapter covers the concepts and terminology in processor design and performance modeling that are relevant for the thesis.

2.1. Computer Architecture in a Nutshell

The capabilities of a processor are determined by the *instruction set architecture* (ISA) that it implements. The ISA describes what operations, so-called *instructions*, the processor can execute and their meaning. We distinguish *instruction instances* and *instruction schemes*: An *instruction instance* consists of a *mnemonic* to determine the performed operation and a sequence of *operands*, which determine the values it operates on. Operands can be the names of registers, memory references, or immediate constants. Each operand has a *width* in bits. An *instruction scheme*, also called *instruction form*, abstracts a set of instruction instances with a common mnemonic and matching operands. Instruction schemes can be instantiated with concrete operands to obtain instruction instances.

Example 2.1. Consider the following instruction schemes for the x86-64 ISA (Intel, 2023b) and possible instantiations. We represent operand positions of instruction schemes in angle brackets $\langle\cdot\rangle$ and annotate them with R, W, or RW in a subscript if the corresponding operand is read, written, or both, when executing the instruction. The x86-64 examples in this thesis follow Intel's assembly syntax, where destination operands come before source operands.[1]

(a) An addition of two 64-bit general-purpose registers (where the first operand is also the destination for the result):

$$\texttt{add}\ \langle\texttt{GPR[64]}\rangle_{RW}\texttt{,}\ \langle\texttt{GPR[64]}\rangle_{R}$$

It can be instantiated by specifying both register operands, for example:

```
add rbx, rdx
```

(b) An addition of an 8-bit immediate constant to a 64-bit number in memory:

$$\texttt{add}\ \langle\texttt{MEM[64]}\rangle_{RW}\texttt{,}\ \langle\texttt{IMM[8]}\rangle_{R}$$

For instantiation, this requires a memory operand and an immediate constant:

```
add qword ptr [rbx], 42
```

[1] See Appendix C for an overview of relevant registers and memory addressing in the x86-64 ISA.

(c) A vector addition of double-precision floating point values from a 128-bit vector register (`xmm0` – `xmm15`) and a 128-bit memory location:

$$\texttt{vaddpd}\ \langle \texttt{XMM} \rangle_W, \langle \texttt{XMM} \rangle_R, \langle \texttt{MEM[128]} \rangle_R$$

This instruction scheme can be instantiated with vector registers of the appropriate width and a memory operand, which can use several registers and constants:

```
vaddpd xmm3, xmm2, xmmword ptr [rax + 4*rdx + 8]
```

⌟

It is surprisingly difficult to determine how many (relevant) instruction schemes an ISA contains. On one hand, the number depends on how instruction instances are grouped into instruction schemes; e.g., how instructions with different operand widths or with special encodings are represented. On the other hand, ISAs like x86-64 have grown over decades with a long history of ISA extensions, some of which are relevant to this day whereas others are no longer supported by contemporary processors. For this work, we base our instruction schemes on the instruction tables used by Abel and Reineke (2019), from which we extract 4,042 instruction schemes. However, we observe only a fraction of these in practice: In our experiments, instances of only 1,036 different instruction schemes were executed when running the SPEC CPU 2017 benchmarks (Bucek *et al.*, 2018).

While the ISA defines what instructions are supported and what they compute, it does not specify *how* the instructions are implemented; this is determined by the processor's *microarchitecture*. Different processor models can use the same microarchitecture, and several different microarchitectures may implement the same instruction set architecture. Decisions in the design of a microarchitecture strongly affect the performance of processors that implement it. Modern microarchitectures are therefore engineering products of immense complexity whose details are kept confidential by the processor manufacturers.

In contrast to early in-order designs, most modern microarchitectures apply out-of-order execution (Hennessy and Patterson, 2017, Chapter 3). This concept is based on the observation that instructions can be executed in any order as long as the results are the same as if they were executed in program order. A processor may execute instructions in parallel and reorder them to any extent that preserves the read-after-write dependencies between the operations and the externally-visible effects specified by the ISA. Current processors are designed to manage several hundred "in-flight" instructions that are considered for reordering; e.g., 256 for AMD's Zen3 microarchitecture (AMD, 2020, Section 2.10.3).

Out-of-order execution is often combined with a scheme to decompose instructions into simpler microarchitecture-specific operations. These so-called *micro-ops* or *µops* are then subject to reordering.

Figure 2.1 shows the relevant parts of a microarchitecture that employs out-of-order execution and µop decomposition. Instructions are fetched and decoded from the instruction cache in program order. The decoder produces µops, which are cached for future re-use. The register management engine resolves write-after-read and write-after-write dependencies by *renaming* the ISA-level operand registers of each operation in terms of a larger number of physical

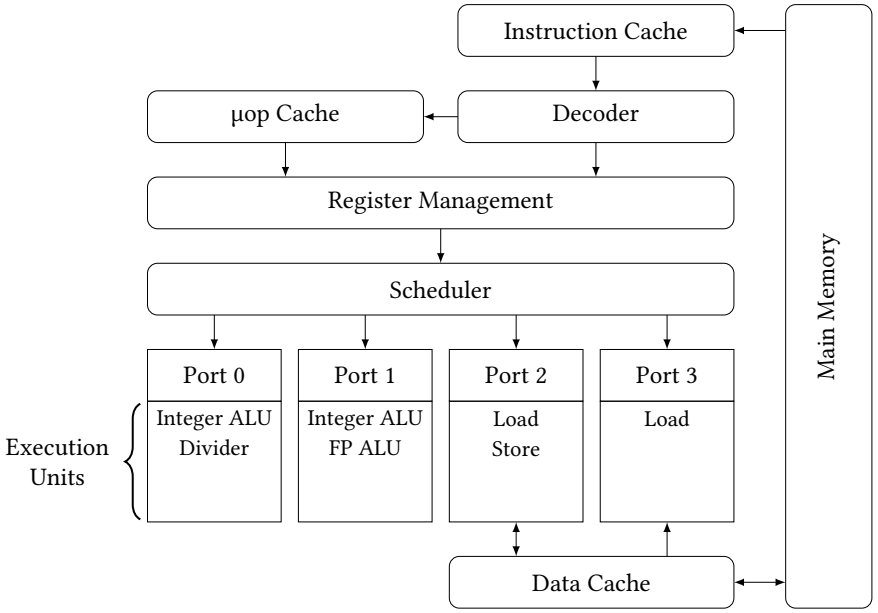


Figure 2.1. Simplified overview of a modern processor design (based on Figure 2-8 of the Intel Software Optimization Manual (Intel, 2023a, Section 2.6)).

registers. A scheduler decides based on operand dependencies and resource availability when and where to execute the μops.

The execution units, which execute the μops, are grouped behind *ports*.[2] Typical examples of execution units are arithmetic logic units (ALUs) for integer or floating point (FP) values, specialized units for complex operations like dividers and multipliers, and load/store units for accessing the system's main memory. Processors can have multiple instances of the same kind of execution unit available at different ports. The number of ports in contemporary x86-64 microarchitectures at the time of writing ranges from eight (e.g., for the Intel Skylake (Intel, 2023a, Section 2.6)) to twelve (e.g., Intel's Golden Cove microarchitecture (Intel, 2023a, Section 2.3)).

The processor operates in discrete steps of execution, so-called clock cycles or *cycles*. How many of these steps are performed per second is determined by the *clock frequency*. Execution units are often pipelined, allowing the ports to start processing a new μop in every cycle. Each μop can be executed on any port that has the necessary execution unit. Once an instruction or a μop has been executed completely, its results are available for subsequent operations and it is *retired*.

The scheme that determines how instructions are decomposed into μops and on which ports those can be processed is the *port mapping* of the microarchitecture. As the port mapping

[2] AMD usually refers to similar constructs in their processors as "pipes"; we use the term "ports" with respect to microarchitectures of any manufacturer.

controls the possible instruction-level parallelism, i.e., to what degree instructions of a single execution thread can be executed in parallel, it plays a significant part in determining how fast a processor can execute given instruction sequences. How exactly the port mapping relates to the execution rate is central to a large portion of this thesis and will be discussed in greater detail in the following chapters.

Computer architects have developed many more techniques to further improve the performance of their processor designs, for instance:

- Modern multicore processors contain multiple processor cores that each include an independent processing pipeline as described above.
- If a processor core implements simultaneous multithreading or hyper-threading, its µop scheduler receives operations from multiple independent instruction streams.
- The processor core can predict the likely outcome of branches and execute subsequent instructions speculatively. In case of a misspeculation, the execution units can be utilized by operations that are not part of the execution specified by the ISA.

These techniques are relevant for the processor's overall performance, but they are not in the scope of this thesis. We only consider individual processor cores for which simultaneous multithreading is disabled and benchmarks where misspeculations are negligible.

Modern processors give software developers insight into their operation through a performance monitoring unit. This unit provides an additional set of registers that can be configured to count occurrences of certain predefined events in the processor. These *hardware performance counters* give statistics on how the processor executes code without affecting the execution itself. They could for instance be configured to count the executed instructions and µops, cache hits and misses, or the number of executed branch instructions. What events can be counted depends on the microarchitecture. Recent Intel processors can count the number of µops executed on each individual port, which is a feature of vital importance for related work by Abel and Reineke (2019) that we will discuss in later chapters. Tools like nanoBench (Abel and Reineke, 2020) and LIKWID (Gruber *et al.*, 2023; Treibig *et al.*, 2010) provide interfaces to configure and access these performance counters.

2.2. Performance Modeling

The complexity of modern processors, combined with the incomplete information provided by the manufacturers, makes estimating how fast a specific program will execute on a given processor challenging. Performance modeling therefore is an active research area, with a wide range of tools and approaches. There are performance models at various scales – from short sequences of machine instructions (Intel, 2012) to entire programs (Binkert *et al.*, 2011) – and aiming for various levels of accuracy – from cycle-accurate (Böhm *et al.*, 2010) to "back-of-the-envelope calculations" (Ofenbeck *et al.*, 2014; Williams *et al.*, 2009).

In this thesis, we focus on performance models that describe how a core of a central processing unit (CPU) executes a single instruction stream and that aim at being cycle accurate.

There are different ways to define a notion of performance in this setting. For instance, one can be interested in the *latency* of a finite instruction sequence. This is the time (or number of cycles) required to execute the instruction sequence from start to end. For long-running systems, measures of the steady-state execution rate are commonly used as performance metrics. In the context of this thesis, the following execution rate measures are of particular interest:

Definition 2.2. Consider a sequence B of n instructions that is executed in indefinite repetition.

- The *throughput* $tp(B)$ is the average number of instances of B executed per clock cycle.
- The *inverse* or *reciprocal throughput* $tp^{-1}(B)$ is the average number of clock cycles required to execute an instance of B:
$$tp^{-1}(B) = \frac{1}{tp(B)}$$
- The *IPC* is the average number of instructions executed per clock cycle:
$$IPC(B) = tp(B) \cdot n = \frac{n}{tp^{-1}(B)}$$
- The *CPI* is the average number of clock cycles required to execute an instruction:
$$CPI(B) = \frac{1}{IPC(B)} = \frac{tp^{-1}(B)}{n}$$

⌟

In this thesis, we usually refer to the execution rate in terms of the inverse throughput $tp^{-1}(B)$ and clarify whenever we deviate. We formalize how and under which conditions the inverse throughput $tp^{-1}(B)$ is determined by the processor's port mapping in the following chapter.

Chapter 3

Port Mapping Inference: The Basics

A major focus of this thesis lies on techniques to infer port mappings, which model how a processor decomposes instructions into micro-operations (µops) and how these are executed. This chapter expands on the formal foundations of the port mapping model that we introduced in the article on PMEvo (Ritter and Hack, 2020). We further provide evidence that the port mapping model captures the behavior of modern processors in the absence of data dependencies and discuss limitations in Section 3.2. Lastly, we formally define port mapping inference problems and present results on their computational complexity in Section 3.3.

The presented port mapping model serves as a foundation for the subsequent chapters of this thesis, where we provide ways to approach the port mapping inference problem.

3.1. Formal Model

We present two port mapping models: The simpler two-level model is convenient to understand the techniques we developed, while the more complex three-level model is closer to the behavior of real processors. Both models describe how a given set $\mathbf{I}$ of instruction schemes is mapped to a given set $\mathbf{P}$ of execution ports of a microarchitecture.

In general, the throughput a given processor achieves for a piece of code does not only depend on the port mapping. Other aspects of the hardware like restrictions in the instruction-decoding frontend of the processor, non-pipelined functional units, and varying latencies of memory accesses also affect performance. Furthermore, properties of the code such as data dependencies contribute substantially to the code's overall execution rate. The presented model only aims to accurately describe the throughput of code whose performance is solely determined by the processor's port mapping. This goal affects the following definition.

Definition 3.1. An *experiment* e is an unordered multiset of instruction schemes. It is represented as a function $e : \mathbf{I} \to \mathbb{N}$ that maps instruction schemes to their numbers of occurrences in the experiment e. ⌟

This definition captures two noteworthy modeling choices: It only considers instruction schemes, which abstract from specific operands, and the experiments do not specify an order among their instruction schemes. The port mapping model therefore does not distinguish experiments based on the specific instances of instruction schemes or the instruction order. This is motivated by our focus on purely port-mapping-bound code, which allows us to assume that

- all instruction instances of a given instruction scheme can be executed on the same functional units with the same performance characteristics and that
- the code is free of dependencies, which enables modern out-of-order processors to freely reorder the instructions when executing them.

We discuss limitations induced by these assumptions and how to measure the throughput of such experiments on a given processor in Section 3.2.

3.1.1. The Two-Level Model

In the first model, the *two-level model*, we assume that each execution port can execute certain kinds of instructions directly. The following definition captures this notion:

Definition 3.2. A *port mapping M in the two-level model* is a bipartite graph $(\mathbf{I} \cup \mathbf{P}, E)$ with the vertices split disjointly into finite, non-empty sets $\mathbf{I}$ of instruction schemes and $\mathbf{P}$ of ports. The edges $E \subseteq \mathbf{I} \times \mathbf{P}$ connect instruction schemes with their ports.

We refer to the ports assigned to an instruction scheme i by $M[i] := \left\{k \;\middle|\; (i,k) \in E\right\}$. We require that for all $i \in \mathbf{I}$, $M[i] \neq \emptyset$, i.e., every instruction scheme has at least one port that can execute it. ⌟

For an example, consider Figure 3.1. The shown two-level port mapping covers four instruction schemes – `mul`, `add`, `sub`, `store` – of a hypothetical instruction set architecture. Only one execution port, p_1, can handle `mul` instructions. For `add` and `sub` instructions, there are more choices: The ports p_1 and p_2 are both able to execute such instructions. Instances of the `store` instruction scheme need to be executed on another port, p_3.

In the two-level model, the processor executes an occurrence of an instruction scheme i by assigning it to one of the ports in $M[i]$. We assume that execution ports are fully pipelined, which means that they can start executing a new instruction in every cycle. Therefore, when an instruction scheme occurrence of an experiment is assigned to a port, it occupies this port exclusively for exactly one cycle. The following definition captures this execution behavior formally:

Definition 3.3. An *execution schedule* for a positive number $N \in \mathbb{N}^+$ of iterations of an experiment e according to the two-level port mapping $M = (\mathbf{I} \cup \mathbf{P}, E)$ is a sequence $Ex := [s_1, \ldots, s_{|Ex|}]$ of partial allocation functions $s_c : \mathbf{P} \rightharpoonup \mathbf{I}$ such that

- all instruction scheme occurrences from each experiment iteration are assigned to a port at some point in the sequence:

$$\forall i \in \mathbf{I}. \left|\left\{(c,k) \;\middle|\; 1 \leq c \leq |Ex| \wedge k \in \mathbf{P} \wedge s_c(k) = i\right\}\right| = N \cdot e(i)$$

- instruction schemes are only assigned to ports that M allows:

$$\forall 1 \leq c \leq |Ex|, i \in \mathbf{I}, k \in \mathbf{P}. \left(s_c(k) = i\right) \Rightarrow k \in M[i]$$

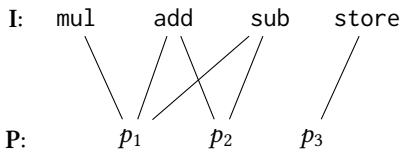


Figure 3.1. A port mapping in the two-level model. (Ritter and Hack, 2020)

We call the length $|Ex|$ of an execution schedule Ex its *execution time*. An *optimal execution schedule* of N iterations of an experiment e according to the port mapping M is an execution schedule with a minimal execution time. We refer to the execution time of such an optimal execution schedule as $T(M, e, N)$. ⌟

This processor model can be viewed from a scheduling perspective: The execution of instructions in a processor corresponds to a scheduling problem. Instructions correspond to independent jobs and the ports are unrelated parallel machines on which a given instruction can either be executed with processing time 1 (if the port mapping allows it), or it cannot be executed (i.e., it has a processing time of ∞).[1]

Example 3.4. Consider the experiment $\{\mathsf{add} \mapsto 2, \mathsf{mul} \mapsto 1, \mathsf{store} \mapsto 1\}$ and the two-level port mapping from Figure 3.1. For a single iteration ($N = 1$), the following is an optimal execution schedule:

$$\big[\{p_1 \mapsto \mathsf{mul}, p_2 \mapsto \mathsf{add}, p_3 \mapsto \mathsf{store}\},\ \{p_1 \mapsto \mathsf{add}\}\big]$$

It has entries for two cycles, therefore the optimal execution time $T(M, e, 1)$ is 2.

For two iterations, i.e., $N = 2$, duplicating the above execution schedule yields a valid schedule with execution time 4:

$$\big[\{p_1 \mapsto \mathsf{mul}, p_2 \mapsto \mathsf{add}, p_3 \mapsto \mathsf{store}\},\ \{p_1 \mapsto \mathsf{add}\},\ \{p_1 \mapsto \mathsf{mul}, p_2 \mapsto \mathsf{add}, p_3 \mapsto \mathsf{store}\},\ \{p_1 \mapsto \mathsf{add}\}\big]$$

However, this execution schedule is not optimal as the iterations can be interleaved to achieve an execution time of 3 cycles:

$$\big[\{p_1 \mapsto \mathsf{mul}, p_2 \mapsto \mathsf{add}, p_3 \mapsto \mathsf{store}\},\ \{p_1 \mapsto \mathsf{add}, p_2 \mapsto \mathsf{add}, p_3 \mapsto \mathsf{store}\},\ \{p_1 \mapsto \mathsf{mul}, p_2 \mapsto \mathsf{add}\}\big]$$

⌟

[1] See, for example, the textbook by Pinedo (2022) for additional background on the classification of scheduling problems.

With this notion of how instructions are executed, we define the throughput that can be achieved with a given port mapping for an experiment as follows:

Definition 3.5. The *modeled inverse throughput* $tp_M^{-1}(e)$ of an experiment e with a two-level port mapping M is the limit

$$\lim_{N \to \infty} \frac{T(M, e, N)}{N}$$

⌟

Theorem 3.6. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment e with a port mapping M always exists, and it is the infimum of the set $\left\{ \frac{T(M,e,N)}{N} \;\middle|\; N \in \mathbb{N}^+ \right\}$.

Proof. See Appendix A.1.1. □

The throughput characterization of Definition 3.5 closely corresponds to the throughput notion discussed in Definition 2.2. It does, however, not provide an obvious constructive method to compute the inverse throughput given a port mapping and an experiment. For this purpose, we use the following alternative throughput formulation based on a linear program (LP), which has first been presented in similar form by Abel and Reineke (2019).[2]

Theorem 3.7. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment $e : I \to \mathbb{N}$ with a port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ is the objective value of an optimal solution to the following linear program:

$$\begin{array}{lll}
\text{minimize} & t & \\
\text{subject to} & \displaystyle\sum_{k \in \mathbf{P}} x_{ik} = e(i) & \text{for all instructions } i \in \mathbf{I} \qquad \text{(A)} \\
& \displaystyle\sum_{i \in \mathbf{I}} x_{ik} = p_k & \text{for all ports } k \in \mathbf{P} \qquad \text{(B)} \\
& p_k \leq t & \text{for all ports } k \in \mathbf{P} \qquad \text{(C)} \\
& x_{ik} \geq 0 & \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} \qquad \text{(D)} \\
& x_{ik} = 0 & \text{if } (i,k) \notin E \qquad \text{(E)}
\end{array}$$

In particular, this linear program is feasible and has a finite optimal objective value.

Proof. See Appendix A.1.2. □

[2] See Appendix B for an overview of the terminology regarding linear programming used in this thesis.

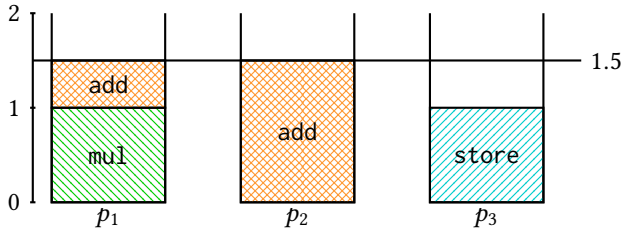


Figure 3.2. Visualization of an optimal solution of the linear program. (Ritter and Hack, 2020)

In other words, $tp_M^{-1}(e)$ is the minimal value that the variable t can have in any valuation of the LP variables that satisfies constraints (A) – (E). This formulation is based on the intuition that an experiment e specifies for each instruction scheme i a corresponding mass $e(i)$ that needs to be distributed among the ports as evenly as possible. The real-valued variables x_{ik} capture the share of the mass $e(i)$ for the instruction scheme i that is assigned to the port k. The constraints (A) and (D) ensure that these variables represent a proper decomposition of the mass of the experiment. With constraint (B), we require that the real-valued variables p_k carry the total mass of instructions assigned to their corresponding port k. Constraint (E) restricts the assignment of each instruction scheme i's mass to ports k that can execute i.

Constraint (C) defines t as an upper bound to the individual per-port masses. Therefore, the objective to minimize t ensures that an optimal solution to the LP distributes the mass among the ports such that the maximal per-port load is minimized.

Example 3.8. We revisit the experiment $\{\text{add} \mapsto 2, \text{mul} \mapsto 1, \text{store} \mapsto 1\}$ and the two-level port mapping from Figure 3.1 used in Example 3.4. Figure 3.2 visualizes a solution to the linear program from Theorem 3.7. The value of each variable x_{ik} is represented as the height of a block for the instruction scheme i in the bucket for port k. The maximal fill of a bucket, 1.5, is equal to the inverse throughput we found in Example 3.4. ⌟

In contrast to the original Definition 3.5, we can use the linear program from Theorem 3.7 to compute the throughput of an experiment as implied by a given port mapping. Linear programs can be solved efficiently, in polynomial time.[3]

A noteworthy special case is the throughput of an instruction scheme i executed in isolation via a *singleton experiment* $\{i \mapsto 1\}$. The corresponding mass is distributed equally among the ports that can execute i, leading to an inverse throughput of $\frac{1}{|M[i]|}$ for any port mapping M.

3.1.2. The Three-Level Model

The two-level model of the previous section simplifies the inner workings of real-world processors in many ways. A critical implementation detail, which we incorporate now, is that modern processors usually decompose instructions into simpler µops to execute them more

[3] See, e.g., the textbook by Bertsimas and Tsitsiklis (1997). We present an alternative way to compute the solution of this linear program with superior performance for practical cases in Section 5.1.4.

efficiently. The three-level model takes this into account by introducing a set **U** of µops as a middle layer to the model.

Definition 3.9. A *port mapping in the three-level model* is a tripartite graph $(\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, F \cup E)$ with disjoint, finite, and non-empty sets **I** of instruction schemes, **U** of µops, and **P** of ports. The labeled edges $F \subseteq \mathbf{I} \times \mathbb{N} \times \mathbf{U}$ connect instruction schemes with µops and the unlabeled edges $E \subseteq \mathbf{U} \times \mathbf{P}$ connect µops with ports.

We require that every µop $u \in \mathbf{U}$ has at least one port $k \in \mathbf{P}$ that can execute it:

$$\forall u \in \mathbf{U}.\ \left\{k \;\middle|\; (u,k) \in E\right\} \neq \emptyset$$

⌟

A labeled edge $(i,n,u) \in F$ in a three-level port mapping means that the µop u occurs n times in the µop decomposition of the instruction scheme i.

Figure 3.3 shows an example for such a three-level port mapping. Here, `add` and `sub` are implemented as one µop u_2 that can be executed on two ports p_1 and p_2. The `mul` and `store` instructions are decomposed into two µops, the former into two of the same kind, u_1, and the latter into two different ones, u_2 and u_3. The `store` instruction has a partial conflict with `add` and `sub` that cannot be represented in the two-level model.

It is important to note the different semantics of the layers of edges: To execute an occurrence of an instruction scheme i, *all* corresponding µops u such that $(i,n,u) \in F$ have to be executed whereas a µop u is executed on *exactly one* of the allowed ports k with $(u,k) \in E$.

We define execution schedules and the modeled throughput analogously to the two-level model.

Definition 3.10. An *execution schedule* for a positive number $N \in \mathbb{N}^+$ of iterations of an experiment e according to the three-level port mapping $M = (\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, F \cup E)$ is a sequence $Ex := [s_1, \ldots, s_{|Ex|}]$ of partial allocation functions $s_c : \mathbf{P} \rightharpoonup \mathbf{U}$ such that

- the µops of all instruction scheme occurrences from each iteration are assigned to a port at some point in the sequence:

$$\forall u \in \mathbf{U}.\ \left|\left\{(c,k) \;\middle|\; 1 \leq c \leq |Ex| \wedge k \in \mathbf{P} \wedge s_c(k) = u\right\}\right| = N \cdot \sum_{(i,n,u) \in F} e(i) \cdot n$$

- µops are only assigned to ports that M allows:

$$\forall 1 \leq c \leq |Ex|, u \in \mathbf{U}, k \in \mathbf{P}.\ (s_c(k) = u) \Rightarrow (u,k) \in E$$

We call the length $|Ex|$ of an execution schedule Ex its *execution time*. An *optimal execution schedule* of N iterations of an experiment e according to the port mapping M is an execution schedule with a minimal execution time. We refer to the execution time of such an optimal execution schedule as $T(M,e,N)$. ⌟

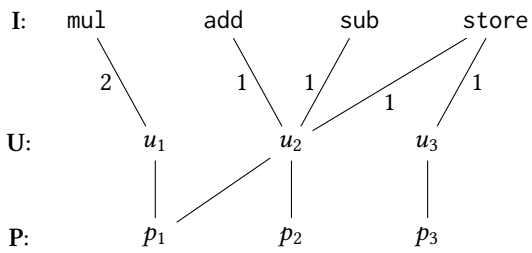


Figure 3.3. Example of a three-level port mapping. (Ritter and Hack, 2020)

Definition 3.11. The *modeled inverse throughput* $tp_M^{-1}(e)$ of an experiment e with a three-level port mapping M is the limit

$$\lim_{N\to\infty} \frac{T(M,e,N)}{N}$$

⌟

The linear program from Theorem 3.7 can be slightly modified to compute the inverse throughput $tp_M^{-1}(e)$ of an experiment $e : \mathbf{I} \to \mathbb{N}$ under the three-level port mapping $M := (\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, F \cup E)$.

Theorem 3.12. Given a three-level port mapping $M := (\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, F \cup E)$, the modeled inverse throughput $tp_M^{-1}(e)$ under M for an experiment $e : I \to \mathbb{N}$ is the objective value of an optimal solution to the following linear program:

$$\begin{array}{lll}
\text{minimize} & t & \\
\text{subject to} & \displaystyle\sum_{k\in\mathbf{P}} x_{uk} = \sum_{(i,n,u)\in F} e(i)\cdot n & \text{for all } u \in \mathbf{U} \quad \text{(A)} \\
& \displaystyle\sum_{u\in\mathbf{U}} x_{uk} = p_k & \text{for all ports } k \in \mathbf{P} \quad \text{(B)} \\
& p_k \le t & \text{for all ports } k \in \mathbf{P} \quad \text{(C)} \\
& x_{uk} \ge 0 & \text{for all µops } u \in \mathbf{U}, \text{ ports } k \in \mathbf{P} \quad \text{(D)} \\
& x_{uk} = 0 & \text{if } (u,k) \notin E \quad \text{(E)}
\end{array}$$

In particular, this linear program is feasible and has a finite optimal objective value.

Proof. Analogous to Theorem 3.7. □

In this linear program, all occurrences of instruction schemes in the two-level LP are replaced by occurrences of µops except for the right-hand side of constraint (A). The right-hand side of (A) ensures that a µop u that occurs n times in the decomposition of the instruction scheme i is taken into account with its appropriate mass.

Remark 3.13. We note two valuable observations about this model:

1. Computing the inverse throughput of an experiment $e : \mathbf{I} \to \mathbb{N}$ with a port mapping $(\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, F \cup E)$ in the three-level model can be reduced to computing the inverse throughput in the two-level model: We instead compute the throughput of the experiment

$$e' = \left\{ u \mapsto \sum_{(i,n,u) \in F} e(i) \cdot n \right\}$$

   with the two-level mapping $(\mathbf{U} \cup \mathbf{P}, E)$. The multiset e' contains the µops that are needed to execute e according to F. These µops are used as instruction schemes for the two-level model. This construction allows us to use algorithms for the simpler two-level model to compute the inverse throughput in the three-level model.

2. Without loss of generality, we can identify µops by the set of ports that they can be executed on, and define $\mathbf{U}$ as the power set $\mathcal{P}(\mathbf{P})$ of the set of ports, excluding the empty set: $\mathbf{U} := \mathcal{P}(\mathbf{P}) \setminus \{\emptyset\}$.

⌟

Lastly, we note that the execution of any experiment in either of the port mapping models is determined by a set of fully utilized bottleneck ports. We will harness this insight throughout the following chapters.

Theorem 3.14. Let M be a two-level or three-level port mapping for a set $\mathbf{P}$ of ports and let e be an experiment with inverse throughput $t^* = tp_M^{-1}(e)$. Let S be the set of all optimal solutions to the corresponding linear program from Theorem 3.7 or Theorem 3.12. Then, there is a non-empty set $BP_M(e) := \bigcap_{s \in S} \{k \mid s[p_k] = t^*\}$ of *bottleneck ports*.

Proof. See Appendix A.1.3. □

For an example, reconsider the LP solution from Example 3.8, again displayed in Figure 3.4. When executing this experiment, p_1 and p_2 are the bottleneck ports. They are both fully utilized for the 1.5 cycles required for the experiment and none of the instructions assigned to p_1 or p_2 could be executed on the remaining port p_3. The only way to lower the utilization of one of the bottleneck ports is by increasing the utilization of the other, leading to a slower execution.

3.2. Evaluating the Port Mapping Model

The port mapping model defined in the previous section abstracts from the behavior of actual processor hardware. Before we derive algorithms to characterize processors based on this model, we should therefore investigate its accuracy. The objective of this section is to measure the inverse throughput of experiments on the actual processor and to check if the results are consistent with the three-level port mapping model.

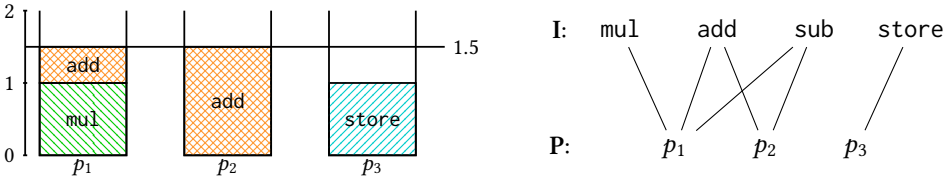


Figure 3.4. Port mapping in the two-level model and corresponding optimal port utilization for the experiment $\{\text{add} \mapsto 2, \text{mul} \mapsto 1, \text{store} \mapsto 1\}$. (Ritter and Hack, 2020)

We first describe a method to measure the steady-state throughput for instruction scheme multisets on a given processor. Then, we compare throughput measurements on an Intel Skylake processor with predictions of the port mapping model. Since Intel's Skylake microarchitecture provides the hardware performance counters that Abel and Reineke (2019) need for their port mapping inference method, there is a reference model available from uops.info for comparison. We present our measurement method in terms of the x86-64 ISA, but it can be applied to other instruction set architectures as well.

3.2.1. Benchmarking the Experiment Throughput

Accurate throughput measurements are key for our goal of inferring implementation details of the microarchitecture. Our measurement strategy follows the intuition of Definitions 3.5 and 3.11: We measure the total number of cycles required to execute a large number N of iterations of an experiment and divide it by N to obtain the inverse throughput for the experiment.

There are a number of practical issues to address when implementing this strategy. Our goal is to measure the number of cycles required to execute a multiset of independent instruction schemes in a steady state. We approach this challenge in two steps: Firstly, a processor does not execute multisets of instruction schemes but lists of instruction instances with concrete operands. We therefore need a strategy to instantiate instruction schemes while avoiding data dependencies. The second step is to benchmark the steady-state execution time of the resulting basic block while limiting the effect of performance bottlenecks other than the port usage. Figure 3.5 visualizes this benchmarking process with an example sequence of instruction schemes.[4] The following subsections discuss how to implement both steps. The described technique builds upon the measurement method we used in the article on PMEvo (Ritter and Hack, 2020) and work by Weis (2019).

Instantiating Instruction Schemes for Port-Mapping-Bound Experiments

Experiments as defined in Definition 3.1 are multisets of instruction schemes. As a first step to instantiate an experiment, we fix an arbitrary order among the instruction schemes to obtain a list of instruction schemes. Since the goal is to instantiate the instruction schemes

[4] See Appendix C for an overview of the relevant registers of the x86-64 ISA.

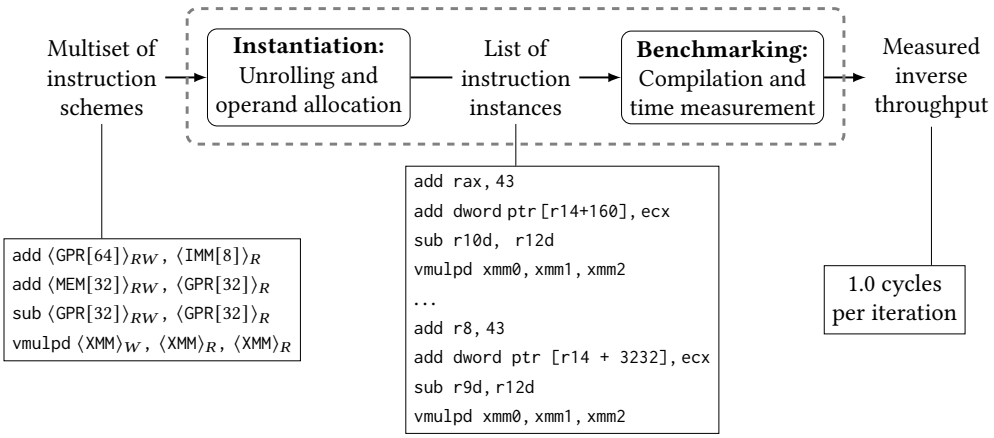


Figure 3.5. Benchmarking workflow with an example input of two read-modify-write additions (one adding an 8-bit immediate constant to a 64-bit register and one adding a 32-bit register value to a 32-bit value in memory), a 32-bit read-modify-write subtraction of one register operand from another, and a vector multiplication of pairs of double-precision floating point values.

such that data dependencies do not affect the throughput, any order is adequate. We test this assumption experimentally in Section 3.2.2.

Next, we allocate concrete operands for each instruction scheme to obtain an executable sequence of instruction instances from the list of instruction schemes. In contrast to code generation in compilers, there is no specific semantics that the operand selection needs to achieve. Instead, the objective is to choose valid operands that avoid erroneous input values for the instructions and that minimize the effect of data dependencies on the throughput.

In the x86-64 instruction set architecture, there are three kinds of operands that we need to consider:

Register operands: Each register operand of an instruction scheme has a group of allowed registers. In the instruction schemes that we use, the registers in such a group are all of the same kind, e.g., only 32-bit general-purpose registers, or only 256-bit vector registers. Register operands can occur as read, written, or read-and-written operands.

Memory operands: Memory operands represent addresses in the system's main memory. They can be expressions of the form

$$[segment : base + scale * index + displacement]$$

where *base* and *index* are general-purpose registers, *scale* and *displacement* are constants, and *segment* is a segment register mainly used in legacy execution modes.[5] Most of these components are optional; our benchmarking routine only uses the *base* and

[5]Recent AVX ISA extensions also allow vector registers in memory operands, we do not consider these in this thesis.

displacement components. Memory operands in instruction schemes also have a fixed bit-width and can occur as read, written, or read-and-written operands.

Immediate operands: An immediate constant is an operand with a fixed value that is encoded as part of the machine code. It has a fixed bit-width and cannot be overwritten.

Of these operand kinds, immediate constants are the easiest to safely instantiate. Since they are only read, they cannot cause data dependencies and their value cannot change when their instruction is executed repeatedly. We therefore instantiate W-bit-wide immediate operands with the fixed value $2^{W-8} + 42$. This is an arbitrary value with convenient properties: The most and least significant bytes are non-zero (ensuring that the constant is not encoded with fewer bits by the assembler) and it avoids values for which one might expect special case handling, like 0, ±1, and maxima or minima of the value range.

The other operand kinds require more care to avoid erroneous values, i.e., values that cause hardware exceptions or lead to untypical performance corner cases. For instance, memory operands always need to point to previously allocated memory regions to avoid memory access faults. Some instructions impose further requirements to their operands: For example, division operations should not receive 0 as a divisor.

Avoiding erroneous values is non-trivial as executing instructions can change the values of their operands. To ensure that no such values occur even when the instantiated code is repeated many times, we reserve registers to carry specific values, like a valid memory base address or a small positive divisor. These registers are not used as written operands of the benchmarked instructions and therefore retain their values throughout the entire microbenchmark. For floating point instructions, subnormal operand values, which require a special encoding and a special treatment by the hardware, can cause performance penalties. We avoid such penalties by configuring the hardware to treat subnormal values as zero.

Exploring in how far operand values affect the execution – and, thus, the throughput – of specific instructions is outside of the scope of this thesis. We therefore omit instruction schemes for which we cannot ensure consistent throughput measurements from the port mapping inference algorithms in the following chapters. For an exploration of such input-dependent behavior we refer to work by Biehl (2021).

Concerning data dependencies, we only need to avoid read-after-write dependencies since write-after-write and write-after-read dependencies are resolved via register renaming in modern out-of-order processors.[6] To avoid read-after-write dependencies, an ideal strategy would avoid reading any written value. This would require that we separate the available registers and memory locations into two groups, one of which is only read whereas the other is only written. In practice, at least for the x86-64 instruction set, this is not possible as this ISA contains many instructions that use an operand both for reading and writing. We therefore partition operand candidates into three sets $\mathbb{O}_R, \mathbb{O}_W, \mathbb{O}_{RW}$: those only used for reading, those only used for writing, and those only used in read-and-written operands. If there are multiple operand candidates that refer to the same register or memory location (we say that they *alias*), only one representative for them is included in any of the sets. For example, of the x86-64 register operands `rax`, `eax`, `ax`, `ah`, and `al`, only one of them may be contained as a

[6] See Section 2.1.

Input: List *schemes* of instruction schemes
1 *resultBasicBlock* ← []
2 *currentTimeStamp* ← 0
3 *lastUses* ← mapping of operand candidates to the timestamp 0
4 **for** $i \in$ *schemes* **do**
5 *chosenOperands* ← {}
6 **for** $op \in i.operands$ **do**
7 *currentTimeStamp* ← *currentTimeStamp* + 1
8 **if** *op is a W-bit immediate operand* **then**
9 $chosenOperands[op] \leftarrow 2^{W-8} + 42$
10 **continue**
11 $\mathbb{O} \leftarrow \begin{cases} \mathbb{O}_R & \text{if } op \text{ is read but not written} \\ \mathbb{O}_W & \text{if } op \text{ is written but not read} \\ \mathbb{O}_{RW} & \text{if } op \text{ is read and written} \end{cases}$
12 $candidates \leftarrow \{(o, r) \mid o \in op.validOperands \wedge o \text{ aliases with } r \in \mathbb{O}\}$
13 $chosen, representative \leftarrow \arg\min_{(o,r) \in candidates} lastUses[r]$
14 $chosenOperands[op] \leftarrow chosen$
15 $lastUses[representative] \leftarrow currentTimeStamp$
16 *resultBasicBlock.append*(*op.instantiate*(*chosenOperands*))
17 **return** *resultBasicBlock*

Algorithm 3.1. Allocating operands for a list of instruction schemes.

representative in any of the sets, since they all refer to different portions of the same register.[7] The memory operands are all derived from a single reserved memory base register with a range of different displacement offsets. This ensures that no memory operands in the $\mathbb{O}$ sets alias.

Since we cannot avoid data dependency chains between read-and-written operands, we instead introduce as many independent dependency chains between these operands in the benchmark as possible. While two instructions in the same dependency chain cannot be reordered, the processor can exploit instruction-level parallelism between the distinct dependency chains. We therefore choose $\mathbb{O}_R, \mathbb{O}_W$ with a small number of operand candidates just sufficient to instantiate each individual instruction scheme and use the remaining majority of operand candidates for $\mathbb{O}_{RW}$.

Algorithm 3.1 shows how we instantiate a list of instruction schemes with concrete operands. Each operand of each instruction scheme is considered individually and in order. Throughout the process, we keep a timestamp counter that is increased whenever a new operand is considered (lines 2, 7) and a mapping for each representative operand candidate to the

[7] Namely: the full 64 bits, the lower 32 bits, the lowest 16 bits, the second-to-lowest byte, and the lowest byte of the a register, respectively.

timestamp when it was last used (lines 3, 15). Immediate operands are initialized directly with a fixed value, as described above (line 9). For register or memory operands, we first determine whether they are read, written, or read and written, and select the appropriate set $\mathbb{O}$ from the operand candidate partition (line 11). Of the concrete operands that would fit at this position in the instruction scheme, we only consider those that are in $\mathbb{O}$ or alias with an operand in $\mathbb{O}$ (line 12). From this set, we pick the candidate whose representative has been used least recently (line 13).

Example 3.15. Consider the experiment from Figure 3.5:

$$\begin{aligned}
&\texttt{add}\ \langle \texttt{GPR[64]} \rangle_{RW}, \langle \texttt{IMM[8]} \rangle_{R} \\
&\texttt{add}\ \langle \texttt{MEM[32]} \rangle_{RW}, \langle \texttt{GPR[32]} \rangle_{R} \\
&\texttt{sub}\ \langle \texttt{GPR[32]} \rangle_{RW}, \langle \texttt{GPR[32]} \rangle_{R} \\
&\texttt{vmulpd}\ \langle \texttt{XMM} \rangle_{W}, \langle \texttt{XMM} \rangle_{R}, \langle \texttt{XMM} \rangle_{R}
\end{aligned}$$

We use the following partition of operand candidates:

$$\begin{aligned}
\mathbb{O}_W &= \{\texttt{rbx}, \texttt{xmm0}, [\texttt{r14} + 32]\} \\
\mathbb{O}_R &= \{\texttt{rcx}, \texttt{r12}, \texttt{xmm1}, \texttt{xmm2}, [\texttt{r14} + 96]\} \\
\mathbb{O}_{RW} &= \{\texttt{rax}, \texttt{r10}, \ldots\} \cup \{\texttt{xmm3}, \ldots\} \cup \{[\texttt{r14} + (32 + i \cdot 64)] \mid 2 \leq i < U\}
\end{aligned}$$

The upper bound U is chosen such that resulting memory addresses are spread among an allocated region, e.g., $U = 63$ for an allocation with the size of a usual 4 KiB page.

The first instruction scheme requires a read-and-written 64-bit general-purpose register and an 8-bit immediate operand. Since no register has been used before, the algorithm chooses any suitable register from $\mathbb{O}_{RW}$ and notes that it has been used at the current time stamp. The immediate constant is instantiated, as described above, with $2^{8-8} + 42 = 43$, resulting in the following instruction instance:

```
add rax, 43
```

The second instruction scheme reads and writes a memory operand and reads from a 32-bit register operand. It is therefore instantiated with a memory operand from $\mathbb{O}_{RW}$, namely $[\texttt{r14} + (32 + 2 \cdot 64)] = [\texttt{r14} + 160]$, and a 32-bit register that aliases with an element of $\mathbb{O}_R$:

```
add dword ptr [r14 + 160], ecx
```

Here, the last-used timestamps of the used memory operand and the `rcx` register are updated.

The `sub` instruction uses two 32-bit registers, the first for reading and writing, the second only for reading. Of the available registers for the first operand, `eax` cannot be chosen since it aliases with the `rax` instruction that has recently been used. Another register aliasing with an element of $\mathbb{O}_{RW}$ is chosen: `r10d`, the lower 32 bits of the `r10` register. Analogously, `r12d`, which aliases with `r12` from $\mathbb{O}_R$, is chosen for the second operand:

```
sub r10d, r12d
```

Lastly, the vector multiplication `vmulpd` needs one written and two read `xmm` vector registers, which the algorithm chooses from $\mathbb{O}_W$ and $\mathbb{O}_R$, respectively:

```
vmulpd xmm0, xmm1, xmm2
```

⌟

This instantiation strategy ensures that subsequent read-and-written operands do not alias, allowing the corresponding instructions to be reordered during execution. We further *unroll* the experiment before the operand allocation algorithm by duplicating the instruction schemes of the experiment multiple times. This way, even if the experiment consists only of a single instruction scheme with a read-and-written operand, we obtain enough instruction instances that can be executed independently. The necessary amount of unrolling depends on a variety of factors. We try for each experiment unrolling factors that lead to circa 40, 80, and 200 instructions in the loop body and use the result with the fastest execution rate per experiment instance.

While we cannot avoid data dependencies between the µops of a single instruction, they do not pose a problem for this instantiation strategy. Unrolling the benchmark ensures that there are enough independent instruction instances whose µops have no inter-instruction dependencies such that the execution of their µops can be interleaved.

However, the strategy falls short for instructions that read from and write to a hardwired operand, like the integer division instructions in the x86-64 ISA, which use the `a` and `d` registers to read the dividend and to write the result, or the add-with-carry instructions that read and write the carry flag register. Repeated instances of such an instruction scheme inherently form a dependency chain, which causes them to be executed sequentially, independent of their port mapping. Abel and Reineke (2019) insert dependency-breaking instructions when benchmarking such instructions. For example, a zeroing idiom like "`xor rax, rax`" sets the `rax` register to zero, allowing the processor's renaming system to use a different physical register for the `rax` register before and after the idiom. Since these blocking instructions can affect the observed throughput, we instead exclude instruction schemes with hardwired read and written operands for our port mapping inference strategies.

Hardwired operands that are either only written or only read do not come with this problem since they don't impose a true read-after-write dependency. We can just add the hardwired operands to the $\mathbb{O}_W$ or $\mathbb{O}_R$ set.

Measuring Basic Block Throughput

Once we have unrolled and instantiated an experiment, we need to measure the execution time of the resulting sequence of instruction instances in a steady state. In general, this is not a well-defined measure:

- The execution time of certain instructions like, e.g., division instructions can depend on their input values, which depends on the processor state at the beginning of the benchmarking process.

- How the steady state is achieved affects the execution time: Placement in a loop causes overhead by the loop code whereas long sequences of repeated instances of the code strain instruction buffers and caches.

- Memory accesses can cause execution time variations in several ways: Data cache misses come with substantial execution time penalties and the values of the involved address registers can affect whether two memory accesses alias (and therefore form a data dependency).

In the microbenchmarks that we use for port mapping inference, we address the first source of uncertainty by initializing all registers and every accessible memory location for the benchmark with arbitrary but fixed values. Since our goal is to gain insights into the port mapping rather than the instruction caches and the decoding phase, we need a long-running loop to achieve a steady state. As the addresses of all memory accesses are loop invariant by construction, this also addresses the effect of data cache misses: Eventually, all accessed memory locations of the loop body are resident in the L1 data cache and therefore do not incur additional delays. Since each accessed memory address differs by a constant offset from a common base register, no input-dependent aliasing can occur.

We implement this strategy in a benchmarking tool that emits the code to be benchmarked within a loop as inline assembly in a C program frame. When compiled and executed, the resulting C program runs the benchmarking loop long enough to achieve a steady state execution, typically 1 ms, and returns the observed execution rate based on time measurements and the processor's clock frequency. Appendix D describes this measurement method in more detail.

Our instantiation strategy can also be combined with alternative microbenchmarking tools, for instance nanoBench (Abel and Reineke, 2020), LIKWID (Gruber *et al.*, 2023; Treibig *et al.*, 2010), and the measurement tool of BHive (Chen *et al.*, 2019). While our tool only uses common Linux and POSIX features and a C compiler for the throughput measurements, these approaches explicitly use microarchitecture-specific performance counters. Our approach is therefore easier to adapt to new microarchitectures. For example, no change is required to switch between Intel's and AMD's recent x86-64 microarchitectures. This comes at the cost of an increased benchmarking time.

3.2.2. Evaluation of the Port Mapping Model

Our work builds on the hypothesis that the linear program from Theorem 3.12 with a processor's port mapping accurately models throughput measurements on the processor as described in the previous section. We test this hypothesis with the Intel Skylake microarchitecture, where a reference port mapping from uops.info (Abel and Reineke, 2019) is available.

Individual Instructions

First, we identify a set of instruction schemes for which our hypothesis is applicable. There are instructions for which benchmarking as described above is not possible or not sensible:

- Jump and branch instructions, which affect control flow, cannot be meaningfully put into a basic block for benchmarking. How these instructions affect the performance depends on whether they are taken and on the processor's branch predictor.

- System instructions, e.g., for interaction with the memory system (prefetching, paging) and for interrupt handling, operate with processor components outside of the port mapping model.

- Combining instructions from the outdated x87, MMX, and SSE floating-point and vector instruction set extensions with the more modern AVX instructions is known to cause delays in the pipeline that are not related to the port mapping.

- For instructions with documented input-dependent behavior, like integer division instructions and instructions with `REP` prefix, a characterization per instruction scheme is insufficient.[8]

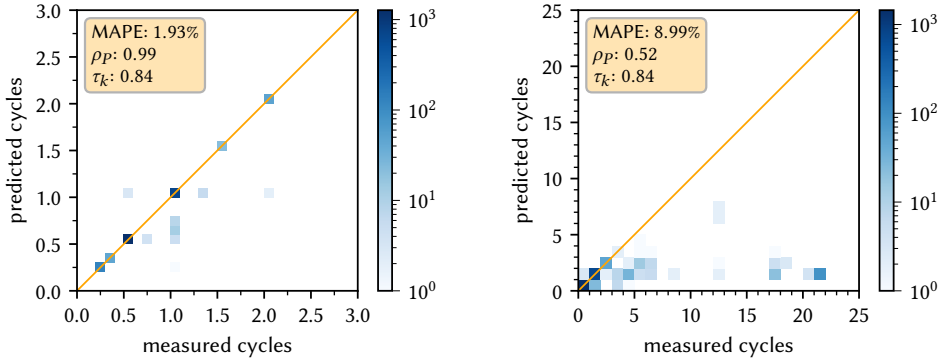
- As discussed in the previous section, the throughput of instructions that read from and write to the same, hard-coded register is dominated by dependencies that we cannot avoid without introducing other confounding factors on the throughput. This includes instructions like `adc` and `sbb`, which read and write the carry flag; `push` and `pop`, which use and modify the stack pointer; and string operations that automatically increment or decrement the value in their memory operand register. Similarly, instruction schemes with read and written operands that must be chosen from the four `ah`–`dh` registers do not allow for enough independent instruction instances.

- `NOP` and register-to-register `MOV` instructions do not use any ports according to the uops.info port mapping. Their throughput is limited by other parts of the processor pipeline.

We extract our instruction schemes from the XML file provided on uops.info (Abel and Reineke, 2019). We exclude the above-mentioned problematic instructions, and restrict the floating-point and vector instructions under consideration to the AVX and AVX2 instruction set extensions. This leaves us with a total of 2498 instruction schemes. For each such instruction scheme i, we measure the achieved inverse throughput of an experiment with only an i instruction on an Intel Core i7 6700 processor with the Skylake microarchitecture.

Figure 3.6 visualizes how these measurements relate to the predictions of the port mapping model with uops.info's Skylake port mapping. The results are shown as heat maps, i.e., two-dimensional histograms. Each experiment is assigned to a bin that is further to the right the more cycles the experiment took in the measurements and further to the top the more cycles the port mapping model predicts. Each bin is displayed as a colored square, with darker colors indicating bins with more experiments. Ideally, predictions and measurements should agree. The closer the colored bins are to the orange diagonal line, the higher is the accuracy of the model. We additionally provide accuracy metrics in the top-left corner of the plot:

[8] Instructions with a `REP` prefix are repeated for a number of times based on register and memory contents.



(a) Restricted to ≤ 3 measured cycles.

(b) Full dataset.

Figure 3.6. Heat maps displaying the inverse throughput in cycles as predicted by the port mapping model (vertical axis) and as measured on the hardware (horizontal axis) for instructions with an inverse throughput of at most 3.0 cycles (a) and for all instructions (b).

- MAPE, the mean absolute percentage error of a collection D of data points consisting of a modeled and a measured value:

$$MAPE(D) = \sum_{(modeled, measured) \in D} \frac{|modeled - measured|}{measured} \cdot 100\%$$

  A small error indicates that model and measurements agree.

- Pearson's linear correlation coefficient ρ_P, which quantifies how close the data points are to a linear relationship.[9] A value of 1.0 indicates maximal correlation, i.e., maximal modeling accuracy, 0.0 represents no correlation, and -1.0 indicates a maximal negative correlation.

- The ranking correlation coefficient τ_K of Kendall (1938, 1945), which describes how close the data is to a monotone relationship. When we order the considered data points in two different ways, one ranked by the modeled value and one ranked by the measured value, Kendall's τ_K is a measure of how similar these orderings are. Similar to ρ_P, 1.0 indicates maximal correlation, 0.0 represents no correlation, and -1.0 indicates a maximal negative correlation.[10]

We explore the results of this evaluation in two different resolutions. Figure 3.6 (a) shows only the experiments with a measured inverse throughput of at most three cycles, these represent 91% of the instruction schemes. The port mapping model follows the measurements very closely for these experiments, with a very low average prediction error and strong linear

[9] See, e.g., Section 3.3 of the textbook by Wasserman (2004) for the definition of Pearson's correlation coefficient.

[10] Specifically, we use the Tau-b statistic, which is adjusted such that correlations of 1.0 and -1.0 are possible in the presence of ties in the rankings.

and rank correlations. Most commonly, between one and four instruction instances can be executed independently, which we observe as inverse throughputs of $\frac{1}{4}$, $\frac{1}{3}$, $\frac{1}{2}$, and 1 cycle. Typically, instructions where the port mapping model does not agree with the measurements have some unusual characteristic in the context of the x86-64 instruction set architecture. This includes, for instance, vector blend instructions with variable masks, which have four operands; `xadd` and `xchg` instructions with two read-and-written operands, and a `mov` instruction with an unusually large 64-bit immediate operand.[11]

When considering all instructions in Figure 3.6 (b), the prediction accuracy of the port mapping model decreases. For instructions with a measured inverse throughput of more than three cycles, the port mapping tends to underestimate the required execution time substantially. This includes any instruction with a `lock` prefix, bit test instructions that access memory, and complex vector or floating point instructions like divisions, square-root computations, or operations of the AES encryption scheme. We conclude that the throughput of these instructions is determined by bottlenecks outside of the port mapping model.

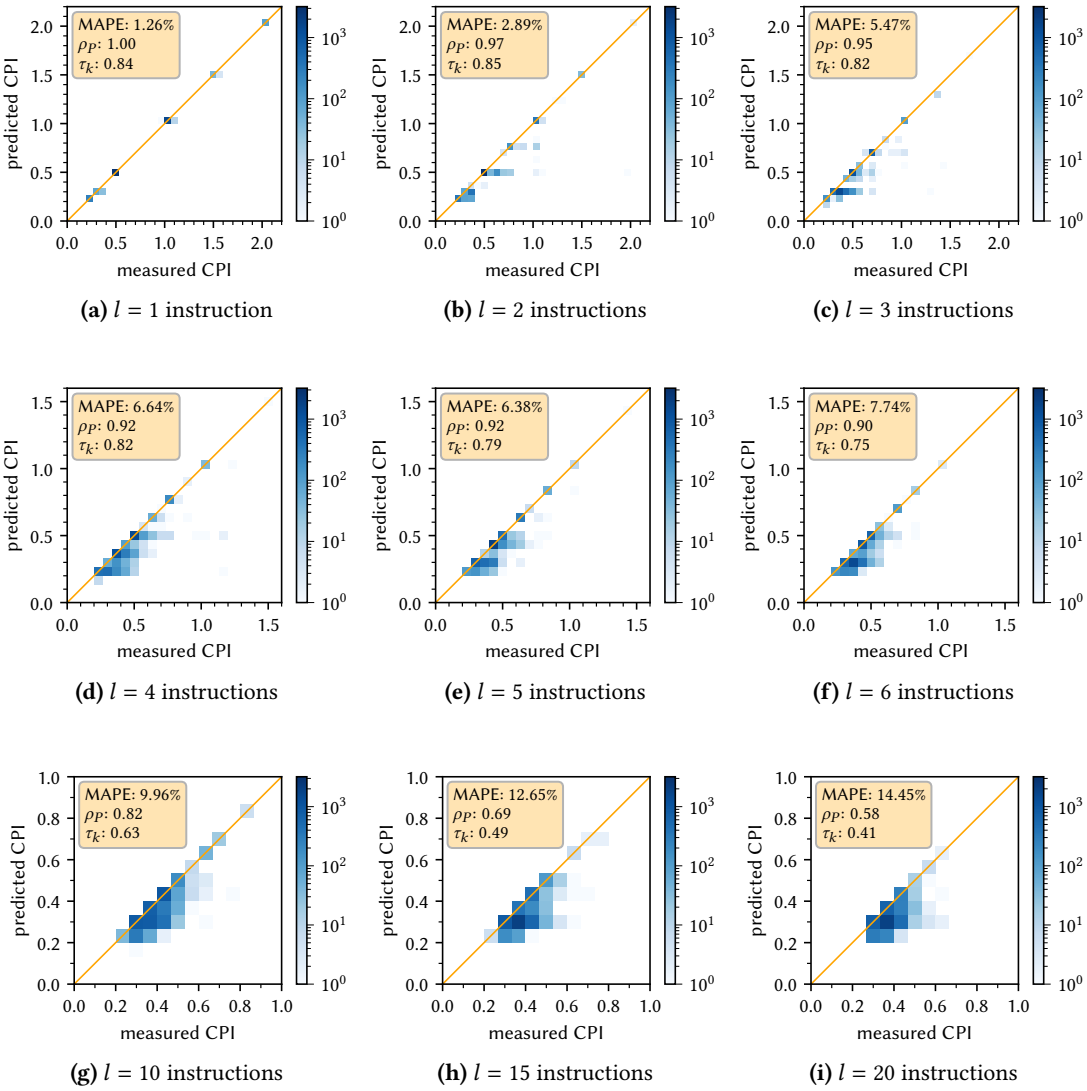
Varying Experiment Length

Next, we investigate in how far the port mapping model remains accurate when multiple instruction schemes are mixed in experiments. For this evaluation, we generate sets of 5,000 experiments each for a range of different experiment sizes l. Each experiment consists of l randomly sampled instruction schemes. We exclude all instruction schemes for which the measurements and the predictions of the port mapping model in the previous section differed by more than 10% to isolate the effect of composing multiple instruction schemes. This leaves a set of 2,234 instruction schemes from which we sample the elements of each experiment uniformly at random. The inverse throughput of each such experiment is measured on an Intel Skylake system and predicted with the uops.info Skylake port mapping as before.

Figure 3.7 displays heat maps for experiments with $l \in \{1, 2, \ldots, 6\}$ instruction schemes ((a)–(f)) and with $l \in \{10, 15, 20\}$ instruction schemes ((g)–(i)). For these heat maps and the summary metrics in the top left corner of each graph, we considered the throughput in cycles per instruction (CPI), i.e., the cycles required to execute the experiments are normalized by the experiment length l.

For the experiments with only one instruction (Figure 3.7 (a)), the heat map shows close to optimal predictions since we excluded any instruction scheme with significant deviations in Figure 3.6 from consideration here. It is worth noting that the value of Kendall's rank correlation metric τ_K is surprisingly low at 0.84 for a data set with such an apparently strong correlation. The reason for this discrepancy is the discrete nature of the predictions of the port mapping model – only the values $\frac{1}{4}$, $\frac{1}{3}$, $\frac{1}{2}$, 1, 1.5, and 2 occur – and the non-discrete nature of the measurements. Following the methodology of other works in the field (Abel and Reineke, 2022; Mendis *et al.*, 2019), we round throughput measurements in cycles to the second decimal place. This is enough resolution for small differences in the throughput measurements due to noise to lead to different ranks where the corresponding predictions tie instead. The resulting

[11] The dissertation of Abel (2020) contains a more detailed exploration of discrepancies between the observable instruction throughput and the port mappings of Intel processors.



(a) $l = 1$ instruction **(b)** $l = 2$ instructions **(c)** $l = 3$ instructions

(d) $l = 4$ instructions **(e)** $l = 5$ instructions **(f)** $l = 6$ instructions

(g) $l = 10$ instructions **(h)** $l = 15$ instructions **(i)** $l = 20$ instructions

Figure 3.7. Heat maps that compare measurements and predictions of the port mapping model for the execution rate on an Intel Skylake CPU in cycles per instruction (CPI). Each heat map covers 5,000 randomly sampled instruction scheme sequences of length l. Each row has a different axis scale to accommodate the different maximal CPIs, but the color scales and the number of bins per CPI-unit are equal in every subplot.

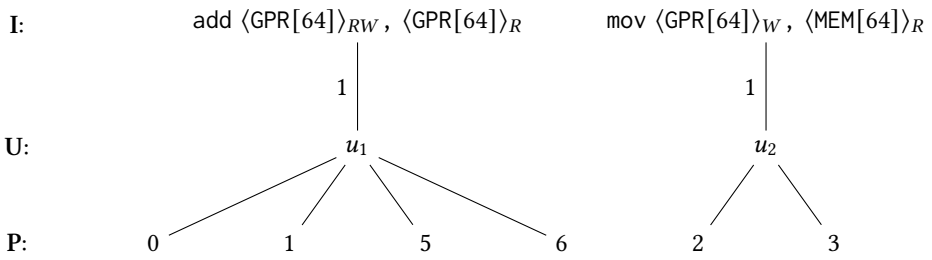


Figure 3.8. Three-level port mapping for selected x86-64 instruction schemes in Intel's Skylake microarchitecture, according to uops.info (Abel and Reineke, 2019).

ranking of the measurements hence differs from the ranking of the predictions, leading to a lower value of τ_K.

Figures 3.7 (b) to 3.7 (f) show that combining more instruction schemes leads to greater discrepancies between throughput measurements and the predictions of the port mapping model. The port mapping model tends to underestimate the execution time here as well, indicating that the throughput is limited by other bottlenecks. While the discrepancies are still minor for experiments with two instruction schemes (2.89% MAPE, Figure 3.7 (b)), the inaccuracy is already pronounced for mixes of six instruction schemes (7.74% MAPE, Figure 3.7 (f)). For larger experiments of 10 or more instruction schemes (Figure 3.7 (g) – Figure 3.7 (i)), the differences are substantial with 10–15% MAPE and decreasing correlation coefficients.

When we base methods to infer port mappings on throughput measurements, we therefore need to ensure that the experiments contain as few instructions as possible or exclude instructions that trigger non-port-mapping bottlenecks when combined with other instructions.

However, not all discrepancies in experiments that combine instruction schemes can be attributed to specific instruction schemes. In the port mapping model, experiments are executed as fast as their instructions (or the µops that constitute them) can be assigned to suitable ports. For a microarchitecture with n ports, the port mapping model therefore may predict throughputs of up to n instructions per cycle. In practice, all modern microarchitectures that we are aware of, including recent designs by Intel and AMD, cannot sustain a full utilization of all execution ports. The culprit is often the decoding frontend (including the caches) or the instruction retirement rate, which limit the processor's peak throughput.

For an example, consider the following experiment with four integer additions and two 64-bit wide loads from memory:

$$4 \times \texttt{add}\ \langle \texttt{GPR[64]} \rangle_{RW}\texttt{,}\ \langle \texttt{GPR[64]} \rangle_{R}$$
$$2 \times \texttt{mov}\ \langle \texttt{GPR[64]} \rangle_{W}\texttt{,}\ \langle \texttt{MEM[64]} \rangle_{R}$$

According to the port mapping (Figure 3.8), the six instruction schemes of this experiment are executed as six µops: four $u_1 = [0, 1, 5, 6]$ addition µops and two $u_1 = [2, 3]$ load µops. As enough ports are available to execute all µops in parallel, we would expect an inverse throughput of 1.0 cycles per iteration, i.e., six instructions executed per cycle. In measurements,

Table 3.1. Arithmetic mean and maximum of CPI differences Δ_{CPI} between permutations of sampled experiments and percentage of samples where Δ_{CPI} exceeds 0.05 CPI, for varying numbers l of instruction schemes per experiment.

l	mean Δ_{CPI}	max Δ_{CPI}	$\Delta_{CPI} > 0.05$
2	0.004	0.18	1.2%
4	0.006	0.56	1.2%
6	0.005	0.30	1.6%
8	0.003	0.09	0.4%
10	0.004	0.31	1.2%
12	0.003	0.17	0.6%
14	0.004	0.16	1.8%
16	0.003	0.09	0.6%
18	0.004	0.15	1.6%
20	0.004	0.21	1.2%

however, we observe that only four instructions are executed per cycle. This observation is consistent with the documented maximal number of instructions that can be retired in a cycle in the Skylake microarchitecture: According to Intel's Software Optimization Guide (Intel, 2023a, Section 2.3), the retirement width has only increased from four to eight µops per cycle in the more recent Golden Cove microarchitecture.

Varying Experiment Order

Lastly, we investigate if excluding the order of the instruction schemes in our definition of experiments (Definition 3.1) is an adequate modeling decision. For this purpose, we consider experiment sizes l between 2 and 20 instructions. For every length l, we construct 500 random sequences of instruction schemes sampled from the same set as for the previous evaluation. We generate 10 random permutations of the instruction scheme list for each such sample, measure their throughputs in cycles per instruction (CPI), and note the difference Δ_{CPI} between the minimal and maximal CPI observations.

Table 3.1 summarizes the results of this evaluation. The arithmetic mean of the differences Δ_{CPI} does not exceed 0.01 CPI for any experiment length. While rare outliers up to 0.56 CPI occur, we observe potentially significant differences of more than 0.05 CPI only for less than 2% of the samples. Neither frequency nor magnitude of these outliers increases notably with the experiment length. Compared to the prediction errors of the port mapping model for similar experiment sizes that we observed in the previous section, the variance induced by different experiment orders is therefore negligible.

3.2.3. Conclusion: Applicability and Limitations

For a substantial portion of the instruction set, the throughput predictions of the port mapping model accurately capture what we observe with our throughput measurement mechanism. However, there are restrictions: The throughput of certain instructions cannot be benchmarked in a meaningful way with the measurement mechanism, e.g., because they affect the control flow, because data dependencies cannot be avoided, or because their throughput depends on the operand values. Moreover, most slow-running instructions with an inverse throughput of more than three cycles and instructions with non-standard operands execute slower than the port mapping model predicts. Throughput measurements alone cannot yield an accurate port mapping for these instructions. Which instructions are affected depends on the microarchitecture under investigation.

Lastly, the throughput prediction accuracy decreases for longer instruction sequences. The higher the number of instruction schemes involved in an experiment, the more often performance bottlenecks outside of the port mapping model are hit.

Port mapping inference algorithms based on throughput measurements need to be aware of these restrictions and should handle them if possible.

3.3. Port Mapping Inference Problems

With the notions from the previous sections in place, we now define the port mapping inference problems addressed in this thesis. For all problems defined here, we assume that the number of ports in the microarchitecture is known. We found this assumption to be true in practice: All manufacturers of processors we investigated provide information about the available execution ports.

We start by defining a foundational concept: the indistinguishability of port mappings.

Definition 3.16. Two port mappings M and M' (in either two- or three-level model) are *indistinguishable* if they yield the same inverse throughput for every experiment e:

$$\forall e : \mathbf{I} \to \mathbb{N}.\ tp^{-1}_{M}(e) = tp^{-1}_{M'}(e)$$

⌟

Since we only rely on throughput measurements to gain insights into the processor, there are certain groups of port mappings that we cannot distinguish. For instance, the two-level port mappings in Figure 3.9 are indistinguishable with throughput measurements since the left can be transformed to the right by renaming the ports ($p_1 \mapsto p_3, p_2 \mapsto p_1, p_3 \mapsto p_2$). The goal in our port mapping inference problems is to find a port mapping that is indistinguishable from the port mapping that the processor under investigation implements.

The first scenario that we consider is the online scenario, where we are given access to a mechanism for measuring the inverse throughput for arbitrary instruction sequences and need to devise suitable microbenchmarks to identify the port mapping:

Definition 3.17. The two-level online port mapping inference problem, OnPMInfer2, is the following task: Given a set $\mathbf{I}$ of instructions, a number k of ports, and an inverse throughput

oracle $O : (\mathbf{I} \to \mathbb{N}) \to \mathbb{R}$, which provides the inverse throughput of any experiment that it is given according to some (unknown) two-level port mapping $M^* = (\mathbf{I} \cup \{1, \ldots, k\}, E^*)$, compute a two-level port mapping $M = (\mathbf{I} \cup \{1, \ldots, k\}, E)$ that is indistinguishable from M^*. The three-level online port mapping inference problem OnPMInfer3 is defined analogously. ⌟

The oracle O in the above definition models the ability to exactly measure the throughput of microbenchmarks on the hardware under test. Variations of this oracle can be of interest, for example:

- The oracle produces results with a non-deterministic (but bounded) measurement error.
- The oracle provides more insight in how the experiment is executed, e.g., how many µops per iteration are executed on each port. This corresponds to the setting of uops.info (Abel and Reineke, 2019).

A simplification to make online port mapping inference problems more tractable is to select a fixed set of experiments beforehand and to only require a port mapping that explains these experiments. We refer to such simplified versions of the problem as *offline* port mapping inference:

Definition 3.18. The two-level offline port mapping inference problem OffPMInfer2 is the following task: Given a set $\mathbf{I}$ of instructions, a number k of ports, and a set $Exps$ of experiments e with measured inverse throughputs $tp^{-1}(e)$, compute a two-level port mapping $M = (\mathbf{I} \cup \{1, \ldots, k\}, E)$ such that it simulates the measured throughputs:

$$\forall e \in Exps.\ tp^{-1}(e) = tp_M^{-1}(e)$$

If no such two-level port mapping exists, return an error value.

A problem instance is *satisfiable* if there is a port mapping as required by the problem; we call these port mappings *satisfying*. Otherwise, the problem instance is *unsatisfiable*. In the decision version OffPMInfer2-D, the task is to decide if the problem instance is satisfiable or not.

The three-level variants OffPMInfer3 and OffPMInfer3-D are defined analogously. ⌟

With diverse experiment sets of increasing size, solutions for the OffPMInfer problems approximate solutions to the OnPMInfer problems.

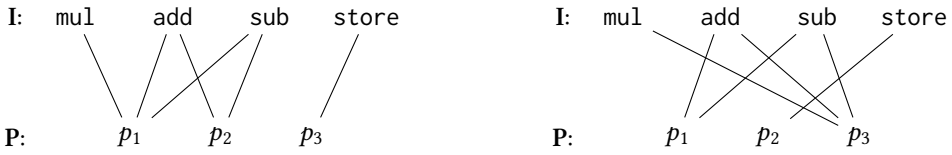


Figure 3.9. Indistinguishable port mappings in the two-level model.

3.3.1. Computational Complexity Results

The goal in this thesis is to infer port mappings for existing microarchitectures, which have fixed numbers of instruction schemes, µops, and ports, with inexact throughput measurements. The asymptotic resource requirements of the algorithms are only of limited relevance as long as practical problem instances are feasible. We therefore cover results on the computational complexity of the above port mapping inference problems only briefly with the following theorems.

Theorem 3.19. The problem OffPMInfer2-D (Definition 3.18) of deciding the existence of a satisfying two-level mapping for a set *Exps* of experiments and k ports is NP-hard. It is NP-complete if we assume a unary encoding of k.

Proof. See Appendix A.1.4. □

Theorem 3.20. The problem OffPMInfer3-D (Definition 3.18) of deciding the existence of a satisfying three-level mapping for a set *Exps* of experiments and k ports is NP-hard. It is NP-complete if we assume a unary encoding of k and the observed inverse throughputs for the experiments.

Proof. See Appendix A.1.5. □

The computational complexity of the online variants of the inference problem remains an open research problem.

Chapter 4

Counter-Example-Guided Port Mapping Inference

In this chapter, we derive algorithms for online port mapping inference as formulated in Definition 3.17 from the linear programs presented in Theorems 3.7 and 3.12 in the previous chapter. Whereas these linear programs compute the unknown throughput of a known experiment with a known port mapping, the goal here is to compute an unknown port mapping by selecting suitable experiments and measuring their throughput.

Our approach is inspired by a range of counter-example-guided algorithms such as symbolic abstraction (Reps *et al.*, 2004), counter-example-guided abstraction refinement (CEGAR) (Clarke *et al.*, 2000), and counter-example-guided inductive synthesis (CEGIS) (Solar-Lezama *et al.*, 2006). The appeal of these algorithms lies in their simple high-level structure that directly incorporates a formal model of the problem under investigation. We discuss a suitable high-level structure for port mapping inference in Section 4.1 before we go into the details for instantiations with the two-level (Section 4.2) and three-level model (Section 4.4).

Since computationally expensive machinery in the form of a satisfiability modulo theories (SMT) solver (Biere *et al.*, 2009) is central to the presented algorithm, we investigate its practicality in Section 4.5. While the SMT-based algorithms do not scale to practical problem sizes in this evaluation, they are instrumental for our further approaches to infer port mappings (Chapters 5 and 6).

4.1. The Inference Algorithm

Algorithm 4.1 shows the high-level structure of our online port mapping inference algorithm. It is centered around the set *Experiments* of microbenchmarks that are annotated with their inverse throughput measured on the processor under investigation. In each iteration, we search a port mapping $m1$ that leads to the measured inverse throughputs in *Experiments* (line 3). If no such mapping is found, the observations do not match the port mapping model and the algorithm terminates unsuccessfully (line 5). Otherwise, we search for a different port mapping $m2$ that is consistent with the measurements in *Experiments*, but that is distinguished from $m1$ by an experiment *newExp* (line 6). This means that $m1$ and $m2$ yield the same throughputs for *Experiments*, but different throughputs for *newExp*. If no such mapping and experiment exist, $m1$ is returned as a solution to the port mapping inference problem, since it is indistinguishable by throughput measurements from the processor's actual

```
1  Experiments ← initialExperiments()
2  while true do
3  |  m1 ← findMapping(Experiments)
4  |  if m1 = None then
5  |  |  return None
6  |  m2, newExp ← findOtherMapping(Experiments, m1)
7  |  if m2 = None then
8  |  |  return m1
9  |  cycles ← measureCycles(newExp)
10 |  Experiments ← Experiments ∪ {(newExp, cycles)}
```

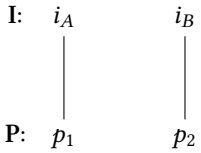
Algorithm 4.1. Counter-example-guided online port mapping inference.

port mapping (line 8). Otherwise, we measure the inverse throughput of *newExp*, add it to *Experiments* (lines 9–10), and continue with the next iteration.

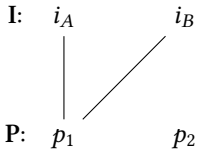
Example 4.1. Assume that we infer a two-level port mapping for an architecture with two instructions i_A, i_B and two ports p_1, p_2. So far, we have measurements indicating that each instruction achieves an inverse throughput of 1.0 cycles per iteration when executed on its own:

$$Experiments = \{(\{i_A \mapsto 1\}, 1.0), (\{i_B \mapsto 1\}, 1.0)\}$$

The *findMapping*(*Experiments*) call will find a port mapping $m1$, for example:



However, this is not the only possible two-level port mapping to explain these measurements. Therefore, *findOtherMapping*(*Experiments*, $m1$) returns a port mapping $m2$, e.g.:



A suitable distinguishing experiment *newExp* is $\{i_A \mapsto 1, i_B \mapsto 1\}$: With $m1$, its inverse throughput is 1.0 cycles per experiment execution while it is 2.0 cycles for $m2$. ⌟

In principle, the algorithm is guaranteed to terminate as we reject at least one of the found port mappings in each iteration. In the two-level setting with fixed numbers of ports and

instructions, there is only a finite number of different port mappings that could be enumerated. The three-level model is slightly more complex: µops could occur with arbitrarily large factors in the port mapping. However, if we require that the *Experiments* set includes a microbenchmark for each individual instruction and their inverse throughputs are bounded by a maximal number t of cycles, we can bound the number of candidate port mappings as well: With k ports, the factors of the µops in the port usage are bounded by $k \cdot t$. As experiments usually rule out more than just a single candidate port mapping, we expect this to be a coarse overestimation of the algorithm's execution time.[1] We investigate the practical running time of the algorithm in Section 4.5.

Algorithm 4.1 describes how we infer port mappings from a high-level view. It relies on several subroutines that we need to provide for a practical implementation:

- The *measureCycles*() procedure (line 9) measures the inverse throughput of an experiment on a given processor. Section 3.2.1 describes the methodology that we use for this task.
- What initial microbenchmarks are provided by *initialExperiments*() in line 1 does not affect the algorithm's correctness: The empty set would be sufficient. In practice, the algorithm can potentially be sped up by providing reasonable experiments, like one to measure the inverse throughput for each individual instruction. We discuss this and other extensions of the algorithm in Section 4.3.
- The remaining procedures, *findMapping* (line 3) and *findOtherMapping* (line 6), which together find two distinguishable satisfying port mappings, are more complex. We describe their SMT-solver-based implementation for port mappings in the two-level and three-level models in the following sections.

4.2. Application in the Two-Level Model

The linear program in Theorem 3.7 formally characterizes the connection between a two-level port mapping (where instructions are mapped directly to the execution ports) and the achieved inverse throughput for an experiment. We derive SMT-solver-powered implementations of *findMapping* and *findOtherMapping* for the inference algorithm based on this linear program.[2] The idea is to augment the linear program such that the port mapping is no longer hardcoded into the constraints, but rather represented by variables of the linear program. We further adjust the formulation such that the resulting inverse throughput is also represented by a free variable. These adjustments enable us to encode *findMapping* and *findOtherMapping* as constraints on the variables of the linear program. An off-the-shelf solver can then produce a satisfying assignment of values to the variables (called a *model*), from which we decode the result.

[1] For a setting with 8 instructions and 4 ports with inverse throughputs of 2 cycles per individual instruction in the three-level model, the presented over-approximation leads to a number of different port mappings that is greater than 10^{100}, which, e.g., greatly exceeds common estimates on the number of atoms in the observable universe.

[2] See Appendix B for an overview of the terminology regarding linear programming and satisfiability modulo theories used in this thesis.

Specifically, we formulate a constraint system

$$relateThroughput[enc_M, enc_e, enc_t]$$

parametrized with collections enc_M, enc_e, enc_t of variables that represent a port mapping, an experiment, and the experiment's inverse throughput, respectively. We construct these constraints such that a model encoding a port mapping M, an experiment e, and a number t satisfies them if and only if $tp_M^{-1}(e) = t$.

The following sections describe how we encode port mappings and experiments with variables in logical formulas (Section 4.2.1), how the constraint sets themselves are derived (Section 4.2.2), and how to use them to implement the *findMapping* and *findOtherMapping* procedures (Section 4.2.3).

4.2.1. Encoding Port Mappings and Experiments

Our goal is that a satisfying model for a system of constraints encodes an experiment and a two-level port mapping. For this purpose, we need to define suitable sets of variables whose values we can interpret as concrete experiments and port mappings.

Experiments – as per Definition 3.1 – are multisets of instruction schemes $i \in \mathbf{I}$. We represent arbitrary multisets with a set $enc_e := \{exp[i] \mid i \in \mathbf{I}\}$ of integer-valued variables. The value of a variable $exp[i]$ determines the number of occurrences of the instruction i in the experiment.

Each such experiment encoding used in a system of constraints needs additional assertions to enforce that the represented experiment is well-formed. We require that each variable is non-negative:

$$\bigwedge_{i \in \mathbf{I}} exp[i] \geq 0$$

Additionally, we can encode constraints on the size of experiments for later optimizations:

$$\sum_{i \in \mathbf{I}} exp[i] \leq bound$$

Such additional assertions are added as conjuncts to the system of constraints.

Example 4.2. Assume an instruction set architecture as in the examples in Section 3.1 with four instructions: `add`, `sub`, `mul`, and `store`. We use four variables to model an experiment:

$$enc_e = \{exp[\texttt{add}], exp[\texttt{sub}], exp[\texttt{mul}], exp[\texttt{store}]\}$$

The following model encodes the experiment $\{\texttt{add} \mapsto 2, \texttt{mul} \mapsto 1, \texttt{store} \mapsto 1\}$:

$$\{exp[\texttt{add}] \mapsto 2, exp[\texttt{sub}] \mapsto 0, exp[\texttt{mul}] \mapsto 1, exp[\texttt{store}] \mapsto 1\}$$

⌟

To represent two-level port mappings, the values of the corresponding variables need to encode bipartite graphs between the instruction schemes $\mathbf{I}$ and the ports $\mathbf{P}$. We use a set $enc_M := \{mapping[i,k] \mid i \in \mathbf{I}, k \in \mathbf{P}\}$ of binary-valued variables. When a variable $mapping[i,k]$ is True in a model, there is an edge from instruction i to port k in the port mapping, i.e., i can be executed on k.

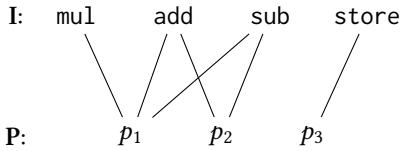
The well-formedness constraints for port mapping encodings require that every instruction has at least one port that can execute it:

$$\bigwedge_{i \in \mathbf{I}} \bigvee_{k \in \mathbf{P}} mapping[i,k]$$

Example 4.3. Assume the instruction set architecture from the examples in Section 3.1 with the four instructions `add`, `sub`, `mul`, and `store`. If we further assume that there are three ports, p_1, p_2, p_3, in the microarchitecture, the encoding enc_M for a two-level port mapping contains a binary variable for each pair of an instruction and a port, as follows:

$$\begin{aligned} enc_M = \{\ & mapping[\texttt{add}, p_1], mapping[\texttt{add}, p_2], mapping[\texttt{add}, p_3], \\ & mapping[\texttt{sub}, p_1], mapping[\texttt{sub}, p_2], mapping[\texttt{sub}, p_3], \\ & mapping[\texttt{mul}, p_1], mapping[\texttt{mul}, p_2], mapping[\texttt{mul}, p_3], \\ & mapping[\texttt{store}, p_1], mapping[\texttt{store}, p_2], mapping[\texttt{store}, p_3]\ \} \end{aligned}$$

Every possible two-level port mapping for this microarchitecture can be represented as a model that assigns values to these variables. Reconsider the example port mapping from Figure 3.1:



The following model encodes this port mapping (variables not present are False):

$$\begin{aligned} \{\ & mapping[\texttt{add}, p_1] \mapsto \text{True}, \quad mapping[\texttt{add}, p_2] \mapsto \text{True}, \\ & mapping[\texttt{sub}, p_1] \mapsto \text{True}, \quad mapping[\texttt{sub}, p_2] \mapsto \text{True}, \\ & mapping[\texttt{mul}, p_1] \mapsto \text{True}, \\ & mapping[\texttt{store}, p_3] \mapsto \text{True}\ \} \end{aligned}$$

⌟

4.2.2. A Parametric SMT Formulation to Relate Port Mapping and Inverse Throughput

Reconsider the linear program from Theorem 3.7:

$$
\begin{array}{lll}
\text{minimize} & t & \\
\text{subject to} & \displaystyle\sum_{k \in \mathbf{P}} x_{ik} = e(i) & \text{for all instructions } i \in \mathbf{I} \quad \text{(A)} \\
& \displaystyle\sum_{i \in \mathbf{I}} x_{ik} = p_k & \text{for all ports } k \in \mathbf{P} \quad \text{(B)} \\
& p_k \leq t & \text{for all ports } k \in \mathbf{P} \quad \text{(C)} \\
& x_{ik} \geq 0 & \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} \quad \text{(D)} \\
& x_{ik} = 0 & \text{if } (i,k) \notin E \quad \text{(E)}
\end{array}
$$

Our goal is to adjust this linear program such that the inverse throughput t, the experiment e, and the port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ occur only as free variables. Then, by asserting any one or two of them to be equal to fixed values, we can use a solver to find satisfying values for the remaining variables.

Inverse Throughput. The inverse throughput is present as a variable t in the linear program, but it is not free: Constraints A-E of the linear program only assert that the experiment can be executed according to the port mapping within at most t cycles. The minimization objective is necessary to ensure that t corresponds to an optimal execution schedule, not just an upper bound. If the value of t was fixed by a constraint, the minimization objective would effectively be disabled. We therefore replace the optimization objective of the linear program with more constraints that ensure optimality of the execution. We use SMT formulas in the theory of linear integer and real arithmetic (LIRA) to obtain a more intuitive formulation with logical disjunctions and implications:[3]

Theorem 4.4. The linear program from Theorem 3.7 is feasible if and only if the following system of constraints is satisfiable:

$$
\begin{array}{lll}
\displaystyle\sum_{k \in \mathbf{P}} x_{ik} = e(i) & \text{for all instructions } i \in \mathbf{I} & \text{(A)} \\
\displaystyle\sum_{i \in \mathbf{I}} x_{ik} = p_k & \text{for all ports } k \in \mathbf{P} & \text{(B)} \\
p_k \leq t & \text{for all ports } k \in \mathbf{P} & \text{(C)} \\
x_{ik} \geq 0 & \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} & \text{(D)} \\
x_{ik} = 0 & \text{if } (i,k) \notin E & \text{(E)}
\end{array}
$$

[3]Note that the multiplications in constraint I both involve a boolean variable. They can therefore be encoded with the SMT if-then-else operator $ite(c,t,f)$ as $b \cdot v :\Leftrightarrow ite(b,v,0)$ without requiring undecidable theories.

$$\bigvee_{k \in \mathbf{P}} q_k \tag{F}$$

$$q_k \leftrightarrow (p_k = t) \qquad \text{for all ports } k \in \mathbf{P} \tag{G}$$

$$j_i \rightarrow q_k \qquad \text{if } (i,k) \in E \tag{H}$$

$$\sum_{i \in \mathbf{I}} j_i \cdot e(i) = \sum_{k \in \mathbf{P}} q_k \cdot t \tag{I}$$

The optimal objective value t^* of the linear program is equal to the value $m[t]$ of the variable t in any satisfying model m of this constraint system.

Proof. See Appendix A.2.1. □

Note that the first five constraints A-E of the constraint system are identical to the constraints of the linear program. They ensure that the x_{ik} variables encode a feasible distribution of the involved instructions to the ports. The remaining constraints are based on an insight that expands on Theorem 3.14: A distribution to ports is optimal if and only if there is a non-empty set Q of bottleneck ports that the distribution utilizes for the full number of cycles with instructions that can only be executed on ports in Q.

We encode this notion in the constraints: A port k is in the set Q of bottleneck ports if, and only if, the boolean variable q_k is True. Constraint F therefore asserts that Q is not empty. With constraint G, we ensure that each bottleneck port is utilized for the full t cycles. The boolean j_i variables encode a set J of instructions i that can only be executed on bottleneck ports in Q, as enforced by constraint H. Lastly, constraint I ensures that only instructions from J contribute to the utilization of the ports in Q.

Experiment and Port Mapping. To introduce the experiment encoding, we use the $exp[i]$ variables instead of the fixed numbers $e(i)$ of occurrences for each instruction i.

The port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ occurs in the constraint set of Theorem 4.4 in the quantification of constraints E and H. We replace these constraints to integrate the port mapping encoding into the constraints with logical implications:

$$x_{ik} = 0 \quad \text{if } (i,k) \notin E \qquad \rightsquigarrow \qquad \neg mapping[i,k] \rightarrow x_{ik} = 0 \quad \text{for all } i \in \mathbf{I}, k \in \mathbf{P} \tag{E}$$

$$j_i \rightarrow q_k \quad \text{if } (i,k) \in E \qquad \rightsquigarrow \qquad mapping[i,k] \rightarrow (j_i \rightarrow q_k) \quad \text{for all } i \in \mathbf{I}, k \in \mathbf{P} \tag{H}$$

This leaves us with the following system of constraints, where all occurrences of the inverse throughput, the experiment, and the port mapping are as free variables:

Definition 4.5. The constraint system $relateThroughput[enc_M, enc_e, enc_t]$ for the two-level setting, which is parametric in variable sets $enc_M = \{mapping[i,k] \mid i \in \mathbf{I}, k \in \mathbf{P}\}$, $enc_e = \{exp[i] \mid i \in \mathbf{I}\}$, and $enc_t = \{t\}$ is defined as follows:

$$\sum_{k \in \mathbf{P}} x_{ik} = exp[i] \qquad \text{for all instructions } i \in \mathbf{I} \tag{A}$$

$$\sum_{i \in \mathbf{I}} x_{ik} = p_k \qquad \text{for all ports } k \in \mathbf{P} \tag{B}$$

$$p_k \leq t \qquad \text{for all ports } k \in \mathbf{P} \tag{C}$$

$$x_{ik} \geq 0 \qquad \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} \tag{D}$$

$$\neg mapping[i,k] \rightarrow x_{ik} = 0 \qquad \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} \tag{E}$$

$$\bigvee_{k \in \mathbf{P}} q_k \tag{F}$$

$$q_k \leftrightarrow (p_k = t) \qquad \text{for all ports } k \in \mathbf{P} \tag{G}$$

$$mapping[i,k] \rightarrow (j_i \rightarrow q_k) \qquad \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} \tag{H}$$

$$\sum_{i \in \mathbf{I}} j_i \cdot exp[i] = \sum_{k \in \mathbf{P}} q_k \cdot t \tag{I}$$

⌟

4.2.3. Implementing the Components of the Counter-Example-Guided Algorithm

The parametric constraint system $relateThroughput[enc_M, enc_e, enc_t]$ is the core of the implementations for the *findMapping* and *findOtherMapping* procedures from Algorithm 4.1.

findMapping(*Experiments*) uses a single free port mapping encoding M_{free}. For each experiment e with inverse throughput t_e, we assert *relateThroughput* constraints for M_{free} and fresh experiment and throughput encodings that are hardwired to e and t_e, respectively:[4]

$$\varphi_{findMapping} := \bigwedge_{(e,t_e) \in Experiments} relateThroughput[M_{free}, e, t_e]$$

The resulting conjunction ensures that the port mapping encoded in a satisfying model yields the observed inverse throughput for all experiments. We use an off-the-shelf SMT solver to check for satisfiability. If the constraints are unsatisfiable, the observed throughputs cannot be explained by a port mapping. Otherwise, we extract and return the port mapping from the values of the encoding variables in the satisfying model produced by the solver.

findOtherMapping(M_1, *Experiments*) includes the same constraints to require that the found port mapping satisfies the experiments, and adds more: Another mapping encoding is hardwired to the input port mapping M_1. For a free experiment encoding e_{free}, we use two more

[4] In practice, we construct simplified formulas where the hardwired values replace the variables and unnecessary constraints are dropped.

instances of the *relateThroughput* constraints to encode the inverse throughputs of both port mapping encodings in two variables t_1, t_2. Lastly, we assert that $t_1 \neq t_2$, i.e., the experiment distinguishes the hard-wired and the free port mapping:

$$\begin{aligned}\varphi_{findOtherMapping} := t_1 \neq t_2 \wedge\ & relateThroughput[M_1, e_{free}, t_1] \\ \wedge\ & relateThroughput[M_{free}, e_{free}, t_2] \wedge \varphi_{findMapping}\end{aligned}$$

In both cases, the auxiliary x_{ik}, p_k, t, q_k, and j_i variables for each occurring *relateThroughput* instance need to be renamed apart.

4.3. Extensions for Practical Applicability

Before we show how the SMT formulation can be extended to the three-level model in Section 4.4, we discuss adjustments to handle problems one encounters when using the SMT formulation in practice.

Handling Measurement Noise

When benchmarking modern processors, inexact measurements due to noise and errors are inevitable. Such inexact measurements are a problem for this method: The constraints in the previous sections encode the exact (in)equality of the observed inverse throughput with the (rational-valued) ideal modeled inverse throughput. We therefore adapt the constraints in practice: A parameter ε constrains the maximal difference between measured and modeled cycles per instruction (CPI) of the experiments.[5] The following constraint encodes the approximated equality of the measured and modeled inverse throughputs t_e and enc_t:

$$\left| \frac{enc_t}{|exp|} - \frac{measuredCycles}{|exp|} \right| < \varepsilon$$

$$\Leftrightarrow |enc_t - measuredCycles| < \varepsilon \cdot |exp|$$

The length $|exp|$ of the experiment in an encoding $enc_e := \left\{ exp[i] \;\middle|\; i \in \mathbf{I} \right\}$ is the sum of all involved variables:

$$\sum_{i \in \mathbf{I}} exp[i]$$

When asserting that the modeled inverse throughputs of the two port mappings in *findOtherMapping* are different, no observed value may be considered equal to both modeled inverse throughputs. Otherwise a found experiment might not rule out any of the candidate mappings. This can be guaranteed if the modeled CPIs differ by at least $2 \cdot \varepsilon$:

$$\left| \frac{t_1}{|exp|} - \frac{t_2}{|exp|} \right| > 2 \cdot \varepsilon$$

$$\Leftrightarrow |t_1 - t_2| > 2 \cdot \varepsilon \cdot |exp|$$

[5] i.e., inverse throughput divided by the length of the experiment.

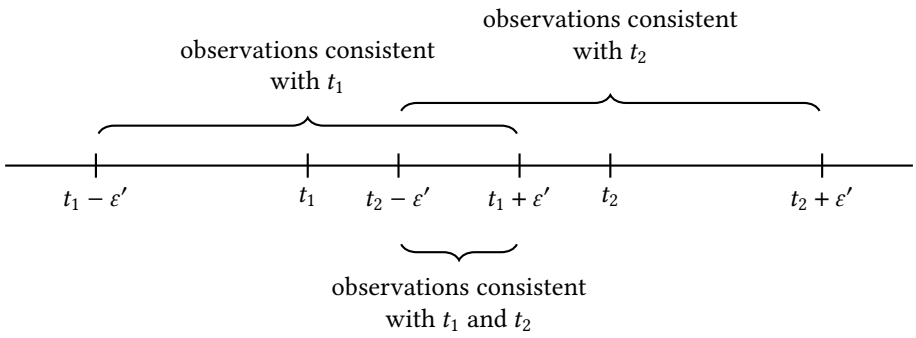


Figure 4.1. Two ideal inverse throughput values, t_1 and t_2, differing by less than $2 \cdot \varepsilon'$. It is possible that throughput observations are consistent with both ideal values.

Figure 4.1 visualizes why $2 \cdot \varepsilon' := 2 \cdot \varepsilon \cdot |exp|$ is necessary as a bound for this constraint on a number line. If the difference between t_1 and t_2 was (strictly) greater than $2 \cdot \varepsilon'$, the region of observations that are consistent with both ideal throughputs would be empty.

Limiting the noise in terms of the CPI rather than any other throughput metric (cf. Definition 2.2) has several benefits:

- In contrast to a constraint on the instructions per cycle (IPC), we avoid a division operation in the constraints.

- Compared with a constraint on (proper) throughput or inverse throughput, a CPI constraint better matches the measurement errors that we expect: Consider two experiments e_1 and e_2, where e_2 contains twice the number of instructions of each kind as e_1. Since our measurement method unrolls the experiment until a specific number of instructions is in the loop body, the corresponding microbenchmark measures the same code, with the same accuracy, only dividing by a different unrolling factor to obtain the cycles per experiment copy:[6]

  $$unrollFactor_1 \cdot tp^{-1}(e_1) = cyclesPerIteration = unrollFactor_2 \cdot tp^{-1}(e_2)$$

  Constraining the accuracy of the inverse throughputs $tp^{-1}(e_1)$ and $tp^{-1}(e_2)$ would therefore impose two different constraints on the accuracy of the measurement of the overall number of *cyclesPerIteration*: A difference of at most $\varepsilon \cdot unrollFactor_1$ and $\varepsilon \cdot unrollFactor_2$ from the true value. Increasing the number of instructions in the experiment would further strengthen the accuracy requirement on the measured *cyclesPerIteration*. Constraining the CPI avoids this counter-intuitive behavior.

[6] See Section 3.2.1 and Appendix D.

Handling Additional Bottlenecks

The port mapping model – and therefore the linear program at the foundation of the *relateThroughput* constraints – assumes that the throughput is only limited by the availability of functional units. As we have seen in Section 3.2.2, this is not the case in practice.

Modern processors usually cannot sustain a full utilization of all ports because of bottlenecks in the decoding frontend (including caches) or the instruction retirement. When a bottleneck limits the execution to at most R_{max} instructions per cycle, experiments that are faster according to the port mapping model are slowed to meet the limit.

We change $relateThroughput[enc_M, enc_e, enc_t]$ such that enc_t is the maximum of the number $tp_M^{-1}(e)$ of cycles according to the model and the peak inverse throughput $|e|/R_{max}$ at the bottleneck. Some port mappings that are distinguishable according to the port mapping model become indistinguishable with this adjustment.

To implement this adjustment, we need to determine the peak throughput of the processor under investigation from documentation or via microbenchmarks before we apply the counter-example-guided port mapping inference algorithm in a realistic setting. A practical benchmarking strategy to determine the peak throughput can proceed as follows:

1. Select a set of instructions with low individual inverse throughputs, preferably below 1.0 cycles.
2. Construct an experiment by picking one instruction and adding it sufficiently often so that it achieves an inverse throughput of 1.0 cycles.
3. Incrementally add more of the remaining instructions to the experiment and keep them if they increase the observed IPC of the experiment.
4. Repeat this process with various processing orders for the instructions. Note the maximal observed IPC as the processor's peak execution rate.

Limiting Experiment Sizes

As presented, the *findOtherMapping* procedure can produce experiments with an arbitrary number of instructions that need to be benchmarked. We have shown in Section 3.2.2 that large numbers of instructions in experiments lead to throughput measurements with significant deviations from the port mapping model. In extreme cases, even the boundaries of µop buffers and the instruction cache can be hit.

An option to avoid such large experiments is enforcing an upper bound on the number of instructions per experiment as described in Section 4.2.1. A fixed upper bound b, however, means that the algorithm is no longer complete: It would only find a port mapping that is indistinguishable from the system under investigation by experiments of length up to b.

We found interleaving the two approaches to be beneficial in practice: We start with a small bound of $b = 1$ and increase it once *findOtherMapping* finds no distinguishing experiment with b instructions anymore. When we encounter a value for b where no new distinguishing experiments of suitable length can be found, we attempt an unbounded run of *findOtherMapping*. If this run also does not find a new experiment, the algorithm terminates; otherwise

it continues with a larger bound b. This stratified approach ensures that we only need to benchmark experiments of minimal size without sacrificing the completeness of the algorithm.

Another related optimization is to add all possible experiments that consist of only a single instruction to the experiment set before the first iteration of the algorithm. This avoids several unnecessary solver calls for distinguishing experiments with just a single instruction with obvious outcomes.

Incremental SMT Solving

In the context of Algorithm 4.1, the implementations of *findMapping* and *findOtherMapping* can make use of incremental solving. State-of-the-art SMT solvers provide an interface for managing a stack of constraints that can be pushed or popped between successive checks for satisfiability. Reusing the SMT solver instance for multiple satisfiability checks in this way allows the solver to reuse information that it previously inferred about the constraint system to speed up subsequent checks.

The constraint sets for every solver call in the *findMapping* and *findOtherMapping* procedures contain the free port mapping encoding and the constraints for a steadily growing list of experiments. We therefore only need to add constraints for each newly introduced experiment to a shared solver instance. The additional constraints in *findOtherMapping* can be pushed to the constraint stack when they are needed and popped afterwards.

4.4. Extension to the Three-Level Model

Similar to the two-level setting, Theorem 3.12 provides a linear program that characterizes how a three-level port mapping determines the inverse throughput of an experiment. A three-level port mapping is essentially a two-level port mapping (where µops take the instruction role) with an additional layer mapping instructions to µops. Our goal is to leverage this close correspondence and to adjust the parametric SMT formulation for the two-level port mapping model from Section 4.2 to the three-level model.

First, we need to change the port mapping encoding to represent three-level mappings. A straightforward encoding uses two sets of variables:

$$enc_M := \left\{ insn2uop[i,u] \;\middle|\; i \in \mathbf{I}, u \in \mathbf{U} \right\} \cup \left\{ uop2port[u,k] \;\middle|\; u \in \mathbf{U}, k \in \mathbf{P} \right\}$$

The integer-valued $insn2uop[i,u]$ variables encode how many instances of a µop u are in the port mapping of the instruction i, while the boolean $uop2port[u,k]$ variables mirror the $mapping[i,k]$ variables from the two-level encoding, mapping µops to their ports. We extend the well-formedness constraints to ensure that every instruction is decomposed into at least one µop and that each µop has at least one port that can execute it:

$$\left(\bigwedge_{i \in \mathbf{I}} \sum_{u \in \mathbf{U}} insns2uops[i,u] > 0 \right) \wedge \left(\bigwedge_{u \in \mathbf{U}} \bigvee_{k \in \mathbf{P}} uops2ports[u,k] \right)$$

We also need to modify the *relateThroughput* constraint system from Definition 4.5. Every occurrence of instructions $i \in \mathbf{I}$ is replaced with µops $u \in \mathbf{U}$, especially in the indices of the x

and j variables. The $uop2port[u,k]$ variables take the role of the $mapping[i,k]$ variables from the two-level setting. Lastly, to connect the instructions with the µops, we adjust references to the number $exp[i]$ of occurrences of an instruction i in the experiment. Each such reference is replaced by the formula for the number of instances of a µop $u \in \mathbf{U}$ that needs to be executed for the experiment, $\sum_{i \in \mathbf{I}} exp[i] \cdot insn2uop[i,u]$. The resulting constraint set looks as follows:

$$\sum_{k \in \mathbf{P}} x_{uk} = \sum_{i \in \mathbf{I}} exp[i] \cdot insn2uop[i,u] \qquad \text{for all µops } u \in \mathbf{U} \qquad \text{(A)}$$

$$\sum_{u \in \mathbf{U}} x_{uk} = p_k \qquad \text{for all ports } k \in \mathbf{P} \qquad \text{(B)}$$

$$p_k \leq t \qquad \text{for all ports } k \in \mathbf{P} \qquad \text{(C)}$$

$$x_{uk} \geq 0 \qquad \text{for all µops } u \in \mathbf{U}, \text{ ports } k \in \mathbf{P} \qquad \text{(D)}$$

$$\neg uop2port[u,k] \rightarrow x_{uk} = 0 \qquad \text{for all µops } u \in \mathbf{U}, \text{ ports } k \in \mathbf{P} \qquad \text{(E)}$$

$$\bigvee_{k \in \mathbf{P}} q_k \qquad \text{(F)}$$

$$q_k \leftrightarrow (p_k = t) \qquad \text{for all ports } k \in \mathbf{P} \qquad \text{(G)}$$

$$uop2port[u,k] \rightarrow (j_u \rightarrow q_k) \qquad \text{for all µops } u \in \mathbf{U}, \text{ ports } k \in \mathbf{P} \qquad \text{(H)}$$

$$\sum_{u \in \mathbf{U}} j_u \cdot \sum_{i \in \mathbf{I}} exp[i] \cdot insn2uop[i,u] = \sum_{k \in \mathbf{P}} q_k \cdot t \qquad \text{(I)}$$

To use this constraint set, we need to determine a set $\mathbf{U}$ of µops. A sound choice would be the power set $\mathcal{P}(\mathbf{P}) \setminus \{\emptyset\}$ of ports, as we observed in Remark 3.13. Every three-level port mapping is representable with this set of µops. This choice of $\mathbf{U}$ leads to a number of µops that is exponential in the number of ports, but it also allows us to simplify the formulas: With $\mathbf{U} := \mathcal{P}(\mathbf{P}) \setminus \{\emptyset\}$, we know for each µop the ports that can execute it. Figure 4.2 visualizes the fixed and variable components of a three-level port mapping encoding with this set of µops. Between the instructions $\mathbf{I}$ and the µops $\mathbf{U}$, we need $|\mathbf{I}| \cdot |\mathbf{U}| = |\mathbf{I}| \cdot (2^{|\mathbf{P}|} - 1)$ integer-valued $insn2uop[i,u]$ variables to determine the port mapping. All connections between µops $\mathbf{U}$ and ports $\mathbf{P}$ are fixed and do not need $uop2port$ variables: A µop $u \in \mathcal{P}(\mathbf{P}) \setminus \{\emptyset\}$ can be executed on a port p if, and only if, $p \in u$. Furthermore, all x_{uk} variables where $k \notin u$ become obsolete since they must be zero in any feasible solution. We can therefore simplify the constraint set as shown in Figure 4.3.

In practice, this still leads to excessively large formulas that make even very small problem sizes prohibitively expensive to solve: We found instances with four instructions and three ports to not terminate within 48 hours. A contributor to the excessive solving times is that the expression for the number of instances of a µop $u \in \mathbf{U}$ that needs to be executed for the experiment, $\sum_{i \in \mathbf{I}} exp[i] \cdot insn2uop[i,u]$, contains a multiplication of two integer variables. The formulas are therefore not in the decidable linear integer and real arithmetic (LIRA) theory but use the generally undecidable non-linear integer and real arithmetic theory. Modern SMT solvers like Z3 (de Moura and Bjørner, 2008) can still find solutions or prove unsatisfiability for some instances of such problems, but termination of the solver can no longer be guaranteed.

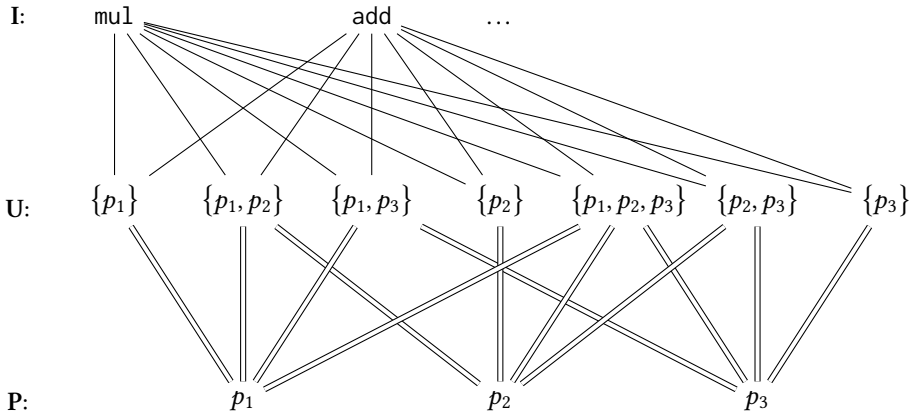


Figure 4.2. Extract of an SMT encoding of a three-level port mapping with fixed µop-to-port mapping corresponding to the constraints in Figure 4.3. Single lines correspond to integer-valued $insn2uop[i, u]$ variables in the formula, double lines are hardwired connections without a variable.

$$\sum_{k \in u} x_{uk} = \sum_{i \in \mathbf{I}} exp[i] \cdot insn2uop[i, u] \qquad \text{for all µops } u \in \mathbf{U} \qquad \text{(A')}$$

$$\sum_{u \in \mathbf{U} \text{ s.t. } k \in u} x_{uk} = p_k \qquad \text{for all ports } k \in \mathbf{P} \qquad \text{(B')}$$

$$p_k \leq t \qquad \text{for all ports } k \in \mathbf{P} \qquad \text{(C)}$$

$$x_{uk} \geq 0 \qquad \text{for all µops } u \in \mathbf{U}, \text{ ports } k \in u \qquad \text{(D')}$$

$$\bigvee_{k \in \mathbf{P}} q_k \qquad \text{(F)}$$

$$q_k \leftrightarrow (p_k = t) \qquad \text{for all ports } k \in \mathbf{P} \qquad \text{(G)}$$

$$(j_u \rightarrow q_k) \qquad \text{for all ports } k \in \mathbf{P}, \text{ µops } u \in \mathbf{U} \text{ s.t. } k \in u \qquad \text{(H')}$$

$$\sum_{u \in \mathbf{U}} j_u \cdot \sum_{i \in \mathbf{I}} exp[i] \cdot insn2uop[i, u] = \sum_{k \in \mathbf{P}} q_k \cdot t \qquad \text{(I)}$$

Figure 4.3. SMT constraints for the three-level model with fixed µop-to-port mapping and $\mathbf{U} = \mathcal{P}(\mathbf{P}) \setminus \{\emptyset\}$.

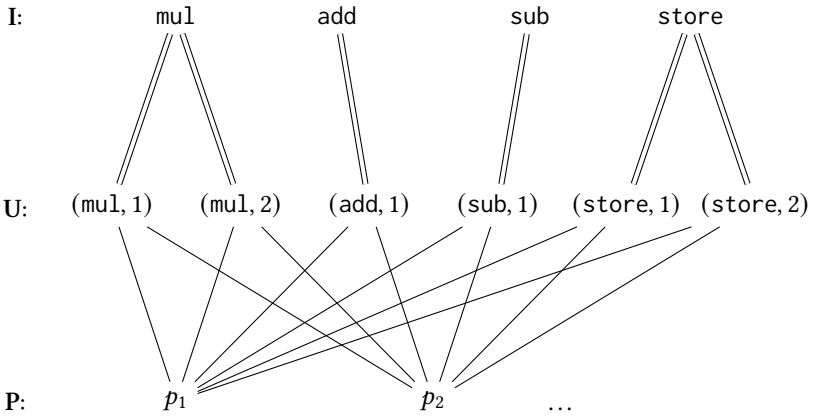


Figure 4.4. Extract of an SMT encoding of a three-level port mapping with fixed instruction-to-µop mapping corresponding to the constraints in Figure 4.5, assuming two µops for `mul`, one each for `add` and `sub`, and two for `store` like in Figure 3.3 on page 19. Single lines correspond to boolean $uop2port[u,k]$ variables in the formula, double lines are hardwired connections without a variable.

$$\sum_{k \in \mathbf{P}} x_{uk} = exp[i] \qquad \text{for all µops } u = (i,r) \in \mathbf{U} \qquad \text{(A")}$$

$$\sum_{u \in \mathbf{U}} x_{uk} = p_k \qquad \text{for all ports } k \in \mathbf{P} \qquad \text{(B)}$$

$$p_k \leq t \qquad \text{for all ports } k \in \mathbf{P} \qquad \text{(C)}$$

$$x_{uk} \geq 0 \qquad \text{for all µops } u \in \mathbf{U}, \text{ ports } k \in \mathbf{P} \qquad \text{(D)}$$

$$\neg uop2port[u,k] \rightarrow x_{uk} = 0 \qquad \text{for all µops } u \in \mathbf{U}, \text{ ports } k \in \mathbf{P} \qquad \text{(E)}$$

$$\bigvee_{k \in \mathbf{P}} q_k \qquad \text{(F)}$$

$$q_k \leftrightarrow (p_k = t) \qquad \text{for all ports } k \in \mathbf{P} \qquad \text{(G)}$$

$$uop2port[u,k] \rightarrow (j_u \rightarrow q_k) \qquad \text{for all µops } u \in \mathbf{U}, \text{ ports } k \in \mathbf{P} \qquad \text{(H)}$$

$$\sum_{u=(i,r) \in \mathbf{U}} j_u \cdot exp[i] = \sum_{k \in \mathbf{P}} q_k \cdot t \qquad \text{(I")}$$

Figure 4.5. SMT constraints for the three-level model with fixed instruction-to-µop mapping and $\mathbf{U} := \left\{(i,r) \;\middle|\; i \in \mathbf{I}, r \in \left\{1, \ldots, uopsPerInsn(i)\right\}\right\}$.

In some scenarios, more information on the µops that occur in the port mapping may be available. For instance, AMD's Zen, Zen+, and Zen2 microarchitectures are documented (AMD, 2019, Section 2.1.15.4.5) to provide a hardware performance counter that allows us to count the µops that are executed when benchmarking an instruction.[7] As a result, we can obtain an assignment *uopsPerInsn* that describes for each instruction the number of µops in its decomposition. This allows us to reduce the size of the constraint set and to avoid non-linear integer arithmetic.

To this end, we define U to include enough µops for every instruction:

$$\mathbf{U} := \left\{(i, r) \;\middle|\; i \in \mathbf{I}, r \in \left\{1, \ldots, \mathit{uopsPerInsn}(i)\right\}\right\}$$

Then, we replace each $\mathit{insn2uop}[i, (i', r)]$ variable with 1 if $i = i'$ and with 0 otherwise. This encodes that an instruction must be mapped to all its µops and to no µops of other instructions. Figure 4.4 visualizes which aspects of the three-level port mapping encoding are fixed with this construction and which remain variable. The integer-valued *insn2uop* variables connecting **I** and **U** are no longer required, only $|\mathbf{U}| \cdot |\mathbf{P}| = \sum_{i \mapsto n \in \mathit{uopsPerInsn}} n \cdot |\mathbf{P}|$ boolean *uop2port* variables remain.

The resulting simplified set of formulas (Figure 4.5) is very similar to the SMT constraints for the two-level model (Definition 4.5): They coincide if *uopsPerInsn* assigns a single µop to each instruction.

For either choice of **U**, the extensions for practical applicability from Section 4.3 apply to the three-level setting unchanged.

4.5. Experimental Evaluation

We evaluate implementations of the algorithm variants from the previous sections with the Z3 SMT solver (de Moura and Bjørner, 2008).[8] The implementations use the optimizations described in Section 4.3: incremental solving, stratified experiment sizes, and an experiment list initialized with throughput measurements for each individual instruction. As the SMT solver with its exponential worst-case running time plays such an integral part in the algorithm, the primary question that arises for this evaluation is if, and in how far, counter-example-guided port mapping inference scales to realistic problem sizes. We separately investigate the two-level and the three-level settings, with varying numbers of instructions and ports for the two-level setting and varying numbers of involved µops in the three-level setting.

Since current microarchitectures are not very diverse in the problem sizes they pose, we perform this evaluation with synthetic benchmarks. For each data point, we randomly generate three ground-truth port mappings with the necessary properties. For each generated ground-truth mapping, we run the port mapping inference algorithm three times with a simulator that provides inverse throughput measurements according to the ground-truth port mapping. We summarize the nine individual results for each data point in the plots in the following sections with a marker for the median result and error bars to the minimal and maximal result observed.

[7] Chapter 6 discusses this setting in more detail.
[8] We use version 4.12.1 of Z3 in this evaluation.

The algorithm uses a noise parameter $\varepsilon_{CPI} = 0.02$ for the shown results. This value can distinguish experiments that utilize five or four ports for one cycle, as these lead to 0.20 CPI or 0.25 CPI, respectively. We found this to be a relevant resolution for real-world settings in experiments performed for Chapter 6. We did not find substantial differences in the results for this evaluation when reducing this parameter to $\varepsilon_{CPI} = 0.001$.

Bottlenecks outside of the port mapping model are not considered for this evaluation: The simulator does not include a non-port-mapping bottleneck and the inference algorithm is configured to not assume one.

We investigate the time required for these inference runs, in which of the subroutines of the algorithm this time is mostly spent, and how many experiments (i.e., counter examples) the algorithm required additionally to the initial set of experiments for the individual instructions. The evaluation was performed on a system with an Intel Core i9-10900K processor (10 cores, 20 threads, 3.7 GHz; the implementation is single-threaded) and 64 GB of RAM.

4.5.1. Two-Level Model

Figure 4.6 shows the results of our evaluation for inferring two-level port mappings. We considered microarchitectures with 4, 6, and 8 ports (represented as individual lines in each plot) and up to 16 instructions.[9] The generated two-level port mappings serving as ground truth map each instruction to 1–4 randomly selected ports.

The execution times in the log-scale plot of Figure 4.6 (a) follow exponential trajectories with rising numbers of instructions. For microarchitectures with 4 ports, the execution times range from fractions of a second at the lower end to around 10 seconds for the case with 16 instructions. Series with more ports rise more quickly, reaching around 1,000 seconds for 6 ports and 16 instructions, and exceeding 10,000 seconds (ca. 2.8 hours) for several cases with 8 ports and only 10 instructions.

With growing execution times, their variance grows as well, even between inference runs with the same ground truth port mapping. For 8 ports and 10 instructions, the algorithm run time for one ground truth mapping varies between 4 and 13 hours, while the required time for another varies only between 45 and 60 minutes.

Overall, the *findOtherMapping* calls dominate the algorithm's running time. For the smallest configurations, *findOtherMapping* causes 80% of the execution time. This percentage rises above 99.9% for the larger configurations.

For comparison, Figure 4.6 (b) shows the number of microbenchmarks queried by the inference algorithm (in addition to the singleton experiments that seed the algorithm's initial experiment list), on a linear scale. These numbers rise less dramatically than the execution time. This indicates that excessive numbers of experiments are not necessary to determine the port mapping. Only the SMT solver calls to find distinguishing experiments or to refute the existence of one become more demanding with growing configurations. In all considered runs, the algorithm queried experiments with at most three to five instructions before finding that no more distinguishing experiment of arbitrary length exists.

[9] The data series for 8 ports do not cover the full range of 16 instructions since their execution times become impractical before this point.

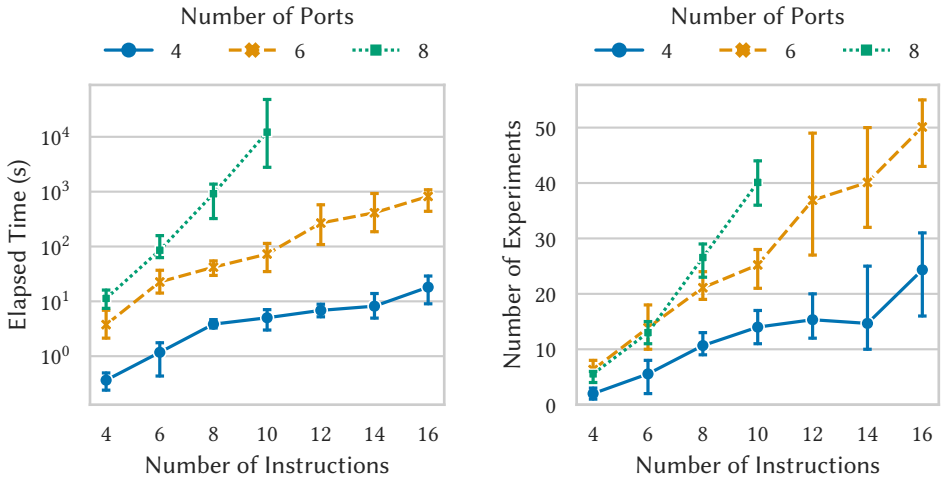


Figure 4.6. Plots for the execution time in seconds with a logarithmic scale (a) and the number of additional experiments queried on a linear scale (b) for counter-example-guided two-level port mapping inference with varying numbers of instructions and ports in the system under investigation at $\varepsilon_{CPI} = 0.02$.

We validated the inferred port mappings by comparing their throughput predictions with the ground truth mapping on 1,000 randomly sampled experiments consisting of five instructions each. With Pearson correlation coefficients greater than 0.95 for all algorithm runs and perfect correlations of 1.0 for most of them, the inferred mappings match the ground truth accurately.

4.5.2. Three-Level Model

Section 4.4 presents two alternative extensions of the SMT formulas to three-level port mappings. We found the formulas of Figure 4.3, which emerge when we choose $\mathbf{U} := \mathcal{P}(\mathbf{P}) \setminus \{\emptyset\}$, to be prohibitively expensive: Instances with four instructions and three ports – simpler than any instance in Figure 4.6 – did not terminate within 48 hours. This evaluation therefore focuses on the second way of extending the formulas to the three-level model, where each instruction has a fixed number of µops that we know beforehand (Figure 4.5). If every instruction is decomposed into a single µop, the formulas for this scenario coincide with the two-level setting. We therefore expect a similar run-time behavior as we observed in the previous Section 4.5.1 in this case. Our goal for this evaluation is to determine how the algorithm's performance changes when we gradually increase the number of µops in the target port mapping.

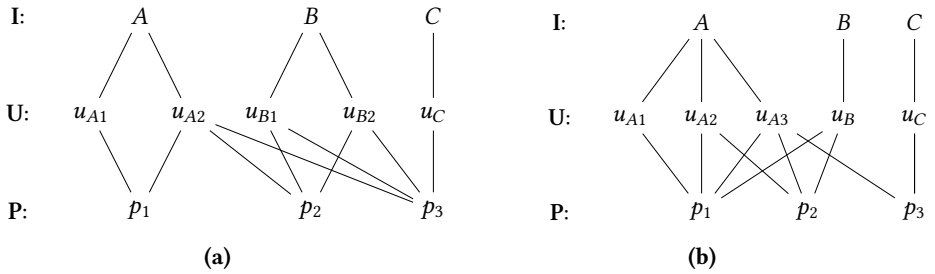


Figure 4.7. Examples for generated port mappings with three instructions and three ports. For port mapping (a), there are **2** instructions with more than one µop (*A* and *B*), and they have **2** µops. In port mapping (b), **1** instruction has more than one µop (*A*), and it has **3** µops.

In contrast to the evaluation of the two-level version in the previous section, all generated port mappings for this evaluation use four instructions and six ports. We instead vary the number of instructions that are decomposed into more than one µop and the number of µops that each such instruction uses. The ground truth port mappings are generated by randomly choosing for each µop between one and four ports that it can be executed on. Figure 4.7 gives examples of port mappings with different values of the varied parameters in a simplified setting with three instructions and three ports. The algorithm obtains the number of µops for each instruction as an input.

Figure 4.8 shows the measurements for this evaluation. Increasing the number of instructions with more than one µop causes the execution time of the algorithm (Figure 4.8 (a)) to rise. The more µops each such instruction has, the stronger the increase. For small numbers of added µops, e.g., one instruction with up to three µops or two instructions with two µops each, however, the added cost is not excessive. Similar to the two-level setting, the time spent in the *findOtherMapping* procedure dominates the execution time.

In the three-level SMT formulation, µops play a similar role as instructions in the two-level formulation. We therefore investigate if experimental configurations with n instructions in the two-level model behave similarly as configurations with a total of n µops in the three-level model. Table 4.1 compares the median execution times for configurations in this evaluation and in the evaluation of the two-level variant (Figure 4.6 (a)) with equal numbers of µops/instructions. Even with the very limited sample size and comparatively large variance in the observations, there is a consistent trend: The algorithm executions for the three-level model are slower than for corresponding configurations in the two-level model.

Figure 4.8 (b) indicates that the growing execution times of the algorithm with increasing numbers of total µops are not caused by larger numbers of required experiments. The numbers of generated experiments range between 3 and 21, with a variance among individual experimental configurations that is too large to determine a clear correlation. With up to 18 instructions, the experiments necessary to infer a three-level port mapping are, however, considerably larger than those for two-level port mappings.

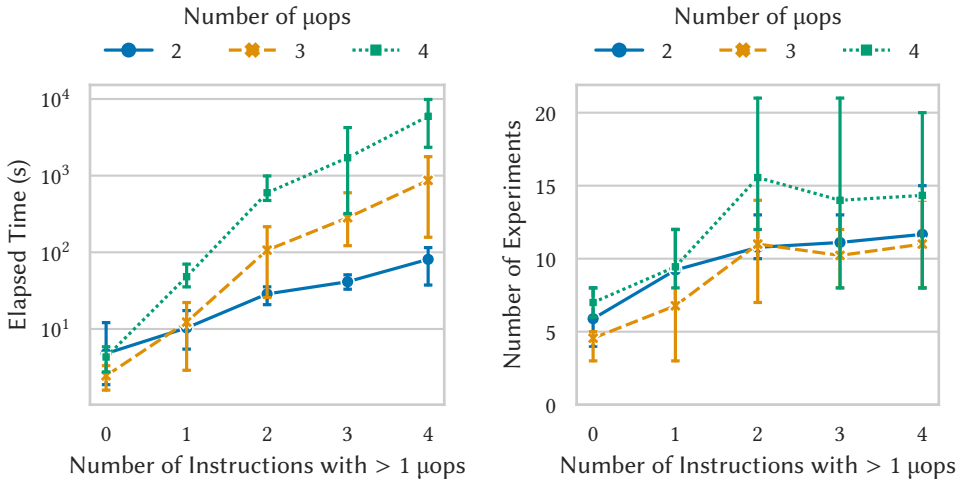


Figure 4.8. Plots for execution time in seconds with a logarithmic scale (a) and number of additional experiments queried on a linear scale (b) for counter-example-guided three-level port mapping inference with 4 instructions and 6 ports in the system under investigation at $\varepsilon_{CPI} = 0.02$. The numbers of instructions with more than one µop and numbers of µops for these instructions are varied.

The inferred port mappings of this evaluation were validated with a comparison of their throughput predictions against the ground truth mapping on 1,000 randomly sampled experiments consisting of five instructions each. The inferred mappings match the ground truth very accurately with Pearson correlation coefficients greater than 0.99 for all algorithm runs.

4.6. Conclusions: Counter-Example-Guided Port Mapping Inference

In this chapter, we derived algorithms for two-level and three-level online port mapping inference from the formal port mapping model of Section 3.1. In theory, the algorithms can infer accurate port mappings for any processor that executes instructions according to the port mapping model.

In practice, the algorithms' execution time is a problem: While we can infer port mappings for small numbers of ports and instructions in the two-level model and for three-level instances that use a known small number of µops, scaling to the thousands of instruction schemes of the x86-64 instruction set architecture is unrealistic. Extending the formulas to the three-level

Table 4.1. Comparison of median algorithm execution times for different configurations with the same total number of µops in the three-level model and instructions in the two-level model.

2-level	3-level	
8 instructions: 42 s	$8 = 4 \times 2 + 0$ µops: 94 s	$8 = 2 \times 3 + 2$ µops: 96 s
10 instructions: 75 s	$10 = 3 \times 3 + 1$ µops: 238 s	$10 = 2 \times 4 + 2$ µops: 542 s
12 instructions: 159 s	$12 = 4 \times 3 + 0$ µops: 884 s	
16 instructions: 920 s	$16 = 4 \times 4 + 0$ µops: 5815 s	

model in other ways than presented in Section 4.4 is possible, but we did not find evidence that any such encoding achieves the necessary increase in performance.

A different approach is therefore necessary to solve the port mapping inference problem practically. The following two chapters discuss alternative approaches that leverage insights from this chapter:

- In Chapter 5, we use an evolutionary algorithm to find port mappings that explain observed throughputs. The fitness metric of the evolutionary algorithm relies on a simulation algorithm derived from the SMT formulation in Theorem 4.4.
- Chapter 6 leverages additional assumptions and knowledge about real-world microarchitectures to relax the port mapping inference problem. As a result, full-fledged online port mapping inference is only necessary for sufficiently small problem instances that can be solved by the counter-example-guided algorithm presented in this chapter.

Chapter 5

Evolving Port Mappings with PMEvo

We have seen in the previous chapter that exact, optimal solutions to the port mapping inference problems are computationally expensive to obtain. For this reason, this chapter presents a method to approximate optimal solutions to the port mapping inference problems. We perform this approximation in two steps:

- We generate experiments following a fixed but possibly incomplete strategy instead of searching for experiments that fully characterize the port mapping.
- We solve the resulting offline port mapping inference problem approximatively with an evolutionary algorithm.

This method, while not exact, allows us to infer port mappings for hundreds of instructions within a practical time budget.

Section 5.1 presents the high-level PMEvo algorithm and its components. An evaluation of PMEvo in the context of its initial publication is included in Section 5.2. In later chapters, we set PMEvo in context with related work (Chapter 7) and we evaluate it in comparison to the more recent Palmed (Derumigny *et al.*, 2022a) and an alternative port mapping inference algorithm (Section 6.3.6).

We originally published the work on PMEvo at the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020) (Ritter and Hack, 2020). This thesis expands upon the PLDI article with a brief discussion about handling additional throughput bottlenecks at the end of Section 5.1.3.

5.1. The PMEvo Framework

PMEvo is a framework to automatically infer port mappings from throughput experiments. Consider Figure 5.1 for an overview of the framework. PMEvo consists of four main stages, which we describe in the following subsections: First, PMEvo generates relevant experiments (Section 5.1.1) and measures their inverse throughput on a given processor – in its original version, PMEvo used a precursor of the method discussed in Section 3.2.1. Next is a preprocessing step that identifies congruent instructions (Section 5.1.2) before the evolutionary optimization is performed (Section 5.1.3).

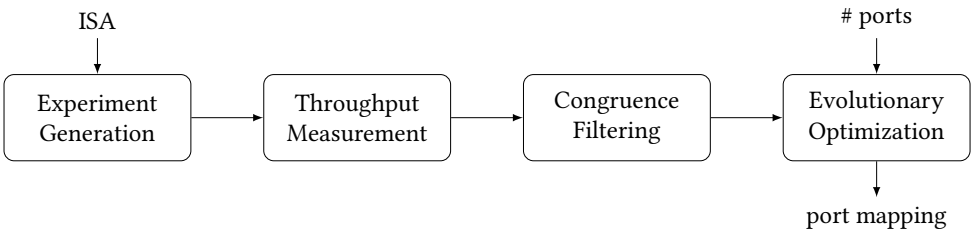


Figure 5.1. Overview of the PMEvo framework. (Ritter and Hack, 2020)

5.1.1. Experiment Generation

The input of the first stage of PMEvo is a description of the instruction set architecture (ISA) supported by the processor under test. This description is a set of instruction schemes, i.e., instructions with typed placeholders for their operands (cf. Section 2.1). The type of the placeholder specifies the operand kind (e.g., memory operand, general-purpose or vector register) and the width of the respective operand.

PMEvo constructs a set of experiments from this information with the following components:

1. for each instruction scheme i, an experiment $\{i \mapsto 1\}$ measuring its individual inverse throughput $tp^{-1}(i)$
2. for each pair (i_A, i_B) of instruction schemes, an experiment $\{i_A \mapsto 1, i_B \mapsto 1\}$
3. for each pair (i_A, i_B) of instruction schemes with $tp^{-1}(i_A) > tp^{-1}(i_B)$, an experiment $\{i_A \mapsto 1, i_B \mapsto n\}$ where
$$n = \left\lceil tp^{-1}(i_A) \,/\, tp^{-1}(i_B) \right\rceil$$

Experiments with this structure lead to different outcomes depending on the port mapping: If the µops of two instruction schemes i_A and i_B require the same ports, experiment (2) will result in an inverse throughput that is the sum of the individual inverse throughputs of i_A and i_B. In case the µops of i_A and i_B are executed by disjoint port sets, the inverse throughput of experiment (3) will be $n \cdot tp^{-1}(i_B)$. More complex partial port conflicts will lead to measured inverse throughputs for these experiments that are harder to interpret manually. It is the task of the evolutionary algorithm to find a mapping that explains these inverse throughputs.

Example 5.1. Assume an instruction set architecture with instructions `add` and `mul`. We will first measure their individual inverse throughputs with single-instruction experiments. Assume that we measure $tp^{-1}(\texttt{add}) = 0.25$ cycles and $tp^{-1}(\texttt{mul}) = 1.0$ cycles. We perform two more experiments for this pair of instructions:

$$\{\texttt{add} \mapsto 1, \texttt{mul} \mapsto 1\}$$
$$\{\texttt{add} \mapsto 4, \texttt{mul} \mapsto 1\}$$

⌟

The evolutionary algorithm is not restricted to experiments of this structure. In theory, longer experiments that combine instances of more than two different instruction schemes can unveil resource conflicts that cannot be covered by these experiments. However, when exploring the experiment design space empirically for existing processors, we did not observe benefits in port mapping quality from more complex experiments with PMEvo.

5.1.2. Congruence Filtering

In a processor microarchitecture, we expect that groups of instruction schemes require the same execution resources. Instruction schemes whose operations are implemented similarly in the processor, e.g., addition and subtraction, often lead to such groups.

PMEvo exploits these patterns to reduce the search space of the evolutionary algorithm. It partitions the set of instruction schemes into congruence classes of instruction schemes that are not distinguishable with the generated experiment set.

In this partitioning, two instruction schemes i_A and i_B are in the same class if and only if the following conditions hold:

- i_A and i_B exhibit equal individual inverse throughputs, i.e., $tp^{-1}(i_A) = tp^{-1}(i_B)$.
- Any two experiments $\{i_A \mapsto m, i_C \mapsto n\}$ and $\{i_B \mapsto m, i_C \mapsto n\}$ that combine these instruction schemes with any other instruction scheme i_C exhibit equal inverse throughputs.

For this purpose, we consider inverse throughputs t_1 and t_2 equal (up to measurement errors) if their symmetric relative difference is limited by a user-specified constant ε, i.e. if

$$\frac{|t_1 - t_2|}{|t_1 + t_2|/2} < \varepsilon$$

For each congruence class, PMEvo selects a representative to be included in the instruction set for the evolutionary algorithm. The evolutionary algorithm then only needs to consider experiments that consist of these representatives.

5.1.3. Evolving Port Mappings

The core of PMEvo is an evolutionary algorithm that searches for a port mapping that accurately explains the observed inverse throughputs for a given set of experiments. Evolutionary algorithms are a well-proven technique to approach optimization problems. They mimic concepts from natural evolution to approximatively optimize complex metrics in non-linear problem settings.[1]

Every evolutionary algorithm is centered around a representation scheme that characterizes the space of possible solutions of the optimization problem. Naturally, the representation that we use is that of port mappings with µop decomposition in the three-level model as described in Section 3.1.2. The sets **I** of instructions and **P** of ports are given by the user. Following

[1] We refer to the textbook by De Jong (2006) for a comprehensive treatment.

```
1 initialize population randomly
2 while not done do
3 |   apply evolutionary operators
4 |   evaluate fitness
5 |   select new population
6 perform local search
7 return fittest individual
```

Algorithm 5.1. Structure of the evolutionary algorithm.

Remark 3.13, we identify each μop with the set of ports that can execute it and allow all non-empty subsets of $\mathbf{P}$ as μops. We define the width $|u| = |\{k \mid (u,k) \in E\}|$ of a μop u as the number of ports that can execute u in a three-level port mapping $M := (\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, F \cup E)$.

PMEvo's evolutionary algorithm follows the structure in Algorithm 5.1. Initially, a set of N port mappings is sampled randomly to form a population. This population is iteratively refined through evolution steps. In each such step, N additional child mappings are generated via evolutionary operators. The resulting population of $2 \cdot N$ port mappings is sorted according to the fitness metric and the best-performing N mappings are selected as the new population. The evolution terminates once the fitness of the population has converged to a single value or an iteration limit is exceeded. By selecting a value for N, the user can find a trade-off between the computational requirements of the evolutionary steps and the quality of the inferred port mapping. After the evolution terminates, PMEvo employs a greedy hill-climbing algorithm to move from the found solutions to a local optimum in the space of possible port mappings.

In the following, we describe the components of the evolutionary algorithm in detail.

Initialization

Each member of the initial population is sampled randomly from the set of possible port mappings as follows. For each instruction i, a random set of 1 to $|\mathbf{P}|$ different μops is sampled. The number of occurrences for each of these μops u in the mapping for i is sampled from the interval $[1, \lceil tp^{-1}(i) \cdot |u| \rceil]$.

The upper bound of this interval is an implication of the throughput model: Executing one u μop puts a load of $1/|u|$ on each of u's ports, leading to an inverse throughput of $1/|u|$ cycles. With $|u|$ instances of these μops, the load on u's ports multiplies by $|u|$, causing an inverse throughput of $u \cdot 1/|u| = 1$ cycle. If at least $t \cdot |u|$ of the μops are executed, the load on u's ports is further multiplied by t, leading to an inverse throughput of at least t cycles. Therefore, an instruction with more than $\lceil t \cdot |u| \rceil$ instances of a μop u in its decomposition requires an inverse throughput of more than t cycles.

Evolutionary Operators

Evolutionary operators create new individuals from existing individuals in the population. The most common operators in evolutionary algorithms are recombination and mutation.

We employ a binary recombination operator that mixes the information of two parent mappings to generate two child mappings. For each instruction i, the lists of occurring µops with their number of occurrences in the parent port mappings are first concatenated into a single list. This list is then shuffled and split randomly into two non-empty parts that form the corresponding assignments for the children. We apply this operator to individuals that are sampled uniformly at random from the population.

When designing the evolutionary algorithm, we tried various random mutation strategies. Experiments showed little benefit over a design without a mutation operator while contributing substantial numbers of fitness computations. Therefore, we eliminated mutation operators from our design to explore larger populations more effectively in the same execution time.

Fitness Metric

PMEvo's evolutionary algorithm solves a multi-objective optimization problem (MOP) approximatively with the goal of minimizing two metrics: The average relative prediction error D_{avg} and the µop volume V. These metrics describe the quality of a port mapping $M = (\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, F \cup E)$ for a set $Exps$ of experiments e with measured inverse throughputs $tp^{-1}(e)$ as follows:

$$D_{avg}(M) = \frac{1}{|Exps|} \sum_{e \in Exps} \frac{|tp_M^{-1}(e) - tp^{-1}(e)|}{tp^{-1}(e)}$$

$$V(M) = \sum_{(i,n,u) \in F} n \cdot |u|$$

A low value for $D_{avg}(M)$ ensures accurate predictions whereas a smaller µop volume indicates a more compact and therefore more interpretable mapping.

We solve the MOP through a priori scalarization, as described, e.g., in Chapter 4.1 of the textbook by Miettinen (1998): We combine the objectives into a single fitness function $\mathcal{F}(M)$ as follows:

$$\mathcal{F}(M) = \Lambda_1(D_{avg}(M)) + \Lambda_2(V(M))$$

Λ_1 and Λ_2 are affine transformations that are chosen in every iteration to normalize both objective metrics to the range $[0, 1000]$. They ensure that the extremal objective values of the current population are mapped to 0 and 1000, respectively, with all other objective values in between.

Combining the accuracy metric D_{avg} with a compactness metric is necessary because inverse throughput measurements usually do not uniquely identify a single port mapping. The port mapping model is flexible enough to allow for a wide range of well-performing mappings with different characteristics. In the available port mappings for Intel microarchitectures by Abel and Reineke (2019), instructions with large numbers of µops that can execute on many different ports are considerably rarer than in the results of an evolutionary algorithm that only optimizes D_{avg}.

By default, this fitness metric does not consider the influence of bottlenecks, e.g., in the processor's frontend, on the measured throughput (cf. Section 3.2.2). We can adjust the above

definition of $D_{avg}(M)$ for a bottleneck of R_{max} instructions per cycle by replacing the simulated inverse throughput $tp_M^{-1}(e)$ with a version that uses the bottleneck throughput if it is slower than the simulated throughput:

$$\widehat{tp_M^{-1}}(e) := \max\left(tp_M^{-1}(e), \frac{|e|}{R_{max}}\right)$$

Local Search

Once the fitness of the port mapping population converges to a uniform level, every individual $M = (\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, F \cup E)$ of the final population is used as a starting point for a greedy local search to further increase the fitness scores. For each entry $(i, n, u) \in F$, indicating that the candidate port mapping includes n instances of μop u for instruction i with $n > 0$, the local search operates as follows:

- Decrement n by one and evaluate the fitness of the resulting port mapping. If the new mapping is at least as fit as the previous candidate, continue to decrement n in steps of one until the fitness decreases or $n = 0$ is reached.
- Otherwise, if decrementing n reduced the fitness of the port mapping candidate, increment it instead. Increment n for as long as the new candidate mapping is strictly fitter than its predecessor.

In either case, use the value of n that yields the best fitness and continue with the next entry of F. The asymmetric construction of this greedy search algorithm – decrementing n is performed if it improves fitness or if fitness does not change, whereas incrementing only happens if it strictly improves fitness – further favors compact, more interpretable port mappings.

Among the optimized port mappings, the fittest one is returned as PMEvo's result.

5.1.4. Efficient Bottleneck Simulation Algorithm

The practical applicability of an evolutionary algorithm depends on how fast the fitness of individuals in the population can be computed. For a given time budget, fitness evaluation speed directly corresponds to the quality of the obtained solution. With faster fitness evaluation, more candidates can be considered, resulting in superior solutions.

Therefore, a critical component of our approach is the efficient simulation of experiments with a given port mapping. Instead of directly solving the linear program from Theorem 3.12, we use a bottleneck simulation algorithm to compute the optimal solution of the linear program. We restrict our presentation here to port mappings in the two-level model for a more concise description. As we have observed in Remark 3.13, this extends to the three-level model straightforwardly.

The bottleneck simulation algorithm implements the following characterization of the inverse throughput of an experiment with a port mapping:

Theorem 5.2. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment e with the port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ can be equivalently characterized as follows:

$$tp_M^{-1}(e) = \max_{Q \subseteq \mathbf{P}} \frac{\sum\{e(i) \mid M[i] \subseteq Q\}}{|Q|} \tag{5.1}$$

$M[i] := \{k \mid (i,k) \in E\}$ denotes the set of ports that can execute an instruction i with M.

Proof. See Appendix A.3.1. □

For any subset $Q \subseteq \mathbf{P}$, the term

$$B(Q) := \frac{\sum\{e(i) \mid M[i] \subseteq Q\}}{|Q|}$$

bounds $tp_M^{-1}(e)$ from below since any instruction i with $M[i] \subseteq Q$ needs to be executed on ports in Q. The fastest way all such instructions could be executed is if all ports in Q are fully utilized with them, which requires $B(Q)$ cycles.

The same insight that also inspired the parametric SMT formulation presented in Theorem 4.4 implies that there is a port subset Q^* such that $tp_M^{-1}(e) = B(Q^*)$: A distribution to ports is optimal if and only if there is a non-empty set Q^* of bottleneck ports that are all utilized for the full number of cycles with instructions that can only be executed on ports in Q^*. In other words, $tp_M^{-1}(e)$ is equal to the total mass of instructions that need to be executed on ports from Q^*, divided by the size of Q^*, i.e., $B(Q^*)$. Consequently, finding a port subset Q that maximizes $B(Q)$ gives us precisely the inverse throughput $tp_M^{-1}(e)$.

Our algorithmic implementation of this characterization computes the max operation in Equation (5.1) by enumerating all subsets of the set of ports and evaluating the corresponding term. The execution time of this algorithm is in $\Theta(2^{|\mathbf{P}|})$, which is substantially more expensive than the polynomial execution time of LP solving (Bertsimas and Tsitsiklis, 1997) from a complexity-theoretic point of view. Nevertheless, this algorithm is considerably faster for practical problems, as we show in Section 5.2.3. On the one hand, this is due to the small number of execution ports available in modern systems. Practical port numbers in typical systems range from eight (e.g., Intel Skylake (Intel, 2023a, Section 2.6) and ARM A72 (ARM, 2015)) over ten (e.g., AMD Ryzen (AMD, 2021b)) to 12 (e.g., Intel Golden Cove (Intel, 2023a, Section 2.3)). On the other hand, thanks to the simplicity of the above algorithm, it is amenable to aggressive performance optimizations such as vectorization.

5.2. Experimental Evaluation

In this section, we describe the evaluation of PMEvo as we have performed it for the original publication (Ritter and Hack, 2020). The evaluated version of PMEvo does not include the dedicated treatment for throughput bottlenecks outside of the port mapping model described in Section 5.1.3. Throughput measurements were performed with an earlier version of the mechanism described in Section 3.2.1. We will revisit PMEvo for another evaluation in Section 6.3.6, where we compare it to the more recent Palmed (Derumigny *et al.*, 2022a) and the alternative port mapping inference algorithm we describe in Chapter 6.

5.2.1. Setup

Evaluated Processors

We use three devices with processors of distinct manufacturers for our evaluation, denoted as SKL, ZEN, and A72 in the following. Relevant parameters are listed in Table 5.1. SKL has a separate pipeline for long-running operations, marked as DIV, that has to be modeled as an additional port. One port of A72 is only used for processing branch instructions (BR). It is omitted in our model as we do not consider instructions that alter control flow. All evaluated systems have frequency scaling and flexible overclocking mechanisms (e.g., Intel Turbo Boost) disabled to facilitate reliable measurements.

A72 and ZEN are of particular interest since they do not provide the per-port performance counters that other approaches rely on (AMD, 2019; ARM, 2016) whereas SKL allows a comparison to related work.

Considered Instructions

We select for each instruction set architecture under test a relevant set of instruction schemes. These sets are derived from the instructions that compilers emit when compiling the SPEC CPU 2017 benchmarks (Bucek *et al.*, 2018). Our instruction schemes for the ARMv8-A ISA are extracted from the instructions that GCC emits.[2] For x86-64, we only extract the used instruction mnemonics from the output of Clang[3] and use the machine-readable instruction table of uops.info (Abel and Reineke, 2019) to generate the corresponding instruction schemes.

We exclude the following instructions from these sets:

- Branch and jump instructions, since their throughput heavily depends on the branch predictor.
- Instructions with implicitly read operands, since these cause dependencies that cannot be resolved through register allocation. Throughput for these could be measured by introducing additional dependency-breaking instructions as done by Abel and Reineke (2019).
- x86 SSE instructions, since these add transition penalties when benchmarked together with AVX instructions.
- All instruction variants that operate on sub-registers, to keep the run time of the evaluation bearable.
- x86 instructions that are not supported by Ithemal (Mendis *et al.*, 2019), to have a common baseline for all comparisons.

The resulting instruction descriptions contain 310 instruction schemes for the x86-64 ISA and 390 instruction schemes for the ARMv8-A ISA.

[2]We used version 4.9.4 of GCC with the `-O3` flag.
[3]We used version 8.0 of Clang with the `-O3 -mavx2` flags.

Table 5.1. Properties of the systems under investigation.

	SKL	ZEN	A72
Manufacturer	Intel	AMD	RockChip
Processor	Core i7 6700	Ryzen 5 2600X	RK3399
Microarchitecture	Skylake	Zen+	Cortex-A72
# Ports	8 + DIV	10	7 + BR
Instruction Set	x86-64	x86-64	ARMv8-A
Clock Frequency	3.4 GHz	3.6 GHz	1.8 GHz
RAM	32 GB	32 GB	4 GB

5.2.2. Model Predictions

Directly measuring the quality of a port mapping is hindered by the lack of ground truth for most processors. We therefore evaluate the inferred port mappings by their ability to accurately predict the measured inverse throughput of port-mapping-bound experiments. For each microarchitecture, we use a different benchmark set of 40,000 experiments, which we instantiate with operands and whose inverse throughput we measure as described in Section 3.2.1. These experiments are sampled uniformly at random from the set of all instruction multi-sets of size 5.

One major use case of PMEvo is to provide port mappings for performance estimation tools. Therefore, we compare the prediction accuracy of PMEvo's mappings to the modeling of port mappings in state-of-the-art performance prediction tools. To this end, we use the same benchmark sets to evaluate IACA (Intel, 2012) (version 3.0), llvm-mca (Di Biagio, 2018) (from LLVM version 8.0.1[4]), Ithemal (Mendis *et al.*, 2019), and the port mapping provided by uops.info (Abel and Reineke, 2019) for their respective supported platforms. Note that these benchmarks specifically stress the port-mapping aspect of these prediction tools because they do not contain any data dependencies. They are therefore not representative to evaluate the overall prediction quality of these tools on compiler-generated code.[5] Section 7.2 discusses the performance estimation tools we evaluate in further detail.

Of these four related approaches, only the port mapping from uops.info is directly comparable to PMEvo's results because it can only predict the inverse throughput of instruction sequences without data dependencies. The other approaches are more general in that they can predict the inverse throughput of arbitrary instruction sequences, but might not be attempting to provide good accuracy for dependency-free code. For example, Ithemal uses a neural network model trained via supervised learning rather than an explicit port mapping model.

[4]Initially, we performed these experiments on the more recent LLVM version 9.0.1 but found a severe regression in prediction accuracy on our experiments compared to version 8.0.1. See Section 8.3.1 for a discussion of this regression that has been fixed after our report.

[5]We refer to the BHive project (Chen *et al.*, 2019) for an evaluation of their accuracy for instruction sequences extracted from code generated for common benchmarks.

Table 5.2. PMEvo mapping characteristics.

	SKL	ZEN	A72
benchmarking time	20 h	27 h	74 h
inference time	5 h	21 h	12 h
instructions found congruent	69%	53%	56%
number of µops	17	15	9

Being trained on collected basic blocks from entire programs where dependencies are to be expected, accurate predictions for dependency-free code might be outside of the scope of Ithemal.

For all three platforms, we ran our PMEvo prototype with a population size of 100,000 and an ε of 0.05 for congruence filtering. Table 5.2 gives numbers on the time required to benchmark inverse throughputs for experiments and to infer a port mapping for all considered platforms. It further shows that the effectiveness of congruence filtering is considerable: The relevant instructions are reduced by 53%–69%. The low number of different µops used in the inferred port mappings indicates that PMEvo developed compact representations for all three platforms. The uops.info port mapping for SKL uses 12 different µops for the same set of instructions.

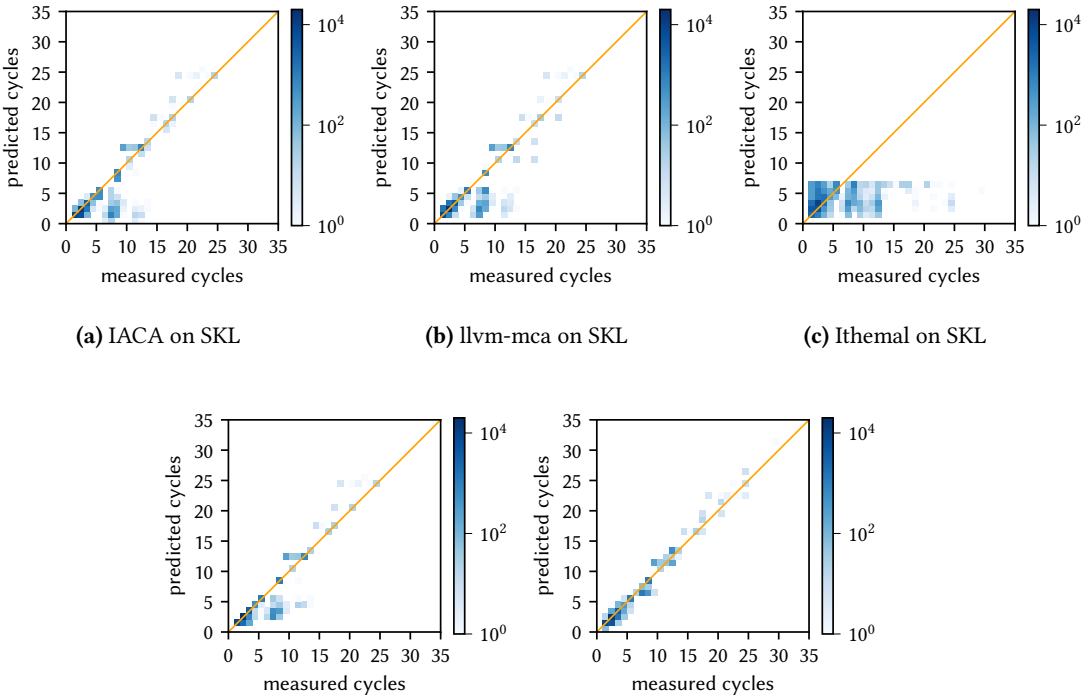
We report the prediction accuracy in terms of the mean absolute percentage error (MAPE), Pearson's correlation coefficient ρ_P, and Kendall's ranking correlation coefficient τ_K. A high ρ_P indicates a linear correlation of predictions and measurements whereas a high τ_K implies that sorting the instruction sequences by predicted or measured inverse throughput leads to similar rankings. The value range for ρ_P and τ_K is $[-1, 1]$, ranging from negative correlation (-1) over no correlation (0) to maximal correlation (1).[6]

Additionally, we visualize the prediction accuracy of our approach in comparison to related work with a heat map for each pair of architecture and prediction mechanism. For each heat map, the experiments are considered as data points with measured and predicted inverse throughput. The heat map visualizes where these data points fall among 35×35 equally sized bins. Each bin's shade represents the number of experiments that lie in it. Ideally, measurement and prediction agree, leading to experiments close to the marked diagonal line. Experiments below the diagonal indicate an underestimation of the inverse throughput, those above are overestimated by the predictor.

SKL

For the Intel Skylake platform, we compare the prediction accuracy of PMEvo to all aforementioned approaches: the port mapping from uops.info, IACA, llvm-mca, and Ithemal using its publicly-available pre-trained network for the Skylake microarchitecture.

[6] See Section 3.2.2 for more on these accuracy metrics.



(a) IACA on SKL **(b)** llvm-mca on SKL **(c)** Ithemal on SKL

(d) uops.info on SKL **(e)** PMEvo on SKL

Figure 5.2. Prediction accuracy on port-mapping-bound experiments for SKL. Each heat map relates predicted and measured inverse throughput in cycles per experiment. Filled bins closer to the diagonal line indicate better predictions. The experiments were set up and measured as described in Section 3.2.1. (Ritter and Hack, 2020)

The inputs for IACA, llvm-mca, and Ithemal consist of the loop body of the experiments, unrolled to a length of ten instructions so that operand allocation can avoid loop-carried dependencies. For the entire set of experiments, we report the results of the tools for this input, divided by the number of experiments in the unrolled loop body.

The accuracy metrics for the five tools are listed in Table 5.3.

IACA, llvm-mca, and uops.info all predict with an average error of less than 10% with high correlation values. This impression is confirmed by the corresponding heat maps in Figures 5.2 (a), 5.2 (b), and 5.2 (d): Most of the experiments are close to the ideal line. They also all show a cluster of experiments below the diagonal line. These can be attributed to the family of bit test instructions (BT, BTC, BTR, BTS), for which the measurable inverse throughput does not agree with the inverse throughput implied by the port usage. The throughput measurements of Abel and Reineke (2019) confirm this discrepancy.

Table 5.3. Prediction accuracy for port-mapping-bound experiments on SKL.

	MAPE	Pearson's ρ_P	Kendall's τ_K
IACA	8.0%	0.86	0.71
llvm-mca	9.7%	0.87	0.71
Ithemal	60.6%	0.35	0.41
uops.info	9.3%	0.91	0.79
PMEvo	14.6%	0.98	0.76

Our approach, PMEvo, has a higher relative error than IACA, llvm-mca, and uops.info, but comparable correlation coefficients. The corresponding heat map in Figure 5.2 (e) shows a distribution close to the diagonal line. The bit test instructions that caused inaccuracies for the other approaches have a representation as multiple µops that map to the same ports. While differing from the real port mapping, this fits better to the observable inverse throughputs.

For Ithemal, we observe lower correlations and a high prediction error. This differs from the evaluation by Mendis *et al.* (2019) where Ithemal exhibits superior results in these metrics in comparison to IACA. Their findings for the accuracy of IACA are consistent with the ones presented here. As already noted, the difference in performance is likely a consequence of the different characteristics of the experiments used here and in the experimental evaluation of their paper: Ithemal is trained and validated on basic blocks emitted from a compiler for entire programs, which exhibit substantially more data dependencies than our experiments.

However, an appropriate interpretation of these results needs to be judicious: A high prediction accuracy for our experiments could have indicated a generalization of Ithemal to dependency-free code. Yet, the observed low prediction accuracy for our inputs does not allow conclusions about Ithemal's performance across real-world programs.

While PMEvo's inferred port mapping provides reasonably accurate throughput predictions, the inferred structures rarely match the port mapping from uops.info. For instance, instructions that only use one µop that can be executed on SKL's four ports with arithmetic logic units according to uops.info use a µop with three available ports in PMEvo's mapping. Generally, the entries of PMEvo's port mapping are simpler than the corresponding ones from uops.info. Double-precision vector additions with and without memory operand, for example, are represented as follows in uops.info's port mapping:

$$\texttt{vaddpd}\ \langle \texttt{YMM} \rangle_W, \langle \texttt{YMM} \rangle_R, \langle \texttt{YMM} \rangle_R \rightsquigarrow \{\{0,1\}\}$$
$$\texttt{vaddpd}\ \langle \texttt{YMM} \rangle_W, \langle \texttt{YMM} \rangle_R, \langle \texttt{MEM[256]} \rangle_R \rightsquigarrow \{\{0,1\},\{2,3\}\}$$

PMEvo's port mapping also identifies an interference between these instructions, but with fewer µops:

$$\texttt{vaddpd}\ \langle \texttt{YMM} \rangle_W, \langle \texttt{YMM} \rangle_R, \langle \texttt{YMM} \rangle_R \rightsquigarrow \{\{2,4\}\}$$
$$\texttt{vaddpd}\ \langle \texttt{YMM} \rangle_W, \langle \texttt{YMM} \rangle_R, \langle \texttt{MEM[256]} \rangle_R \rightsquigarrow \{\{4,6\}\}$$

Table 5.4. Prediction accuracy for port-mapping-bound experiments on ZEN and A72.

	MAPE	Pearson's ρ_P	Kendall's τ_K
PMEvo (ZEN)	13.5%	0.94	0.76
llvm-mca (ZEN)	50.8%	0.86	0.40
PMEvo (A72)	21.4%	0.68	0.64
llvm-mca (A72)	65.7%	0.67	0.51

As the uops.info port mapping is the result of a constructive inference algorithm that exploits hardware performance counters, we expect that it is a more faithful characterization of the Skylake microarchitecture than PMEvo's mapping.

ZEN and A72

For the AMD and ARM microarchitectures, we compare PMEvo's results only to llvm-mca since the other approaches are only available for Intel architectures.

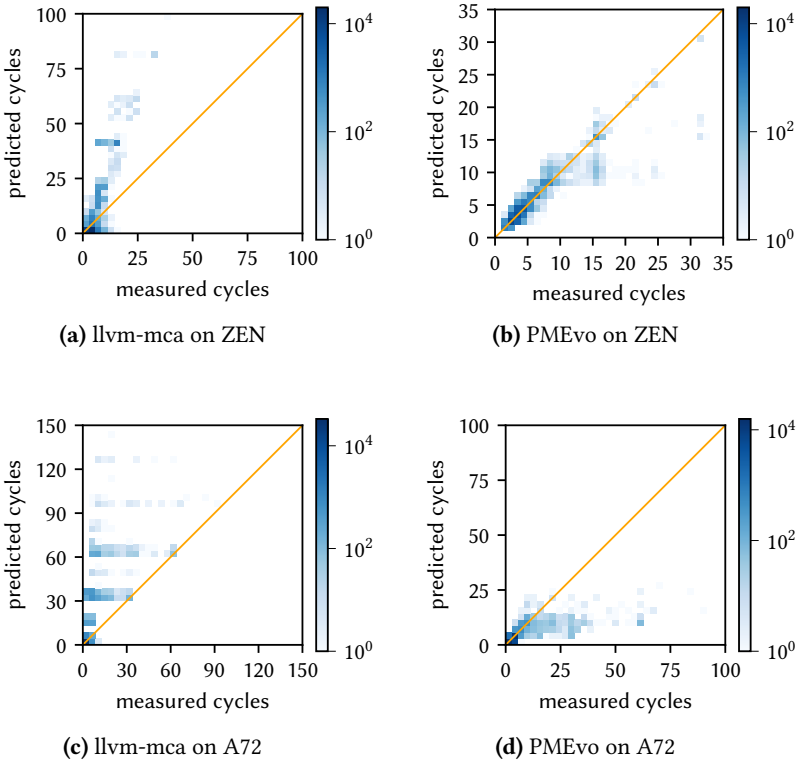
The metrics for both architectures in Table 5.4 show a common trend: PMEvo exhibits a considerably smaller prediction error than llvm-mca.

For ZEN, PMEvo inferred a port mapping that predicts with close to equal accuracy as its SKL mapping. With 21.4%, the prediction error of the PMEvo mapping for A72 is notably higher while correlations are lower. This observation is confirmed by the corresponding heat maps in Figure 5.3. PMEvo on A72 is prone to underestimating experiments with longer running times. We attribute this to A72's less elaborate out-of-order execution engine (according to the respective optimization guides (AMD, 2021b; ARM, 2015; Intel, 2023a)), which renders the experiments less representative for the port mapping. Limits on the extent to which instructions can be reordered and the number of concurrently dispatched µops may slow the execution in ways that the port mapping model does not capture.

In contrast to its results for SKL, llvm-mca has substantially larger prediction errors. The heat maps indicate a significant overestimation of the inverse throughput. One possible explanation is that these architectures are less in the focus of the developers than SKL and the respective port mapping models might not yet be as elaborate and accurate as the one for SKL. Especially for these two architectures, the models derived with PMEvo may significantly increase the accuracy of llvm-mca's throughput prediction.

5.2.3. Performance of the Bottleneck Simulation Algorithm

This section explores the performance behavior of the bottleneck simulation algorithm as presented in Section 5.1.4. For this purpose, we compare our optimized implementation of the bottleneck simulation algorithm to a realization of the linear program from Theorem 3.12 in



(a) llvm-mca on ZEN

(b) PMEvo on ZEN

(c) llvm-mca on A72

(d) PMEvo on A72

Figure 5.3. Prediction accuracy on port-mapping-bound experiments for ZEN and A72, note the differently scaled axes. Each heat map relates predicted and measured inverse throughput in cycles per experiment. Filled bins closer to the diagonal line indicate better predictions. The experiments were set up and measured as described in Section 3.2.1. (Ritter and Hack, 2020)

the state-of-the-art LP solver Gurobi (Gurobi Optimization, LLC, 2023).[7] The running times reported for the LP version include model construction via the Gurobi C++ API as well as the actual solving time. Since Gurobi's running times with multiple threads were indistinguishable from single-threaded mode, all reported numbers were obtained with a single thread on SKL.

There are two significant parameters that influence the execution time of both simulation methods: the number of ports in the microarchitecture and the length of the experiments.[8] We evaluate these parameters with randomly generated microarchitectures with the appropriate number of ports and an artificial instruction set architecture of 100 instructions. Figure 5.4

[7] We use version 7.5.2 of Gurobi for this evaluation.

[8] The number of instructions in the instruction set architecture is not relevant, since both implementations ignore instructions that do not occur in the experiment.

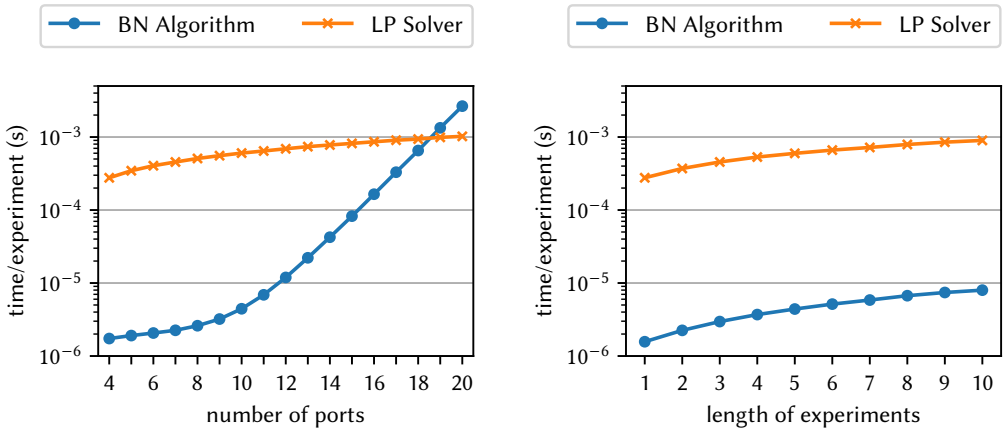


Figure 5.4. Execution time comparison of the bottleneck simulation algorithm and the LP solver with varying port numbers with experiments of length 4 (a) and with varying length of experiments with 10 ports (b). Both have their vertical axis in a logarithmic scale.

displays the results of this evaluation. For each (number of ports, length of experiments) configuration, 128 randomly sampled experiments were simulated with each of 8 randomly sampled three-level mappings. The resulting seconds per experiment value for each pair of experiment and mapping is the arithmetic mean over 1000 simulations. The points in the graph mark the median of these values for each (number of ports, length of experiments) configuration. In all cases, the variance in execution time around this median is too low to be visible in the graphs; error bars are therefore omitted.

Influence of the Number of Ports Figure 5.4 (a) shows the results for experiments consisting of 4 instructions with a varying number of ports. For port numbers up to 12 as they occur in contemporary platforms, the bottleneck simulation algorithm outperforms the linear program by two orders of magnitude. Starting from 12 ports, the simulation time with the bottleneck simulation algorithm rises with a stronger incline. The bottleneck simulation algorithm reaches the simulation time of the LP implementation at about 18 ports. With the same inputs, the simulation time via the LP solver grows substantially slower with the number of ports. We conclude that the exponential run-time behavior of the bottleneck simulation algorithm, as explained in Section 5.1.4, has a negligible impact for inputs of interest.

Influence of the Length of Experiments The experiments we use for the evolutionary algorithm are of very limited length to allow reliable execution on actual processors. Nevertheless, exploring the behavior with different lengths of experiments is worthwhile for the discussion of the bottleneck simulation algorithm. The results for varying lengths

of experiments in an architecture with 10 ports are displayed in Figure 5.4 (b). Here, the bottleneck simulation algorithm consistently outperforms the LP solver by two orders of magnitude. The execution time for both methods grows sub-exponentially with the length of experiments, with an almost identical incline in the log-scale plot. This indicates that the rate at which the execution time rises with growing experiment length for the LP solver is considerably higher than for the bottleneck simulation algorithm.

5.3. Conclusions: PMEvo

PMEvo is the first published automatic port mapping inference approach that does not depend on specific hardware performance counters. Instead, it measures the throughput of a fixed set of port-mapping-bound microbenchmarks and applies an evolutionary algorithm to search for port mappings whose throughput predictions coincide with the measurements. The inherently parallelizable evolutionary algorithm in combination with our vectorizable bottleneck simulation algorithm to compute port mapping fitness allow us to leverage modern processing hardware effectively for port mapping inference.

As PMEvo only depends on time and clock-frequency measurements, it is widely applicable, allowing us to infer port mappings for microarchitectures where none have been available before. A drawback of PMEvo is that it only approximates solutions to the online port mapping inference problem. While the resulting port mappings may provide adequate throughput prediction accuracy, their structure usually does not match the actual hardware. When comparing PMEvo's inferred port mapping for Intel's Skylake microarchitecture to available results by Abel and Reineke (2019), we find substantial structural differences. In contrast to the port mapping inference algorithm of uops.info (Abel and Reineke, 2019), PMEvo cannot provide explanatory benchmarks that witness each inferred port usage and therefore bolster confidence in the resulting models. We address these limitations of PMEvo in the following Chapter 6.

Chapter 6

Explainable Port Mapping Inference with Sparse Performance Counters

In Chapter 5, we presented a method to infer port mappings that is applicable when we can only observe the execution time of microbenchmarks. While the resulting port mappings are suitable to estimate how instructions execute in parallel, we cannot ensure that inferred mappings closely mirror the structure of the actual hardware. The port mapping inference algorithm for uops.info by Abel and Reineke (2019) can provide more confidence in the structure of the resulting port mappings. Using hardware performance counters, they can observe on which ports of the processor µops are executed when running a microbenchmark. This allows them to design microbenchmarks that serve as an explanation of the resulting port usage: When the uops.info algorithm determines that a µop u for an instruction can be executed on a set p of ports, there are experiments that show that u can be executed on any port in p as well as experiments demonstrating that u cannot be executed on any other port.

In this chapter, we establish that not all performance counters used in the uops.info algorithm are required for such an explainable algorithm when the investigated processor adheres to the port mapping model. We present an algorithm in a similar style as the uops.info algorithm that does not use per-port µop counters. Aside from time measurements, only one performance counter is required to count the total number of µops executed in a microbenchmark. In contrast to the uops.info algorithm, our adaptation therefore applies to AMD's Zen microarchitectures, which are documented to provide this performance counter (AMD, 2019, Section 2.1.15.4.5). Our algorithm thus alleviates a practical limitation of the original algorithm, while providing similar insights into the structure of the actual hardware.

The gain in applicability, however, comes at the cost of robustness: Our algorithm requires that the processor operates very closely to the port mapping model and that we can perform very accurate throughput measurements. We evaluate this approach and its trade-offs with a detailed case study of the AMD Zen+ microarchitecture in Section 6.3. In Section 7.1, we highlight how our algorithm differs from Palmed by Derumigny *et al.* (2022a), who also adapted the basic idea of the uops.info algorithm to infer a different kind of resource model.

The work described in this chapter has been published at the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024) (Ritter and Hack, 2024).

6.1. Starting Point: The uops.info Algorithm

The port usage algorithm presented by Abel and Reineke (2019, Section 5.1) relies on blocking instructions. A blocking instruction for a subset $Q \subseteq \mathbf{P}$ of ports in a three-level port mapping $M := (\mathbf{I} \cup \mathbf{U} \cup \mathbf{P}, F \cup E)$ is an instruction that is decomposed into a single µop that can be executed on exactly the ports in Q. Formally, $i \in \mathbf{I}$ is a blocking instruction for $Q \subseteq \mathbf{P}$ if, and only if,

$$\exists u \in \mathbf{U}.\ \left\{(i', n', u') \in F \;\middle|\; i = i'\right\} = \left\{(i, 1, u)\right\} \wedge \left\{k \;\middle|\; (u, k) \in E\right\} = Q$$

The algorithm needs blocking instructions for the port sets of every µop in the port mapping. Abel and Reineke find suitable blocking instructions with microbenchmarks. In contrast to the experiments that we have seen so far in this thesis, Abel and Reineke do not only observe the number of cycles required to execute the code in a steady state, but also the results of several hardware performance counters. They use performance counters that register the number of µops executed for each individual execution port. These counters make it straightforward to find and characterize blocking instructions: An instruction i is a blocking instruction if we count a single µop per executed i instruction and it blocks exactly the ports where µops are counted when i is executed repeatedly without data dependencies.

Example 6.1. Consider the instruction scheme of a 32-bit integer addition with only register operands on the Intel Skylake microarchitecture:

$$\texttt{add}\ \langle \texttt{GPR[32]} \rangle_{RW}\texttt{,}\ \langle \texttt{GPR[32]} \rangle_{R}$$

When we benchmark this instruction scheme in a steady state, the hardware performance counters of the Intel architecture show that

(a) an add instruction requires 0.25 cycles on average, i.e., four of them can execute in a single cycle,

(b) the processor executes one µop per add instruction, and

(c) the µops are executed in equal parts on ports 0, 1, 5, and 6 of the microarchitecture (which has a total of 8 ports).

This add instruction scheme can therefore block the port set $\{0, 1, 5, 6\}$. Observation (c) requires per-port µop counters that are not present in AMD's processors. ⌟

Abel and Reineke select one blocking instruction for each occurring port set. A special case are the µops required to store data into memory: On Intel's microarchitectures, these come only in pairs: one µop for the address generation and one to transfer the data. Abel and Reineke use a single blocking instruction for both µops. They then proceed according to Algorithm 6.1.

Each instruction i is investigated individually to infer its port usage. The *foundUops* multiset of µops for the instruction i (line 2) is filled throughout a run of the algorithm. The algorithm benchmarks the interaction of i with each blocking instruction B for a set Q of ports, starting with blocking instructions for the smallest port sets and proceeding to increasing port set

Input: instruction under investigation i
1 $blockingInsns \leftarrow$ tuples of blocking instructions with their blocked ports, sorted by ascending number of blocked ports
2 $foundUops \leftarrow \{\}$
3 **for** $(B, Q) \in blockingInsns$ **do**
4 $\quad k \leftarrow$ sufficient number of blocking instructions B to flood Q
5 $\quad uops \leftarrow measureUopsOnPorts(\{B \mapsto k, i \mapsto 1\}, Q)$
6 $\quad surplusUops \leftarrow uops - k$
7 $\quad$ **for** $Q', n \in foundUops$ **do**
8 $\quad\quad$ **if** $Q' \subset Q$ **then**
9 $\quad\quad\quad surplusUops \leftarrow surplusUops - n$
10 $\quad$ **if** $surplusUops > 0$ **then**
11 $\quad\quad foundUops[Q] \leftarrow surplusUops$
12 **return** $foundUops$

Algorithm 6.1. Port mapping inference for uops.info. (Abel and Reineke, 2019)

sizes. Each microbenchmark contains the instruction i and enough copies of the considered blocking instruction B such that any µop that *can* be executed on ports outside of Q *is* executed on these alternative ports (lines 4, 5). The number k of blocking instruction copies needs to be high enough to ensure that each blocked port in Q receives at least as many µops as i uses:

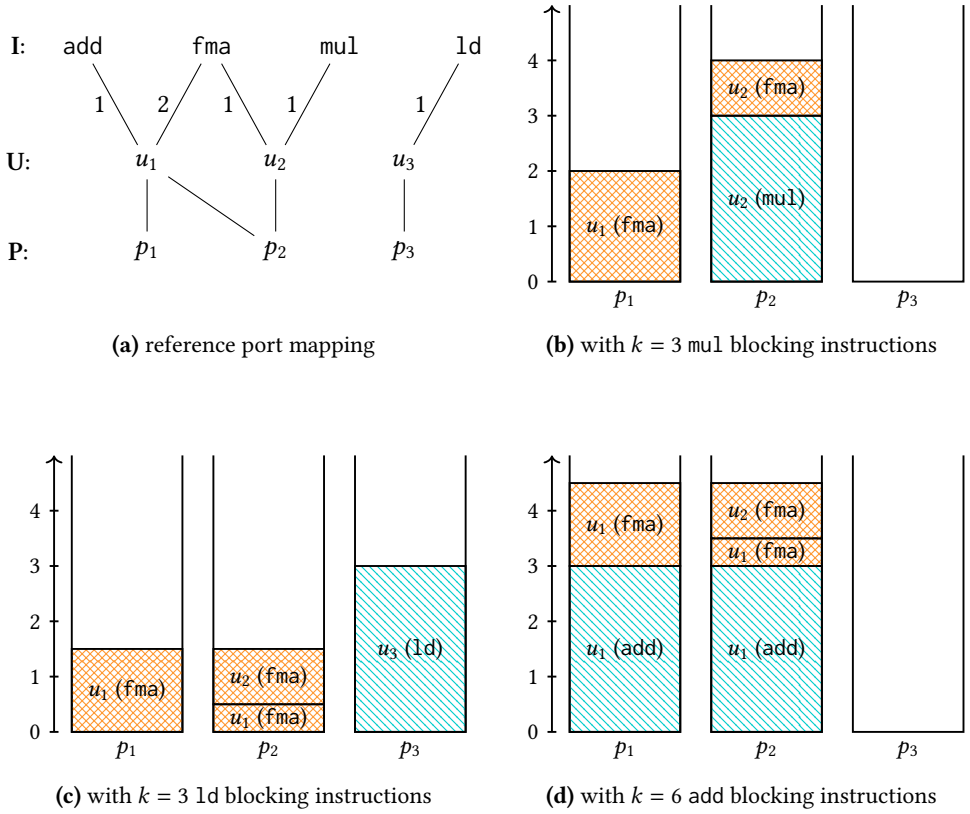
$$k \geq |Q| \cdot \mu opsOf(i) \tag{6.1}$$

Otherwise, ports in Q might be unoccupied while µops of i are issued, allowing µops of i to be executed on Q even though they could be executed on other ports.

When running this microbenchmark, Abel and Reineke count executed µops on ports in Q via per-port performance counters. The result is the sum of the number k of blocking instructions and the number of µops of i that only use ports in Q. These surplus µops include not only µops that can be executed on *all* ports of Q but also those that only have a subset of Q available. Because we assume that there is a blocking instruction for the port set of each occurring µop and because the blocking instructions are sorted by ascending number of blocked ports, all µops for proper subsets of Q were characterized in previous iterations of the loop. We can therefore subtract the numbers of these previously characterized µops from the surplus µops (lines 7–9) to obtain the number of µops of i that can use every port in Q (line 11). Once interactions with each blocking instruction are investigated, the port usage of i is completely characterized.

As a performance optimization, Algorithm 6.1 can terminate early once the number of characterized µops for i is equal to the total number of µops observed when benchmarking i on its own.

Example 6.2. Consider a processor that operates according to the port mapping in Figure 6.1 (a). With per-port µop counters, the uops.info algorithm finds three blocking instruc-



(a) reference port mapping

(b) with $k = 3$ `mul` blocking instructions

(c) with $k = 3$ `ld` blocking instructions

(d) with $k = 6$ `add` blocking instructions

Figure 6.1. Example port mapping (a) and possible steady-state distributions of µops per port in benchmarks with different blocking instructions (b-d).

tions: `mul` for the port set $\{p_2\}$, `ld` for $\{p_3\}$, and `add` for the ports $\{p_1, p_2\}$. For this example, we use $k := |Q| \cdot \mu opsOf(i)$ blocking instructions per benchmark.

When characterizing the port usage of the `fma` instruction, the algorithm first benchmarks `fma` with the blocking instructions for port sets of size 1, i.e., `mul` and `ld` (with $k = 1 \cdot 3 = 3$). For the benchmark with `mul` as blocking instruction, µops are distributed to the ports as shown in Figure 6.1 (b) in steady state execution. Measurements with per-port µop counters therefore indicate that four µops are executed on the port set $\{p_2\}$. With $k = 3$ blocking instructions and no µops that were characterized for smaller port sets, the algorithm concludes that `fma` uses $4 - 3 = 1$ µop that can be executed exclusively on $\{p_2\}$. For a similar benchmark with `ld` as blocking instruction (Figure 6.1 (c)), we observe only the blocking instructions on the port set $\{p_3\}$. This implies that `fma` has no µops that use (only) $\{p_3\}$.

Next, `fma` is benchmarked with the `add` instruction, which blocks a port set of size 2 (i.e., $k = 2 \cdot 3 = 6$). We measure a total of nine µops on the ports $\{p_1, p_2\}$. Subtracting the six

blocking instructions, three surplus µops remain. One of these surplus µops is explained by the $\{p_2\}$ µop that we found previously. The remaining two µops therefore need to be executed on the port set $\{p_1, p_2\}$.

No more blocking instructions remain, leaving us with the correct port usage for `fma`:

$$\{\{p_2\} \mapsto 1, \{p_1, p_2\} \mapsto 2\}$$

⌟

For the choice of the number k of blocking instructions (line 4 in Algorithm 6.1), Abel and Reineke (2019) propose in their article the product of the maximum latency of the instruction i under investigation and the number of ports of the microarchitecture. This choice does not satisfy Constraint 6.1: We found that it is, in general, not sufficient to fully flood the blocked ports, see Appendix E.1 for a counter example. The uops.info implementation[1] uses a different, more complex term for the number k of blocking instructions:

$$\begin{aligned} k_1 &\leftarrow 2 \cdot |Q| \cdot \max(1, \lfloor tp^{-1}\left(\{i \mapsto 1\}\right) \rfloor) \\ k_2 &\leftarrow |Q| \cdot \mu opsOf(i) \\ k &\leftarrow \min\big(\max(k_1, k_2, 10), 100\big) \end{aligned}$$

The result is a number of blocking instructions between 10 and 100 that depends on the size $|Q|$ of the blocked port set, the number $tp^{-1}\left(\{i \mapsto 1\}\right)$ of cycles required to execute i on its own in a steady state, and the total number $\mu opsOf(i)$ of µops into which i is decomposed. Since this term does not share the problem of the paper version for instructions i with a reasonable number of µops, we use it for our adapted algorithm as well. Compared to an implementation that only satisfies Constraint 6.1 tightly, we expect the typically larger number of blocking instructions in this term to be more resilient against non-optimal scheduling decisions in the hardware.

A key benefit of this port mapping inference algorithm is that the performed microbenchmarks serve as witnesses for the result: For each instruction i and each port set Q, there is an experiment explaining if i uses a µop for Q.

6.2. Our Adapted Algorithm

For AMD's Zen microarchitectures and several ARM designs, the uops.info algorithm is not applicable as they lack performance counters for executed µops per port. The key insight of this work is that the problems requiring per-port µop counters in the uops.info algorithm can be solved without them if we assume that the processor follows the port mapping model for some (unknown) port mapping. In the following, we provide alternatives for the relevant parts of the uops.info algorithm that do not use per-port µop counters. The only performance counter used, besides time measurements, counts the total number of µops executed for a microbenchmark.

[1] `https://github.com/andreas-abel/nanoBench/blob/faf75236cade57f7927f9ee949ebc679fc7864b7/tools/cpuBench/cpuBench.py#L3393`

6.2.1. Identifying Unique Blocking Instructions

The uops.info algorithm uses per-port performance counters to identify and characterize the blocking instructions. We propose an alternative approach:

1. Count the µops issued when executing each instruction individually. Each instruction with only a single µop is a candidate. Since these candidate instructions correspond to a single µop each, we can treat them as instructions in the simpler two-level model described in Section 3.1.1.

2. Determine for each candidate instruction i the number of ports that can execute its µop. It is equal to the number of instances of i that can be executed per cycle, i.e., the (non-inverse) throughput $tp\left(\left\{i \mapsto 1\right\}\right) = 1/tp^{-1}\left(\left\{i \mapsto 1\right\}\right)$, which we can measure.

3. Filter blocking instruction candidates that block the same port set so that only one blocking instruction remains for each port set. Two blocking instruction candidates cannot be redundant if their port sets have different sizes, as determined in the previous step. For two candidates i and j with equally sized port sets, we exploit the criterion of Theorem 6.3 (below) to check their port sets for equality: If

$$tp^{-1}\left(\left\{i \mapsto 1, j \mapsto 1\right\}\right) = tp^{-1}\left(\left\{i \mapsto 1\right\}\right) + tp^{-1}\left(\left\{j \mapsto 1\right\}\right)$$

   holds, i.e., the inverse throughputs are additive, the port sets of i and j are equal. We test this condition with throughput benchmarks.

4. Characterize the port usage of the found unique blocking instructions. This is a port mapping inference problem in the two-level model (since each involved instruction is known to correspond to a single µop). We use the counter-example-guided online port mapping inference algorithm described in Chapter 4 for this purpose. Since we have already characterized the number of ports per instruction, we additionally encode this information as constraints on the inferred port mappings.

The result is a set of blocking instructions with corresponding port usages that are indistinguishable from the port mapping used by the hardware.

Theorem 6.3. Let $M = (\mathbf{I} \cup \mathbf{P}, E)$ be a two-level port mapping and let $i, j \in \mathbf{I}$ such that $|M[i]| = |M[j]|$ and such that the following holds:

$$tp_M^{-1}\left(\left\{i \mapsto 1, j \mapsto 1\right\}\right) = tp_M^{-1}\left(\left\{i \mapsto 1\right\}\right) + tp_M^{-1}\left(\left\{j \mapsto 1\right\}\right)$$

Then, i and j use the same set of ports: $M[i] = M[j]$.

Proof. See Appendix A.4.1. □

We have seen that the SMT-based counter-example-guided algorithm from Chapter 4 does not scale to infer port mappings for realistic numbers of instructions on its own. It is, however, applicable in this setting. Since the task is limited to unique blocking instructions, only a small number of instructions, which behave according to the two-level model, need to be

considered. For example, the port mapping that Abel and Reineke (2019) inferred for the Intel Skylake microarchitecture contains only 12 distinct port sets.

Furthermore, the counter-example-guided inference algorithm handles situations like the storing µops of Intel microarchitectures, where the µops only occur in a fixed constellation without proper blocking instructions: The extension to the three-level model with a fixed instruction-to-µop mapping that we present in Section 4.4 and evaluate in Section 4.5.2 can allow more than one µop for specific instructions while treating the remaining blocking instructions according to the two-level model. We explore this in the case study in Section 6.3.4.

6.2.2. Determining How Many µops Cannot Evade the Blocking Instructions

The uops.info algorithm also relies on the per-port µop counters to determine how many µops are executed on ports that are flooded with blocking instructions. In the port mapping model, we do not need performance counters for this purpose. As in the algorithm, let the experiment $e := \{B \mapsto k, i \mapsto 1\}$ consist of the instruction i under investigation and k blocking instructions B for a port set Q. Consider further an experiment $e' := \{B \mapsto k\}$ that benchmarks the blocking instructions alone. If all µops of i can be executed on unblocked ports, $tp^{-1}(e) = tp^{-1}(e')$ holds. Otherwise, each µop of i that needs to be executed on Q utilizes one of the flooded ports for one cycle per iteration. Each such µop therefore adds $1/|Q|$ to the observed inverse throughput, i.e.:

$$tp^{-1}(\{B \mapsto k, i \mapsto 1\}) = tp^{-1}(\{B \mapsto k\}) + \frac{\text{µops of } i \text{ executed on } Q}{|Q|}$$

We can therefore compute the number of non-evading µops with the following formula:

$$\text{µops of } i \text{ executed on } Q = \left(tp^{-1}(\{B \mapsto k, i \mapsto 1\}) - tp^{-1}(\{B \mapsto k\})\right) \cdot |Q|$$

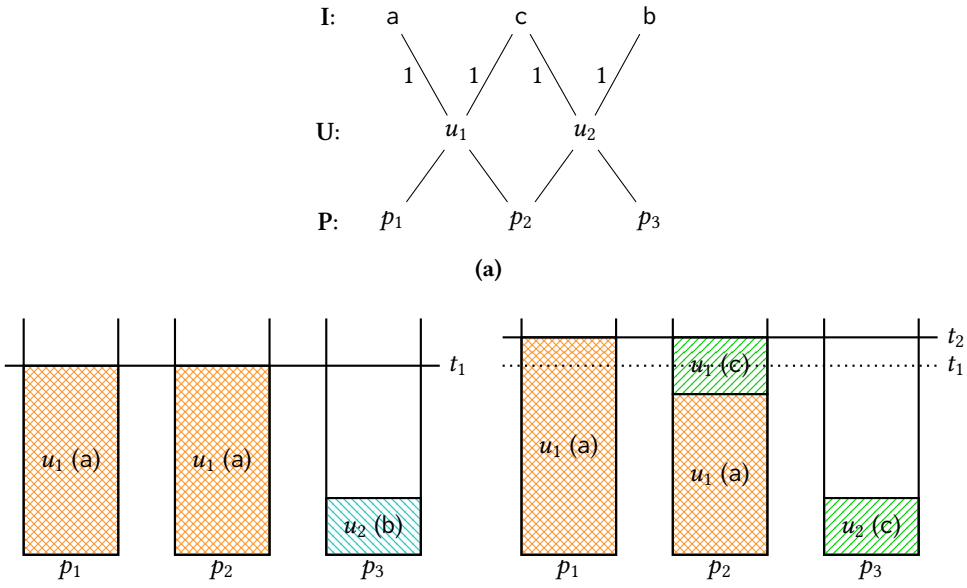
Example 6.4. Consider the three-level port mapping in Figure 6.2 (a). The instructions a and b are blocking instructions for the port sets $\{p_1, p_2\}$ and $\{p_2, p_3\}$, respectively. When we supply an experiment with a sufficient number k of copies of instruction a, the p_1 and p_2 ports are flooded with corresponding u_1 µops. If, as in Figure 6.2 (b), the tested instruction's µops (one u_2 µop for b) can be executed on other ports, they will only be executed on these other ports. The resulting inverse throughput t_1 is therefore the same as if only the blocking instructions were executed:

$$t_1 = \frac{k}{|\{p_1, p_2\}|}$$

In case a number n of µops of the instruction under test can only be executed on the flooded ports, like the u_1 µop of instruction c in Figure 6.2 (c), the load on all flooded ports rises. The inverse throughput t_2 is therefore higher:

$$t_2 = \frac{k+n}{|\{p_1, p_2\}|} = t_1 + \frac{n}{|\{p_1, p_2\}|} = t_1 + \frac{n}{2}$$

⌟



(b) Instruction b under test.

(c) Instruction c under test.

Figure 6.2. Example with a three-level port mapping (a) and schematic executions where (b) all µops under test evade the flooded ports or (c) one µop cannot evade.

6.2.3. Handling Pipeline Bottlenecks

As we have shown in Section 3.2.2, there is at least one way in which we expect modern processors to not behave according to the port mapping model. On modern x86-64 architectures, sustained full utilization of all execution ports is usually not possible because of bottlenecks in other parts of the CPU, commonly the decoding frontend and its caches or the instruction retirement. With a bottleneck that bounds the execution rate to at most R_{max} instructions per cycle, experiments that are faster according to the port mapping model are slowed to meet the limit.

Such bottlenecks can affect the correctness of our algorithm. When characterizing the port set sizes of a blocking instruction i (step 2 in Section 6.2.1) for a microarchitecture with a bottleneck of R_{max} instructions per cycle, one of three cases could occur:

- i can use less than R_{max} ports. Then, the microbenchmark is unaffected by the bottleneck.
- i can use exactly R_{max} ports. This can occur, e.g., on many Intel microarchitectures.
- i can use more than R_{max} ports. We cannot distinguish this from the previous case via microbenchmarks. However, this is unlikely to occur in practice, since such a processor would have resources that can never be fully utilized.

Furthermore, checking blocking instruction candidates from the latter two cases for equivalence (step 3 in Section 6.2.1) is futile: Any combination of two blocking instructions for R_{max} ports leads to at least R_{max} instructions executed per cycle according to the port mapping model. With an execution rate that is limited to at most R_{max} instructions per cycle, the condition for equivalence in Theorem 6.3 is therefore always fulfilled. In case the microarchitecture under investigation has blocking instructions for more than one port set of size $\geq R_{max}$, the algorithm thus yields wrong results.

After identifying unique blocking instructions in step 3 of Section 6.2.1, we therefore determine the processor's peak execution rate via measurements. This can be done by iteratively expanding a microbenchmark with instances of all blocking instructions as long as they increase the number of executed instructions per cycle (IPC). We use the maximal IPC that we observe with such a microbenchmark as the processor's peak execution rate. If the identified bottleneck execution rate is equal to the size of the port set of any blocking instruction, an implementation of the adapted algorithm can warn the user of the potentially wrong results.

With the bottleneck characterized, we apply the extension to the counter-example-guided port mapping inference algorithm described in Section 4.3 to handle the bottleneck in step 4 of Section 6.2.1. Some port mappings that are distinguishable according to the port mapping model become indistinguishable with this adjustment.

For the detection of evading instructions described in Section 6.2.2, a bottleneck is also only problematic if it can be hit with instances of a single blocking instruction scheme B. In this case, an experiment consisting only of copies of B already reaches the bottleneck with R_{max} instructions per cycle. We would then need to observe more than R_{max} instructions executed per cycle to determine that another instruction's µops can evade the blocked ports, which is impossible. Otherwise, if there is a gap between the maximal number of available ports per µop and the peak execution rate, the experiment consisting of k copies of the blocking instruction B yields at most $(R_{max} - 1)$ instructions per cycle and the algorithm works unchanged.

6.2.4. Supported Microarchitectures

Our algorithm has the following requirements:

- We need to measure the number of cycles required to execute a piece of code. Such functionality is commonplace in contemporary Intel, AMD, and ARM microarchitectures.
- There needs to be a counter for the total number of µops executed for a piece of code. Recent Intel Core architectures support this, and AMD's Zen, Zen+, and Zen2 are documented to support this as well.[2]
- The processor's throughput bottleneck should not be reachable by executing only instructions of the same kind. At the time of writing, this is the case for AMD's Zen-family microarchitectures (up to Zen4), with at most 4 ports per µop and a peak execution rate of at least 5 µops per cycle. Most Intel Core microarchitectures to this date do not fulfill this

[2] See Section 6.3.1 for more on this and the subsequent Zen-family microarchitectures.

requirement: The limit on the number of retired µops per cycle has only been increased from 4 to 8 with the Golden Cove microarchitecture,[3] while the maximal number of available ports per µop only increased from 4 to 5. (Intel, 2023a, Section 2.3)

Additionally, as the original uops.info algorithm, we assume that there is a blocking instruction for most µops of the microarchitecture. Where this requirement is not met, replacement instructions with a throughput dominated by the respective µop need to be specified manually. In the following case study, we show that, contrary to official documentation, such µops occur in AMD's Zen+ microarchitecture.

6.3. Case Study: The AMD Zen+ Microarchitecture

We evaluate our adapted explainable port mapping inference algorithm with the AMD Zen+ microarchitecture. This allows us to use PMEvo (Chapter 5) and Palmed (Derumigny *et al.*, 2022a) as points of comparison in Section 6.3.6. Since Zen+ does not have full per-port µop counters, the original uops.info algorithm is not applicable. We also compare our results to the available documentation for Zen+:

- AMD's Software Optimization Guide (AMD, 2021b) describes the microarchitecture, including a table that documents latencies and throughputs of instructions, if they are microcoded, and, for simple instructions, their execution units.
- Agner Fog's microarchitecture guide (Fog, 2023) analyzes the similar Zen architecture based on manual microbenchmarks.
- Fog (2022) and uops.info (Abel and Reineke, 2019) provide tables with measured latencies, throughputs, and numbers of µops of individual instructions. They include the port usage of floating point (FP)/vector instructions since per-port performance counters for these units are available (AMD, 2019, Section 2.1.15.4.1).
- WikiChip collects information on Zen and Zen+, in parts from AMD's marketing resources. (WikiChip, 2023a,b)

Our test system has an AMD Ryzen 5 2600X processor and 32 GB of RAM. It runs the port mapping inference algorithm and automatically performs microbenchmarks when required. Simultaneous multi-threading and frequency scaling are disabled as far as possible. Similar to findings by Fog (2023, Section 22.20), we found that (cycle-)accurate measurements are more difficult to achieve with AMD's Ryzen processors compared to contemporary Intel processors: Aggressive frequency scaling optimizations by AMD make extensive warm-up phases before each measurement and median-aggregation over several runs essential. We measure inverse throughput with the procedure described in Section 3.2.1 with an extension to read out the processor's performance counter for retired µops.[4] For each throughput measurement, we take the median over 11 repeated microbenchmark runs. We consider two

[3] Golden Cove is used in Alder Lake and Sapphire Rapids processors for client and server applications.
[4] We use the `PMCx0C1` ("Retired Uops") counter. (AMD, 2019, Section 2.1.15.4.5)

throughput measurements equal if the implied cycles per instruction (CPI) differ by at most $\varepsilon_{CPI} = 0.02$. This value allows us to distinguish if experiments use five ports (0.20 CPI) or four ports (0.25 CPI).

We take the x86-64 instruction schemes from uops.info and remove control flow and system instructions as well as instructions with known input-dependent performance characteristics (cf. Section 3.2.2). For floating-point and vector operations, we only consider instructions from the AVX and AVX2 instruction set extensions. This gives us 2,980 instruction schemes. We further reduce this set of instruction schemes when the need arises throughout the stages of the algorithm.

6.3.1. Identifying Blocking Instruction Candidates with Singleton Experiments

The first stage of the algorithm benchmarks every single instruction under investigation individually and measures its inverse throughput and the number of µops into which it is decomposed. Instructions that are executed with a single µop are blocking instructions.

Counting µops. Compared to AMD's documentation, we measure unexpected numbers of µops, e.g., for this instruction scheme:

$$\texttt{add}\ \langle \texttt{GPR[32]} \rangle_{RW}, \langle \texttt{MEM[32]} \rangle_{R}$$

It loads a value from memory, adds it to the value of a register, and writes the result back into the same register. According to AMD's Software Optimization Guide (AMD, 2021b, Table 1. "Typical Instruction Mappings"), such an instruction should be decomposed into two µops: one that loads and one that adds. However, the "Retired Uops" performance counter only increases by one for each such instruction. We make the same observation for any instruction involving memory operands, both with our measurement tool and with nanoBench (Abel and Reineke, 2020). The instruction tables by Fog (2022), which are also based on this performance counter, agree with our observations. The involved information sources are therefore in contradiction:

- the µop decomposition described in the AMD Software Optimization Guide (AMD, 2021b, Table 1. "Typical Instruction Mappings"),
- the documentation of the "Retired Uops" hardware performance counter in the AMD Processor Programming Reference (AMD, 2019, Section 2.1.15.4.5) (or its implementation in hardware), and
- our measurement methodology, as well as the one used by Abel and Reineke (2020) and Fog (2022).

While our inquiry with AMD's support remains unanswered, there is evidence that the "Retired Uops" performance counter, `PMCx0C1`, counts *macro-ops* instead of µops: We observe counter values that are consistent with the Software Optimization Guide's macro-op numbers.

Furthermore, the Processor Programming Reference for the more recent Zen 3 and 4 microarchitectures (AMD, 2021a, Section 2.1.15.4.5) documents that the performance counter with this identifier counts macro-ops on these microarchitectures.

As described in the AMD Software Optimization Guide for Zen+ (AMD, 2021b, Section 2.3 "Instruction Decomposition"), AMD's macro-ops are a representation between x86-64 instructions and the µops that are executed by the execution units.[5] The Zen+ microarchitecture implements many instructions with a single macro-op, whereas for example 256-bit-wide vector operations are implemented as two 128-bit-wide macro-ops. Complex instructions are microcoded with a greater number of macro-ops.

To the best of our knowledge, there is no detailed published information on how macro-ops are decomposed into µops and no suitable performance counter for experimental characterization on Zen+. Since our algorithm requires a count of µops, we postulate a macro-op-to-µop correspondence in the Zen+ microarchitecture, based on the high-level description and examples from AMD's Software Optimization Guide (AMD, 2021b, Section 2.3 "Instruction Decomposition"):

Postulate 6.5. Let n be the number of macro-ops observed when executing a basic block *bb* on the AMD Zen+ microarchitecture. We obtain the µop count by adding

- 1 for each memory operand with a width of at most 128 bits (excluding "load effective address" (`lea`) and loading move (`mov`) instructions),
- 2 for each memory operand with a width of 256 bits (as they are implemented as two 128-bit operations).

⌟

This strategy deviates in one aspect from the Software Optimization Guide: According to examples there, `mov` instructions that *store* a value from a general-purpose register to memory do not require an additional µop. This contradicts our observations:

- A store-`mov` together with four simple register-additions takes 1.25 cycles. Therefore, it has a µop that is restricted to the four ports with arithmetic logic units (ALUs).
- A `vmovapd` vector-register-to-memory store (documented with a store µop and one to deliver the stored data) together with the four additions takes only 1.0 cycles. Hence, no µops of this instruction are restricted to the ALU ports.
- A storing `mov` with a storing `vmovapd` leads to an inverse throughput of 2 cycles. These instructions therefore interfere, i.e., a µop of the `mov` instruction uses a port that the `vmovapd` instruction also needs.

Hence, the storing `mov` instruction has a µop that is restricted to one or more ALU ports and one for a non-ALU port. Therefore, similar to Intel architectures (Abel and Reineke, 2019, Section 5.1.1), there is no proper blocking instruction for memory store µops.

[5] AMD's macro-ops are also referred to as "ops", "MOPs", "complex OPs", or "COPs". The term is used differently by Intel in the context of their microarchitectures.

Problematic Instructions. For several instruction schemes, we observe violations of the algorithm's assumptions:

- `nop`s and 32 or 64-bit-wide register-to-register `mov`s use no ports: They execute at a rate of five instructions per cycle (IPC) whereas, according to documentation, no functional unit occurs at more than four ports, i.e., at most four instructions of a kind per cycle should be possible. The processor resolves such `mov`s via register renaming (Fog, 2023, Section 22.13) and implements `nop`s without µops. No port mapping is necessary for these cases.
- Some floating-point instructions execute slower than the port mapping model permits, e.g., divisions, square-root computations and approximate reciprocals.
- A `mov` of a 64-bit immediate into a general-purpose register causes unreliable measurements: The throughput observations cluster into groups with 1.0 instructions per cycle and 2.0 instructions per cycle (according to the uops.info tables, we would expect an IPC of 1.0). As these constants are unusual in the ISA, they use special handling in the hardware. For example, instructions with 64-bit immediate constants are documented to block multiple slots in the op-cache of the Zen+ microarchitecture (AMD, 2021b, Section 2.9).

We exclude all these instruction schemes, leaving 2,323 remaining schemes. Of these, 691 are identified as blocking instruction candidates.

6.3.2. Filtering Equivalent Blocking Instructions

The next step of the algorithm performs microbenchmarks for pairs of blocking instruction candidates with equally-sized port sets. If the number of cycles required to execute both instructions is equal to the sum of corresponding numbers for the individual instructions, the candidates are equivalent.

We encounter further problematic instructions in these experiments: Conditional move instructions, AES de/encryption operations, numerical conversions of the `vcvt*` family, and double-precision floating-point multiplications cause unstable measurements when benchmarked with other instructions. Floating-point/vector operations with three read operands, like fused multiply-and-add instructions and some vector blending operations, do not fit the port mapping model either in Zen+. While these operations can execute on two of the four ports of the floating-point unit, they use data lines of a third port. (AMD, 2021b, Section 2.11) This third port meanwhile has to idle, which we observe as contradicting equivalence information. We exclude these instructions from the following steps of the algorithm.

This leaves us with 1,887 instruction schemes in total, with 563 blocking instruction candidates. Of these candidates, 13 are identified as unique blocking instructions. Table 6.1 shows them with the number of candidates per equivalence class.

This selection of blocking instructions is consistent with the partial data available on uops.info: If we found two candidates equivalent and if they are covered by uops.info, then their reported port usages are equal. uops.info does not cover 266 of our 563 candidates.

AMD's instruction tables do not agree for 33 instruction schemes. For instance, AMD's documentation states that all of the following vector comparisons for integers of varying size can execute on the same two ports:

Table 6.1. Classes of blocking instructions for AMD Zen+ as identified in our adapted port mapping inference algorithm. Representatives were selected manually for clarity.

# Ports	Instruction Scheme	# Equiv.	Description
4	add $\langle$GPR[32]$\rangle_{RW}$, $\langle$GPR[32]$\rangle_R$	242	ALU ops
	vpor $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$	21	logical vector ops
3	vpaddd $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$	30	vector int. arith.
2	vminps $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$	143	FP compare, mul.
	vbroadcastss $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$	50	vector layouting
	vpaddsw $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$	17	saturating vector ops
	vaddps $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$	10	FP additions
	mov $\langle$GPR[32]$\rangle_W$, $\langle$MEM[32]$\rangle_R$	6	memory loads
1	vpslld $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$	27	vector shifts
	vpmuldq $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$	10	elaborate vector mul.
	imul $\langle$GPR[32]$\rangle_{RW}$, $\langle$GPR[32]$\rangle_R$	9	integer mul.
	vroundps $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$IMM[8]$\rangle_R$	4	vector rounding
	vmovd $\langle$XMM$\rangle_W$, $\langle$GPR[32]$\rangle_R$	2	vector-to-GPR mov

vpcmp**gtq** $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$
vpcmp**eqq** $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$
vpcmp**gtb** $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$XMM$\rangle_R$

In our measurements, only the second (testing equality for 2×64-bit integers) has two ports, whereas the first and third have one and three ports available (greater-than tests for 2×64-bit and 16×8-bit integers, respectively). Fog's table and uops.info agree with our measurements; this appears to be an error in AMD's documentation.

6.3.3. Measuring the Throughput Bottleneck

With the technique from Section 6.2.3, we observe a peak throughput of five instructions per cycle on the AMD Zen+ microarchitecture. This throughput is achieved with several different combinations of blocking instructions, e.g.:

$2\times$ mov $\langle$GPR[32]$\rangle_W$, $\langle$MEM[32]$\rangle_R$
$2\times$ vbroadcastss $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$
$1\times$ vroundps $\langle$XMM$\rangle_W$, $\langle$XMM$\rangle_R$, $\langle$IMM[8]$\rangle_R$

This peak IPC for single-µop instructions is consistent with the measurements of Fog (2023, Section 22.21): He found that the op-cache of the Zen architecture can issue up to five instructions per cycle. While he also reports that up to six µops per cycle can be executed,

this can only be observed when at least one instruction is decomposed into more than one µop. Since this step only considers blocking instructions, which are executed as a single µop, this case cannot occur in our setting.

As no µop has more than four ports available, the throughput bottleneck therefore does not affect our characterization of the microarchitecture as discussed in Section 6.2.3.

6.3.4. Computing a Mapping for the Blocking Instructions

Here, we compute a port mapping for the blocking instructions with the counter-example-guided port mapping inference algorithm presented in Chapter 4 with the extensions discussed in Section 4.3. We use version 4.12.1 of the Z3 SMT solver (de Moura and Bjørner, 2008) and select the same value of $\varepsilon_{CPI} = 0.02$ as in the rest of the algorithm to address noisy measurements. Following AMD's documentation (AMD, 2021b), we use a set of 10 ports: $\mathbf{P} = \{0, 1, \ldots, 9\}$.

As there are no proper blocking instructions for store operations, we add "improper" blocking instructions manually:

- `mov` $\langle$`MEM[32]`$\rangle_W$, $\langle$`GPR[32]`$\rangle_R$, which stores a 32-bit value from a general-purpose register into memory, and
- `vmovapd` $\langle$`MEM[128]`$\rangle_W$, $\langle$`XMM`$\rangle_R$, which stores a 128-bit value from a vector register into memory.

While we expect to use only the `mov` instruction in place of a blocking instruction for the store µop, both are required to infer that the store µop does not use an ALU instruction, cf. Section 6.3.1. We use the three-level version of the counter-example-guided port mapping inference algorithm with a fixed instruction-to-µop mapping. This instruction-to-µop mapping is chosen to ensure that all proper blocking instructions use only one µop and the above two improper blocking instructions use exactly two µops. We additionally encode for both improper blocking instructions that one of their µops must be equal to one of the µops of a proper blocking instruction. With these constraints, we avoid the prohibitively long execution times that come with the unconstrained three-level inference algorithm.

For three blocking instructions, the generated experiments exhibit throughputs outside of the port mapping model:

- The `imul` scheme for scalar integer multiplications, e.g., when combined with four additions:

  $$4\times \texttt{add}\ \langle\texttt{GPR[32]}\rangle_{RW}, \langle\texttt{GPR[32]}\rangle_R$$
  $$1\times \texttt{imul}\ \langle\texttt{GPR[32]}\rangle_{RW}, \langle\texttt{GPR[32]}\rangle_R$$

  Since `add` has four ports and `imul` is restricted to one, two inverse throughputs are possible in the port mapping model: 1.25 cycles, if `imul` uses a port of the `add` instruction, or 1.0 cycles, if their ports are disjoint. While AMD's Software Optimization Guide (AMD, 2021b, Section 2.10.2) indicates the former case, we measure ca. 1.5 cycles for this experiment, matching neither case.

Table 6.2. Documented and inferred port usage of the blocking instructions for the AMD Zen+ microarchitecture as shown in Table 6.1. The inferred ports were renamed bijectively for ease of comparison with available documentation.

Instruction Scheme	Doc. Ports	Inferred Ports
add ⟨GPR[32]⟩$_{RW}$, ⟨GPR[32]⟩$_R$	ALU	[6,7,8,9]
vpor ⟨XMM⟩$_W$, ⟨XMM⟩$_R$, ⟨XMM⟩$_R$	FP 0,1,2,3	[0,1,2,3]
vpaddd ⟨XMM⟩$_W$, ⟨XMM⟩$_R$, ⟨XMM⟩$_R$	FP 0,1,3	[0,1,3]
vminps ⟨XMM⟩$_W$, ⟨XMM⟩$_R$, ⟨XMM⟩$_R$	FP 0,1	[0,1]
vbroadcastss ⟨XMM⟩$_W$, ⟨XMM⟩$_R$	FP 1,2	[1,2]
vpaddsw ⟨XMM⟩$_W$, ⟨XMM⟩$_R$, ⟨XMM⟩$_R$	FP 0,3	[0,3]
vaddps ⟨XMM⟩$_W$, ⟨XMM⟩$_R$, ⟨XMM⟩$_R$	FP 2,3	[2,3]
mov ⟨GPR[32]⟩$_W$, ⟨MEM[32]⟩$_R$	AGU	[4,5]
vpslld ⟨XMM⟩$_W$, ⟨XMM⟩$_R$, ⟨XMM⟩$_R$	FP 2	[2]
vroundps ⟨XMM⟩$_W$, ⟨XMM⟩$_R$, ⟨IMM[8]⟩$_R$	FP 3	[3]
mov ⟨MEM[32]⟩$_W$, ⟨GPR[32]⟩$_R$	AGU	[5] + [6,7,8,9]
vmovapd ⟨MEM[128]⟩$_W$, ⟨XMM⟩$_R$	FP 2	[5] + [2]

- vpmuldq, which represents complex vector multiplication operations,[6] leads to experiments that run slower than their port usage would imply. Accommodating for this deviation from the ideal throughputs would require a larger ε_{CPI}, which would lead to a loss of accuracy for other instructions.

- For vmovd, we observe inconsistent resource conflicts when combined with different instructions. As this instruction scheme is untypical in that it transfers data between vector registers and the general-purpose registers, its throughput might depend on resources outside of the port mapping model.

In the algorithm, these inconsistencies lead to unsatisfiable SMT constraints in the *findMapping* method, where we attempt to find a port mapping that satisfies the inverse throughputs measured in the experiments so far. We exclude them and instructions with the same mnemonics (as we expect them to share aspects of the problematic instructions) from this investigation.

In three runs with the remaining blocking instructions, the algorithm terminated within 12–20 hours after generating 55–59 experiments with up to five instructions. Table 6.2 shows the inferred port mapping together with the documented port usage. For vector and floating-point instructions, where documented port usages are available, our port mapping is equivalent.

[6] This specific instruction multiplies the 32-bit integers at even-numbered vector lanes in the source registers without overflows into a vector of 64-bit integers.

Results for the add blocking instruction differ across repeated algorithm runs in whether a port is shared with the floating-point instructions: Besides the mapping in Table 6.2, "[6,7,8,9]", variants that use FP ports like "[0,6,7,8]" and "[1,6,7,8]" are possible. These variants are indistinguishable with the throughput bottleneck of five instructions per cycle. Which result we get depends on choices of the SMT solver. This ambiguity would be resolved with a less tight bottleneck or with blocking instructions for the individual FP ports or fine-grained subsets of the ALU ports. We use "[6,7,8,9]" in the rest of the algorithm as it is consistent with the documentation.

The results for the improper blocking instructions (at the bottom of the table) are consistent with the expectations: They have a µop (presumably for storing to memory) for port 5 in common. vmovapd has an additional µop for port 2, which uops.info reports as its port usage. For mov, the additional µop is an ALU µop, matching our observations from Section 6.3.1. This µop could also be restricted to a subset of the ALU ports, the blocking instructions are not sufficient to distinguish these cases.

6.3.5. Computing the Remaining Port Mapping

In the final step of the algorithm, we systematically benchmark each instruction that uses more than one µop against the suite of blocking instructions. To combat unstable measurements, we run this part of the algorithm three times and only report the port usage for an instruction if at least two of the runs agree. We use mov $\langle \mathtt{MEM}[32] \rangle_W$, $\langle \mathtt{GPR}[32] \rangle_R$ to block the store port 5.

The results follow regular patterns for most instructions:

- 256-bit wide AVX instructions (which operate on ymm vector registers) use µops of the same kinds as the corresponding 128-bit xmm variants, only with twice the number, for instance:

$$\begin{aligned} &\mathtt{vpcmpeqq}\ \langle \mathtt{XMM} \rangle_W, \langle \mathtt{XMM} \rangle_R, \langle \mathtt{XMM} \rangle_R && \rightsquigarrow 1 \times [0,3] \\ &\mathtt{vpcmpeqq}\ \langle \mathtt{YMM} \rangle_W, \langle \mathtt{YMM} \rangle_R, \langle \mathtt{YMM} \rangle_R && \rightsquigarrow 2 \times [0,3] \end{aligned}$$

- Instruction schemes with a read memory operand differ from their register-only counterparts by one load µop (two for double-pumped 256-bit AVX instructions). For example:

$$\begin{aligned} &\mathtt{add}\ \langle \mathtt{GPR}[32] \rangle_{RW}, \langle \mathtt{GPR}[32] \rangle_R && \rightsquigarrow 1 \times [6,7,8,9] \\ &\mathtt{add}\ \langle \mathtt{GPR}[32] \rangle_{RW}, \langle \mathtt{MEM}[32] \rangle_R && \rightsquigarrow 1 \times [6,7,8,9] + 1 \times [4,5] \end{aligned}$$

  and

$$\begin{aligned} &\mathtt{vpcmpeqq}\ \langle \mathtt{YMM} \rangle_W, \langle \mathtt{YMM} \rangle_R, \langle \mathtt{YMM} \rangle_R && \rightsquigarrow 2 \times [0,3] \\ &\mathtt{vpcmpeqq}\ \langle \mathtt{YMM} \rangle_W, \langle \mathtt{YMM} \rangle_R, \langle \mathtt{MEM}[256] \rangle_R && \rightsquigarrow 2 \times [0,3] + 2 \times [4,5] \end{aligned}$$

  This complies with our postulated macro-op-to-µop decomposition (Postulate 6.5).

- Simple scalar integer instructions with a read and written memory operand use an ALU µop and a store µop:

$$\begin{aligned} &\mathtt{add}\ \langle \mathtt{GPR}[32] \rangle_{RW}, \langle \mathtt{GPR}[32] \rangle_R && \rightsquigarrow 1 \times [6,7,8,9] \\ &\mathtt{add}\ \langle \mathtt{MEM}[32] \rangle_{RW}, \langle \mathtt{GPR}[32] \rangle_R && \rightsquigarrow 1 \times [6,7,8,9] + 1 \times [5] \end{aligned}$$

  If these operate on less than 32 bits, they require an additional µop on the address generation units to handle the less common access widths:

$$\texttt{add}\ \langle \texttt{GPR[8]} \rangle_{RW}, \langle \texttt{GPR[8]} \rangle_{R} \quad \rightsquigarrow 1 \times [6,7,8,9]$$
$$\texttt{add}\ \langle \texttt{MEM[8]} \rangle_{RW}, \langle \texttt{GPR[8]} \rangle_{R} \quad \rightsquigarrow 1 \times [6,7,8,9] + 1 \times [4,5] + 1 \times [5]$$

This is a deviation from our postulated macro-op-to-µop decomposition. For the wider instruction variants, however, we can confirm that Zen+ does not use separate µops for the two memory operations in read-modify-write instructions, which stands in contrast to Intel's microarchitectures.

Overall, 70% of the remaining 1,819 considered instruction schemes fall into these regular patterns.

For complex instructions, we observe unexpected results. For instance:

$$\texttt{bsf}\ \langle \texttt{GPR[64]} \rangle_{W}, \langle \texttt{MEM[64]} \rangle_{R} \quad \rightsquigarrow 9 \times [6,7,8,9] + 1 \times [4,5] + 9 \times [0,1,2,3]$$

$$\texttt{vphaddw}\ \langle \texttt{XMM} \rangle_{W}, \langle \texttt{XMM} \rangle_{R}, \langle \texttt{XMM} \rangle_{R} \quad \rightsquigarrow \begin{array}{l} 1 \times [0,1,2,3] + 1 \times [0,1,3] \\ \quad + 2 \times [1,2] + 4 \times [6,7,8,9] \end{array}$$

The former is a *bit scan forward* instruction, which finds the least significant bit set in its read (memory) operand and stores its index into the written operand.[7] The latter is a horizontal vector addition that adds pairs of adjacent 16-bit elements of a vector register across the typical vector lanes.[8] Their inferred port usages are unexpected in two ways: They contain more µops than reported by the performance counter (8+1 counted and adjusted for a memory operand for `bsf` and 4 counted for `vphaddw`) and they include µops for unlikely ports. For the scalar integer operation `bsf`, we would not expect a utilization of vector/floating-point ports $[0,1,2,3]$, whereas the vector operation `vphaddw` is unlikely to use the scalar ALU ports $[6,7,8,9]$. We suspect that these µops are spurious observations caused by the processor's microcode sequencer. For instructions with many µops, the processor's instruction decoder only emits an entry point address for the microcode sequencer ROM. The microcode sequencer then emits the relevant operations from the ROM. Our observations match a microcode sequencer that emits four operations per cycle while stalling the remaining frontend, including the operation cache. Rather than µops that cannot execute on unblocked ports, we measure the overhead of this bottleneck. This occurs for 8% of the 1,819 considered instruction schemes.

For 7% of the instruction schemes, e.g., for bit shift operations on vector registers, the experiments yield throughputs that are unstable or outside the port mapping model.

This last stage of the algorithm took 8–10 hours. The last two stages of the algorithm dominate the running time of the algorithm, with a total of 20–28 hours. Overall, we inferred a port mapping for 1,700 of the initial 2,980 instruction schemes. uops.info has no port mapping for 1,142 (67%) of these 1,700 supported instruction schemes.

6.3.6. Prediction Accuracy

We evaluate the inferred Zen+ port mapping quantitatively by comparing its throughput prediction accuracy against our previous approach, PMEvo, (cf. Chapter 5) and Palmed by

[7] See `https://www.felixcloutier.com/x86/bsf`.
[8] See `https://www.felixcloutier.com/x86/phaddw:phaddd`.

Table 6.3. Comparison of IPC prediction accuracy for Zen+ models generated by PMEvo, Palmed, and our explainable port mapping inference (PMI) algorithm.

	MAPE	ρ_P	τ_K
PMEvo	28.0%	0.83	0.72
Palmed	35.2%	0.79	0.66
explainable PMI	6.6%	0.96	0.90

Derumigny *et al.* (2022a).[9] As port mappings model only the utilization of functional units, we focus on instruction sequences whose throughput is not limited by data dependencies.

To predict the throughput of an experiment e with the inferred mapping, we apply the fast bottleneck simulation algorithm from Section 5.1.4 and obtain the number t of cycles of an optimal execution with respect to the port mapping. If this number is faster than the bottleneck of 5 IPC allows, we report an inverse throughput of $5/|e|$ cycles, and t otherwise. For Palmed, we use the most recent available model for the Zen architecture. For PMEvo, we combine the original implementation with the modernized measurement setup described in Section 3.2.1 and infer a new port mapping. We seed the population of its evolutionary algorithm with 50,000 random port mappings and let it run until evolution converges after ca. 59 hours.[10] While PMEvo only uses experiments with at most two different instruction schemes, it benchmarks such experiments for all pairs of instruction schemes. When considering thousands of instruction schemes, this leads to substantially more microbenchmarks than in the runs of our explainable inference algorithm. Therefore, to keep benchmarking times for PMEvo manageable, we restrict this evaluation to instruction schemes that occur in compiled binaries for the SPEC CPU2017 benchmarks (Bucek *et al.*, 2018) and are covered by our inferred explainable mapping.[11] Overall, this leaves us with 577 instruction schemes.

We evaluate the performance models on a set of 5,000 randomly generated experiments. Each experiment consists of five instruction schemes that were selected uniformly at random from the considered 577 instruction schemes. We apply the method described in Section 3.2.1 to benchmark their throughput in instructions per cycles (IPC) on the Zen+ hardware. In particular, the instruction schemes are instantiated with operands to avoid data dependencies that would limit the observable throughput.

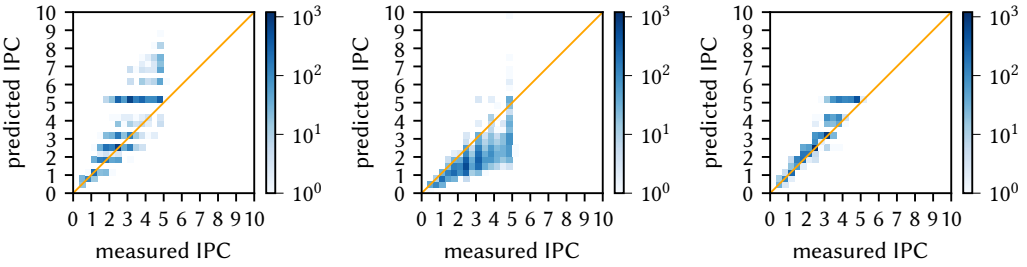
Table 6.3 shows the IPC prediction accuracy in terms of mean absolute percentage error (MAPE), Pearson's correlation coefficient ρ_P, and Kendall's τ_K for each tool. A high ρ_P indicates a linear correlation of predictions and measurements whereas a high τ_K implies that sorting the instruction sequences by predicted or measured IPC leads to similar rankings. Both metrics can range between -1 and 1.[12]

[9] See Chapter 7 for a conceptual comparison to these approaches.

[10] We follow the original PMEvo design that does not adjust for the IPC bottleneck. Incorporating the adjustment only causes minor differences in the accuracy metrics of this evaluation.

[11] Benchmarking PMEvo's experiments for this setup took 11 days.

[12] See Section 3.2.2 for more on these accuracy metrics.



(a) PMEvo **(b)** Palmed **(c)** explainable PMI

Figure 6.3. Heat maps of predicted vs. measured execution rate in instructions per cycle (IPC) for Zen+ models generated by PMEvo, Palmed, and our explainable port mapping inference (PMI) algorithm.

The predictions of PMEvo and Palmed share a similar level of accuracy, with significant correlations but rather high errors of 28–35%.[13] Our inferred explainable port mapping is substantially more accurate with an error of 6.6% and very strong linear and rank correlations. The heat maps in Figure 6.3 quantify each model's prediction accuracy in more detail. They group the basic blocks into bins based on the IPC we observed in the benchmarks and the predictions of each model. Bins are displayed darker the more basic blocks they contain; the closer the darker bins are to the diagonal line, the closer are predictions and observations. The heat map for our explainable model, Figure 6.3 (c), is notably closer to the diagonal than PMEvo's and Palmed's with few outliers.

The structure of PMEvo's mapping differs substantially from our inferred explainable port mapping: There, most instructions have only a single kind of µop in their port usage. Our explainable approach captures structures of the microarchitecture that PMEvo's optimization does not resolve. As shown in Figure 6.3 (b), Palmed's conjunctive mapping usually predicts slower executions than what we measure in microbenchmarks. As Palmed depends on assumptions in its measurement infrastructure, we cannot evaluate whether its model would be more consistent with our throughput measurements if it used our microbenchmarking setup.

6.4. Conclusions: Explainable Port Mapping Inference with Sparse Performance Counters

We have shown that per-port µop counters are not necessary to apply a uops.info-style explainable port mapping inference algorithm. With techniques that build on the port mapping model and our counter-example-guided online port mapping inference algorithm (cf. Chapter 4), we

[13] Note that, in contrast to previous chapters, the prediction accuracy here is evaluated in terms of IPC rather than cycles or CPI to enable a comparison with the evaluation by Derumigny *et al.* (2022a).

can efficiently infer the port usage for a large number of instructions if the processor under investigation follows the port mapping model.

Our case study of the AMD Zen+ microarchitecture indicates that the approach is also practical for a large portion of the processor's instructions. However, there are practical hindrances like throughput bottlenecks in parts of the processor, misdocumented performance counters, and complex micro-coded or non-pipelined instructions. Nevertheless, we uncovered details of the Zen+ microarchitecture that have, to the best of our knowledge, not been previously documented. We inferred the first explainable three-level port mapping for over 1,000 instruction schemes on Zen+ that were out of scope for previous work and demonstrated its ability to accurately model performance characteristics of the microarchitecture.

Chapter 7

Related Work on Port Mapping Inference

In this chapter, we discuss how the port mapping inference methods presented in the Chapters 4 to 6 compare to previous and current efforts in the field. To this end, Section 7.1 explores alternative approaches to infer port mappings and similar properties of the hardware. Since basic block throughput predictors are the most common use cases for port mappings, Section 7.2 discusses these tools and how they model port-mapping-related information.

7.1. Inferring Port Mappings

The **instruction tables by Fog (2022)** are a well established source for experimentally validated information on instruction latency, throughput, and port usage for a wide range of x86 microarchitectures. They are the product of hand-crafted microbenchmarks that use hardware performance counters to count the number of executed cycles and the number of executed µops per port. Abel and Reineke (2019) show that the reported port usage by Fog is only an under-approximation of the usable ports.

For cases where these counters are not available, Fog uses experiments that execute instructions with unknown port usage together with instructions whose port usage is known from some other resource. Observing the running time allows to identify interfering instruction combinations.

The timing tables by Granlund (2019) fall into a similar category, but they only provide latency and throughput measurements, without information on the port usage of the instructions.

For **uops.info**, Abel and Reineke (2019) automated the laborious design of microbenchmarks to measure latency, throughput, and port usage. Their algorithm to estimate port usage overcomes the inaccuracy of Fog's approach by using blocking instructions, as discussed in Section 6.1. Similar to Fog's method, their method relies on per-port hardware performance counters. While Abel and Reineke provide throughput and latency measurements for x86-64 microarchitectures by Intel and AMD, they only give complete port mappings for the Intel platforms as only these provide the required performance counters.

Johnson (2021) adapted the techniques used for uops.info by Abel and Reineke to analyze the microarchitecture of Apple's M1 CPU. The M1 implements an ARM ISA, but comes with adequate performance counters to apply the methodology of Abel and Reineke.

Another similar approach, initiated by Google, is **EXEgesis** (Chatelet, 2018; Google, 2018; LLVM, 2023a). One part of this project seeks to extract latencies, throughputs, and port usage for Intel architectures from manufacturer-provided documentation. This requires automatically parsing documents that were intended for human readers: a fragile and work-intensive process. Since the provided documentation does not include all relevant information, the EXEgesis developers also created tools to infer the missing information via experiments. This led to the second part of the EXEgesis project, llvm-exegesis (Chatelet, 2018), a tool inside the LLVM framework (Lattner and Adve, 2004) that automatically generates benchmarks similar to those used by Fog. For measuring port usage, llvm-exegesis depends on per-port performance counters just as the previously discussed approaches.

The main difference between these prior approaches and our port mapping inference methods is that our approaches do not rely on per-port performance counters. Since the prior approaches can work with more information, their strategies for port mapping inference can be less complex than ours while being less dependent on exact throughput measurements. However, they are restricted to processors with suitable performance counters. The processors by AMD and many ARM platforms are therefore out of their scope, whereas our approaches are applicable.

PMEvo (Chapter 5) only requires widely available hardware support to measure the number of cycles required to execute a benchmark and no specific hardware performance counters. It is therefore able to infer port mappings for AMD's x86-64 microarchitectures and ARM architectures. An advantage of uops.info compared to PMEvo is that the uops.info algorithm comes to its results through microbenchmarks that can be validated by users of the performance model. For every µop that is included or not included in the inferred port usage of an instruction, the uops.info algorithm finds a microbenchmark that witnesses this decision. The randomized and approximative nature of PMEvo's evolutionary algorithm cannot provide this level of confidence. Inspecting PMEvo's port mappings shows that, while they may accurately predict the throughput of port-mapping-bound microbenchmarks, they do not necessarily follow the structure of the actual hardware.

The variation of the uops.info algorithm that we present in Chapter 6 addresses this weakness of PMEvo. As it follows the same high-level structure as the uops.info algorithm, it provides similar witnessing microbenchmarks to justify the inference results. We found the resulting port mappings to follow available hardware descriptions more closely than the results of PMEvo. This increased quality of the inference results comes at the cost of stronger requirements on the investigated microarchitecture: We require hardware support to count the total number of executed µops in a microbenchmark. While this is still a weaker requirement than the per-port performance counters required for the original uops.info algorithm, deviations from the port mapping model in the processor's behavior, e.g., due to tight frontend bottlenecks, can further limit the applicability of our algorithm.

Palmed by Derumigny *et al.* (2022a), which has been concurrently developed and published after our work on PMEvo, falls into a different category. They share our goal of inferring a performance model for the resource usage of instructions without requiring the extensive performance counters used by approaches like uops.info. However, the model that they produce differs from the standard (three-level) port mappings that our methods and the other related approaches aim for. Palmed's conjunctive mappings do not explicitly model how

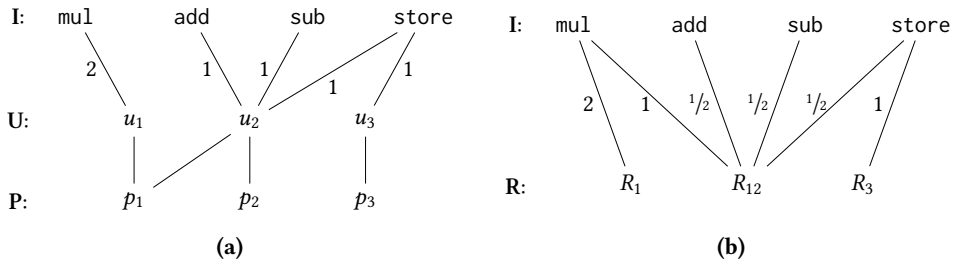


Figure 7.1. Example port mapping in the three-level model (a) and a mapping in Palmed's conjunctive model that yields equal throughput predictions (b).

instructions are decomposed into µops. Instead, they model instructions as using abstract resources, which do not directly correspond to the execution ports of the processor. If two instructions decompose into µops that compete for the same ports, Palmed's model for these instructions will contain an abstract resource that both instructions use.

For an example, consider the conjunctive mapping in Figure 7.1 (b). The weight of an edge between an instruction scheme i and an abstract resource $r \in \mathbf{R} := \{R_1, R_{12}, R_3\}$ describes the load that each occurrence of i in an experiment e puts on r. Derumigny *et al.* model the inverse throughput $tp^{-1}(e)$ as the maximal accumulated load on any resource. For the experiment $e := \{\texttt{mul} \mapsto 1, \texttt{add} \mapsto 2, \texttt{store} \mapsto 1\}$, R_1 receives a load of 2, R_{12} receives a load of 2.5, and R_3 receives a load of 1. The resulting modeled inverse throughput is therefore 2.5 cycles. The three-level port mapping in Figure 7.1 (a) yields the same result with two u_1 µops and three u_2 µops that are executed on the bottleneck ports $\{p_1, p_2\}$.

To infer a conjunctive mapping, one needs to determine adequate real-valued edge weights. In the appendix of the extended version of the Palmed article, Derumigny *et al.* (2022b) provide an algorithm to construct a conjunctive mapping – with a potentially exponential number of abstract resources – that yields equal predictions as a given three-level port mapping.

The key benefit of this resource model is that Palmed can solve a simpler integer linear program (ILP) where we use an expensive SMT formulation in Chapter 4 and Chapter 6 or approximate with an evolutionary algorithm in Chapter 5.

Like the explainable inference algorithm we present in Chapter 6, Palmed takes inspiration from the uops.info algorithm. Derumigny *et al.* proceed in two phases, first finding a core mapping for a small instruction set and then inferring models for all other instructions based on the core mapping. Rather than identifying blocking instructions with a µop counter, they select basic instructions for the core mapping heuristically based on throughput benchmarks. Palmed constructs a set of abstract resources that represent possible bottlenecks in the execution of core mapping instructions with an ILP-based algorithm. It further generates for each resource a kernel of basic instructions that saturates the resource, similar to how blocking instructions flood their corresponding set of ports. Palmed then runs individual benchmarks with the saturating kernels for every instruction that is not in the core mapping and uses a linear program to compute the pressure that the instruction puts on the corresponding abstract resources.

Besides port sets, Palmed's resource model inherently represents other potential bottlenecks like the maximal execution rate of the frontend, which our approaches need to treat explicitly. However, conjunctive mappings do not integrate straightforwardly with existing performance models based on three-level port mappings since the inferred abstract resources have no clear correspondence to documented aspects of the microarchitecture.

7.2. Basic Block Throughput Predictors

In recent years, a variety of tools have been proposed to estimate the throughput a processor achieves when executing a given instruction sequence. Traditional approaches in this field rely on a – typically hand-crafted – model of the processor's microarchitecture that captures parameters such as instruction latencies, port mapping, and µop queue sizes of the hardware. This model is used to simulate the execution of one or more instances of the basic block whose performance is to be predicted.

Basic block throughput predictors reduce the complexity of the simulation task by relying on assumptions about the context in which basic blocks are executed. The most common ones are that all memory accesses hit the fastest cache and that the basic block is the body of an innermost loop that is executed indefinitely. With these assumptions, the main domain of application for such throughput predictors is in the optimization of short, very hot code regions in programs where performance is crucial. Depending on their model, these tools can provide insights into the performance of the analyzed code, e.g., which component of the processor is most likely to cause a performance bottleneck.

In the following, we describe approaches that match this description and how they model the processor's port mapping. Additionally, we discuss recent approaches that use machine learning to avoid the tedious task of creating a microarchitectural model by hand.

7.2.1. Traditional Approaches

The Intel Architecture Code Analyzer **IACA** (Intel, 2012) provides for a given instruction sequence a throughput prediction, an estimation of where in the processor core the bottleneck for the execution lies, and a distribution of µops to ports. It is a closed-source tool that the processor manufacturer Intel provides for some of its x86-64 microarchitectures. IACA can therefore make use of unpublished internal information for its performance predictions. Nevertheless, previous research, e.g., by Abel and Reineke (2019), has shown cases where the prediction of IACA differs substantially from the observable behavior. As IACA provides a distribution of µops to ports, we expect its proprietary model to include a port mapping. In April 2019, Intel announced that IACA has reached its end of life.

The open-source tools llvm-mca (Di Biagio, 2018; LLVM, 2023b), OSACA (Laukemann *et al.*, 2018), and uiCA (Abel and Reineke, 2022) all produce similar reports as IACA with a throughput prediction and an estimated execution port utilization.

The LLVM Machine Code Analyzer **llvm-mca** (Di Biagio, 2018; LLVM, 2023b) bases its processor model on the instruction scheduling models of the LLVM compiler infrastructure (Lattner and Adve, 2004). These scheduling models are the result of human fine-tuning

effort, proprietary knowledge contributed by processor designers, and experiments via llvm-exegesis (Chatelet, 2018). As LLVM has scheduling models for a variety of microarchitectures – including x86-64 architectures by Intel and AMD as well as various ARM designs – llvm-mca is widely applicable.

LLVM's scheduling models were, however, designed with code generation and instruction scheduling, not simulation and performance prediction in mind (Di Biagio, 2020). Instead of collecting information about how instructions are decomposed into µops and how these are executed, LLVM's models only capture resource usage at the instruction level. This abstraction of how the processor operates limits llvm-mca's accuracy (Di Biagio, 2020). Nonetheless, while LLVM's scheduling models do not capture all details of a port mapping, knowledge of the port mapping can help to make these models more accurate.

The performance models of **OSACA** (Laukemann *et al.*, 2018) are based on published information by the manufacturers, custom microbenchmarks, and the instruction tables of Fog (2022) and uops.info (Abel and Reineke, 2019). OSACA has models for x86-64 microarchitectures by AMD and Intel, as well as several ARM microarchitectures.

Where port mapping information is available, OSACA's processor models include explicit three-level port mappings. Laukemann *et al.* implement means of validating their port mappings via experiments with throughput measurements and note that experiments with multiple different instructions can uncover new details of the port mapping. Our approaches can be seen as systematic extensions of this line of work to derive new port mappings.

uiCA (Abel and Reineke, 2022) relies on latency data and port mappings from uops.info (Abel and Reineke, 2019) for a fine-grained simulation of the execution of a basic block. It further includes detailed reverse-engineered models of the decoding frontends of the supported microarchitectures. uiCA is specialized in Intel's x86-64 microarchitectures and therefore does not provide models for AMD or ARM architectures. Abel and Reineke report that uiCA outperforms the previous tools in terms of throughput prediction accuracy on a modified version of the BHive benchmark set (Chen *et al.*, 2019) for the supported platforms.[1]

With their follow-up work, **Facile**, Abel *et al.* (2023) observed that the port mapping is one of few components that need to be modeled to achieve high throughput prediction accuracy on the BHive benchmark set. Facile relies on the same microarchitectural models as uiCA, but uses them differently. It considers potential causes for bottlenecks individually – including the port mapping, dependency chains, and various components of the decoding frontend – and estimates how each potential bottleneck on its own would limit the throughput of the analyzed basic block. Facile reports the slowest among these individual estimates as its throughput prediction. In the evaluation of Abel *et al.*, this approach achieves similar throughput prediction accuracy on BHive as uiCA but requires significantly less analysis time.

Facile employs a simplified variant of the bottleneck simulation algorithm that we present in Section 5.1.4. Where our algorithm enumerates all port subsets as potential bottlenecks, their algorithm only considers port combinations required by pairs of µops. In general, this simplified variant does not always yield the same result as our bottleneck simulation algorithm and the equivalent linear program, see Appendix E.2 for an example.

[1] We discuss the BHive benchmarks in more detail in Section 8.4.1.

The Code Quality Analyzer (**CQA**) of MAQAO (Rubial *et al.*, 2014) searches for opportunities to improve the performance of innermost loops. Besides throughput estimation and a break-down of utilized resources in the processor's decoding frontend, CQA gives a detailed analysis of how effectively the input code uses single-instruction-multiple-data operations and recommendations for improvement. The tool supports x86-64 microarchitectures by AMD and Intel, and several ARM architectures.

CQA initially came without a detailed model of the out-of-order backend of the supported microarchitectures. Since version 2.18.0, CQA has the capabilities to report a port usage breakdown for analyzed code. The CQA models combine port mapping data from uops.info, the instruction tables by Fog, and documentation by the manufacturers.[2]

7.2.2. Learning-Based Approaches

A recent line of work leverages machine learning to reduce the effort required to build accurate microarchitectural models for basic block throughput prediction. The earliest work, **Ithemal** (Mendis *et al.*, 2019), uses a hierarchical neural network based on long short-term memory (LSTM) cells for throughput prediction. Basic blocks under investigation are split into instructions whose mnemonics and operands are tokenized as input for the neural network. The neural network outputs a real-valued estimate of the achieved throughput. Ithemal is trained in a supervised setting, based on a corpus of basic blocks with annotated throughput measurements from the actual hardware. Mendis *et al.* extract the basic blocks for their training data from a varied collection of compiled benchmark programs.

The benefit of this approach is that no information about the microarchitecture, like the frontend organization or the number of execution ports, are needed when training an Ithemal model for a new processor. In evaluations against the above traditional approaches, e.g., by Abel *et al.* (2023), only uiCA and Facile outperform Ithemal in terms of throughput prediction accuracy.

A drawback of the Ithemal approach is that the resulting processor model is not straightforward to interpret. Since Ithemal's neural network does not include a dedicated port mapping model, there is no straightforward way to extract a port mapping from the learned parameters. **DiffTune** (Renda *et al.*, 2020), addresses this drawback by learning parameters of llvm-mca's microarchitectural models from Ithemal. Renda *et al.* use surrogate learning with an Ithemal-based model to learn values for llvm-mca parameters concerning the port mapping, instruction latencies, and the frontend bandwidth. In particular, the learned parameters include `PortMap` values for each instruction that determine for how many cycles the instruction uses each port. As discussed above, these parameters are oblivious of the port usage of individual µops and therefore do not capture every detail of the processor's port mapping.

Abel (2022) reports that they outperform the original llvm-mca parameters as well as those learned with DiffTune in terms of throughput prediction accuracy on the BHive benchmark set (Chen *et al.*, 2019) with a parameter set for llvm-mca that, among other changes, sets every `PortMap` value to zero. DiffTune's and llvm-mca's prediction accuracy on the BHive benchmarks therefore appears to be determined by factors other than the `PortMap` parameters.

[2]This was disclosed in personal communication by the maintainers of MAQAO.

Granite (Sýkora *et al.*, 2022) advances the Ithemal concept with an updated neural network structure. Sýkora *et al.* use a graph neural network that operates on the input basic block's data flow graph, instead of the purely syntactic token string used by Ithemal. Additionally, they design their model architecture such that one model can be used to predict the throughput for a range of microarchitectures. They report an improvement in prediction accuracy over Ithemal, but do not include a comparison with uiCA.

PerfVec (Li *et al.*, 2023) aims at performance prediction for entire execution traces of programs. While these could be considered as very long basic blocks, the difference in magnitude – billions of instructions rather than less than ten or twenty – requires different techniques. PerfVec models the performance-relevant features of the instructions and the microarchitecture separately to reduce training complexity. Adapting it to new microarchitectures only requires training a new microarchitecture model whereas the instruction model remains unchanged. PerfVec combines contributions of both models to predict each instruction's contribution to the latency of the execution trace with a linear predictor.

Granite and PerfVec both share Ithemal's black-box nature with no straightforward way to extract a port mapping from the learned parameters.

CATREEN (Amalou *et al.*, 2022) is a machine-learning-based performance predictor on the basic block level, but it has a different focus than the previously discussed tools. On one hand, it considers basic blocks in their execution context. Instead of predicting a single execution time for a given basic block, CATREEN's predictions additionally depend on which basic blocks were executed before the basic block under investigation. This allows CATREEN to model behaviors of the branch predictor and caches that are not in the scope of the other tools. On the other hand, CATREEN is designed for in-order microarchitectures, compared to the more complex out-of-order architectures targeted by the remaining approaches. Since the port mapping is a property of out-of-order architectures, it is not relevant to CATREEN.

Chapter 8

AnICA: Analyzing Inconsistencies in Microarchitectural Code Analyzers

In the previous chapters, we have discussed ways to model and infer a processor's port mapping. Basic block throughput prediction tools are a central use case for inferred port mappings. They need a microarchitectural model to estimate how a processor executes a given instruction sequence. So far, we treated these tools as motivating related work (Section 7.2) and as points of comparison (Sections 5.2 and 6.3).

In this chapter, we investigate basic block throughput prediction tools like IACA (Intel, 2012), llvm-mca (Di Biagio, 2018), OSACA (Laukemann *et al.*, 2018), uiCA (Abel and Reineke, 2022), Ithemal (Mendis *et al.*, 2019), and DiffTune (Renda *et al.*, 2020) more directly. These tools typically aim for close to cycle-accurate performance estimations, but cannot perform an exact simulation of the target hardware because no exact model is available.[1] Their performance predictions are commonly based on assumptions, e.g., that all memory accesses hit the fastest cache and that execution of the basic block is in a steady state, i.e., it is the body of an innermost loop that is executed indefinitely. But even with these simplifying assumptions, the task of predicting the throughput for a given basic block is challenging. Modern processors split instructions into undocumented µops and reorder them as freely as the data dependencies allow.[2] Numerous undocumented buffers and execution units make the processor fast, but they also impede accurate throughput estimation. Moreover, as noted, e.g., by Abel and Reineke (2022), there is often not a single well-defined throughput for a given basic block on a microarchitecture:

- On many microarchitectures, the execution time of some instructions depends on their input values.
- Basic blocks might contain data dependencies if certain input values are pointers to the same memory location.
- Depending on whether the basic block is repeated through a loop or through concatenating many copies, different bottlenecks determine the throughput.

Tools that predict the throughput of arbitrary basic blocks need to rely on assumptions about the processor's behavior for all such cases. Very often, basic block throughput predictors do not come with an explicit statement of these assumptions.

[1] See Section 7.2 for a discussion of the individual tools.
[2] See Section 2.1.

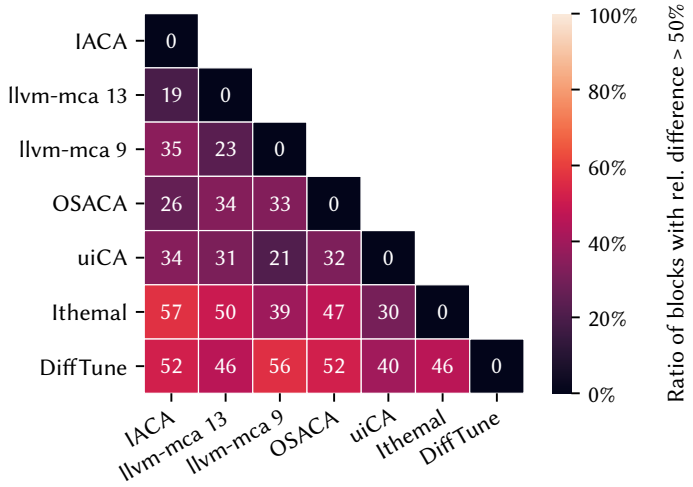


Figure 8.1. Heat map showing the percentage of basic blocks with inverse throughput estimates that deviate by more than 50% for each pair of predictors.

When working with basic block throughput predictors for the evaluation of PMEvo (Section 5.2), we regularly encountered basic blocks for which throughput predictors for the same microarchitecture would yield considerably different results. Figure 8.1 shows the results of an experiment in which we randomly generate 10,000 basic blocks consisting of 4 instructions each[3] and let several throughput predictors give their estimate for these blocks assuming the Intel Haswell[4] microarchitecture. For each pair of throughput predictors, the heat map contains an entry indicating the percentage of basic blocks for which the difference between the inverse throughput estimates exceeds 50% of their mean.

Overall, all pairs of predictors exhibit substantial numbers of inconsistencies. As we can see from the inconsistencies in different versions of llvm-mca (23% of the basic blocks in Figure 8.1 are predicted inconsistently between llvm-mca versions 9 and 13), even closely related implementations are affected. There may be several reasons for these deviations, e.g.:

- The individual performance models fail to capture relevant parts of the execution.
- The tools are built with different (implicit or explicit) assumptions.
- The learning-based tools need more training data.
- The implementations contain bugs.

In any of these cases, finding and characterizing such inconsistencies is valuable. If the cause is unintentional, finding inconsistencies helps to improve the tools. If the inconsistencies are

[3] A basic block is generated by individually sampling instructions from the machine-readable x86-64 ISA description from uops.info and instantiating them with valid operands.

[4] Haswell, which was introduced in 2013, is the only microarchitecture supported by all compared tools.

the result of deliberate choices of the developers, identifying them helps to explore the limits of the tools. Our goal in this chapter of the thesis is therefore to discover inconsistencies in the results of instruction throughput predictors and to give insight into their causes.

To this end, we present AnICA, our work on analyzing inconsistencies in microarchitectural code analyzers, as published at OOPSLA 2022 (Ritter and Hack, 2022). The idea of AnICA is to apply differential testing (McKeeman, 1998) to a pair of basic block throughput predictors. We randomly sample basic blocks and compare the outputs of the tools under investigation. If the tools do not agree, we found an inconsistency. Similar techniques are used in a variety of domains, prominently for compilers (McKeeman, 1998), SSL/TLS certificate validators (Brubaker *et al.*, 2014; Chen and Su, 2015), but also for software that simulates aspects of processors like instruction decoders (Jay and Miller, 2018; Paleari *et al.*, 2010; Woodruff *et al.*, 2021).

The existing approaches, however, do not transfer well to basic block throughput predictors since those provide an unusual setting for differential testing. The heat map in Figure 8.1 shows that finding inputs that exhibit inconsistencies is not difficult. Hence, elaborate methods for searching the input space are not necessary. However, just listing the large number of inconsistencies we find would not be very helpful to understand and improve the tools under test. The focus of AnICA is therefore to find compact characterizations of large classes of input basic blocks that cause inconsistencies. We apply concepts from abstract interpretation (Cousot and Cousot, 1977) to find these compact characterizations and present them together with witnesses for their derivation. These witnesses give insights in two directions:

- They contain examples of represented basic blocks that exhibit inconsistencies.
- They show the boundaries of the problem through similar basic blocks that do not exhibit inconsistencies.

For many of the tool combinations shown in Figure 8.1, ten of AnICA's characterizations are sufficient to capture more than half of the several thousand encountered inconsistencies. We investigate results of AnICA in case studies showing that the results are helpful for improving performance models in several ways: to find modeling bugs and regressions from one tool version to the next, to understand differing modeling assumptions, and to identify underrepresented constructs in the training sets of learned predictors.

In this thesis, we expand upon the OOPSLA publication with formal discussions of properties of the AnICA generalization algorithm and the presented basic block abstraction in the Sections 8.1.3 and 8.1.4.

8.1. The AnICA Algorithm

On a high level, AnICA follows the structure of differential testing (McKeeman, 1998): An AnICA *campaign* searches for inconsistencies between a fixed pair of throughput predictors and reports them as *discoveries*. We generate valid input basic blocks, give them to both tools under investigation, and compare their results. The throughput predictors are required to support a common instruction set architecture (ISA), i.e., they need to have compatible input

formats. Given a basic block, they should output a real-valued estimate for the number of cycles required for its execution or report an error.

Differential testing is a natural fit to overcome the lack of a clear ground truth when comparing basic block throughput predictors, with their implicit and explicit assumptions on the input. Differences in the assumptions of the tools under investigation are visible as inconsistencies for basic blocks that are affected by these assumptions.

Valuable insight can also be gained from comparing the results of a throughput predictor to measurements on the actual hardware rather than other predictors. AnICA naturally supports this, by using a microbenchmarking tool as one of the tools under investigation.[5] However, even microbenchmarks on the actual hardware rely on assumptions that may not hold when the investigated basic block is executed in a different context, for instance by initializing registers and memory with specific values. Therefore, we also use the perspective of differential testing for such comparisons with hardware measurements to acknowledge that the involved microbenchmarking tool is also influenced by assumptions and not a definitive ground truth.

As shown in this chapter's introduction, a key challenge for AnICA is that inconsistencies are so common in the randomly sampled inputs that just reporting all basic blocks with inconsistencies leads to an impractical number of reports. Therefore, we center the algorithm around the idea of *abstract basic blocks*, or *abstract blocks* for short: compact representations characterizing sets of basic blocks by common properties. AnICA aims for several goals to make the abstract blocks that are reported as discoveries useful:

- The represented basic blocks should be *concise*, i.e., not contain instructions that are irrelevant to the underlying problem. This makes them easy to interpret.
- Each discovery should be *general* by representing as many relevant basic blocks as possible. The more general the discoveries are, the fewer of them need to be inspected.
- The discoveries should be *pertinent*, i.e., not represent basic blocks that do not exhibit inconsistencies in the tools under investigation. Significant numbers of such cases would make the characterization unreliable.

Since the results of AnICA are used to hint at existing problems or to show limitations of the tools – rather than, e.g., to prove the absence of inconsistencies – none of these goals are strict formal requirements. This fact allows us to employ approximations rather than heavy formal machinery at several points in the following sections.

AnICA's high-level structure, serving as a table of contents for the remainder of this section, is shown in Algorithm 8.1. We randomly sample a basic block and check whether it is *interesting*, i.e., if the throughput predictors under test exhibit an inconsistency (lines 3–5). AnICA minimizes interesting basic blocks (line 6) by greedily removing as many instructions as we can while keeping the block interesting. If the minimized basic block is already represented by a previously discovered abstract block, we do not need to further investigate it (lines 7–8). Otherwise, the basic block is generalized to an abstract block, which is then noted as a new discovery (lines 9–10).

[5] We explore this in a case study in Section 8.3.3.

```
1  discoveries ← {};
2  while termination condition not reached do
3  |  candidate ← sampleBB() ;                    // Section 8.1.5
4  |  if candidate is not interesting then        // Section 8.1.1
5  |  |  continue;
6  |  minBB ← minimize(candidate);
7  |  if any d ∈ discoveries subsumes minBB then  // Section 8.1.6
8  |  |  continue;
9  |  newDisc ← generalize(minBB) ;               // Section 8.1.3
10 |  discoveries ← discovery ∪ {newDisc};
11 return filterSubsumed(discoveries) ;           // Section 8.1.6
```

Algorithm 8.1. Discovering inconsistencies.

We repeat this process until some termination condition is reached (line 2), e.g., a time budget is exhausted or a number of subsequent samples did not produce new discoveries. Finally, we check for each discovered abstract block a whether there is a subsequent one whose represented basic blocks include all of a. Such subsumed discoveries provide only redundant information and are therefore filtered from the results (line 11).

In the following subsections, we describe the components of Algorithm 8.1 in detail as indicated in the comments.

8.1.1. Interestingness Metric

Not every difference in the output of the tools under investigation is relevant. Since they predict real-valued average execution times based on vastly different models, small deviations are to be expected. Our definition of interestingness therefore takes small variations into account:

A basic block is *interesting* if it causes a tool under investigation to crash, or if the relative difference between their predictions $pred_a$ and $pred_b$ exceeds a specified threshold:

$$\frac{|pred_a - pred_b|}{avg(pred_a, pred_b)} = \frac{|pred_a - pred_b| \cdot 2}{pred_a + pred_b} > threshold$$

As we cannot assume any of the predictions to be the "correct" one, this definition normalizes the absolute difference between the predictions by their arithmetic mean. The interestingness threshold is a parameter of the algorithm that influences what inconsistencies are found.

Other definitions of interestingness are conceivable and may be useful. For example, our AnICA implementation also provides an alternative criterion based on the absolute difference:

$$|pred_a - pred_b| > threshold$$

Which criterion is the most suitable depends on the inconsistencies we are searching for. The relative difference is effective for focusing on interesting inconsistencies when the

predicted numbers of cycles grow larger. With the absolute difference criterion, it is easier to investigate inconsistencies of a few cycles for short-running basic blocks.

8.1.2. Basic Block Abstraction

We borrow concepts and notation from abstract interpretation (Cousot and Cousot, 1977) to describe our representation of sets of basic blocks. Abstract interpretation is a technique commonly used in static program analysis. It provides a framework to reason about approximations of the possible behaviors of programs. A key insight of the technique is to represent subsets of conceivable program behaviors (denoted as elements of the *concrete domain*) by elements of an *abstract domain*. This is beneficial since the concrete domain of sets of program behaviors is generally too large to work with, whereas the abstract domain can be compact.

In AnICA, we apply this notion of abstraction to a different application domain. Instead of program behaviors, we abstract basic blocks. Therefore, our concrete domain contains sets of basic blocks and the abstract domain represents features of these basic blocks such as instruction mnemonics, use of memory, and operand dependencies.

Formally, our concrete domain $\mathbb{C}$ is the power set of the set $\mathbb{B}$ of instruction sequences from an instruction set $\mathbf{I}$, ordered by set inclusion $\subseteq$:

$$\mathbb{C} := \mathcal{P}(\mathbb{B}) \qquad \text{with } \mathbb{B} := \mathbf{I}^+$$

An abstract domain $\mathbb{A}$ is a set with a partial order $\sqsubseteq$ that relates domain elements by their generality: If $a \sqsubseteq b$ holds for two elements $a, b \in \mathbb{A}$, b represents at least all elements of the concrete domain that a represents. While the abstract domain may be infinite, we require the ascending chain condition, i.e., infinite sequences of strictly more general elements in the abstract domain are not allowed.

As usual in abstract interpretation, the AnICA algorithm is independent of the specific abstract domain. The abstract domain only needs to relate to the concrete domain via two functions: a *concretization function* $\gamma : \mathbb{A} \to \mathbb{C}$ and a *representation function* $\beta : \mathbb{B} \to \mathbb{A}$.

The concretization function γ maps each element of the abstract domain to a set containing all basic blocks that it represents. Conversely, the representation function β maps individual basic blocks to a representation in the abstract domain.[6] Figure 8.2 visualizes these functions and the constraints we impose on them to constitute an abstract domain:

$$\forall b \in \mathbb{B}.\ b \in \gamma(\beta(b)) \tag{8.1}$$

$$\forall a, a' \in \mathbb{A}.\ a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a') \tag{8.2}$$

These constraints ensure that the functions are consistent with the orders of the domains: Equation (8.1) assures that a basic block b is part of the concretization of its representative $\beta(b)$. Equation (8.2) requires that γ is monotone with respect to the domain orders, i.e., that if one abstract block is more general than another, it represents more concrete basic blocks.

[6] We require β instead of the more common *abstraction function* $\alpha : \mathbb{C} \to \mathbb{A}$ since it tends to be easier to define and since our generalization algorithm only ever needs to abstract individual concrete basic blocks. With an abstraction function α, β could be defined as $\beta(x) = \alpha(\{x\})$.

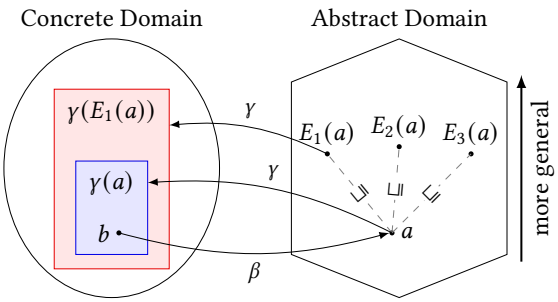


Figure 8.2. Relationships between elements of the concrete and the abstract domain. (Ritter and Hack, 2022)

While β is often straightforward to implement, γ is unwieldy: If implemented explicitly, it would need to produce very large sets of concrete basic blocks. To avoid this problem, abstract domains in AnICA do not come with an explicit concretization function, but with a *concretization sampler* $\widetilde{\gamma}(a)$ that randomly samples a basic block from $\gamma(a)$.

Our algorithm does not require an explicit generality relation $\sqsubseteq$ either. Instead, we use a set $Exps \subseteq \mathbb{A} \rightharpoonup \mathbb{A}$ of partial *expansion* functions that each map abstract blocks to their immediate successors in the generality relation.

In practice, these expansion functions each describe a way to modify abstract blocks in order to obtain a slightly more general abstract block. Formally, each expansion function $E \in Exps$ needs to be strictly ascending and monotone:

$$\forall a \in \mathrm{dom}(E).\ a \sqsubseteq E(a) \wedge a \neq E(a)$$
$$\forall a, a' \in \mathrm{dom}(E).\ a \sqsubseteq a' \Rightarrow E(a) \sqsubseteq E(a')$$

If there is an immediate successor a' to a in the generality order, there should be an expansion function $E \in Exps$ such that $E(a) = a'$. However, we require that for each abstract block a, the number of expansion functions $E \in Exps$ such that $a \in \mathrm{dom}(E)$ is finite.

Example 8.1. For an informal notion of what an abstract block for the x86-64 instruction set architecture looks like, consider the following example:

Instructions:
1. mnemonic: `mov`; memory: read
2. mnemonic: `add`; category: arithmetic; memory: read+written

Aliasing:
- operand 1 of instruction 1 must alias with operand 2 of instruction 2

(AB1)

This abstract block represents all basic blocks consisting of two instructions that satisfy constraints on their mnemonics, their category, their use of memory, and the aliasing of

their operands.[7] For instance, the following concrete basic block is represented by the above abstract block:

```
1 mov rbx, [rdx + 42]
2 add [r8], rbx
```

The mnemonics fit their constraints, the first instruction only reads from memory at the location [rdx + 42], whereas the second one reads and writes memory at [r8]. The aliasing constraint is satisfied by using the common rbx register.[8]

An expansion function could for example drop the constraint on the mnemonic of the second instruction. The result of this expansion function is the following abstract block:

Instructions:
1. mnemonic: mov; memory: read
2. category: arithmetic; memory: read+written

Aliasing:
- operand 1 of instruction 1 must alias with operand 2 of instruction 2

It represents all basic blocks represented by the previous one, and more: All arithmetic instructions are now allowed as the second instruction. ⌟

The following definition collects the above requirements we pose on an abstract domain.

Definition 8.2. A *valid* abstract domain $(\mathbb{A}, \sqsubseteq)$ with a concretization function $\gamma : \mathbb{A} \to \mathbb{C}$, a representation function $\beta : \mathbb{B} \to \mathbb{A}$, and a set $Exps \subseteq \mathbb{A} \rightharpoonup \mathbb{A}$ of expansion functions satisfies the following constraints:

1. $(\mathbb{A}, \sqsubseteq)$ is partially ordered.
2. $(\mathbb{A}, \sqsubseteq)$ contains no infinite ascending chains.
3. All expansion functions $E \in Exps$ are strictly ascending:
$$\forall a \in \mathrm{dom}(E).\ a \sqsubseteq E(a) \wedge a \neq E(a)$$
4. For each abstract block, the number of applicable expansion functions is finite.
5. For each abstract block, every direct successor in $\sqsubseteq$ is reachable via an expansion function:
$$\forall a, b \in \mathbb{A}.\ \left(a \sqsubset b \wedge \nexists c \in \mathbb{A}.\ a \sqsubset c \sqsubset b\right) \Rightarrow \exists E \in Exps.\ E(a) = b$$
6. The expansion functions are monotone:
$$\forall a, a' \in \mathrm{dom}(E).\ a \sqsubseteq a' \Rightarrow E(a) \sqsubseteq E(a')$$

[7] We use the term "alias" here for instruction operands that refer to the same data. It is therefore not restricted to memory operands, but also refers to (fully or partially) overlapping register operands.

[8] See Appendix C for an overview of the registers of the x86-64 ISA.

7. The concretization function is monotone:
$$\forall a, a' \in \mathbb{A}.\ a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a')$$
8. Concretization and representation functions satisfy the soundness constraint:
$$\forall b \in \mathbb{B}.\ b \in \gamma(\beta(b))$$

⌟

Next, we describe how AnICA automatically generalizes interesting basic blocks to concise and pertinent abstract blocks. With this generalization algorithm in mind, we then formalize the details of a modular abstract domain for the x86-64 instruction set architecture in Section 8.1.4.

8.1.3. Generalization Algorithm

AnICA generalizes an interesting basic block b as shown in Algorithm 8.2. The first result candidate is the representative $\beta(b)$, an abstract block that represents the given basic block b as specifically as possible in the abstract domain (line 1). After validating that the initial candidate is interesting (line 2), we choose an expansion to make the candidate more general (lines 7–8). If the expanded abstract block is still interesting, we use it as a new candidate (line 9). Otherwise, we note this expansion as rejected (line 10) and choose a different one. As expansions are monotone and ascending, a once rejected expansion cannot be useful later in generalization.

Once no expansion is left, we return the now general and still pertinent candidate (lines 5–6). Termination is guaranteed as the abstract domain has no infinite ascending chains and the set of expansions that apply to an abstract block is finite, cf. Theorem 8.4.

We extend our definition of interestingness (Section 8.1.1) from basic blocks to abstract blocks for this algorithm. Ideally, an abstract block should be deemed interesting if all represented basic blocks are interesting. As this is prohibitively expensive to check, we approximate this property. We randomly sample represented basic blocks with the concretization sampler $\widetilde{\gamma}$

```
   Input: basic block b
1  absBB ← β(b);
2  if absBB is not interesting then return b;
3  rejected ← {};
4  while True do
5  |  avail ← {E ∈ Exps | absBB ∈ dom(E)} \ rejected;
6  |  if avail = {} then return absBB ;
7  |  exp ← choose(avail);
8  |  t ← exp(absBB);
9  |  if t is interesting then absBB ← t ;
10 |  else rejected ← rejected ∪ {exp} ;
```

Algorithm 8.2. Generalization Algorithm.

and consider the abstract block interesting if all samples are interesting. The number of samples is a parameter of AnICA.

Example 8.3. Assume that the predictors under test disagree on the latency of reading a value from memory that was written immediately before. Figure 8.3 visualizes a simplified run of the generalization algorithm for this problem.

The first hypothesis for an abstract block (1) is the representation of a concrete basic block exhibiting this behavior. As the represented basic blocks all show the assumed inconsistency, this abstract block is interesting (marked with green double borders). In the next step, the algorithm expands the aliasing requirement and reaches abstract block (2). With $\top$, we denote that the component is unconstrained. Since the sampled basic blocks are then no longer restricted to using the same memory location, they are not uniformly interesting, which causes this expansion to be rejected (marked with single red borders). When we expand the mnemonic of the second instruction, the abstract block (3) continues to only cover interesting basic blocks. Allowing any of the instructions to not use memory (4,5) leads to more rejections. Finally, this leaves only the mnemonic of the first instruction to be expanded (6), which results in another interesting basic block. After that, all components of the abstract blocks are either $\top$ or only affected by rejected expansions. Hence, the algorithm terminates returning abstract block (6). ⌟

We formulate and prove two statements about this algorithm. The first theorem concerns the algorithm's termination, whereas the second one states that, when approximations are idealized, the algorithm achieves our goals of generality and pertinence for the resulting abstract blocks.

Theorem 8.4. The generalization algorithm always terminates if a valid abstract domain is used.

Proof. See Appendix A.5.1. □

Theorem 8.5. Given an ideal check for the interestingness of an abstract block and a valid abstract domain, the result of the generalization algorithm is

1. interesting and
2. maximal, i.e., every more general abstract block is not interesting,

if the initial representative $\beta(b)$ is interesting. A check for interestingness $chk : \mathbb{A} \to \{\mathsf{True}, \mathsf{False}\}$ is ideal if

$$chk(a) \Leftrightarrow \forall b \in \gamma(a).\ b \text{ is interesting.}$$

Proof. See Appendix A.5.2. □

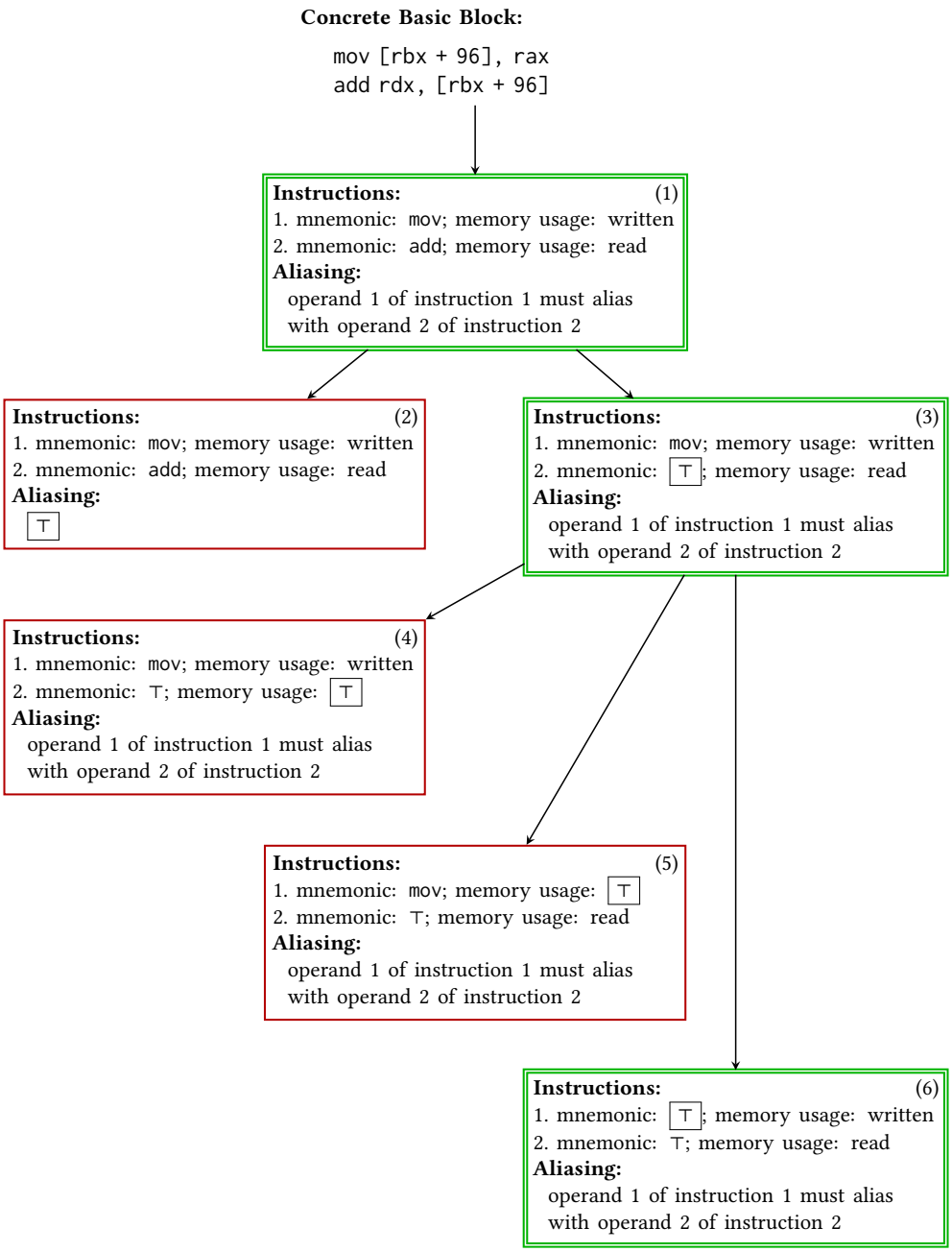


Figure 8.3. An example generalization tree. (Ritter and Hack, 2022)

The order in which expansions are chosen in the generalization algorithm (line 7 in Algorithm 8.2) affects the result. We approximate an optimal expansion order by generalizing each candidate several times with different random expansion orders. Since we prune subsumed discoveries from the results, we can try arbitrarily many different expansion orders without degrading the quality of our discovery results.

The straightforward nature of the generalization algorithm is helpful when interpreting AnICA's results. As Parnin and Orso (2011) noted, tools that automatically find bugs and present them only abstractly to users do not necessarily help to fix the bugs. They formulate the observation that "[p]roviding overviews that cluster results and explanations that include data values [and] test case information [...] could make faults easier to identify and tools ultimately more effective." (Parnin and Orso, 2011, Section 6.1)

Our generalization algorithm naturally produces such clustered results and explanations in the form of a generalization decision tree like the one in Figure 8.3. Each decision in this tree is justified by the set of basic blocks that was sampled and evaluated to gauge their interestingness. Our implementation of AnICA therefore includes a graphical interface to inspect the generalization trees of its discoveries to provide users with detailed information and concrete debuggable inputs for the tools under investigation.

The basic blocks that justify the rejection of an expansion are particularly insightful. They highlight the limits of an inconsistency's scope in a way that a plain clustering of inconsistent basic blocks could not. We demonstrate in Section 8.2 how we can use such results to identify behaviors of throughput predictors that run counter to common expectations.

8.1.4. Our Abstract Domain

We now define an abstract domain for the x86-64 instruction set architecture (ISA) as this architecture is supported by most available basic block throughput predictors. The AnICA algorithm is not conceptually limited to this ISA and similar domains can be designed for, e.g., ARM architectures.

Top-Level Abstraction

Our abstract domain $(\mathbb{A}, \sqsubseteq_{\mathbb{A}})$ separates constraints on the individual instructions of the represented basic blocks from constraints on how they interact via their operands. An abstract block thus consists of a sequence of abstract instructions from an instruction abstraction $\mathbb{A}_{in}$ and abstract alias information from an aliasing abstraction $\mathbb{A}_{al}$:

$$\mathbb{A} := \mathbb{A}_{in}^{+} \times \mathbb{A}_{al} \tag{8.3}$$

The partial order $\sqsubseteq_{\mathbb{A}}$ only relates abstract blocks with the same number of abstract instructions and is defined through partial orders $\sqsubseteq_{in}$ and $\sqsubseteq_{al}$ among its components:

$$\begin{aligned}(x_{in}, x_{al}) \sqsubseteq_{\mathbb{A}} (y_{in}, y_{al}) :\Leftrightarrow\ & x_{al} \sqsubseteq_{al} y_{al} \wedge |x_{in}| = |y_{in}| \\ & \wedge (\forall 1 \leq k \leq |x_{in}|.\ x_{in}[k] \sqsubseteq_{in} y_{in}[k])\end{aligned} \tag{8.4}$$

In a similar way, the concretization $\gamma_{\mathbb{A}} : \mathbb{A} \to \mathcal{P}(\mathbb{B})$ and representation $\beta_{\mathbb{A}} : \mathbb{B} \to \mathbb{A}$ functions are defined based on per-component-functions:

$$b \in \gamma_{\mathbb{A}}((a_{in}, a_{al})) :\Leftrightarrow b \in \gamma_{al}(a_{al}) \wedge |a_{in}| = |b| \\ \wedge (\forall 1 \leq k \leq |a_{in}|.\ b[k] \in \gamma_{in}(a_{in}[k])) \tag{8.5}$$

$$\beta_{\mathbb{A}}(b) := \left([\beta_{in}(b[i])]_{i=1,\ldots,|b|}, \beta_{al}(b)\right) \tag{8.6}$$

Expansion functions expand one component of the abstract block, i.e., an abstract instruction or the aliasing abstraction, and leave all other components untouched:

$$\begin{aligned} &Exps_{\mathbb{A}} := \\ &\quad \left\{\lambda(a_{in}, a_{al}).\ (a_{in}[k \mapsto E(a_{in}[k])], a_{al}) \text{ if } k \leq |a_{in}| \wedge a_{in}[k] \in \mathsf{dom}(E) \;\middle|\; k \in \mathbb{N}, E \in Exps_{in}\right\} \\ &\quad \cup \left\{\lambda(a_{in}, a_{al}).\ (a_{in}, E(a_{al})) \text{ if } a_{al} \in \mathsf{dom}(E) \;\middle|\; E \in Exps_{al}\right\} \end{aligned} \tag{8.7}$$

Note that here and in the following, definitions of sets of expansion functions follow a common scheme with some term t and predicates P and Q:

$$Exps := \left\{\lambda x.\ t(x, y) \text{ if } P(x, y) \;\middle|\; Q(y)\right\}$$

This structure with two separate predicates is required since *Exps* is a set of partial functions. With "$\lambda x.\ t(x, y)$ if $P(x, y)$", we denote an individual partial function with the function parameter x that is only defined if the predicate $P(x, y)$ is satisfied. The surrounding set definition $\left\{ \cdots \;\middle|\; Q(y)\right\}$ specifies that *Exps* contains such a partial function for each value of y that satisfies $Q(y)$.

Instruction Abstraction

The set $\mathbb{A}_{in}$ contains abstract instructions that describe sets of instruction schemes, i.e., instruction representations that are parametric in their operands (cf. Section 2.1). We extract the instruction schemes for the x86-64 ISA from uops.info (Abel and Reineke, 2019). Additionally, we collect for each instruction scheme several features such as the mnemonic, the explicit and implicit operand types, whether and how it uses memory, and to which instruction category and ISA extension it belongs. Table 8.1 shows examples for such features and their values. Our domain groups instruction schemes through constraints on these features. The form of these constraints for a feature f is determined by its feature abstraction $\mathbb{A}_F[f]$. Table 8.2 introduces the feature abstractions that we use for our abstract domain.

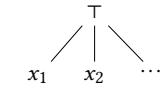
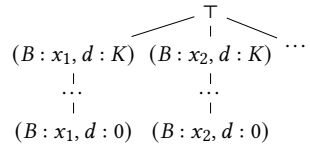
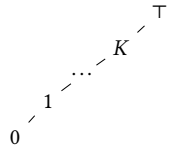
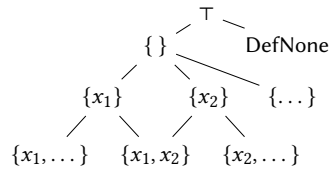
For simple features like the category and ISA extension to which the instruction belongs or the presence of a `lock` or `rep` prefix, we use the *singleton* abstraction. It expresses that all represented instruction schemes have a specific value for the feature, e.g., that they are all part of the AVX vector extension, or that they do (or do not) have a `lock` prefix to make memory accesses atomic.

For the mnemonic, we use the *edit distances* abstraction. It constrains the represented mnemonics by an upper bound d on the Levenshtein editing distance from a base string B.

Table 8.1. Values of various features f for two example instruction schemes.

f	$f(\texttt{vaddpd}\ \langle\texttt{XMM}\rangle_W, \langle\texttt{XMM}\rangle_R, \langle\texttt{XMM}\rangle_R)$	$f(\texttt{lock sub}\ \langle\texttt{MEM[32]}\rangle_{RW}, \langle\texttt{IMM[32]}\rangle_R)$
mnemonic	vaddpd	sub
memory usage	$\emptyset$	$\{\mathrm{R}, \mathrm{W}, \mathrm{Size}: 32\}$
operand types	$\{\langle\texttt{XMM}\rangle_W, \langle\texttt{XMM}\rangle_R\}$	$\{\langle\texttt{MEM[32]}\rangle_{RW}, \langle\texttt{IMM[32]}\rangle_R, \texttt{FLAGS}_W\}$
has lock prefix	no	yes
ISA-Extension	AVX	base
µops in Skylake	$\{\{0,1\},\{2,3\}\}$	$\{4\times\{0,1,5,6\}, 2\times\{0,6\}, 2\times\{2,3\}, \{4\}\}$

Table 8.2. Feature domains used in AnICA. The domains are shown as Hasse diagrams, where the partial order is indicated through the lines: If x is connected to y and y is closer to the top, $x \sqsubseteq_F y$ holds. The representation function $\beta^{(f)}$ of the Log Sizes domain improves upon the insufficient version from the paper. (Ritter and Hack, 2022)

Domain	Hasse Diagram	Used for	$\gamma_X^{(f)}(av)$ for $av \neq \top$	$\beta_X^{(f)}(i)$
Singletons		Category, ISA-Extension, Prefixes	$\{i \mid f(i) = av\}$	$f(i)$
Edit Distances		Mnemonic	$\{i \mid dist(f(i), B) \leq d\}$	$(B: f(i), d: 0)$
Log Sizes		Number of µops	$\{i \mid \lvert f(i)\rvert < 2^{av}\}$	if $f(i) = \emptyset$: 0 if $\lvert f(i)\rvert \geq 2^K$: ⊤ otherwise: $\lfloor \log_2(\lvert f(i)\rvert) + 1 \rfloor$
Subset-or-None		Memory Usage, Operand Types	if av = DefNone: $\{i \mid f(i) = \emptyset\}$ otherwise: $\{i \mid f(i) \supseteq av\}$	if $f(i) = \emptyset$: DefNone otherwise: $f(i)$

If they share their base, abstract values are ordered by their value of d, which is limited by a maximum bound K.[9] This abstraction allows AnICA to group instruction schemes with similar mnemonics: An abstract value representing only `vaddpd` (an addition for vectors of double-precision floats) can be expanded to also represent `vaddps` (the same operation with single-precision) and the scalar versions `vaddsd` and `vaddss`. The edit distance abstraction is heuristic in nature: neither do all similar instructions have similar mnemonics nor are all instructions with similar mnemonics similar themselves. We nevertheless found this abstraction to be helpful in practice since, for instance, mnemonic suffixes that rarely affect the instruction's performance behavior are common in modern ISAs (e.g., specifying the floating-point format and vector width, or the condition for conditional move instructions).

We use the *log sizes* abstraction for the (multi-)set of micro operations required to execute an instruction. AnICA can therefore group instruction schemes by their complexity: An abstract value represents all instruction schemes that are decomposed into less than 2^k µops for a certain k. To avoid infinite ascending chains of abstractions, a maximal value for k is a parameter K of this abstraction.[10]

Whether and how an instruction accesses memory affects its performance significantly. Our domain uses the fine-grained *subset-or-none* abstraction with subsets of $\{\mathrm{R}, \mathrm{W}, \mathrm{Size} : n\}$ to represent memory usage. This enables AnICA to relax constraints on memory usage step by step. An abstract instruction representing only instruction schemes that read (R) and write (W) n bits in memory can be expanded by dropping any of these constraints. The expanded abstract instruction might, e.g., represent all instruction schemes that at least read n bits from memory. With DefNone, only instruction schemes that do not access memory are represented. We also use this abstraction for the set of operand types that occur in the instruction schemes. Notably, we use this abstraction only with finite sets.

Formally, abstract instructions are tuples of elements of feature abstractions $\mathbb{A}_F[f_j]$ for each considered feature f_j:

$$\mathbb{A}_{in} := \mathbb{A}_F[f_1] \times \cdots \times \mathbb{A}_F[f_N] \tag{8.8}$$

The partial order among abstract instructions relies on the partial orders of the involved feature abstractions:

$$(x_1, \cdots, x_N) \sqsubseteq_{in} (y_1, \cdots, y_N) :\Leftrightarrow \bigwedge_{j \in [1,N]} x_j \sqsubseteq_F y_j \tag{8.9}$$

The last two columns of Table 8.2 define the feature concretization and representation functions $\gamma_X^{(f)}$ and $\beta_X^{(f)}$ for each feature abstraction X. They are parameterized by the feature f for which they are used and refer with $f(i)$ to the value of the instruction scheme i for this feature.

The feature concretization functions $\gamma_X^{(f)}$ map *abstract values* from the feature abstraction X to their set of represented instruction schemes i. All feature abstractions have a maximal abstract value $\top$, which represents the absence of any constraint. Therefore, its concretization

[9] In practice, we use $K = 3$ for the mnemonic abstraction.
[10] In practice, we use $K = 5$ to abstract the number of µops.

is the same for every domain: the entire set of available instruction schemes. The feature representation functions $\beta_X^{(f)}$ define the value from the feature abstraction that best describes an instruction scheme with the value $f(i)$ for the feature f.

We use these feature concretization and representation functions to define the corresponding functions for the instruction abstraction. An abstract instruction imposes the conjunction of the per-feature constraints on the represented instruction schemes. Therefore, it concretizes to the intersection of the per-feature concretizations $\gamma^{(f)}_{\mathbb{A}_F[f]}$ applied to the abstract instruction's component $ai[f]$ for each feature f:

$$\gamma_{in}(ai) := \bigcap_{f \in Features} \gamma^{(f)}_{\mathbb{A}_F[f]}(ai[f]) \tag{8.10}$$

To obtain a representative abstract instruction for a concrete one, we apply the representation functions for each feature:

$$\beta_{in}(i) := \left(\beta^{(f_1)}_{\mathbb{A}_F[f_1]}(i), \ldots, \beta^{(f_N)}_{\mathbb{A}_F[f_N]}(i)\right) \tag{8.11}$$

Analogously, an expansion function for an abstract instruction ai takes a non-$\top$ component and replaces it with one of its immediate successors in the generalization order:

$$Exps_{in} := \left\{\lambda ai.\ ai[f \mapsto y] \text{ if } ai[f] = x \;\middle|\; f \in Features, x \in \mathbb{A}_F[f] \setminus \{\top\}, y \text{ succeeds } x \text{ in } \sqsubseteq_{\mathbb{A}_F[f]}\right\} \tag{8.12}$$

Aliasing Abstraction

The subcomponent $\mathbb{A}_{al}$ represents aliasing constraints among operands of instructions. We refer to an operand of an instruction via a pair $(idx_i, idx_o) \in Idx := (\mathbb{N} \times \mathbb{N})$ of indexes into the sequence of instructions and into the sequence of operands of the instruction. The operand idx_o of instruction idx_i in the basic block b is denoted as $b[(idx_i, idx_o)]$.

An aliasing constraint for a pair of such instruction operand designators can state that they *must* or *must not* alias, or that no constraint applies (denoted as $\top$). The aliasing subcomponent is therefore defined as a mapping as follows:

$$\mathbb{A}_{al} := (Idx \times Idx) \to \{\texttt{must}, \texttt{mustnot}, \top\} \tag{8.13}$$

A value g of the aliasing abstraction is more general than or as general as another h if g imposes the same or a weaker constraint than h for *every* pair of operands:

$$h \sqsubseteq_{al} g :\Leftrightarrow \forall x \in (Idx \times Idx).\ g(x) = \top \vee h(x) = g(x) \tag{8.14}$$

Only finitely many entries in an element of the aliasing abstraction may differ from $\top$.

This component of the abstraction is more intricate than one might expect at first: The instruction abstraction can represent sets of vastly different instruction sequences. One abstract instruction could for example represent a 2-operand integer addition operation and a 3-operand floating point addition. Consequently, the aliasing abstraction needs to handle

cases where operands do not match, i.e., cannot possibly alias, or where they are not present at all in some of the represented basic blocks.

We handle these cases with a concretization function that only applies constraints on operands that match and are present in the concrete basic block:

$$\begin{aligned} b \in \gamma_{al}(h) :\Leftrightarrow \bigwedge_{((i_1,i_2)\mapsto x)\in h} \Big(& \big(b[i_1] \text{ and } b[i_2] \text{ exist and match}\big) \\ & \Rightarrow \Big(x = \top \vee \big(x = \texttt{must} \wedge \big(b[i_1] \text{ and } b[i_2] \text{ alias}\big)\big) \\ & \qquad \vee \big(x = \texttt{mustnot} \wedge \big(b[i_1] \text{ and } b[i_2] \text{ do not alias}\big)\big)\Big)\Big) \end{aligned} \tag{8.15}$$

The representation function β_{al} is defined to capture the must-alias and must-not-alias relations between matching operands of the concrete basic block:

$$\beta_{al}(b) := \lambda(i_1, i_2).\ \begin{cases} \texttt{must} & \text{if } b[i_1], b[i_2] \text{ exist, match, and alias} \\ \texttt{mustnot} & \text{if } b[i_1], b[i_2] \text{ exist, match, and do not alias} \\ \top & \text{otherwise} \end{cases} \tag{8.16}$$

It is straightforward to decide for a pair of register operands whether they alias: They alias if and only if they are the same or if one is a sub-register of the other.[11] Whether memory operands alias depends on the values of registers. We approximate this by considering two memory operands aliasing if they are syntactically identical, and not aliasing otherwise. In general, this is not a sound approximation: Two memory operands can look entirely different but refer to the same address or vice versa. It is only adequate for our use case because the basic block sampling method described in the following section manages memory operands such that they alias if and only if they are syntactically identical. As there are only finitely many instructions and operands in any basic block, β_{al} only produces assignments with a finite number of non-$\top$ entries, as required.

The expansion functions for the aliasing component of an abstract block each replace a non-$\top$ entry in the aliasing abstraction with $\top$:

$$Exps_{al} := \left\{\lambda h.\ h[x \mapsto \top] \text{ if } h(x) \neq \top \;\middle|\; x \in Idx \times Idx\right\} \tag{8.17}$$

Properties of the Abstract Domain

The termination and correctness statements of the generalization algorithm given with Theorem 8.4 and Theorem 8.5 are conditional on using a valid abstract domain, as defined in Definition 8.2. With the following theorem, we establish that the domain presented in this section fulfills this requirement:

Theorem 8.6. The presented construction is a valid abstract domain.

Proof. See Appendix A.5.3. □

[11] We do not consider the legacy floating point extensions x87 and MMX; register aliasing is more complicated for the x87 register stack.

8.1.5. Sampling Represented Basic Blocks

Our generalization algorithm relies on a method $\widetilde{\gamma}$ to randomly sample basic blocks that are represented by a given abstract block. For arbitrary elements of our abstract domain, this is a hard problem: Sampling a block that fulfills the aliasing constraints essentially corresponds to a graph-coloring register allocation problem (Chaitin *et al.*, 1981) because concrete registers have to be found that comply with the aliasing constraints. Since there are no restrictions on these constraints, arbitrary interference graphs can emerge in general which renders sampling NP-hard in theory.

As the concretization sampler $\widetilde{\gamma}$ is a very common operation in AnICA's generalization algorithm, we do not implement a complete solution to the NP-hard sampling problem. Instead, we proceed greedily as follows:

1. For each abstract instruction, randomly choose a represented instruction scheme.

2. If the schemes have fixed operands, select those and set all related must-alias operands accordingly.[12]

3. Repeatedly: Where an operand is not yet selected, randomly choose one that is not forbidden through must-not-alias constraints. Set all must-alias operands accordingly.

We restrict what registers may be used as register operands to have distinct registers available for the base registers of memory operands that cannot be overwritten. If two memory operands are required to alias, we instantiate them with the same combination of base register and displacement. In case of a no-alias constraint on memory operands, we use different combinations of base register and displacement.

If, at any point in this algorithm, no selection is possible without violating the alias constraints or the requirements of the instruction schemes, the sampling fails and needs to be repeated. For an example, consider the following abstract block for the x86-64 ISA with two unconstrained abstract instructions and a must-not alias constraint:

Instructions:
1. $\top$
2. $\top$

Aliasing:
- operand 2 of instruction 1 must not alias with operand 2 of instruction 2

If, in the first step of the sampling algorithm, we choose shift instructions with variable shift amount for both instructions, sampling will fail: Both instructions need to have register `c` as second operand for the shift amount, which would violate the alias constraint.

[12] For example, shifts in x86-64 use the `c` register for their shift amount.

In practice, sampling rarely fails for the short instruction sequences that we sample. It affected 0.01% of the ca. 4.8×10^6 sampling operations in the campaigns presented in Section 8.2.3.[13]

8.1.6. Checking for Subsumption

In Algorithm 8.1, we check whether concrete or abstract blocks are subsumed by an abstract block to avoid unnecessary generalizations and to prune irrelevant discoveries. The fixed number and positions of instructions in our abstract domain simplify the sampling of basic blocks, but they hinder us here. The concretization $\gamma_{\mathbb{A}}(a)$ of an abstract block a does not contain basic blocks that we would like to consider subsumed by a. For an example, reconsider abstract block AB1 from Example 8.1:

Instructions:
1. mnemonic: `mov`; memory: read
2. mnemonic: `add`; category: arithmetic; memory: read+written

Aliasing:
- operand 1 of instruction 1 must alias with operand 2 of instruction 2

(AB1)

The following basic block would not be included in the concretization of AB1:

```
1 add [r8], rbx
2 mov rbx, [rdx + 42]
```

However, the instructions here are the same as in the example represented by AB1, only in a different order that has no impact on the block's sustained throughput. The throughput is determined by the rate at which the basic block can be executed repeatedly for an indefinite number of iterations. What determines the throughput of a basic block bb is therefore the trace of instructions resulting from repeating bb a large number of times. When a basic block bb' results from rotating bb (i.e., removing a sequence of instructions from the beginning and appending it to the end), its trace differs from the one of bb only by short pre- and suffixes whose influence on the execution time vanishes with a growing number of repetitions.

Similarly, if this basic block exhibits an inconsistency in the predictions, it is likely to have the same reason as AB1 as it contains the characteristics described by the abstract block:

```
1 mov rbx, [rdx + 42]
2 nop
3 add [r8], rbx
```

[13] An alternative approach would be to encode the aliasing constraints as SMT formulas and use, e.g., the approach of Dutra *et al.* (2019) to sample satisfying solutions. In comparison to our approach this would eliminate the chance of sampling errors at the cost of an increased execution time of the sampling steps.

Yet, it is not included in $\gamma_{\mathbb{A}}(\text{AB1})$ since it contains three instead of two instructions.

We therefore do not rely on the partial order of the abstract domain to implement the subsumption checks in Algorithm 8.1. Instead, we check for the following definition:

Definition 8.7. An abstract block (a^1_{in}, a^1_{al}) *subsumes* another (a^2_{in}, a^2_{al}) if there is a mapping $m : I^1 \to I^2$ from the indexes I^1 of the abstract instructions of a^1_{in} to the indexes I^2 of a^2_{in} s.t.

$$\forall i, j \in I^1.\ i \neq j \Rightarrow m(i) \neq m(j) \tag{C1}$$

$$\forall i \in I^1.\ \gamma_{in}(a^2_{in}[m(i)]) \subseteq \gamma_{in}(a^1_{in}[i]) \tag{C2}$$

$$\forall((i, op_1), (j, op_2) \mapsto x) \in a^1_{al}.\ x \neq \top \Rightarrow a^2_{al}((m(i), op_1), (m(j), op_2)) = x \tag{C3}$$

$$\forall i \in I^1.\ \forall k \in I^2 \text{ between } m(i) \text{ and } m((i+1) \bmod |I^1|).\ \nexists i'.\ m(i') = k \tag{C4}$$

An abstract block *a subsumes* a concrete basic block b if it subsumes $\beta_{\mathbb{A}}(b)$. ⌟

In other words, m needs to be injective (C1) and map abstract instructions to at least as specific ones (C2). Furthermore, the aliasing constraints imposed by a^2_{al} on the mapped instructions need to be at least as strong as those imposed by a^1_{al} (C3). Lastly, the order of the mapped instructions $m(i)$ in a^2_{in} needs to be a rotation of the order of the instructions i in a^1_{in} (C4). All instructions of a^1_{in} need to have a counterpart in a^2_{in}, but not vice versa.

We encode these constraints in a boolean formula that is satisfiable if and only if such a mapping exists and use a SAT solver to discharge them. In an AnICA campaign, subsumption checks are not numerous and in our experience, SAT solvers can solve the formulas quickly.

8.1.7. Ranking Abstract Basic Blocks

When evaluating the usefulness of AnICA discoveries, as well as for guiding developers interested in improving throughput predictors, it is helpful to rank abstract basic blocks by a notion of importance. In the following, we describe three approaches to ranking abstract basic blocks that we found useful when evaluating AnICA and carrying out the case studies presented in Section 8.3.

Ranking by Interestingness

Every abstract block that results from AnICA's generalization has been checked for interestingness. This means that we sampled a number of represented concrete basic blocks and computed the (relative or absolute) difference between the predictions of the tools under investigation for the basic blocks for each discovery. A natural metric for the relevance of the abstract block is therefore the mean prediction difference over the set of sampled basic blocks. The higher it is, the more dramatic is the inconsistency characterized by the abstract block. For inputs that crash a throughput predictor, we set this metric to infinity to indicate maximal interestingness.

Ranking by Generality

Inconsistencies do not need to come with large deviations in the predictions to indicate a significant difference in the tools under investigation. We therefore use *generality* as an alternative metric for ranking abstract blocks. The idea is that we want to find discoveries that affect large classes of concrete basic blocks.

There are several conceivable options to define such a metric, with different trade-offs. We need to consider the effort required to compute them – an expensive choice would be to sample a large number of basic blocks and check how many of them are subsumed by each discovery. It is also not obvious how represented basic blocks should be weighted. For instance, should instruction schemes with wide immediate constants be considered more general because each possible immediate value counts as a different instruction?

We chose a notion of generality that is inexpensive to compute and operates, like our generalization algorithm, on the granularity of instruction schemes: An abstract block's generality is the minimal number of instruction schemes represented by any of its abstract instructions. While this is a simplification of reality – it ignores aliasing constraints and the number of abstract instructions in the abstract block – this metric was instrumental for finding several examples for our case studies.

Maximizing the Number of Subsumed Basic Blocks

If users of AnICA have a concrete set of basic blocks that they consider particularly relevant, e.g., extracted from an important benchmark set, this can be leveraged to a custom-tailored notion of generality. For instance, we can rank AnICA's abstract blocks by the number of basic blocks from the set that they subsume.

An extension to this strategy is to solve the following integer linear program (ILP) to obtain a subset of k discoveries that subsume a maximal portion of the basic block set B from the set *AbsBlocks* of AnICA's discoveries:[14]

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j \in B} \mathsf{BB.covered}[j] \\
\text{subject to} \quad & \sum_{i \in \mathit{AbsBlocks}} \mathsf{AB.used}[i] \leq k \\
& \sum_{i \in \mathit{AbsBlocks} \,\wedge\, i \text{ subsumes } j} \mathsf{AB.used}[i] \geq \mathsf{BB.covered}[j] && \text{for all } j \in B \\
& \mathsf{AB.used}[i] \in \{0,1\} && \text{for all } i \in \mathit{AbsBlocks} \\
& \mathsf{BB.covered}[j] \in \{0,1\} && \text{for all } j \in B
\end{aligned}
$$

The ILP uses two groups of binary variables: an $\mathsf{AB.used}[i]$ variable for each abstract block i and a $\mathsf{BB.covered}[j]$ variable for each concrete basic block j. If one of the AB.used variables is 1, the corresponding abstract block is chosen as one of the k maximally subsuming discoveries. The first constraint of the ILP ensures that no more than k abstract blocks are

[14] See Appendix B for an overview of the terminology regarding integer linear programming used in this thesis.

selected. If one of the BB.covered variables is 1, the corresponding concrete basic block is subsumed by at least one of the chosen discoveries. We encode this relationship with the second constraint of the ILP: BB.covered$[j]$ cannot be greater than 0 unless an abstract block i that subsumes it is chosen with AB.used$[i]$. With the ILP's objective term, we require that an optimal solution covers as many concrete basic blocks as possible.

A set of abstract blocks extracted from the values of the AB.used variables in an optimal solution to the ILP is a maximally diverse selection of AnICA discoveries. With an appropriate selection of the parameter k, AnICA's results can thus be summarized as concisely as desired.

8.2. Experimental Evaluation

The main goal of AnICA is to provide insights into the basic block throughput predictors under investigation. Since this goal is not easily quantified, we evaluate AnICA in two parts: a general investigation of how inconsistencies are generalized (Section 8.2.3) and a number of detailed case studies to give examples of actual insights gained (Section 8.3).

8.2.1. Considered Tools

We compare a broad range of throughput predictors:

- the Intel Architecture Code Analyzer, IACA (Intel, 2012), version 3.0
- LLVM's Machine Code Analyzer, llvm-mca (Di Biagio, 2018), as contained in LLVM version 13 (if not stated otherwise)
- the Open Source Architecture Code Analyzer, OSACA (Laukemann *et al.*, 2018), version 0.4.6
- the uops.info Code Analyzer, uiCA (Abel and Reineke, 2022)[15]
- the neural-network-based Ithemal (Mendis *et al.*, 2019), with the provided model that was trained on basic blocks from the BHive data set (Chen *et al.*, 2019)[16]
- the modified llvm-mca version of DiffTune (Renda *et al.*, 2020), with the parameters provided by the authors[17]

See Section 7.2 of this thesis for an overview of these tools. We did not compare the MAQAO Code Quality Analyzer (Rubial *et al.*, 2014) as the current version at the time of evaluation was only documented to support entire loops as input for its throughput prediction, which not all of the other tools support.[18] We also omit Facile (Abel *et al.*, 2023), Granite (Sýkora *et al.*, 2022) and PerfVec (Li *et al.*, 2023) since, at the time of evaluation, they were not available.

[15] `https://github.com/andreas-abel/uiCA`, commit `71f2eb6`

[16] `https://github.com/ithemal/Ithemal`, commit `47a5734` and `https://github.com/ithemal/Ithemal-models`, commit `87c2468`

[17] `https://github.com/ithemal/DiffTune`, commit `9992f69`

[18] Our AnICA implementation nevertheless supports tools like MAQAO's CQA, with an option to wrap each basic block in a loop when it is given as input to the tools.

8.2.2. AnICA Parameters

We use the following parameters for AnICA:

- Threshold that the relative difference between two predictions must exceed to be considered interesting: 0.5
- Number of samples to check whether an abstract block is interesting: 100
- Maximal length of sampled basic blocks for discovery: 5 instructions
- Number of randomized generalizations per basic block: 5

In preliminary experiments, we found that variations in the latter three parameters do not affect AnICA's results substantially in terms of the metrics used in the following evaluation. Only if they are selected widely out of range (e.g., only using very few samples to check for interestingness or only investigating very short basic blocks), the performance declines. The threshold of the interestingness metric is of more relevance since it determines what inconsistencies are found. The selected value is relatively large, which causes AnICA to focus on substantial output differences. We found such inconsistencies to be more likely to hint at conceptual differences like the handling of memory dependencies (cf. Section 8.3.1).

We extract the instruction schemes used for sampling from uops.info (Abel and Reineke, 2019). For the evaluation, we exclude instruction schemes if they satisfy any of the following conditions:

- They are in a single-instruction-multiple-data (SIMD) or floating point extension other than versions 1 and 2 of the Advanced Vector Extensions (AVX1&2).
- They are not measured by uops.info.[19]
- They affect control flow.
- They need to be executed in privileged mode.

This leaves us with 2940 instruction schemes. For each campaign, we further exclude all instruction schemes that are not supported by one of the tools under investigation.[20] We configure the throughput predictors to assume the Intel Haswell microarchitecture since it is the only one supported by all considered tools.

The AnICA campaigns ran on a system with an Intel Core i9-10900K processor (10 cores, 20 threads, 3.7 GHz) and 64 GB of RAM. Running the predictors to evaluate the interestingness of basic blocks, which constitutes most of the execution time, is performed with 20 concurrent threads.

8.2.3. Generalization of Inconsistencies

The heat map shown in Figure 8.1 at the beginning of this chapter, repeated in Figure 8.4, demonstrates that we can find a large number of inconsistencies among the tools through

[19] This criterion is intended to exclude instructions that are not supported in the microarchitecture, e.g., because they are from outdated ISA extensions.

[20] We consider an instruction scheme supported by a tool if the tool gives a non-zero prediction for a basic block consisting of only an instance of the instruction scheme.

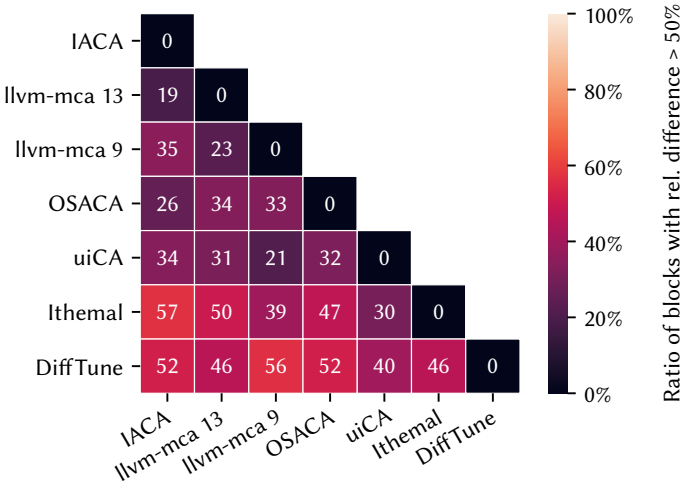


Figure 8.4. Heat map showing the percentage of basic blocks with inverse throughput estimates that deviate by more than 50% for each pair of predictors. Identical to Figure 8.1.

random testing; enough that investigating them all by hand would be infeasible. This section evaluates how AnICA summarizes these inconsistencies.

The evaluation is based on the same data as Figures 8.1 and 8.4: a test set of 10,000 randomly sampled basic blocks consisting of 4 instructions each. We sample these as described in Section 8.1.5 from an abstract block with 4 instructions and no constraints. We ran AnICA for each pair of tools until around 150 discoveries were found. Table 8.3 contains a column for each AnICA campaign.[21] The first line repeats the data from Figure 8.4: the percentage of basic blocks in the test set that are interesting, i.e., for which the relative difference of the predictions exceeds 50% of their average.

The second line shows the percentage of the set of interesting basic blocks from the test set that are subsumed (cf. Section 8.1.6) by an AnICA discovery. At 74% to 97%, these ratios are very high for comparisons of IACA, llvm-mca, uiCA, and OSACA. This indicates that AnICA inferred general descriptions of the differences between these tools.

The third line further demonstrates that AnICA effectively condenses the inconsistent basic blocks for manual inspection: It gives the ratio of interesting basic blocks in the test set that are subsumed by a subset of only ten discoveries of the AnICA campaign. In eight of the campaigns, these numbers were higher than 50%, meaning that in these cases only ten of AnICA's discoveries are sufficient to plausibly explain more than half of the inconsistently predicted basic blocks. In every case except for the Ithemal/uiCA campaign, ten of the AnICA discoveries subsume more than 1,000 inconsistent basic blocks from the dataset, ranging up to 3,060 subsumed inconsistencies in the IACA/Ithemal campaign. The discovery subsets for this

[21] For brevity, we use llvm-mca version 9 only for a comparison to version 13 of the same tool.

Table 8.3. AnICA campaigns to find 150 inconsistencies, with metrics on how many basic blocks from Figure 8.4 they explain, ordered by the percentage of interesting basic blocks subsumed.

	IACA, OSACA	llvm-mca 13, 9	IACA, uiCA	OSACA, uiCA	llvm-mca, uiCA	llvm-mca, OSACA	IACA, llvm-mca	DiffTune, OSACA	IACA, Ithemal	Ithemal, OSACA
BBs interesting	26%	23%	34%	32%	31%	34%	19%	52%	57%	47%
int. BBs covered	97%	97%	91%	85%	83%	77%	74%	69%	68%	66%
…by top 10	68%	92%	82%	53%	72%	55%	70%	34%	54%	33%
run time (h:m)	6:34	0:32	1:35	6:59	1:19	5:28	0:38	9:29	4:34	8:13

	DiffTune, llvm-mca	DiffTune, IACA	Ithemal, llvm-mca	DiffTune, uiCA	DiffTune, Ithemal	Ithemal, uiCA
BBs interesting	46%	52%	50%	40%	46%	30%
int. BBs covered	63%	62%	62%	59%	57%	38%
…by top 10	31%	32%	49%	34%	29%	16%
run time (h:m)	5:55	5:54	5:05	6:25	10:02	5:15

metric were computed with the strategy to maximize the number of subsumed basic blocks presented in Section 8.1.7, applied to the interesting basic blocks in the test set.

The time required to find these discoveries, as displayed in the last line, mainly depends on how fast the tools produce their predictions. While not the focus of this work, we observe that in this setup, IACA, llvm-mca, and uiCA were considerably faster than OSACA, Ithemal, and DiffTune.

The campaigns that include Ithemal and DiffTune still cover a substantial number of inconsistencies, but AnICA finds less potential for generalization here than in the other campaigns. We can identify reasons for this observation from the results for these campaigns: AnICA's generalizations terminate early in several instances where these tools produce results that run counter to common expectations.

For example, Ithemal produces different results for basic blocks that only differ in the specific register that they use, as can be seen in its predictions for basic blocks consisting of a single "rotate left" operation:

Basic Block	`rol r12, cl`	`rol r10, cl`
Predicted Inverse Throughput	0.35 cycles	1.01 cycles

All other tools predict equal throughputs for these blocks.

AnICA groups instructions by instruction schemes, i.e., a form that abstracts from the specific operands of the instruction. It therefore does not generalize inconsistencies that are

not independent of the concrete registers used. Most throughput predictors share this notion and do not change their prediction if, e.g., operand registers in the basic block are replaced (while preserving dependencies). This assumption is evidently not enforced in Ithemal's neural network.

AnICA's results demonstrate this issue and therefore justify the conclusion that Ithemal might benefit from training data where basic blocks are included multiple times with different but semantically equivalent register allocations. Since the measured throughputs for these would be the same, the neural network might learn to abstract from the specific register used.

DiffTune learns parameters for llvm-mca and can therefore not produce different predictions based on the specific operands of the instructions as Ithemal does. We can, however, observe that instructions that are very similar are predicted differently by DiffTune. For example, AnICA finds that the abstract block in Figure 8.5 (e) represents an inconsistency between DiffTune and IACA. This abstract block covers arithmetic right shift instructions, which – as the witnessing experiments in AnICA's generalization decision tree show – DiffTune predicts slower than IACA if they use memory and faster if they do not use memory. However, this discovery also indicates that instructions with a mnemonic that is only slightly different, like the logical right shift operations `shr`, are not predicted inconsistently. From the experiments that reject the expansion to a mnemonic edit distance of 1, we can see that DiffTune gives different predictions for `shr` and `sar` instructions, in contrast to most other tools. This different treatment of similar instructions invites for a closer inspection, but it restricts AnICA's generalizations.

In summary, we observe that AnICA's generalization is very effective for the majority of considered tools. Where generalization is not as effective, the results are nevertheless insightful and point to concrete problems.

8.3. Case Studies

The previous section shows that AnICA is able to summarize thousands of inconsistencies between throughput predictors by a small number of abstract blocks. For DiffTune and Ithemal, it also presents first lessons learned from AnICA's results. We further investigated AnICA's discoveries and found several kinds of insights, for which we present examples in the following:

- AnICA uncovers different assumptions in the tools that can lead to dramatically different predictions. (Section 8.3.1)
- AnICA finds newly introduced regression bugs in subsequent versions of the same tool (Section 8.3.1) as well as long-existing bugs (Section 8.3.2).
- AnICA characterizes a variety of inaccuracies in llvm-mca's model for the AMD Zen+ microarchitecture, as well as an unusual quirk in the microarchitecture itself. (Section 8.3.3)

Instructions:
1. category: logical;
   memory: read+written

(a) Inconsistency between llvm-mca 13 & 9 and between llvm-mca 13 & DiffTune

Instructions:
1. memory: read+written;
   category: binary
2. memory: written

Aliasing:
- operand 1 of instruction 1 must alias with operand 1 of instruction 2

(b) Inconsistency between uiCA & Ithemal

Instructions:
1. memory: read+written;
   requires less than 8 μops

(c) Inconsistency between IACA & uiCA

Instructions:
1. mnemonic: `vpsubq` + 1 edit;
   memory: read
2. mnemonic: `fxrstor[64]`

(d) Crash in llvm-mca 12

Instructions:
1. mnemonic: `sar` + 0 edits

(e) Inconsistency between IACA & DiffTune

Figure 8.5. Abstract blocks causing inconsistent behavior found by AnICA. Feature abstractions are summarized for brevity. (Ritter and Hack, 2022)

8.3.1. Memory Dependencies

Data dependencies through memory operands are a challenge for basic block throughput predictors. If subsequent writes to and reads from memory refer to the same location, the write needs to be completed before the read.[22] If they access disjoint memory locations, they can execute independently. However, which of the cases applies may not be obvious or depend on the inputs. AnICA's results show that the tools handle these cases quite differently.

Table 8.4 shows how the throughput predictors handle memory dependencies on three example basic blocks: one with a guaranteed memory dependency (first line), one with independent instructions (second line), and one where the instructions may be independent, given suitable register values (third line). We see three plausible inverse throughput prediction results for executing such basic blocks in a loop:

- Two cycles, if there are no dependencies through memory and each instruction uses the processor's store unit for one cycle.
- Six cycles, if there is no memory dependency between the two instructions, but each instruction depends on its own result from the previous iteration. They then form two dependency chains with a latency of 6 cycles, which can be executed in parallel.
- Around 12 cycles, if all memory accesses depend on each other, forming a single large dependency chain.

[22] More specifically, the written value needs at least to be computed and put into a store buffer, from which it can be forwarded to subsequent reads.

Table 8.4. Predictions for the cycles required to execute basic blocks that differ in their memory dependencies.

Basic Block	uiCA	OSACA	IACA	llvm-mca 13	llvm-mca 13 alias	llvm-mca 9	DiffTune	Ithemal
`add [rcx+16],rbx; add [`**`rcx+16`**`],rbx`	12.0	12.0	2.0	2.1	14.0	14.0	14.1	5.9
`add [rcx+16],rbx; add [`**`rcx+128`**`],rbx`	6.0	6.0	2.0	2.1	14.0	14.0	14.1	5.9
`add [rcx+16],rbx; add [`**`rdx+16`**`],rbx`	6.0	6.0	2.0	2.1	14.0	14.0	14.1	6.0

llvm-mca in its default setting (LLVM, 2023b) and IACA[23] assume the first case. AnICA shows that for llvm-mca in the outdated version 9, this was not the case. It discovers, e.g., that the abstract block in Figure 8.5 (a) represents an inconsistency between llvm-mca versions 9 and 13. The older version was affected by a bug that led to predictions as if all memory accesses aliased. While we first discovered and reported this bug manually in the context of the PMEvo evaluation (Section 5.2), AnICA automatically finds this regression with a minimal example for reproducing the bug.

We also find that DiffTune's learned parameters for llvm-mca attempt to bypass llvm-mca's assumption that memory accesses do not alias: AnICA finds the same abstract block in Figure 8.5 (a) in the campaign for llvm-mca 13 and DiffTune. The example basic blocks for this discovery show that DiffTune also predicts throughputs as if all accesses were dependent on each other. Regarding this, Renda *et al.* (2020) remark in their evaluation that DiffTune learned a "degenerately high" latency for instructions that read and write memory from the same location. llvm-mca also provides an override switch to assume that all memory accesses alias, which leads to results similar to DiffTune's.

AnICA's discoveries like the one in Figure 8.5 (c) indicate that uiCA, OSACA, and Ithemal do not share IACA's assumption that no memory operations alias. These three tools recognize the data dependency formed by a single instruction that reads *and* writes with itself, therefore AnICA reports no discovery like those in Figure 8.5 (a) and 8.5 (c) between them. However, the abstract block displayed in Figure 8.5 (b) allows us to identify the difference in line 1 of Table 8.4: uiCA and OSACA rightly assume that the two memory locations alias if they are identical. AnICA provides the basic block shown in line 1 of the table for this abstract block.

For a user of these tools, this discrepancy can significantly affect the outcome: If the memory operands in the application alias in a non-obvious way, the observed cycles would exceed the results of uiCA and OSACA by a factor of two, and those of IACA and the default setting of llvm-mca by a factor of 6.

Since this inconsistency affects a large number of instruction schemes, corresponding discoveries were easy to find in AnICA's report with discoveries ranked by their generality (cf. Section 8.1.7).

[23] as previously noted by Abel and Reineke (2019)

8.3.2. FXRSTOR Crash in llvm-mca

To uncover crashes, AnICA can compare a single tool with itself. Figure 8.5 (d) shows an abstract block we found when investigating the llvm-mca version from LLVM release 12. This abstract block crashes the tool with an assertion, which AnICA always counts as interesting. `FXRSTOR` instructions, which restore the state of floating point control registers from memory, require the processor to execute a large number of µops. If one of the resources that it accesses is also used by a different instruction nearby, e.g., a vector subtraction with a memory operand, a bug in llvm-mca is triggered.

For LLVM release 13, this bug has been reported and fixed independently of our research.[24] The report includes a large input with more than 300 instructions to trigger the bug. AnICA automatically discovered the issue and provides a minimal input of just two instructions.

This discovery appears prominently in AnICA's results when the discoveries are ranked by their interestingness (cf. Section 8.1.7) since crashes in a tool under investigation are reported as maximally interesting.

8.3.3. Comparing llvm-mca to Measurements

AnICA has little requirements on the tools under investigation. They only need to produce an inverse throughput estimate for a given basic block. Consequently, we can also use AnICA to compare a throughput predictor to a tool that runs input basic blocks as microbenchmarks on the actual hardware.

In this case study, we apply AnICA to compare the predictions of llvm-mca for the AMD Zen/Zen+ microarchitecture to microbenchmarks performed with nanoBench (Abel and Reineke, 2020) on an AMD Ryzen 5 2600X processor. For these discoveries, we configured AnICA to consider an abstract block interesting if the absolute difference between measured and predicted inverse throughput of all of 50 sampled basic blocks is at least 2 cycles. This ensures that we can identify the more subtle inconsistencies in the results. We further restrict the instructions considered by AnICA for sampling and generalization: We exclude instruction schemes that read *and* write memory to avoid discovering more variants of the problem described in Section 8.3.1.

It is important to note that nanoBench is just another tool under investigation, not a ground truth. We configure nanoBench to group 10 instances of the measured instructions in a loop body that is executed 100 times while the passing processor cycles are measured with a hardware performance counter after 10 warm-up iterations. We use defaults for the remaining settings of nanoBench, which entails among other things that most registers are initialized with arbitrary values (except for those used as memory addresses). These assumptions affect the measured cycles, rendering the measurements unsuitable for use as definitive ground truth. The strength of AnICA's differential testing perspective is that neither tool needs to be assumed as "correct" to obtain interesting insights.

Table 8.5 shows the selected AnICA discoveries that we discuss in the following. We refer to the discoveries by the identifier in the first column. The second column contains the abstract block reported as discovery by AnICA. With the generalization decision tree and

[24] `https://llvm.org/bz50725`

Table 8.5. Abstract blocks capturing inconsistent behavior found by AnICA in llvm-mca/nanoBench campaigns on the AMD Zen+ microarchitecture. The descriptions are not generated by AnICA. The "Resulting Cycles" column displays the number of cycles predicted by llvm-mca (mca) and the cycles measured by nanoBench (nb) for the basic block in the preceding column.

	Abstract Block	Example Basic Block	Resulting Cycles	Simplified Description
(A)	Instructions: 1. optypes: $\{\texttt{DF}_R\}$	`cmpsq`	mca: 100 nb: 3.0	llvm-mca models complex instructions, certain shifts, and horizontal vector operations very pessimistically.
(B)	Instructions: 1. optypes: $\{\texttt{cl}_R\}$ isa-set: I386	`shld r11, rdx, cl`	mca: 100 nb: 3.0	
(C)	Instructions: 1. mnemonic: haddpd + 3 edits; category: SSE3	`hsubpd xmm15, xmm12`	mca: 100 nb: 6.5	
(D)	Instructions: 1. mnemonic: bsf + 1 edit optypes: $\{\langle\texttt{GPR[64]}\rangle_W\}$ Aliasing: • operand 1 of instruction 1 must not alias with operand 2 of instruction 1	`bsr rcx, r11`	mca: 0.3 nb: 4.0	LLVM's model for bit-scan instructions implies a wrong throughput.
(E)	Instructions: 1. mnemonic: add + 2 edits optypes: $\{\langle\texttt{GPR[64]}\rangle_{RW}\}$ memory: DefNone 2. mnemonic: and + 3 edits optypes: $\{\langle\texttt{GPR[64]}\rangle_{RW}\}$ memory: read Aliasing: • operand 1 of instruction 1 must alias with operand 1 of instruction 2	`add r8, 0x2a` `adc r8, qword ptr [r14]`	mca: 5.03 nb: 2.0	llvm-mca misses that memory loads can start before other operands are available.
(F)	Instructions: 1. mnemonic: shl + 2 edits optypes: {0x0} isa-set: I186 2. optypes: $\{\texttt{OF}_R\}$	`shl r9w, 0x0` `setno r11b`	mca: 1.04 nb: 25.7	Reading flags after shifting by 0 incurs a penalty on Zen+.

the corresponding evaluated basic blocks, AnICA provides more additional information than we can present here. We therefore instead only show one example basic block sampled from the abstract block in the third column and the results of nanoBench (nb) and llvm-mca (mca) for it in the fourth column. In the last column, we annotate a short summary of the problem characterized by the discovery.

Microcoded Instructions When ranking AnICA discoveries by their interestingness (cf. Section 8.1.7), the ones that stand out the most are those concerning instructions that llvm-mca predicts to require 100 cycles to execute. This mainly affects microcoded instructions, e.g., the string operations, which commonly read the direction flag register `DF`, summarized by discovery (A). When the processor's instruction decoder encounters such instructions, it produces a (potentially large and/or varying) number of μops that need to be executed by the processor's functional units. LLVM's Zen+ scheduling model (and consequently llvm-mca) handles most such instructions in a coarse way that just assigns them a latency of 100 cycles.

However, this strategy is also used for more unexpected instructions like certain bit shifts (discovery (B)) and horizontal vector operations (discovery (C)). While these modeling decisions are not per se bugs, they can make the Zen+ model of llvm-mca effectively unusable for any task that uses such instructions. LLVM's issue tracker contains a report for this behavior that has been submitted independently of our work.[25]

Bit-Scan Instructions Discovery (D) shows an apparent bug in LLVM's Zen+ scheduling model for bit-scan instructions.[26] The measurements with nanoBench, as well as the instruction latency table provided by AMD (2021b), show that a `BSR` instruction has a latency of 4 cycles and requires 4 cycles to be executed in a steady state (an execution rate of 0.25 instructions per cycle).

The aliasing component of AnICA's abstract block (D) shows that llvm-mca predicts the latency consistently with measurements: In similar basic blocks that violate the must-not-alias constraint of abstract block (D), an instance of the bit-scan instruction can only be executed once the previous instance produced its result, the throughput is therefore determined by the instruction's latency. As the must-not-alias constraint could not be dropped during generalization, llvm-mca models this case – and therefore the instruction's latency – consistently.

Without operand aliasing, llvm-mca underestimates the required execution time. The scheduling model of LLVM (2021, line 235) provides an explanation: The affected instructions are represented with a plausible latency, but they are modeled to block only one of the architecture's four arithmetic logic units. Therefore, llvm-mca assumes that up to four independent instances of bit-scan instructions can be executed per cycle. We reported this and four other results from similar AnICA discoveries to the LLVM developers. These reports

[25] `https://github.com/llvm/llvm-project/issues/53242`

[26] The bit-scan instructions `BSF`/`BSR` determine the index of the least/most significant bit set in their second operand and write it to the first operand.

show errors in LLVM's Zen+ scheduling model for a total of 72 of our instruction schemes. The bugs were confirmed and fixed by the developers.[27]

Inaccuracies in Load Operand Usage With discovery (E), we learn that llvm-mca overestimates the time required to execute instructions that depend on the result of a preceding instruction and load from memory. The hardware is evidently able to issue a new instruction in every cycle for the corresponding example basic block; the load latency (4 cycles for L1 cache hits on Zen+) completely overlaps with the remaining computation. llvm-mca's model does not account for this behavior: Here, the loading instruction always starts executing with the instruction it depends on, causing the load latency to be visible. This problem has also been independently reported in the LLVM issue tracker.[28]

A Microarchitectural Quirk AnICA's results not only highlight oddities in prediction tools, they can also show unusual behavior in the processor under test. The discovery (F) shows how AnICA automatically found a microarchitectural quirk of the Zen architectures that has been previously described by Abel (2020). Bit shifts by zero cause a severe execution time penalty if they are followed by instructions that read the flag registers. Such no-op shifts are a special case in the x86-64 ISA as only for a shift amount of zero the flag registers are not updated.[29] llvm-mca's model omits the unexpected handling of this corner case in AMD's Zen architectures.

In all of the above cases, AnICA automatically discovered an unexpected inconsistency and provided helpful insight with its generalization. Such insights could otherwise only be gained through tedious manual effort. The fact that we, additionally to finding new bugs, automatically rediscovered several previously reported problems in the llvm-mca predictions indicates that AnICA finds problems that are relevant to the users of llvm-mca.

8.4. Related Work

To the best of our knowledge, AnICA is the first work to apply differential testing to microarchitectural code analyzers. This section describes other approaches to evaluate such tools and contrasts AnICA to previous work in differential testing.

8.4.1. Testing Throughput Predictors

Most of the available basic block throughput predictors come with an evaluation of their prediction accuracy. A common approach to evaluate throughput predictors is to measure the relative error from and the correlation with execution time measurements on a chosen set of basic blocks. This is done for OSACA (Laukemann *et al.*, 2018), Ithemal (Mendis *et al.*, 2019), DiffTune (Renda *et al.*, 2020), and uiCA (Abel and Reineke, 2022). They all use

[27] `https://github.com/llvm/llvm-project/issues/54811` and `https://github.com/llvm/llvm-project/issues/54889`

[28] `https://github.com/llvm/llvm-project/issues/50899`

[29] See, e.g., `https://www.felixcloutier.com/x86/sal:sar:shl:shr`.

basic blocks that were extracted from the binaries of common benchmarks and open source programs whose throughput was measured using various methodologies. Of particular note is BHive (Chen *et al.*, 2019), which is used in the evaluations of DiffTune and uiCA. It is an openly available set of such basic blocks with annotated measured inverse throughputs for several Intel microarchitectures. The evaluation of uiCA identifies cases where assumptions made for the ground truth measurements affect which tool is "more accurate" than another, motivating our differential approach.

Evaluating the prediction accuracy on basic blocks from compiled programs is helpful when the expected use of the tools is on similar basic blocks. However, such basic blocks are lacking when we want to explore inconsistencies of the tools systematically: Of the 2940 instruction schemes that we use in our evaluation, 2002 (i.e., 68%) do not occur in any basic block of the BHive data set and 525 (i.e., 18%) of the instruction schemes are enough to represent 99% of the BHive basic blocks. The BHive benchmarks therefore leave a gap in the input space when testing throughput predictors that AnICA addresses.

BHive includes an approach to help developers identify problems with their throughput predictors. They cluster basic blocks from the data set based on their use of execution units in the processor (e.g., "vectorized code" and "code with mainly memory operations"). If a tool performs particularly poor on a cluster of basic blocks, the developers can focus on improving support for the associated category. These categories are, however, considerably less specific than the inconsistencies that AnICA reports to the user.

Abel (2022) investigated the prediction accuracy of DiffTune, providing a very simple set of parameters for llvm-mca that outperform the learned DiffTune parameters on the BHive data set in terms of prediction accuracy. These findings are consistent with the unexpected predictions we encountered in our DiffTune campaigns (Section 8.2.3).

EXEgesis (Chatelet, 2018; Google, 2018; LLVM, 2023a) is a project to validate LLVM's performance models and, consequently, llvm-mca. For a given instruction scheme, EXEgesis executes a microbenchmark on the target machine and measures its performance characteristics. EXEgesis can compare the measured performance to the corresponding information in LLVM's scheduling model. In contrast to AnICA, EXEgesis does not generate experiments with multiple instructions to test their interactions and it is closely integrated with LLVM. Comparisons with other predictors are therefore not supported.

Approaches that infer models for components of throughput predictors are also evaluated against existing ones on a measured ground truth. Our previously published approach for inferring port mappings, PMEvo (Ritter and Hack, 2020, cf. Chapter 5 of this thesis), and Palmed (Derumigny *et al.*, 2022a) both use basic blocks without data dependencies, whose throughput is bound by the processor's functional units. For Palmed, the basic blocks mirror basic blocks observed in the binaries of benchmark suites (without the dependencies). The basic blocks we used for PMEvo are more similar to the ones we use here: They are randomly sampled in a way that avoids data dependencies. The evaluation of uops.info (Abel and Reineke, 2019) points out some inconsistencies in IACA, but focuses on the usage of resources for single instructions.

COMET (Chaudhary *et al.*, 2023), which was published after our article on AnICA, explains the predictions of a basic block throughput predictor with a technique similar to AnICA's.

Given a basic block b, COMET searches a set of features of b that determines the predicted throughput, i.e., changing any other feature of b does not affect the throughput prediction significantly. A key component of the approach is the random generation of basic blocks that differ from b in a given set F of features while the remaining features are preserved. In AnICA's framework, this operation corresponds to setting the components of the abstract block $\beta(b)$ that correspond to features in F to $\top$ and sampling from the resulting abstract block.

COMET considers the number of instructions of the basic block, their opcodes, and the dependencies between the instructions as basic block features. This feature selection is less expressive than the abstract domain we describe in Section 8.1.4 for AnICA: It loosely corresponds to replacing the instruction abstraction by a single *singleton* component for the mnemonic/opcode.

AnICA's basic block generalization algorithm could be applied to COMET's setting with an interestingness metric that considers basic blocks interesting if their predicted throughput is (almost) equal to the prediction for b. When we start the generalization algorithm from b with such an interestingness metric, the non-$\top$ components of the resulting abstract block correspond to features of b that determine the throughput prediction.

COMET applies a general algorithm for explaining model predictions, the Anchors algorithm (Ribeiro *et al.*, 2018). On a high level, this algorithm operates dually to our generalization algorithm:

- AnICA's generalization starts with a pertinent characterization of an interesting input, low in the generality order, and goes "up" in the order as long as the result remains pertinent. For each step "up", we use random sampling to find a feature to "expand" while preserving pertinence.

- The Anchors algorithm starts with a most general characterization, maximal in the generality order, and narrows it "down" in the order until a pertinent characterization is found. For each step "down", they follow a sample-based algorithm to identify which feature should be "shrunk" to increase pertinence.

A future line of research could be to investigate if AnICA's perspective on generalizing interesting inputs offers new insights for this setting of explaining models.

8.4.2. Differential Testing

Differential testing (McKeeman, 1998) is commonly used to find bugs in tools where no ground truth is available. There are general frameworks for differential testing tools like Nezha (Petsios *et al.*, 2017), but they mainly focus on effectively exploring a sparse space of inconsistencies. As the space of inconsistencies among basic block throughput predictors is not sparse, there is little benefit in using these frameworks.

AnICA's use of minimization and abstraction can be seen as a form of triage in the usual nomenclature (Manès *et al.*, 2021). The concepts and notations borrowed from abstract interpretation give us a way to systematically implement a generalized deduplication of inputs.

Previous research already used differential testing for other tools operating on machine code. Several differential fuzzing approaches (Jay and Miller, 2018; Paleari *et al.*, 2010; Woodruff *et al.*, 2021) focus on instruction decoders. However, these works differ from our setting in their goal and, consequently, in the inputs that they generate. They generate bit sequences that are (or are close to) machine instructions, for which they check the results of a group of instruction decoders. Since we aim to find inconsistencies in the throughput predictions, we only produce valid instruction sequences. Woodruff *et al.* (2021) note that they encounter large numbers of nearly identical discoveries that are difficult to deduplicate, making human analysis essential. This mirrors our motivation to use abstraction to reduce the manual investigation effort for analyzing the discoveries.

Revizor (Oleksenko *et al.*, 2021) is a differential testing approach that also generates random instruction sequences. They compare a CPU's behavior with that of a simulation that does not leak information to find side channel attacks. In contrast to AnICA, their instruction sequences include control flow. They define a number of patterns on the dependencies between consecutive instructions that are similar to the constraints represented by our aliasing abstraction. However, Revizor uses these patterns only as a metric to control the size of the instruction sequences that they sample. Since abstraction is central to AnICA, we designed the basic block abstraction to cover more complex alias constraints as well as constraints on the involved instructions, which are beyond the scope of Revizor's patterns.

8.5. Possible Extensions

A strength of AnICA is that the throughput predictors under investigation are treated as black boxes. The resulting flexibility opens a range of further use cases for AnICA with no or minor adjustments to the implementation.

Comparing Different Benchmarking Assumptions

When benchmarking the execution time of basic blocks, tools like nanoBench (Abel and Reineke, 2020) have to make assumptions on how the blocks should be executed. For instance, they need to initialize registers and memory regions with specific values and choose whether basic blocks should be wrapped in a loop or concatenated sufficiently often. If these choices are configurable (as with nanoBench), AnICA can investigate the effect of different configuration decisions on the measurements. From discovery (F) in our llvm-mca case study (Section 8.3.3), we would, e.g., expect to find inconsistencies depending on whether the registers are initialized with 0 or not.

Comparing Measurements on Different Microarchitectures

We have presented results for comparing pairs of throughput predictor tools (Section 8.2.3) as well as for comparing a throughput predictor to measurements on the modeled hardware (Section 8.3.3). A natural next step would be to compare measurements on two different hardware implementations of an instruction set architecture to each other. This would allow

us to investigate performance differences of subsequent generations of CPUs by the same manufacturer, or different trade-offs made by two manufacturers in competing CPU models.

Comparing Port Usage Models

The AnICA algorithm can also be applied to subcomponents of performance models that affect only individual aspects of basic block throughput prediction. For instance, approaches like uops.info (Abel and Reineke, 2019), our work PMEvo (cf. Chapter 5), and Palmed (Derumigny *et al.*, 2022a) build models for how individual instructions use a CPU's execution resources. These models are able to predict the throughput of basic blocks without data dependencies.

AnICA could therefore investigate differences between the models produced by the individual approaches, as well as deviations between a model and measurements on the actual hardware. For this application domain, the presented basic block abstraction should be adjusted such that only basic blocks with as few data dependencies as possible are sampled. Consequently, the aliasing component of the basic block abstraction then does not capture meaningful information anymore and may be dropped.

Since these approaches infer their models from microbenchmarks, the results of AnICA may be helpful to improve the models by characterizing classes of benchmarks that are missing.

8.6. Conclusions: AnICA

State-of-the-art tools for basic block throughput prediction often do not agree in their results, for a variety of reasons. With AnICA, we apply differential testing to understand these tools and to identify ways of improving them. By borrowing notions from abstract interpretation, we can draw from a pool of well-established techniques and formalisms to generalize inconsistencies in a systematic way. Our evaluation shows that AnICA can summarize thousands of inconsistencies in a few dozen descriptions that directly lead to high-level insights into the different behavior of the tools.

Chapter 9

Conclusions and Outlook

Performance models for processors are an important prerequisite for building software that uses the available hardware to its fullest potential. Accurate performance models are however difficult to construct because of the complexity of modern processors and lacking documentation. In this dissertation, we have presented theoretical and practical advancements in two facets of the CPU performance modeling field. First, we have introduced three novel methods for inferring the port mapping of an out-of-order processor from throughput measurements. The port mapping describes the processor's ability to exploit instruction-level parallelism as it models how instructions are decomposed into µops and how these are executed on the processor's functional units. Each of our port mapping inference methods implements a different trade-off between applicability and accuracy.

Our first, counter-example-guided port mapping inference algorithm finds a port mapping that throughput measurements cannot distinguish from the processor's actual port mapping. It automatically constructs instruction sequences for benchmarking that are sufficient to ascertain such an indistinguishable port mapping based on a formal port mapping model. The algorithm only requires that we can measure the throughput of instruction sequences as determined by the processor's port mapping without other interfering bottlenecks.

The strong guarantees of the algorithm come at the cost of practical running time. As it relies on a satisfiability-modulo-theories solver to construct experiments for throughput measurements, the counter-example-guided inference algorithm does not scale to practical problem sizes. With realistic port numbers, inference problems for mappings with more than ten instructions can take days to solve, whereas modern instruction set architectures contain hundreds or thousands of instruction variants.

The second port mapping inference method, PMEvo, enables throughput-based port mapping inference for practical problem sizes. We achieve this through approximations. Instead of dynamically constructing a complete set of throughput benchmarks that characterize the processor's port mapping, PMEvo follows a fixed benchmarking strategy to determine the throughput of each individual instruction and of each pair of instructions. PMEvo then uses a randomized evolutionary algorithm that searches the space of possible port mappings for one that explains the observed throughputs.

We applied PMEvo to infer port mappings for microarchitectures by Intel, AMD, and ARM. The designs of the latter two manufacturers are not supported by previous performance-counter-based inference approaches. The inferred port mappings model the observed throughput accurately, but their structure rarely follows the available documentation of the

processors since throughput measurements for PMEvo's benchmarks do not fully characterize the processor's port mapping. To the best of our knowledge, PMEvo remains the only available and practical approach to automatically infer a three-level port mapping solely based on throughput measurements.

Our third port mapping inference method finds port mappings that model the observed throughput accurately and whose structure agrees with available documentation. The algorithm follows the high-level structure of a performance-counter-based port mapping inference approach from prior work, but eliminates most uses of hardware performance counters. Instead of performance counters for the executed µops at each of the processor's ports, our algorithm only needs a single hardware performance counter for the total number of executed µops in a benchmark. While this imposes a stronger requirement on the hardware than our previous two inference methods, it still enables port mapping inference for AMD processors that the previous performance-counter-based approaches do not support.

We demonstrate this port mapping inference method with a case study on AMD's Zen+ microarchitecture. While practical hindrances like throughput bottlenecks and complex microcode prohibit accurate port mappings for some instructions, a large portion of the instruction set architecture is covered by this approach. In contrast to PMEvo, the resulting port mapping is in most cases consistent with available documentation and comes with specific benchmarks to show where the documentation does not agree with the observable behavior for the other cases. This study uncovers previously undocumented details of the Zen+ microarchitecture and results in the first explainable port mapping for over 1,000 instruction schemes on Zen+ that were out of scope for previous performance-counter-based approaches. In comparison to the state of the art in throughput-based port mapping inference – including PMEvo – the resulting port mapping achieves superior throughput prediction accuracy.

The second facet of CPU performance modeling that this thesis addresses is the analysis of inconsistencies between basic block throughput predictors. There is a variety of tools that share the goal of predicting the throughput a processor achieves for short instruction sequences. However, these approaches often disagree in their predictions. With AnICA, we propose an approach to test basic block throughput predictors differentially against other throughput predictors and measurements on the modeled hardware. AnICA automatically discovers inconsistencies in throughput estimates and derives compact characterizations of classes of inconsistencies with an abstract-interpretation-based generalization algorithm.

We demonstrate that AnICA can summarize thousands of inconsistencies with a few dozen descriptions that provide high-level insights into the behavior of the investigated tools. In a series of case studies, AnICA exposes subtle modeling differences between the tools, identifies underrepresented constructs in the training sets of learned predictors, and pinpoints a long-standing crash in llvm-mca with a two-instruction test case. We further show that AnICA automatically finds and characterizes inaccuracies in llvm-mca's model for AMD's Zen+ microarchitecture. Our reports of these inaccuracies lead to improvements in LLVM's upstream scheduling models for Zen+.

Future Directions

For inferring port mappings as well as for analyzing the results of throughput predictors, our practical contributions are founded on underlying theoretical frameworks. These frameworks open up interesting directions for future research.

On the port mapping side, the formal port mapping model and the SMT formulation that we derived from it can be used for other purposes than port mapping inference. One idea is to automatically check if – and under which conditions – strategies to construct microbenchmarks are sufficient to identify a processor's port mapping. Early experimentation with this idea lead us to an error in the paper version of the uops.info algorithm[1] and to the insight that underlies our explainable port mapping inference method: The benchmarks of the uops.info algorithm are sufficient to determine the port mapping even if per-port performance counters are not available.

The framework that underlies AnICA is not restricted to analyzing basic block throughput predictors. At its core is a method for identifying and systematically generalizing interesting structures. Applying this method in other domains may yield interesting results. The AnICA algorithm with a different abstract domain could for instance be used to explain machine learning models by characterizing classes of inputs that lead to interesting predictions of black-box models.

On a broader scale, the port mapping inference field could benefit from a cooperation with hardware designers. Our complex throughput-based inference methods are only necessary because processors do not provide the performance counters required for the prior performance-counter-based inference methods. Integrating such performance counters into more microarchitectures would make inferring performance models considerably easier. A further step would be to automatically derive open performance models directly from the hardware designs instead of relying on delicate microbenchmarks.

Lastly, our case studies with AnICA have shown that differing implicit assumptions are a major cause for inconsistencies between basic block throughput predictors. The tools choose to assume, e.g., that different memory operands do or do not alias. As the actual behavior depends on the context in which the basic block is executed, throughput predictors for individual basic blocks cannot model this accurately. Users of these tools may however know which behaviors the throughput predictor should assume for their case, e.g., based on profiling information. For such cases, a performance predictor that makes all assumptions explicit and user-configurable could advance the state of the art in basic block throughput prediction considerably.

[1] See Appendix E.1.

Appendix

Appendix A

Proofs

A.1. Proofs for Chapter 3

A.1.1. Proof for Theorem 3.6

Theorem 3.6. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment e with a port mapping M always exists, and it is the infimum of the set $\left\{\frac{T(M,e,N)}{N} \;\middle|\; N \in \mathbb{N}^+\right\}$.

For Context:

Definition 3.2. A *port mapping M in the two-level model* is a bipartite graph $(\mathbf{I} \cup \mathbf{P}, E)$ with the vertices split disjointly into finite, non-empty sets $\mathbf{I}$ of instruction schemes and $\mathbf{P}$ of ports. The edges $E \subseteq \mathbf{I} \times \mathbf{P}$ connect instruction schemes with their ports.

We refer to the ports assigned to an instruction scheme i by $M[i] := \{k \mid (i,k) \in E\}$. We require that for all $i \in \mathbf{I}$, $M[i] \neq \emptyset$, i.e., every instruction scheme has at least one port that can execute it. ⌟

Definition 3.3. An *execution schedule* for a positive number $N \in \mathbb{N}^+$ of iterations of an experiment e according to the two-level port mapping $M = (\mathbf{I} \cup \mathbf{P}, E)$ is a sequence $Ex := [s_1, \ldots, s_{|Ex|}]$ of partial allocation functions $s_c : \mathbf{P} \rightharpoonup \mathbf{I}$ such that

- all instruction scheme occurrences from each experiment iteration are assigned to a port at some point in the sequence:

$$\forall i \in \mathbf{I}.\ \left|\left\{(c,k) \;\middle|\; 1 \leq c \leq |Ex| \wedge k \in \mathbf{P} \wedge s_c(k) = i\right\}\right| = N \cdot e(i)$$

- instruction schemes are only assigned to ports that M allows:

$$\forall 1 \leq c \leq |Ex|, i \in \mathbf{I}, k \in \mathbf{P}.\ \left(s_c(k) = i\right) \Rightarrow k \in M[i]$$

We call the length $|Ex|$ of an execution schedule Ex its *execution time*. An *optimal execution schedule* of N iterations of an experiment e according to the port mapping M is an execution schedule with a minimal execution time. We refer to the execution time of such an optimal execution schedule as $T(M, e, N)$. ⌟

Definition 3.5. The *modeled inverse throughput* $tp_M^{-1}(e)$ of an experiment e with a two-level port mapping M is the limit

$$\lim_{N \to \infty} \frac{T(M, e, N)}{N}$$

⌟

As a first observation, we note that there are valid execution schedules for any port mapping M and any number N of iterations of any experiment e: By Definition 3.2, every instruction $i \in \mathbf{I}$ has at least one port $p(i) \in M[i]$ that can execute it. Therefore, we can construct an execution schedule for one instance of the instructions $i_1, \ldots, i_{|e|}$ in the instruction multiset e as follows:

$$Ex = \left[\left\{i_1 \mapsto p(i_1)\right\}, \ldots, \left\{i_{|e|} \mapsto p(i_{|e|})\right\}\right]$$

This schedule executes each instruction in an individual cycle on a port allowed by the port mapping. By concatenating this schedule N times, we obtain a schedule for N iterations of e that satisfies the constraints of Definition 3.3. The set of execution schedules for N iterations of e with the port mapping M is therefore non-empty. Since the execution time of a schedule is defined as the length of a list, schedule execution times can only be non-negative integers. $T(M, e, N)$ as the smallest such integer is therefore well-defined.

The set $\mathcal{T} := \left\{ \frac{T(M,e,N)}{N} \;\middle|\; N \in \mathbb{N}^+ \right\}$ is thus clearly a non-empty set of real numbers. It is bounded from below by 0, therefore the Dedekind completeness of the real numbers ensures that it has an infimum b in the real numbers. This infimum b can either be part of $\mathcal{T}$, or for every $\varepsilon > 0$, there is an element b' of $\mathcal{T}$ such that $|b' - b| < \varepsilon$. (Observation (A))[1]

For every $k \in \mathbb{N}^+$, we can write any integer $N \geq k$ as $N = k \cdot J + l$ with integers J and l such that $J := \left\lfloor \frac{N}{k} \right\rfloor \cdot k > 0$ and $0 \leq l < k$. We observe:

$$T(M, e, k \cdot J + l) \leq J \cdot T(M, e, k) + l \cdot T(M, e, 1)$$

This follows from the definition of execution schedules: For a valid (but not necessarily optimal) schedule for $k \cdot J + l$ iterations, we can take an optimal schedule for k iterations, concatenate it J times, and append an optimal schedule for a single iteration l times.

[1] Theorem 3.7 and the corresponding proof in Appendix A.1.2 imply that b is indeed contained in $\mathcal{T}$.

This implies the following relationship:

$$\begin{aligned}\frac{T(M,e,k\cdot J+l)}{k\cdot J+l} &\leq \frac{J\cdot T(M,e,k)+l\cdot T(M,e,1)}{k\cdot J+l}\\ &\leq \frac{J\cdot T(M,e,k)+k\cdot T(M,e,1)}{k\cdot J} && \text{since } 0\leq l<k\\ &= \frac{T(M,e,k)}{k}+\frac{T(M,e,1)}{J}\end{aligned}$$

For a fixed integer k and any integer $N \geq k$, we therefore have:

$$\begin{aligned}\frac{T(M,e,N)}{N} &= \frac{T(M,e,k\cdot J+l)}{k\cdot J+l}\\ &\leq \frac{T(M,e,k)}{k}+\frac{T(M,e,1)}{J}=\frac{T(M,e,k)}{k}+\frac{T(M,e,1)}{\left\lfloor\frac{N}{k}\right\rfloor\cdot k} && \text{(Observation (B))}\end{aligned}$$

We now show that the sequence $\left(\frac{T(M,e,N)}{N}\right)_{N=1}^{\infty}$ converges to the infimum b based on the above observations. Let $\varepsilon > 0$. By Observation (A), there is an element b' of the sequence such that $|b'-b| < \frac{\varepsilon}{2}$ (b may be equal to b'). Since b' is an element of the sequence, there is some k' such that $b' = \frac{T(M,e,k')}{k'}$. From Observation (B), we know that the sequence $\left(\frac{T(M,e,N)}{N}\right)_{N=k'}^{\infty}$ is bounded from above by the following sequence:

$$(B_N)_{N=k'}^{\infty} := \left(\frac{T(M,e,k')}{k'}+\frac{T(M,e,1)}{\left\lfloor\frac{N}{k'}\right\rfloor\cdot k'}\right)_{N=k'}^{\infty}$$

This bounding sequence converges to $\frac{T(M,e,k')}{k'} = b'$ for $N \to \infty$ since the right summand vanishes for sufficiently large values of N. Hence, there is an N' such that all subsequent entries of the bounding sequence have a distance less than $\frac{\varepsilon}{2}$ from b'. Without loss of generality, $N' \geq k'$.

Therefore, the following holds for all $N'' \geq N'$:

$$\begin{aligned}&|B_{N''}-b'|<\frac{\varepsilon}{2} \quad\text{and}\quad |b'-b|<\frac{\varepsilon}{2}\\ \Rightarrow\ &|B_{N''}-b'|+|b'-b|<\frac{\varepsilon}{2}+\frac{\varepsilon}{2}\\ \Rightarrow\ &|B_{N''}-b'+b'-b|<\frac{\varepsilon}{2}+\frac{\varepsilon}{2} && \text{by the triangle inequality}\\ \Rightarrow\ &|B_{N''}-b|<\varepsilon\end{aligned}$$

As $N'' \geq k'$, we further know that the following holds:

$$\begin{aligned}
&\frac{T(M,e,N'')}{N''} \leq B_{N''} && \text{by Observation (B)}\\
\Rightarrow &\frac{T(M,e,N'')}{N''} - b \leq B_{N''} - b\\
\Rightarrow &\left|\frac{T(M,e,N'')}{N''} - b\right| \leq |B_{N''} - b| && \text{since } \frac{T(M,e,N'')}{N''} - b \geq 0\\
\Rightarrow &\left|\frac{T(M,e,N'')}{N''} - b\right| < \varepsilon
\end{aligned}$$

Consequently, for every $\varepsilon > 0$, there is a number $N' \in \mathbb{N}$ such that for all integers $N'' \geq N'$, the value of the sequence $\left(\frac{T(M,e,N)}{N}\right)_{N=1}^{\infty}$ at position N'' differs by less than ε from b. Thus, the sequence converges to b. □

A.1.2. Proof for Theorem 3.7

Theorem 3.7. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment $e : I \to \mathbb{N}$ with a port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ is the objective value of an optimal solution to the following linear program:

$$
\begin{array}{lll}
\text{minimize} & t & \\
\text{subject to} & \displaystyle\sum_{k \in \mathbf{P}} x_{ik} = e(i) & \text{for all instructions } i \in \mathbf{I} \quad \text{(A)} \\
& \displaystyle\sum_{i \in \mathbf{I}} x_{ik} = p_k & \text{for all ports } k \in \mathbf{P} \quad \text{(B)} \\
& p_k \leq t & \text{for all ports } k \in \mathbf{P} \quad \text{(C)} \\
& x_{ik} \geq 0 & \text{for all instructions } i \in \mathbf{I}, \text{ports } k \in \mathbf{P} \quad \text{(D)} \\
& x_{ik} = 0 & \text{if } (i,k) \notin E \quad \text{(E)}
\end{array}
$$

In particular, this linear program is feasible and has a finite optimal objective value.

For Context:

Definition 3.3. An *execution schedule* for a positive number $N \in \mathbb{N}^+$ of iterations of an experiment e according to the two-level port mapping $M = (\mathbf{I} \cup \mathbf{P}, E)$ is a sequence $Ex := [s_1, \ldots, s_{|Ex|}]$ of partial allocation functions $s_c : \mathbf{P} \rightharpoonup \mathbf{I}$ such that

- all instruction scheme occurrences from each experiment iteration are assigned to a port at some point in the sequence:

$$\forall i \in \mathbf{I}. \left| \left\{ (c,k) \;\middle|\; 1 \leq c \leq |Ex| \wedge k \in \mathbf{P} \wedge s_c(k) = i \right\} \right| = N \cdot e(i)$$

- instruction schemes are only assigned to ports that M allows:

$$\forall 1 \leq c \leq |Ex|, i \in \mathbf{I}, k \in \mathbf{P}. \left(s_c(k) = i \right) \Rightarrow k \in M[i]$$

We call the length $|Ex|$ of an execution schedule Ex its *execution time.* An *optimal execution schedule* of N iterations of an experiment e according to the port mapping M is an execution schedule with a minimal execution time. We refer to the execution time of such an optimal execution schedule as $T(M, e, N)$. ⌟

Theorem 3.6. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment e with a port mapping M always exists, and it is the infimum of the set $\left\{ \frac{T(M,e,N)}{N} \;\middle|\; N \in \mathbb{N}^+ \right\}$.

We proceed by showing that each execution schedule implies a feasible LP solution and that each optimal LP solution implies a valid execution schedule. Lastly, we argue that any execution schedule that requires fewer cycles per iteration than one implied by an optimal LP solution would imply an LP solution with a better-than-optimal objective value. Optimal LP solutions therefore must correspond to execution schedules that use the minimal number of cycles per iteration.

An execution schedule for any N implies a feasible solution to the LP: Let $Ex = [s_1, \ldots, s_{|Ex|}]$ be an execution schedule for $N \in \mathbb{N}^+$ iterations of an experiment e according to the port mapping M. We define a solution S for the LP as follows:

$$S[x_{ik}] := \frac{\left|\left\{c \mid 1 \leq c \leq |Ex| \wedge s_c(k) = i\right\}\right|}{N} \tag{A.1}$$

$$S[p_k] := \sum_{i \in \mathbf{I}} S[x_{ik}] \tag{A.2}$$

$$S[t] := \frac{|Ex|}{N} \tag{A.3}$$

This solution is feasible, i.e., it fulfills all constraints of the LP:

- Constraint (A) holds for any $i \in \mathbf{I}$:

$$\begin{aligned}
\sum_{k \in \mathbf{P}} S[x_{ik}] &= \sum_{k \in \mathbf{P}} \frac{\left|\left\{c \mid 1 \leq c \leq |Ex| \wedge s_c(k) = i\right\}\right|}{N} && \text{definition of } S[x_{ik}] \\
&= \frac{1}{N} \cdot \sum_{k \in \mathbf{P}} \left|\left\{c \mid 1 \leq c \leq |Ex| \wedge s_c(k) = i\right\}\right| \\
&= \frac{1}{N} \cdot \sum_{k \in \mathbf{P}} \left|\left\{(c,k) \mid 1 \leq c \leq |Ex| \wedge s_c(k) = i\right\}\right| \\
&= \frac{1}{N} \cdot \left|\bigcup_{k \in \mathbf{P}} \left\{(c,k) \mid 1 \leq c \leq |Ex| \wedge s_c(k) = i\right\}\right| && \text{size of disjoint union} \\
&= \frac{1}{N} \cdot \left|\left\{(c,k) \mid 1 \leq c \leq |Ex| \wedge k \in \mathbf{P} \wedge s_c(k) = i\right\}\right| \\
&= \frac{1}{N} \cdot (N \cdot e(i)) = e(i) && \text{first constraint of Definition 3.3}
\end{aligned}$$

- That Constraint (B) holds follows immediately from the definition of $S[p_k]$.

- Constraint (C) holds for any $k \in \mathbf{P}$:

$$\begin{aligned}
S[p_k] &= \sum_{i \in \mathbf{I}} S[x_{ik}] && \text{definition of } S[p_k] \\
&= \sum_{i \in \mathbf{I}} \frac{\left|\left\{c \;\middle|\; 1 \le c \le |Ex| \wedge s_c(k) = i\right\}\right|}{N} && \text{definition of } S[x_{ik}] \\
&= \frac{1}{N} \cdot \sum_{i \in \mathbf{I}} \left|\left\{c \;\middle|\; 1 \le c \le |Ex| \wedge s_c(k) = i\right\}\right| \\
&= \frac{1}{N} \cdot \left|\left\{c \;\middle|\; 1 \le c \le |Ex| \wedge s_c(k) \in \mathbf{I}\right\}\right| && (*) \\
&\le \frac{1}{N} \cdot \left|\left\{c \;\middle|\; 1 \le c \le |Ex|\right\}\right| \\
&= \frac{|Ex|}{N} \\
&= S[t] && \text{definition of } S[t]
\end{aligned}$$

  For the step marked with $(*)$, we observe that the term before the transformation counts the number of corresponding entries for port k in Ex for each instruction i and sums the results. This is equivalent to counting the total number of defined entries for port k in Ex for any instruction.

- Constraint (D) holds, since the $S[x_{ik}]$ are defined as the quotient of a non-negative number and a positive number, which is non-negative.

- Constraint (E) holds, as $\left\{c \;\middle|\; s_c \in Ex \wedge s_c(k) = i\right\}$ can only contain any entry if $s_c(k) = i$ for some s_c in Ex. By the second constraint of Definition 3.3, $s_c(k) = i$ implies that $k \in M[i]$.

Therefore, any valid (but not necessarily optimal) execution schedule Ex for $N \in \mathbb{N}^+$ iterations of an experiment implies a feasible (but not necessarily optimal) solution to the corresponding LP, with an objective value of $\frac{|Ex|}{N}$. (Observation (A))

The LP is feasible and has a finite optimal objective value: As Theorem 3.6 implies that execution schedules exist for every port mapping M and experiment e, feasible solutions to the LP also exist by Observation (A).

By Constraint (C), any variable p_k – there is at least one since $\mathbf{P}$ may not be empty by definition – is a lower bound to the objective term t. By Constraint (B), the value of p_k is the sum of the values of x_{ik} variables – there is at least one since $\mathbf{I}$ is required to be non-empty. The x_{ik} variables are non-negative by Constraint (D). Therefore, the LP's objective term cannot be minimized below 0. Hence, the LP has an optimal solution.

An optimal LP solution implies an execution schedule for some N: The linear program has only rational coefficients. Therefore, there is a rational optimal solution S^* for the LP with

a rational objective value t^*.[2] Since the values $S^*[x_{ik}]$ for all x_{ik} variables in S^* are rational, there is a number $L \in \mathbb{N}^+$ (e.g., the least common multiple of all divisors) such that for each x_{ik}, $L \cdot S^*[x_{ik}]$ is an integer.

We define for each port k a sequence σ_k of instructions, where each instruction i occurs $L \cdot S^*[x_{ik}]$ times:

$$\sigma_k := \Big[\underbrace{i_1, \ldots, i_1}_{L \cdot S^*[x_{i_1 k}] \text{ times}} , \ldots, \underbrace{i_{|\mathbf{I}|}, \ldots, i_{|\mathbf{I}|}}_{L \cdot S^*[x_{i_{|\mathbf{I}|} k}] \text{ times}} \Big]$$

These sequences allow us to define an execution schedule $Ex^* = [s_1, \ldots, s_{|Ex^*|}]$ for L iterations of e as follows:

$$s_c(k) := \sigma_k[c]$$

We can show that Ex^* is a valid execution schedule for L instances of the experiment e, following Definition 3.3: Let $i \in \mathbf{I}$ be an instruction. Then the following holds:

$$\begin{aligned}
&\left|\left\{(c,k) \;\middle|\; 1 \le c \le |Ex^*| \wedge k \in \mathbf{P} \wedge s_c(k) = i\right\}\right| \\
&\quad = \left|\left\{(c,k) \;\middle|\; 1 \le c \le |Ex^*| \wedge k \in \mathbf{P} \wedge \sigma_k[c] = i\right\}\right| && \text{definition of } Ex^* \\
&\quad = \left|\bigcup_{k \in \mathbf{P}} \left\{(c,k) \;\middle|\; 1 \le c \le |Ex^*| \wedge \sigma_k[c] = i\right\}\right| && \text{partition by port} \\
&\quad = \sum_{k \in \mathbf{P}} \left|\left\{(c,k) \;\middle|\; 1 \le c \le |Ex^*| \wedge \sigma_k[c] = i\right\}\right| && \text{size of disjoint union} \\
&\quad = \sum_{k \in \mathbf{P}} L \cdot S^*[x_{ik}] && \text{definition of } \sigma_k \\
&\quad = L \cdot \sum_{k \in \mathbf{P}} S^*[x_{ik}] \\
&\quad = L \cdot e(i) && \text{Constraint (A)}
\end{aligned}$$

Hence, the first constraint of Definition 3.3 is fulfilled. Constraint (E) ensures that only instructions that can be executed on port k occur in the sequence σ_k, therefore the second constraint of the definition is also fulfilled.

[2] Whenever a linear program has an optimal solution, it has a basic feasible solution that is also optimal. Each basic feasible solution of an LP satisfies a linear system of equalities consisting of constraints of the LP where inequalities are replaced by equalities. For an LP with rational coefficients, the basic feasible solutions – including optimal ones – are therefore rational.

Lastly, we observe that the length of the execution schedule Ex^* is given by the maximal sequence length $\max_{k \in \mathbf{P}} |\sigma_k|$. The length of a sequence σ_k relates to the LP's objective value t^* as follows:

$$\begin{aligned}
|\sigma_k| &= \sum_{i \in \mathbf{I}} L \cdot S^*[x_{ik}] && \text{definition of } Ex^* \\
&= L \cdot \sum_{i \in \mathbf{I}} S^*[x_{ik}] \\
&= L \cdot S^*[p_k] && \text{Constraint (B)} \\
&\leq L \cdot S^*[t] && \text{Constraint (C)} \\
&= L \cdot t^*
\end{aligned}$$

Since the optimization goal is to minimize t, the above inequality needs to be tight for at least one port k. Therefore, $|Ex^*| = \max_{k \in \mathbf{P}} |\sigma_k| = L \cdot t^*$.

The execution time of the implied schedule is the modeled inverse throughput: We show that $t^* = \inf\left\{ \frac{T(M,e,N)}{N} \;\middle|\; N \in \mathbb{N}^+ \right\}$.

As we have seen, $L \cdot t^*$ is the execution time of an execution schedule for L iterations of the experiment e with the port mapping M. Therefore, $T(M, e, L) \leq L \cdot t^*$ and consequently $\frac{T(M,e,L)}{L} \leq t^*$ must hold. Thus, t^* cannot be smaller than $\inf\left\{ \frac{T(M,e,N)}{N} \;\middle|\; N \in \mathbb{N}^+ \right\}$.

Assume there is some N^* such that $\frac{T(M,e,N^*)}{N^*} < t^*$. By Observation (A), the corresponding optimal execution schedule implies a feasible solution for the LP with objective value $\frac{T(M,e,N^*)}{N^*} < t^*$. This contradicts the optimality of t^*, hence $\frac{T(M,e,N)}{N} \geq t^*$ for every N. Consequently, t^* is the infimum of $\left\{ \frac{T(M,e,N)}{N} \;\middle|\; N \in \mathbb{N}^+ \right\}$, which is equal to the modeled inverse throughput $tp_M^{-1}(e)$ by Theorem 3.6. □

A.1.3. Proof for Theorem 3.14

Theorem 3.14. Let M be a two-level or three-level port mapping for a set $\mathbf{P}$ of ports and let e be an experiment with inverse throughput $t^* = tp_M^{-1}(e)$. Let S be the set of all optimal solutions to the corresponding linear program from Theorem 3.7 or Theorem 3.12. Then, there is a non-empty set $BP_M(e) := \bigcap_{s \in S} \{k \mid s[p_k] = t^*\}$ of *bottleneck ports*.

For Context:

Theorem 3.7. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment $e : I \to \mathbb{N}$ with a port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ is the objective value of an optimal solution to the following linear program:

$$
\begin{array}{lll}
\text{minimize} & t & \\
\text{subject to} & \displaystyle\sum_{k \in \mathbf{P}} x_{ik} = e(i) & \text{for all instructions } i \in \mathbf{I} \quad \text{(A)} \\
& \displaystyle\sum_{i \in \mathbf{I}} x_{ik} = p_k & \text{for all ports } k \in \mathbf{P} \quad \text{(B)} \\
& p_k \leq t & \text{for all ports } k \in \mathbf{P} \quad \text{(C)} \\
& x_{ik} \geq 0 & \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} \quad \text{(D)} \\
& x_{ik} = 0 & \text{if } (i,k) \notin E \quad \text{(E)}
\end{array}
$$

In particular, this linear program is feasible and has a finite optimal objective value.

Let M be a two-level port mapping (the argument for three-level port mappings follows analogously).

Assume the set $BP_M(e)$ was empty. We know that the linear program is feasible (since any execution schedule implies a feasible solution to the LP, see Appendix A.1.2). Therefore, S is not empty.

For each optimal solution $s \in S$, the set $Q(s) := \{k \mid s[p_k] = t^*\}$ is not empty: otherwise, t^* could not be the optimal objective value of the LP. Let $s^* \in S$ be an optimal solution. For each port $k \in Q(s^*)$, there needs to be another optimal solution $s' \in S$ such that $k \notin Q(s')$ – otherwise, k would be in $BP_M(e)$ – and, consequently, $s'[p_k] < t^*$. Let S' be the set containing s^* and the other solutions s' for each $k \in Q(s^*)$.

We now consider the midpoint of the solutions in S', i.e., an assignment s_m of variables to values such that for each variable v of the LP, $s_m[v] = \sum\{s[v] \mid s \in S'\}/|S'|$. For each port k, $s_m[p_k] < t^*$ holds, since $s_m[p_k]$ is the arithmetic mean of a finite set of values with no value larger than t^*, but at least one value smaller than t^*. From LP theory, we know that the set of feasible solutions of a linear program is a convex polyhedron, meaning that all points

between feasible solutions are also feasible solutions. As a consequence, the midpoint of a set of optimal solutions like S' must also be included in the convex set of feasible solutions. This contradicts the premise that t^* is the optimal objective value, since s_m is a feasible solution with a lower objective value. □

A.1.4. Proof for Theorem 3.19

Theorem 3.19. The problem OffPMInfer2-D (Definition 3.18) of deciding the existence of a satisfying two-level mapping for a set *Exps* of experiments and k ports is NP-hard. It is NP-complete if we assume a unary encoding of k.

For Context:

Definition 3.18. The two-level offline port mapping inference problem OffPMInfer2 is the following task: Given a set $\mathbf{I}$ of instructions, a number k of ports, and a set *Exps* of experiments e with measured inverse throughputs $tp^{-1}(e)$, compute a two-level port mapping $M = (\mathbf{I} \cup \{1, \ldots, k\}, E)$ such that it simulates the measured throughputs:

$$\forall e \in Exps.\ tp^{-1}(e) = tp_M^{-1}(e)$$

If no such two-level port mapping exists, return an error value.

A problem instance is *satisfiable* if there is a port mapping as required by the problem; we call these port mappings *satisfying*. Otherwise, the problem instance is *unsatisfiable*. In the decision version OffPMInfer2-D, the task is to decide if the problem instance is satisfiable or not.

The three-level variants OffPMInfer3 and OffPMInfer3-D are defined analogously. ⌟

We show two results: OffPMInfer2-D is NP-hard, and OffPMInfer2-D is in NP if k is unary encoded.

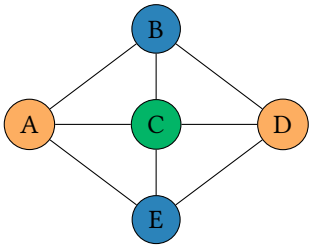
Hardness. To prove NP-hardness, we reduce the well-known NP-complete graph k-coloring problem to OffPMInfer2-D. We are given an undirected graph $G = (V, E)$ to color. The following OffPMInfer2-D instance is satisfiable if and only if G is k-colorable:

$$\begin{aligned} \mathbf{I} &= V \\ |\mathbf{P}| &= k \\ Exps &= \{(\{v \mapsto 1\}, 1) \mid v \in V\} \cup \{(\{u \mapsto 1, v \mapsto 1\}, 1) \mid \{u, v\} \in E\} \end{aligned}$$

The idea is to represent graph nodes as instructions and colors as ports. A graph coloring then directly corresponds to an instruction-to-port mapping. The experiments enforce that a satisfying port mapping implies a valid graph coloring. Consider Figure A.1 for an example of this construction for a 3-colorable graph and a satisfying mapping to three ports.

We argue that a k-colorable graph implies a satisfying port mapping and vice versa:

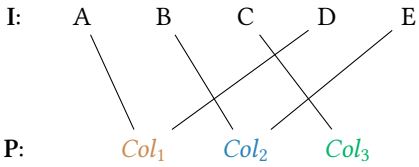
⇒ Let G be k-colorable with a node-to-color mapping M. M is also a satisfying port mapping for the corresponding OffPMInfer2-D instance: The per-node experiments are satisfied



(a)

$$\mathbf{I} := \{A, B, C, D, E\} \qquad |\mathbf{P}| := k$$
$$\begin{aligned}\mathsf{Exps} := \{&(\{A\}, 1), (\{B\}, 1), (\{C\}, 1), (\{D\}, 1),\\ &(\{E\}, 1), (\{A, B\}, 1), (\{A, C\}, 1),\\ &(\{A, E\}, 1), (\{B, C\}, 1), (\{B, D\}, 1),\\ &(\{C, D\}, 1), (\{C, E\}, 1), (\{D, E\}, 1)\}\end{aligned}$$

(b)



(c)

Figure A.1. A 3-colorable graph (a), the corresponding offline port mapping inference instance (b), and a satisfying port mapping (c).

since M maps each node to a single color, meaning that each instruction in the port mapping can be executed on exactly one port. Executing such a singleton experiment therefore blocks the corresponding port for one cycle in every iteration, leading to an inverse throughput of one cycle. For each of the per-edge experiments, we know that the corresponding instructions (graph nodes) are mapped to different ports (colors). Hence, these experiments can be executed with the involved ports blocked for one cycle per iteration. All experiment constraints are therefore satisfied.

$\Leftarrow$ Let M be a satisfying port mapping for the OffPMInfer2-D instance that corresponds to a graph G. M maps each instruction to a single port: Otherwise, if an instruction could be executed on $n > 1$ ports, the inverse throughput of the corresponding experiment would need to be $\frac{1}{n}$ cycles, violating the requirements of the singleton experiments. M can therefore be interpreted as a functional assignment of graph nodes to colors.

Let a, b be two adjacent nodes of G. From the per-edge experiments, we know that the inverse throughput of $\{a, b\}$ with the port mapping M is one cycle. If $M[a]$ was equal to $M[b]$, this inverse throughput could not be achieved: When two instructions that are assigned to the same port are executed together in an experiment, they block this port for two cycles per experiment instance. The adjacent nodes a and b are thus mapped to a different color (port).

Lastly, we note that a OffPMInfer2-D instance constructed in this way consists of one experiment per graph edge and one experiment per graph node, it can therefore be encoded with polynomial space requirements.

Membership. To show that OffPMInfer2-D is in NP, we have to verify that a two-level port mapping is a solution to a given problem instance in a time that grows polynomially with the input. Theorem 3.7 provides a linear program of polynomial size (with respect to the size of the sets of instructions and ports, which are part of the instance) whose solution is the throughput of an experiment with a given port mapping. Since solving a linear program is possible in polynomial time (Bertsimas and Tsitsiklis, 1997), it requires only polynomial time to evaluate all experiments with the given candidate port mapping and compare the results with the throughputs given in the problem instance.

□

A.1.5. Proof for Theorem 3.20

Theorem 3.20. The problem OffPMInfer3-D (Definition 3.18) of deciding the existence of a satisfying three-level mapping for a set *Exps* of experiments and k ports is NP-hard. It is NP-complete if we assume a unary encoding of k and the observed inverse throughputs for the experiments.

For Context:

Theorem 3.14. Let M be a two-level or three-level port mapping for a set $\mathbf{P}$ of ports and let e be an experiment with inverse throughput $t^* = tp_M^{-1}(e)$. Let S be the set of all optimal solutions to the corresponding linear program from Theorem 3.7 or Theorem 3.12. Then, there is a non-empty set $BP_M(e) := \bigcap_{s \in S} \{k \mid s[p_k] = t^*\}$ of *bottleneck ports*.

Definition 3.18. The two-level offline port mapping inference problem OffPMInfer2 is the following task: Given a set $\mathbf{I}$ of instructions, a number k of ports, and a set *Exps* of experiments e with measured inverse throughputs $tp^{-1}(e)$, compute a two-level port mapping $M = (\mathbf{I} \uplus \{1, \ldots, k\}, E)$ such that it simulates the measured throughputs:

$$\forall e \in Exps.\ tp^{-1}(e) = tp_M^{-1}(e)$$

If no such two-level port mapping exists, return an error value.

A problem instance is *satisfiable* if there is a port mapping as required by the problem; we call these port mappings *satisfying*. Otherwise, the problem instance is *unsatisfiable*. In the decision version OffPMInfer2-D, the task is to decide if the problem instance is satisfiable or not.

The three-level variants OffPMInfer3 and OffPMInfer3-D are defined analogously. ⌟

Hardness. We cannot use a direct reduction from OffPMInfer2-D since the existence of a satisfying three-level port mapping for a problem instance does not imply a two-level one in general.[3] Instead, we show that the specific problem instance used for the reduction in the proof of Theorem 3.19 has a satisfying two-level mapping if and only if there is a satisfying three-level mapping.

One of the constituting implications of this statement is straightforward: If *any* problem instance has a satisfying two-level mapping M, the three-level mapping that maps each instruction i to a µop that can be executed on $M[i]$ is also satisfying (see Figure A.2 for an

[3] For example, the three-level mapping in Figure 3.3 on page 19 cannot be represented as a two-level port mapping.

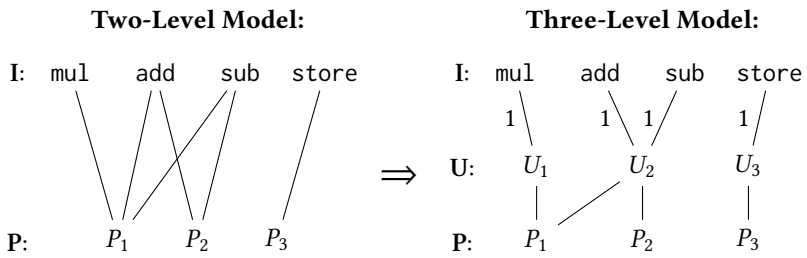


Figure A.2. Example for the straightforward translation of a two-level port mapping to the three-level model.

example). To prove the other implication, we need properties of the specific problem instance we use to encode a graph (V, E):

$$\begin{aligned} \mathbf{I} &= V \\ |\mathbf{P}| &= k \\ \mathit{Exps} &= \{(\{v \mapsto 1\}, 1) \mid v \in V\} \cup \{(\{u \mapsto 1, v \mapsto 1\}, 1) \mid \{u, v\} \in E\} \end{aligned}$$

Let M_1 be a satisfying three-level port mapping for this problem instance. We construct a two-level port mapping M_2 where each instruction is mapped to one of its bottleneck ports (cf. Theorem 3.14) with M_1:[4]

$$M_2 := \left(\mathbf{I} \cup \{1, \ldots, k\}, \left\{i \mapsto \left(\varepsilon x.\ x \in BP_{M_1}(\{i \mapsto 1\})\right) \;\middle|\; i \in \mathbf{I}\right\}\right)$$

We now assume that M_2 is not a solution to the OffPMInfer2-D problem and show that this implies a contradiction. As M_2 is not satisfying, there needs to be an experiment $e \in \mathit{Exps}$ whose throughput is simulated incorrectly by M_2. Since M_2 maps every instruction to exactly one port, the inverse throughput of every experiment consisting of only one instruction is 1, as required by *Exps*. The experiment e therefore needs to be of the form $\{i \mapsto 1, j \mapsto 1\}$, and $tp^{-1}_{M_2}(e) \neq 1$.

By construction, M_2 maps both, i and j, to exactly one port. The inverse throughput $tp^{-1}_{M_2}(e)$ can therefore only be 2 (if both instructions are mapped to the same port) or 1 (if they map to different ports). Since we ruled out the latter case, the former needs to apply, i.e.,

$$M_2[i] = \{k\} = M_2[j].$$

This however means that k is a bottleneck port for $\{i \mapsto 1\}$ and for $\{j \mapsto 1\}$ in M_1. Therefore, there is no way to execute the µops of i in M_1 with an inverse throughput of 1 cycle without utilizing port k for 1 cycle per iteration (and the same holds for the µops of j).

[4] With $\varepsilon x.\ P(x)$ we denote an arbitrary value x that satisfies the predicate $P(x)$.

As M_1 satisfies the problem instance, $tp^{-1}_{M_1}\left(\left\{i \mapsto 1, j \mapsto 1\right\}\right) = 1$ holds. Therefore, a solution to the corresponding linear program uses port k for less than 1 cycle for the µops of at least one of i and j; without loss of generality: j. Since the solution still needs to accommodate all µops of j without utilizing any port for more than 1 cycle, we can extract an optimal solution for the LP for the experiment $\left\{j \mapsto 1\right\}$ with M_1 where the bottleneck port k is underutilized, a contradiction.

In the proof for Theorem 3.19, we have shown that a graph (V, E) is k-colorable if and only if there is a satisfying two-level port mapping for the corresponding problem instance. The above construction shows that there is a satisfying two-level port mapping for such a problem instance if and only if there is a satisfying three-level port mapping. Together, this proves that (V, E) is k-colorable if and only if a satisfying three-level port mapping for the problem instance exists, i.e., we can solve a k-coloring problem by solving a OffPMInfer3-D instance with polynomial input size.

Membership. To show that OffPMInfer3-D is in NP, we have to prove that for every satisfiable problem instance there is a satisfying polynomially-sized three-level port mapping that we can validate in a time that grows polynomially with the input. This is more involved than in the two-level setting: In general, three-level port mappings do not necessarily qualify as a checkable certificate for a solution to the OffPMInfer3-D instance since they could use exponentially many µops arbitrarily often. Their size is therefore not polynomially bounded as it would be required for a certificate (see, e.g., the text book by Arora and Barak (2009, Chapter 2)).

We can however show that there is a polynomial bound to the total number of µops involved in an experiment e with inverse throughput t: In the worst case, each of the k ports is fully occupied for t cycles with an iteration of the experiment. The summed total utilization is therefore $t \cdot k$ cycles. Each involved µop contributes 1 cycle to this total utilization, possibly spread among several ports. Thus, only $t \cdot k$ µops can be involved. Therefore, and since we assume k and t to be encoded unarily in the input, the parts of a satisfying port mapping that are constrained by the experiments in the instance are polynomially bounded. Any satisfying port mapping for a set of experiments can therefore be transformed into one that is polynomially bounded by replacing the port usage for unconstrained instructions with some trivial port usage.

Such polynomially-bounded port mappings can be checked in polynomial time with the linear program from Theorem 3.12.

□

A.2. Proofs for Chapter 4

A.2.1. Proof for Theorem 4.4

Theorem 4.4. The linear program from Theorem 3.7 is feasible if and only if the following system of constraints is satisfiable:

$$\sum_{k \in \mathbf{P}} x_{ik} = e(i) \qquad \text{for all instructions } i \in \mathbf{I} \tag{A}$$

$$\sum_{i \in \mathbf{I}} x_{ik} = p_k \qquad \text{for all ports } k \in \mathbf{P} \tag{B}$$

$$p_k \leq t \qquad \text{for all ports } k \in \mathbf{P} \tag{C}$$

$$x_{ik} \geq 0 \qquad \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} \tag{D}$$

$$x_{ik} = 0 \qquad \text{if } (i,k) \notin E \tag{E}$$

$$\bigvee_{k \in \mathbf{P}} q_k \tag{F}$$

$$q_k \leftrightarrow (p_k = t) \qquad \text{for all ports } k \in \mathbf{P} \tag{G}$$

$$j_i \rightarrow q_k \qquad \text{if } (i,k) \in E \tag{H}$$

$$\sum_{i \in \mathbf{I}} j_i \cdot e(i) = \sum_{k \in \mathbf{P}} q_k \cdot t \tag{I}$$

The optimal objective value t^* of the linear program is equal to the value $m[t]$ of the variable t in any satisfying model m of this constraint system.

For Context:

Theorem 3.7. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment $e : I \rightarrow \mathbb{N}$ with a port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ is the objective value of an optimal solution to the following linear program:

$$\begin{aligned} &\text{minimize} && t \\ &\text{subject to} && \sum_{k \in \mathbf{P}} x_{ik} = e(i) && \text{for all instructions } i \in \mathbf{I} && \text{(A)} \\ & && \sum_{i \in \mathbf{I}} x_{ik} = p_k && \text{for all ports } k \in \mathbf{P} && \text{(B)} \\ & && p_k \leq t && \text{for all ports } k \in \mathbf{P} && \text{(C)} \\ & && x_{ik} \geq 0 && \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} && \text{(D)} \\ & && x_{ik} = 0 && \text{if } (i,k) \notin E && \text{(E)} \end{aligned}$$

In particular, this linear program is feasible and has a finite optimal objective value.

As a part of the proof for this theorem, we first formulate and prove a helpful intermediate result:

Lemma A.1. Let $M := (\mathbf{I} \cup \mathbf{P}, E)$ be a two-level port mapping, let e be an experiment, and let

- S be the set of optimal solutions and t^* the optimal objective value for the linear program from Theorem 3.7 for M and e,
- $Q(s) := \left\{k \in \mathbf{P} \;\middle|\; s[p_k] = t\right\}$ be the set of ports that are fully utilized in such an optimal solution $s \in S$, and
- $J(s) := \left\{i \in \mathbf{I} \;\middle|\; \exists k \in Q(s).\ s[x_{ik}] > 0\right\}$ be the set of instructions that are executed on ports in $Q(s)$.

There is an optimal solution $\hat{s} \in S$ such that instructions in $J(\hat{s})$ can only be executed on ports from $Q(\hat{s})$:

$$\forall i \in J(\hat{s}).\ M[i] \subseteq Q(\hat{s})$$

Proof. From Theorem 3.7, we know that S is not empty. Let $s \in S$ and let

$$J^\Delta(s) := \left\{i \in J(s) \;\middle|\; \exists k \notin Q(s).\ (i,k) \in E\right\}$$

If $J^\Delta(s)$ is empty, s fulfills the requirements of $\hat{s}$. Otherwise, the following algorithm can modify s to obtain a feasible solution s' with identical objective value and an empty $J^\Delta(s')$:

1. Pick an $i' \in J^\Delta(s)$ and a $k' \notin Q(s)$ such that $(i', k') \in M$.
2. Determine the set $Q'(s) := \left\{k \in Q(s) \;\middle|\; s[x_{i'k}] > 0\right\}$ of ports in $Q(s)$ that execute mass of i'.
3. Determine the smallest amount of i'-mass that we can remove from every port in $Q(s)$ that have i'-mass

   $$x_{removable} := \min\left\{s[x_{i'k}] \;\middle|\; k \in Q'(s)\right\}$$

   and the least upper bound to the mass that we can add to k' without reaching a total mass of t^* on k'

   $$x_{addable} := t^* - s[p_{k'}].$$

   Both are greater than zero since $i' \in J^\Delta(s) \subseteq J(s)$ and $k' \notin Q(s)$. Then,

   $$\varepsilon := \min(x_{removable}, x_{addable}/|Q'(s)|)$$

   bounds the amount of i' mass that we can remove from every port in $Q'(s)$ and add to k' from above. It is greater than zero as well.

4. Define a new solution s':

$$\begin{aligned} s'[x_{i'k'}] &= s[x_{i'k'}] + \frac{\varepsilon}{2} \cdot |Q'(s)| \\ s'[x_{i'k}] &= s[x_{i'k}] - \frac{\varepsilon}{2} && \text{for ports } k \in Q'(s) \\ s'[x_{ik}] &= s[x_{ik}] && \text{otherwise} \\ s'[p_k] &= \sum_i s'[x_{ik}] && \text{for ports } k \\ s'[t] &= \max_k s[p_k] \end{aligned}$$

5. If the set $J^{\Delta}(s')$ that is implied by s' is empty, stop. Otherwise, go to (1) with s' as the new starting solution s.

After every step of the algorithm, the updated mapping s' is still a feasible solution to the linear program:

- Constraint (A) holds since the values for the x_{ik} variables only change for $i = i'$ and

$$\sum_{k \in \mathbf{P}} s'[x_{i'k}] = \left(\sum_{k \in \mathbf{P}} s[x'_i k] \right) + \frac{\varepsilon}{2} \cdot |Q'(s)| - \frac{\varepsilon}{2} \cdot |Q'(s)| = \sum_{k \in \mathbf{P}} s[x'_i k] = e(i')$$

- The constraints (B) and (C) hold trivially because of how s' is constructed.
- Constraint (D) could only be violated if values of x_{ik} variables would be reduced below zero. This is impossible since $\frac{\varepsilon}{2}$ is chosen to be smaller than the value of every x_{ik} variable that is to be reduced.
- As the algorithm only modifies values of x_{ik} variables for which $(i, k) \in M$, the integrity of (E) is preserved.

Furthermore, the updated solution s' preserves optimality: The objective value t is tied to the maximal occuring port pressure p_k. The only k for which $s'[p_k]$ is greater than $s[p_k]$ is k'. However, the following holds:

$$\begin{aligned} s'[p'_k] = \sum_{i \in \mathbf{I}} s'[x_{ik'}] = \left(\sum_{i \in \mathbf{I}} s[x_{ik'}] \right) + \frac{\varepsilon}{2} \cdot |Q'(s)| &\leq \left(\sum_{i \in \mathbf{I}} s[x_{ik'}] \right) + \frac{t^* - s[p_{k'}]}{2 \cdot |Q'(s)|} \cdot |Q'(s)| \\ = s[p_{k'}] + \frac{t^* - s[p_{k'}]}{2} &= \frac{t^* + s[p_{k'}]}{2} \leq t^* \end{aligned}$$

Therefore, no p_k is increased to a value greater than t^*. As a feasible solution cannot have a smaller objective value than the optimal solution s, at least one port k with $s'[k] = t^*$ remains.

The transformation algorithm terminates since at least one of the finite number of ports in $Q(s)$ is removed in each iteration. □

We now show both directions of the equivalence between LP and SMT constraints individually. Let $M := (\mathbf{I} \cup \mathbf{P}, E)$ be a two-level port mapping and let e be an experiment.

- Let the corresponding SMT formulation be satisfiable with a model m that fulfills constraints (A)–(I) and let $m[t]$ be the value of the t variable in the model m. We define two sets capturing bottleneck ports and instructions:

$$Q = \{k \in \mathbf{P} \mid m[q_k] = 1\}$$

$$J = \{i \in \mathbf{I} \mid m[j_i] = 1\}$$

These enable the following shorthand notations for any functions $f : \mathbf{I} \to \mathbb{R}$ and $g : \mathbf{P} \to \mathbb{R}$:

$$\sum_{i \in \mathbf{I}} j_i \cdot f(i) = \sum_{i \in J} f(i)$$

$$\sum_{k \in \mathbf{P}} q_k \cdot f(k) = \sum_{k \in Q} f(k)$$

Since all constraints of the linear program are also constraints of the SMT formulation, m is a feasible solution for the LP. We therefore only need to show that its objective value $m[t]$ is optimal. Assume m was not an optimal solution of the LP, i.e., there is a "better" solution s^* such that $s^*[t] < m[t]$. Since $s^*[p_k] \leq s^*[t]$ (as s^* has to satisfy constraint (C)) and $m[t] = m[p_k]$ for all $k \in Q$ (by constraint (G)), we can conclude that $s^*[p_k] < m[p_k]$ for all $k \in Q$. Because of constraint (F), we know that Q is not empty.

We find the following to hold because of constraint (I):

$$\sum_{k \in Q} s^*[p_k] < \sum_{k \in Q} m[p_k] = \sum_{k \in Q} m[t] \overset{\text{(I)}}{=} \sum_{i \in J} e(i) \tag{A.4}$$

By constraint (B) and because (D) ensures that $s^*[x_{ik}] > 0$ for any $i \in \mathbf{I}$ and $k \in \mathbf{P}$, we can rewrite and bound the left-hand side of this inequality:

$$\sum_{k \in Q} s^*[p_k] \overset{\text{(B)}}{=} \sum_{k \in Q} \sum_{i \in \mathbf{I}} s^*[x_{ik}] \geq \sum_{k \in Q} \sum_{i \in J} s^*[x_{ik}] \tag{A.5}$$

Inequalities (A.4) and (A.5) together yield the following inequality:

$$\sum_{k \in Q} \sum_{i \in J} s^*[x_{ik}] < \sum_{i \in J} e(i) \tag{A.6}$$

Using constraint (A) and by reordering and decomposing sums, we can further deduce:

$$\sum_{i \in J} e(i) \overset{\text{(A)}}{=} \sum_{i \in J} \sum_{k \in \mathbf{P}} s^*[x_{ik}] = \sum_{k \in \mathbf{P}} \sum_{i \in J} s^*[x_{ik}] = \sum_{k \in Q} \sum_{i \in J} s^*[x_{ik}] + \sum_{k \notin Q} \sum_{i \in J} s^*[x_{ik}]$$

By Inequality (A.6), the left summand $\sum_{k \in Q} \sum_{i \in J} s^*[x_{ik}]$ is strictly smaller than $\sum_{i \in J} e(i)$. This implies that the right summand $\sum_{k \notin Q} \sum_{i \in J} s^*[x_{ik}]$ is strictly greater than zero. In other words, mass from some bottleneck instruction $i \in J$ is scheduled to a non-bottleneck port $k \notin Q$. With constraint (E), this entails that $(i, k) \in E$ whereas constraint (H) requires that $(i, k) \notin E$, a contradiction. Hence, there cannot be such a "better" solution s^*, m is an optimal solution to the LP.

- Let the LP be feasible with a solution s with optimal objective value t^*. In the following, we show how to construct a model m that satisfies the constraints of the SMT formulation with $m[t] = t^*$. We start by defining sets of bottleneck ports and bottleneck instructions:

$$Q = \{k \in \mathbf{P} \mid s[p_k] = t\}$$

$$J = \{i \in \mathbf{I} \mid \exists k \in Q.\ s[x_{ik}] > 0\}$$

By Lemma A.1, we can assume that the set

$$J^\Delta = \left\{i \in J \;\middle|\; \exists k \notin Q.\ (i, k) \in E\right\} = \left\{i \in J \;\middle|\; M[i] \not\subseteq Q\right\}$$

is empty without loss of generality.

We construct our model m for the SMT formulation as follows:

$$\begin{array}{rl}
m[x_{ik}] = s[x_{ik}] & \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} \\
m[p_k] = s[p_k] & \text{for ports } k \in \mathbf{P} \\
m[t] = s[t] & \\
m[q_k] \leftrightarrow (k \in Q) & \text{for ports } k \in \mathbf{P} \\
m[j_i] \leftrightarrow (i \in J) & \text{for instructions } i \in \mathbf{I}
\end{array}$$

This model satisfies the SMT constraints (A)-(I):

  - The first five constraints (A)-(E) hold as s is a solution of the LP, which already requires these constraints.
  - Constraint (F) holds as s is optimal, i.e., there is a k such that $s[p_k] = t^*$.
  - The construction of Q ensures that (G) holds.
  - Constraint (H) follows from the emptiness of J^Δ: Let $(i, k) \in E$ such that $i \in J$, which entails $m[j_i] = 1$. As $J^\Delta = \emptyset$, $k \in Q$ holds. Therefore, $m[q_k]$ is true.

– Lastly, relying on constraints whose satisfaction we have already shown, we can see that constraint (I) also holds:

$$\begin{aligned}
\sum_{i \in \mathbf{I}} m[j_i] \cdot e(i) &= \sum_{i \in J} e(i) && \text{by the definition of } m[j_i] \\
&= \sum_{i \in J} \sum_{k \in \mathbf{P}} m[x_{ik}] && \text{by (A)} \\
&= \sum_{i \in J} \sum_{k \in Q} m[x_{ik}] && \text{as } J^{\Delta} = \emptyset \text{ and by (E)} \\
&= \sum_{i \in \mathbf{I}} \sum_{k \in Q} m[x_{ik}] && \text{by the definition of } J \text{ and by (D)} \\
&= \sum_{k \in Q} \sum_{i \in \mathbf{I}} m[x_{ik}] && \text{reorder sums} \\
&= \sum_{k \in Q} m[p_k] && \text{by (B)} \\
&= \sum_{k \in \mathbf{P}} m[q_k] \cdot m[p_k] && \text{by the definition of } Q \\
&= \sum_{k \in \mathbf{P}} m[q_k] \cdot m[t] && \text{by (G)}
\end{aligned}$$

We therefore constructed a satisfying model m for the SMT constraints with $m[t] = t^*$.

Together, this proves the theorem. □

A.3. Proofs for Chapter 5

A.3.1. Proof for Theorem 5.2

Theorem 5.2. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment e with the port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ can be equivalently characterized as follows:

$$tp_M^{-1}(e) = \max_{Q \subseteq \mathbf{P}} \frac{\sum\{e(i) \mid M[i] \subseteq Q\}}{|Q|} \tag{5.1}$$

$M[i] := \{k \mid (i,k) \in E\}$ denotes the set of ports that can execute an instruction i with M.

For Context:

Theorem 3.7. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment $e : I \to \mathbb{N}$ with a port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ is the objective value of an optimal solution to the following linear program:

$$\begin{array}{lll}
\text{minimize} & t & \\
\text{subject to} & \displaystyle\sum_{k \in \mathbf{P}} x_{ik} = e(i) & \text{for all instructions } i \in \mathbf{I} \quad \text{(A)} \\
& \displaystyle\sum_{i \in \mathbf{I}} x_{ik} = p_k & \text{for all ports } k \in \mathbf{P} \quad \text{(B)} \\
& p_k \le t & \text{for all ports } k \in \mathbf{P} \quad \text{(C)} \\
& x_{ik} \ge 0 & \text{for all instructions } i \in \mathbf{I}, \text{ ports } k \in \mathbf{P} \quad \text{(D)} \\
& x_{ik} = 0 & \text{if } (i,k) \notin E \quad \text{(E)}
\end{array}$$

In particular, this linear program is feasible and has a finite optimal objective value.

Lemma A.1. Let $M := (\mathbf{I} \cup \mathbf{P}, E)$ be a two-level port mapping, let e be an experiment, and let

- S be the set of optimal solutions and t^* the optimal objective value for the linear program from Theorem 3.7 for M and e,
- $Q(s) := \left\{k \in \mathbf{P} \;\middle|\; s[p_k] = t\right\}$ be the set of ports that are fully utilized in such an optimal solution $s \in S$, and
- $J(s) := \left\{i \in \mathbf{I} \;\middle|\; \exists k \in Q(s).\ s[x_{ik}] > 0\right\}$ be the set of instructions that are executed on ports in $Q(s)$.

There is an optimal solution $\hat{s} \in S$ such that instructions in $J(\hat{s})$ can only be executed on ports from $Q(\hat{s})$:

$$\forall i \in J(\hat{s}).\ M[i] \subseteq Q(\hat{s})$$

Let $S(M, e)$ be defined as follows:

$$S(M,e) := \left\{ \frac{\sum\{e(i) \mid M[i] \subseteq Q\}}{|Q|} \;\middle|\; Q \subseteq \mathbf{P} \right\}$$

With this notation, the right-hand side of Equation (5.1) can be written as

$$\max S(M,e) =: \hat{tp}_M^{-1}(e)$$

The proof proceeds by showing that the result $\hat{tp}_M^{-1}(e)$ of the bottleneck simulation algorithm is equal to the throughput $tp_M^{-1}(e)$ according to Theorem 3.7 for any experiment e and any (two-level) port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$. We do this by showing that $tp_M^{-1}(e)$ is included in $S(M, e)$ **(I)** and that each element of $S(M, e)$ is upper-bounded by $tp_M^{-1}(e)$ **(II)**.

(I) Let s be an optimal feasible solution of the linear program. We denote the value of a variable v chosen in s by $s[v]$. Since s is optimal, there is a non-empty maximal set $Q \subseteq P$ such that for all $k \in Q$ holds that

$$\sum_{i \in \mathbf{I}} s[x_{ik}] = s[p_k] = s[t] = tp_M^{-1}(e) \tag{A.7}$$

By Lemma A.1, we assume without loss of generality that each instruction that s executes on a port in Q can only be executed on ports in Q, that is:

$$k \in Q \wedge s[x_{ik}] > 0 \Rightarrow M[i] \subseteq Q \tag{A.8}$$

By defining $J := \{i \mid M[i] \subseteq Q\}$, we identify the following equalities:

$$\sum_{i \in J} e(i) \overset{(A)}{=} \sum_{i \in J} \sum_{k \in \mathbf{P}} s[x_{ik}] \overset{(E)}{=} \sum_{i \in J} \sum_{k \in Q} s[x_{ik}] = \sum_{k \in Q} \sum_{i \in J} s[x_{ik}]$$
$$\overset{(A.8)}{=} \sum_{k \in Q} \sum_{i \in \mathbf{I}} s[x_{ik}] \overset{(A.7)}{=} \sum_{k \in Q} s[t] = tp_M^{-1}(e) \cdot |Q|$$

The equality of the leftmost term and the rightmost term proves that $tp_M^{-1}(e) \in S(M, e)$:

$$\sum_{i \in J} e(i) = tp_M^{-1}(e) \cdot |Q|$$
$$\Leftrightarrow \quad tp_M^{-1}(e) = \frac{\sum_{i \in J} e(i)}{|Q|} = \frac{\sum\{e(i) \mid M[i] \subseteq Q\}}{|Q|}$$

(II) Let $Q' \subseteq \mathbf{P}$ and $t' := \sum\{e(i) \mid M[i] \subseteq Q'\}/|Q'|$. We assume $t' > tp_M^{-1}(e)$ and show that this leads to a contradiction, proving that $tp_M^{-1}(e)$ is an upper bound to each element of $S(M, e)$.

For this argument, we form the dual of the linear program.[5] First, we transform the LP to an equivalent form that allows for a more straightforward construction of the dual:

[5] See Chapter 4.2 in the textbook by Bertsimas and Tsitsiklis (1997)

$$
\begin{aligned}
&\min \quad t \\
&\text{s.t.} \sum_{k \in \mathbf{P}} x_{ik} = e(i) \quad \text{for } i \in \mathbf{I} \qquad (A) \\
&\qquad \sum_{i \in \mathbf{I}} x_{ik} = p_k \quad \text{for } k \in \mathbf{P} \qquad (B) \\
&\qquad p_k \leq t \qquad \text{for } k \in \mathbf{P} \qquad (C) \\
&\qquad x_{ik} \geq 0 \qquad \text{for } i \in \mathbf{I}, k \in \mathbf{P} \; (D) \\
&\qquad x_{ik} = 0 \qquad \text{if } (i,k) \notin E \qquad (E)
\end{aligned}
\quad \rightsquigarrow \quad
\begin{aligned}
&\min \quad t \\
&\text{s.t.} \sum_{k \in \mathbf{P}} x_{ik} = e(i) \quad \text{for } i \in \mathbf{I} \qquad (A) \\
&\qquad \sum_{i \in \mathbf{I}} x_{ik} - t \leq 0 \quad \text{for } k \in \mathbf{P} \qquad (C') \\
&\qquad x_{ik} \geq 0 \qquad \text{for } i \in \mathbf{I}, k \in \mathbf{P} \; (D) \\
&\qquad \overline{m}_{ik} \cdot x_{ik} = 0 \qquad \text{for } i \in \mathbf{I}, k \in \mathbf{P} \; (E')
\end{aligned}
$$

This transformation incorporates information about the edges E of the port mapping via boolean coefficients $\overline{m}_{ik} = 1 \Leftrightarrow (i,k) \notin E$. If $\overline{m}_{ik} = 1$, the constraint (E') restricts the x_{ik} variable to 0, otherwise $\overline{m}_{ik}$ is 0 and constraint (E') does not restrict x_{ik}. Furthermore, the transformation eliminates the p_k variables by substituting their occurrences with $\sum_{i \in \mathbf{I}} x_{ik}$. Lastly, we drop the now obsolete constraint (B) and adjust constraint (C) such that variables only occur in its left-hand side. This transformation does not affect the optimal objective value of the linear program.

The dual to the transformed linear program is as follows:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{i \in \mathbf{I}} e(i) \cdot y_i \\
\text{subject to} \quad & y_i + z_k + \overline{m}_{ik} \cdot v_{ik} \leq 0 \qquad \text{for all } i \in \mathbf{I}, k \in \mathbf{P} \\
& \sum_{k \in \mathbf{P}} -z_k = 1 \\
& z_k \leq 0 \qquad \text{for all } k \in \mathbf{P}
\end{aligned}
$$

Here, the y_i, z_k, and v_{ik} are real-valued variables.

By the strong duality theorem[6] for linear programs, an optimal solution for this dual linear program has the same objective $tp_M^{-1}(e)$ as an optimal solution for the primal linear program.

Given the assumption that $t' > tp_M^{-1}(e)$, we construct a solution s' for the dual with a higher objective value, which contradicts the optimality of $tp_M^{-1}(e)$. The construction of s' is as follows for each $i \in \mathbf{I}$ and $k \in \mathbf{P}$:

$$
\begin{aligned}
s'[z_k] &= \frac{-1}{|Q'|} && \text{if } k \in Q' \\
s'[y_i] &= \frac{1}{Q'} && \text{if } M[i] \subseteq Q' \\
s'[v_{ik}] &= -1
\end{aligned}
$$

All other variables are set to 0. This solution fulfills all constraints:

1. With variables set as above, violating the constraint $y_i + z_k + \overline{m}_{ik} \cdot v_{ik} \leq 0$ for any $i \in \mathbf{I}$, $k \in \mathbf{P}$ would require all of the following:

[6] See Theorem 4.4 in the textbook by Bertsimas and Tsitsiklis (1997)

- $s'[y_i] = \frac{1}{Q'}$, which implies that $M[i] \subseteq Q'$
- $s'[z_k] = 0$, which implies that $k \notin Q'$
- $\overline{m}_{ik} = 0$, which implies that $k \in M[i]$

It is impossible that all three of these hold, therefore the solution satisfies this constraint.

2. $\sum_{k \in \mathbf{P}} -s'[z_k] = |Q'| \cdot -\frac{-1}{|Q'|} = 1$
3. $s'[z_k]$ is either $-1/|Q'|$ or 0, either is less than or equal to 0.

Furthermore, the solution has the following objective value:

$$\sum_{i \in \mathbf{I}} e(i) \cdot y_i = \frac{\sum\{e(i) \mid M[i] \subseteq Q'\}}{|Q'|} = t' > tp_M^{-1}(e)$$

This contradicts the optimality of $tp_M^{-1}(e)$.

Overall, we conclude that $tp_M^{-1}(e)$ is the maximal element of the set $S(M, e)$. This proves the correctness of the bottleneck simulation algorithm. □

A.4. Proofs for Chapter 6

A.4.1. Proof for Theorem 6.3

Theorem 6.3. Let $M = (\mathbf{I} \cup \mathbf{P}, E)$ be a two-level port mapping and let $i, j \in \mathbf{I}$ such that $|M[i]| = |M[j]|$ and such that the following holds:

$$tp_M^{-1}(\{i \mapsto 1, j \mapsto 1\}) = tp_M^{-1}(\{i \mapsto 1\}) + tp_M^{-1}(\{j \mapsto 1\})$$

Then, i and j use the same set of ports: $M[i] = M[j]$.

For Context:

Theorem 5.2. The modeled inverse throughput $tp_M^{-1}(e)$ of an experiment e with the port mapping $M := (\mathbf{I} \cup \mathbf{P}, E)$ can be equivalently characterized as follows:

$$tp_M^{-1}(e) = \max_{Q \subseteq \mathbf{P}} \frac{\sum\{e(i) \mid M[i] \subseteq Q\}}{|Q|} \tag{5.1}$$

$M[i] := \{k \mid (i,k) \in E\}$ denotes the set of ports that can execute an instruction i with M.

Let M, i, j be as required in the theorem and let $n := |M[i]| = |M[j]|$. By Theorem 5.2, we know that

$$tp_M^{-1}(\{i \mapsto 1\}) = \max_{Q \subseteq \mathbf{P}} \frac{1 \text{ if } M[i] \subseteq Q \text{ else } 0}{|Q|} = \max_{M[i] \subseteq Q \subseteq \mathbf{P}} \frac{1}{|Q|} = \frac{1}{|M[i]|} = \frac{1}{n} \tag{A.9}$$

$$tp_M^{-1}(\{j \mapsto 1\}) = \max_{Q \subseteq \mathbf{P}} \frac{1 \text{ if } M[j] \subseteq Q \text{ else } 0}{|Q|} = \max_{M[j] \subseteq Q \subseteq \mathbf{P}} \frac{1}{|Q|} = \frac{1}{|M[j]|} = \frac{1}{n} \tag{A.10}$$

$$tp_M^{-1}(\{i \mapsto 1, j \mapsto 1\}) = \max_{Q \subseteq \mathbf{P}} \frac{(1 \text{ if } M[i] \subseteq Q \text{ else } 0) + (1 \text{ if } M[j] \subseteq Q \text{ else } 0)}{|Q|} \tag{A.11}$$

We now investigate possible values of the term that is subject to maximization in Equation (A.11) for different choices of Q:

- If $M[i] \not\subseteq Q$ and $M[j] \not\subseteq Q$ both hold, the term is 0.
- If $M[i] \subseteq Q$ and $M[j] \not\subseteq Q$, the term is $\frac{1}{|Q|}$. The largest value achievable this way uses a minimal $Q = M[i]$, leading to the value $\frac{1}{|M[i]|} = \frac{1}{n}$.
- Analogously, if $M[i] \not\subseteq Q$ and $M[j] \subseteq Q$, the maximal achievable value is $\frac{1}{|M[j]|} = \frac{1}{n}$.
- If $M[i] \subseteq Q$ and $M[j] \subseteq Q$, the term is $\frac{2}{|Q|}$. The largest value achievable this way uses a minimal $Q = M[i] \cup M[j]$, leading to a value of $\frac{2}{|M[i] \cup M[j]|}$.

The inverse throughput $tp_M^{-1}(\{i \mapsto 1, j \mapsto 1\})$ is the maximal of these four possible values. Since we require that

$$tp_M^{-1}(\{i \mapsto 1, j \mapsto 1\}) = tp_M^{-1}(\{i \mapsto 1\}) + tp_M^{-1}(\{j \mapsto 1\}) = \frac{1}{|M[i]|} + \frac{1}{|M[j]|} = \frac{2}{n},$$

we can rule out the first three of the above cases, as they cannot yield a sufficiently large value. Therefore, the following needs to hold:

$$\frac{2}{|M[i] \cup M[j]|} = tp_M^{-1}(\{i \mapsto 1, j \mapsto 1\}) = \frac{2}{n}$$

If $M[i] \neq M[j]$, and, consequently, $|M[i] \cup M[j]| > |n|$ held, this could not be the case. Therefore, $M[i] = M[j] = M[i] \cup M[j]$ needs to hold, which proves the statement. □

A.5. Proofs for Chapter 8

A.5.1. Proof for Theorem 8.4

Theorem 8.4. The generalization algorithm always terminates if a valid abstract domain is used.

For Context:

Input: basic block b
1 $absBB \leftarrow \beta(b)$;
2 **if** $absBB$ is not interesting **then return** b;
3 $rejected \leftarrow \{\}$;
4 **while** *True* **do**
5 $\quad avail \leftarrow \{E \in Exps \mid absBB \in \mathsf{dom}(E)\} \setminus rejected$;
6 $\quad$ **if** $avail = \{\}$ **then return** $absBB$;
7 $\quad exp \leftarrow choose(avail)$;
8 $\quad t \leftarrow exp(absBB)$;
9 $\quad$ **if** t is interesting **then** $absBB \leftarrow t$;
10 $\quad$ **else** $rejected \leftarrow rejected \cup \{exp\}$;

Algorithm A.1. Generalization Algorithm.

Definition 8.2. A *valid* abstract domain $(\mathbb{A}, \sqsubseteq)$ with a concretization function $\gamma : \mathbb{A} \to \mathbb{C}$, a representation function $\beta : \mathbb{B} \to \mathbb{A}$, and a set $Exps \subseteq \mathbb{A} \rightharpoonup \mathbb{A}$ of expansion functions satisfies the following constraints:

1. $(\mathbb{A}, \sqsubseteq)$ is partially ordered.
2. $(\mathbb{A}, \sqsubseteq)$ contains no infinite ascending chains.
3. All expansion functions $E \in Exps$ are strictly ascending:
$$\forall a \in \mathsf{dom}(E).\ a \sqsubseteq E(a) \wedge a \neq E(a)$$
4. For each abstract block, the number of applicable expansion functions is finite.
5. For each abstract block, every direct successor in $\sqsubseteq$ is reachable via an expansion function:
$$\forall a, b \in \mathbb{A}.\ \left(a \sqsubset b \wedge \nexists c \in \mathbb{A}.\ a \sqsubset c \sqsubset b\right) \Rightarrow \exists E \in Exps.\ E(a) = b$$
6. The expansion functions are monotone:
$$\forall a, a' \in \mathsf{dom}(E).\ a \sqsubseteq a' \Rightarrow E(a) \sqsubseteq E(a')$$
7. The concretization function is monotone:
$$\forall a, a' \in \mathbb{A}.\ a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a')$$

8. Concretization and representation functions satisfy the soundness constraint:

$$\forall b \in \mathbb{B}.\ b \in \gamma(\beta(b))$$

⌟

In each (non-terminating) iteration of the generalization algorithm, one of two operations is performed: Either the current abstract block is replaced by an expanded version (line 9), or an expansion is rejected (line 10). Neither of these operations can occur infinitely often in an execution of the algorithm:

Assume there was an execution of the generalization algorithm with infinitely many expansion steps. Since expansion functions are strictly ascending (Requirement 3), the sequence of values for *absBB* in this execution is an infinite ascending chain, which is forbidden by our requirements on an abstract domain (Requirement 2).

Assume there was an execution with infinitely many rejection steps. As there cannot be infinitely many expansion steps in the execution, the execution eventually expands to a final abstract block $absBB^{\dashv}$. In an execution with infinitely many rejection steps, there need to be infinitely many expansions applicable to (and rejected for) $absBB^{\dashv}$. This contradicts the requirement that only a finite number of expansions is applicable for each abstract block (Requirement 4).

Therefore, the generalization algorithm can only perform a finite number of operations and eventually terminates. □

A.5.2. Proof for Theorem 8.5

Theorem 8.5. Given an ideal check for the interestingness of an abstract block and a valid abstract domain, the result of the generalization algorithm is

1. interesting and
2. maximal, i.e., every more general abstract block is not interesting,

if the initial representative $\beta(b)$ is interesting. A check for interestingness $chk : \mathbb{A} \to \{\mathsf{True}, \mathsf{False}\}$ is ideal if

$$chk(a) \Leftrightarrow \forall b \in \gamma(a).\ b \text{ is interesting.}$$

For Context:

Definition 8.2. A *valid* abstract domain $(\mathbb{A}, \sqsubseteq)$ with a concretization function $\gamma : \mathbb{A} \to \mathbb{C}$, a representation function $\beta : \mathbb{B} \to \mathbb{A}$, and a set $Exps \subseteq \mathbb{A} \rightharpoonup \mathbb{A}$ of expansion functions satisfies the following constraints:

1. $(\mathbb{A}, \sqsubseteq)$ is partially ordered.
2. $(\mathbb{A}, \sqsubseteq)$ contains no infinite ascending chains.
3. All expansion functions $E \in Exps$ are strictly ascending:
$$\forall a \in \mathsf{dom}(E).\ a \sqsubseteq E(a) \wedge a \neq E(a)$$
4. For each abstract block, the number of applicable expansion functions is finite.
5. For each abstract block, every direct successor in $\sqsubseteq$ is reachable via an expansion function:
$$\forall a, b \in \mathbb{A}.\ \left(a \sqsubset b \wedge \nexists c \in \mathbb{A}.\ a \sqsubset c \sqsubset b\right) \Rightarrow \exists E \in Exps.\ E(a) = b$$
6. The expansion functions are monotone:
$$\forall a, a' \in \mathsf{dom}(E).\ a \sqsubseteq a' \Rightarrow E(a) \sqsubseteq E(a')$$
7. The concretization function is monotone:
$$\forall a, a' \in \mathbb{A}.\ a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a')$$
8. Concretization and representation functions satisfy the soundness constraint:
$$\forall b \in \mathbb{B}.\ b \in \gamma(\beta(b))$$

⌟

With a complete induction over the number of iterations in the algorithm, we can show that *absBB* is interesting in every iteration of the algorithm. For the induction base, the check in line 2 ensures that *absBB* is interesting at the beginning of the initial iteration. In each

iteration, *absBB* is either replaced with an interesting abstract block or remains the same. If it is interesting before an iteration – as the induction hypothesis guarantees – it is therefore also interesting after the iteration. Consequently, the result of the algorithm is also interesting.

Assume that the result a of the generalization is not maximal, i.e., there is a strictly more general abstract block a^* than the result that is interesting. As a^* is more general than a, there is a chain $a \sqsubseteq a' \sqsubseteq \cdots \sqsubseteq a^*$ of immediate successors in the generality order from a to a^*. Each abstract block on this chain is at most as general as a^* and therefore also interesting. Since we assume that each immediate successor in the generality order is reachable through an expansion function (Requirement 5), there is an applicable expansion function E such that $E(a) = a'$. The generalization terminated with the result a, therefore E was rejected in a previous iteration.

The expansion E can only be rejected if its result $E(a_p)$ for a previous value $a_p \sqsubseteq a$ for *absBB* is not interesting, i.e., a basic block b^* in $\gamma(E(a_p))$ is not interesting. Expansions and the concretization function γ are required to be monotone (Requirements 6 and 7), therefore the following holds:

$$a_p \sqsubseteq a \Rightarrow E(a_p) \sqsubseteq E(a) \Rightarrow \gamma(E(a_p)) \subseteq \gamma(E(a))$$

So, the non-interesting basic block b^* is also in the set $\gamma(E(a))$, which means that $E(a)$ is not interesting, a contradiction. Hence, the assumption is wrong: There cannot be a more general interesting abstract block a^*. □

A.5.3. Proof for Theorem 8.6

Theorem 8.6. The presented construction is a valid abstract domain.

For Context:

Definition 8.2. A *valid* abstract domain $(\mathbb{A}, \sqsubseteq)$ with a concretization function $\gamma : \mathbb{A} \to \mathbb{C}$, a representation function $\beta : \mathbb{B} \to \mathbb{A}$, and a set $\mathit{Exps} \subseteq \mathbb{A} \rightharpoonup \mathbb{A}$ of expansion functions satisfies the following constraints:

1. $(\mathbb{A}, \sqsubseteq)$ is partially ordered.
2. $(\mathbb{A}, \sqsubseteq)$ contains no infinite ascending chains.
3. All expansion functions $E \in \mathit{Exps}$ are strictly ascending:
$$\forall a \in \mathsf{dom}(E).\ a \sqsubseteq E(a) \wedge a \neq E(a)$$
4. For each abstract block, the number of applicable expansion functions is finite.
5. For each abstract block, every direct successor in $\sqsubseteq$ is reachable via an expansion function:
$$\forall a, b \in \mathbb{A}.\ \left(a \sqsubset b \wedge \nexists c \in \mathbb{A}.\ a \sqsubset c \sqsubset b\right) \Rightarrow \exists E \in \mathit{Exps}.\ E(a) = b$$
6. The expansion functions are monotone:
$$\forall a, a' \in \mathsf{dom}(E).\ a \sqsubseteq a' \Rightarrow E(a) \sqsubseteq E(a')$$
7. The concretization function is monotone:
$$\forall a, a' \in \mathbb{A}.\ a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a')$$
8. Concretization and representation functions satisfy the soundness constraint:
$$\forall b \in \mathbb{B}.\ b \in \gamma(\beta(b))$$

⌟

The proof for this theorem mostly consists of bookkeeping tasks as the abstract domain was constructed with these requirements in mind. We prove the claims of the theorem in order:

1. *The domain is partially ordered.* We start by noting that each of the feature domains is partially ordered. These domain components are by definition ordered by the reflexive and transitive closure of the adjacency relation of the Hasse diagrams in Table 8.2 on page 118. This adjacency relation is free of cycles since elements with a lower vertical position are only related to elements with a strictly higher vertical position. For this reason, the adjacency relation as well as its reflexive and transitive closure are antisymmetric. Consequently, the feature domains are partially ordered. Since the instruction order $\sqsubseteq_{in}$

(Equation (8.9)) is the product order of the orders of several feature domains, it is also a partial order.

Similarly, the aliasing abstraction (Equation (8.14)) is a partially ordered set:

- The aliasing order $\sqsubseteq_{al}$ is reflexive: Let $h \in \mathbb{A}_{al}$. Then:

$$h \sqsubseteq_{al} h \Leftrightarrow \forall x \in (Idx \times Idx).\ h(x) = \top \vee h(x) = h(x)$$

  Since $h(x) = h(x)$ always holds, this condition is fulfilled.

- The aliasing order $\sqsubseteq_{al}$ is antisymmetric: Let $h, g \in \mathbb{A}_{al}$ such that $h \sqsubseteq_{al} g$ and $h \neq g$. Then:

$$h \sqsubseteq_{al} g \Leftrightarrow \forall x \in (Idx \times Idx).\ g(x) = \top \vee h(x) = g(x)$$

  Since $h \neq g$, there is an $\hat{x} \in (Idx \times Idx)$ such that $h(\hat{x}) \neq g(\hat{x})$. Therefore, $g(\hat{x}) = \top$ and $h(\hat{x}) \neq \top$, which implies that $g \sqsubseteq_{al} h$ cannot hold.

- The aliasing order $\sqsubseteq_{al}$ is transitive: Let $h, g, f \in \mathbb{A}_{al}$ such that $h \sqsubseteq_{al} g$ and $g \sqsubseteq_{al} f$. This means:

$$\begin{aligned}
&\big(\forall x \in (Idx \times Idx).\ g(x) = \top \vee h(x) = g(x)\big) \wedge \\
&\quad \big(\forall x \in (Idx \times Idx).\ f(x) = \top \vee g(x) = f(x)\big) \\
\Rightarrow\ &\forall x \in (Idx \times Idx).\ \Big(\big(g(x) = \top \vee h(x) = g(x)\big) \wedge \big(f(x) = \top \vee g(x) = f(x)\big)\Big) \\
\Rightarrow\ &\forall x \in (Idx \times Idx).\ \Big(\big(f(x) = \top \wedge (g(x) = \top \vee h(x) = g(x))\big) \\
&\qquad \vee \big(g(x) = f(x) \wedge (g(x) = \top \vee h(x) = g(x))\big)\Big) \\
\Rightarrow\ &\forall x \in (Idx \times Idx).\ f(x) = \top \vee \big(g(x) = f(x) \wedge (g(x) = \top \vee h(x) = g(x))\big) \\
\Rightarrow\ &\forall x \in (Idx \times Idx).\ f(x) = \top \vee \big(h(x) = f(x) \vee f(x) = g(x) = \top\big) \\
\Rightarrow\ &\forall x \in (Idx \times Idx).\ f(x) = \top \vee h(x) = f(x) \\
\Rightarrow\ &h \sqsubseteq_{al} f
\end{aligned}$$

We lift this insight to the full abstract domain order (Equation (8.4)):

- The domain order $\sqsubseteq_{\mathbb{A}}$ is reflexive: Let $(a_{in}, a_{al}) \in \mathbb{A}$. Then:

$$\begin{aligned}
&(a_{in}, a_{al}) \sqsubseteq_{\mathbb{A}} (a_{in}, a_{al}) \\
\Leftrightarrow\ &a_{al} \sqsubseteq_{al} a_{al} \wedge |a_{in}| = |a_{in}| \wedge (\forall 1 \leq k \leq |a_{in}|.\ a_{in}[k] \sqsubseteq_{in} a_{in}[k])
\end{aligned}$$

  The first conjunct is true since $\sqsubseteq_{al}$ is reflexive, the second because $=$ is reflexive, and the third because $\sqsubseteq_{in}$ is reflexive.

- The domain order $\sqsubseteq_{\mathbb{A}}$ is antisymmetric: Let $(a_{in}, a_{al}), (b_{in}, b_{al}) \in \mathbb{A}$ be abstract blocks such that $(a_{in}, a_{al}) \sqsubseteq_{\mathbb{A}} (b_{in}, b_{al})$ and $(a_{in}, a_{al}) \neq (b_{in}, b_{al})$. Then:

$$a_{al} \sqsubseteq_{al} b_{al} \wedge |a_{in}| = |b_{in}| \wedge (\forall 1 \leq k \leq |a_{in}|.\ a_{in}[k] \sqsubseteq_{in} b_{in}[k])$$

Since $(a_{in}, a_{al}) \neq (b_{in}, b_{al})$, at least one of $a_{in} \neq b_{in}$ and $a_{al} \neq b_{al}$ holds. As $\sqsubseteq_{in}$ and $\sqsubseteq_{al}$ are both antisymmetric, $b_{al} \sqsubseteq_{al} a_{al}$ or $b_{in}[k] \sqsubseteq_{in} a_{in}[k]$ for some $1 \leq k \leq |a_{in}| = |b_{in}|$ is false. Therefore, $(b_{in}, b_{al}) \not\sqsubseteq_{\mathbb{A}} (a_{in}, a_{al})$.

- The domain order $\sqsubseteq_{\mathbb{A}}$ is transitive: Let $(a_{in}, a_{al}), (b_{in}, b_{al}), (c_{in}, c_{al}) \in \mathbb{A}$ be abstract blocks such that $(a_{in}, a_{al}) \sqsubseteq_{\mathbb{A}} (b_{in}, b_{al})$ and $(b_{in}, b_{al}) \sqsubseteq_{\mathbb{A}} (c_{in}, c_{al})$. Then:

$$
\begin{aligned}
&\big(a_{al} \sqsubseteq_{al} b_{al} \land |a_{in}| = |b_{in}| \land (\forall 1 \leq k \leq |a_{in}|.\ a_{in}[k] \sqsubseteq_{in} b_{in}[k])\big) \land \\
&\quad \big(b_{al} \sqsubseteq_{al} c_{al} \land |b_{in}| = |c_{in}| \land (\forall 1 \leq k \leq |b_{in}|.\ b_{in}[k] \sqsubseteq_{in} c_{in}[k])\big) \\
&\Rightarrow \begin{array}{l} \big(a_{al} \sqsubseteq_{al} b_{al} \sqsubseteq_{al} c_{al}\big) \land \big(|a_{in}| = |b_{in}| = |c_{in}|\big) \\ \quad \land \big(\forall 1 \leq k \leq |a_{in}| = |b_{in}|.\ a_{in}[k] \sqsubseteq_{in} b_{in}[k] \sqsubseteq_{in} c_{in}[k]\big) \end{array} \\
&\Rightarrow \big(a_{al} \sqsubseteq_{al} c_{al}\big) \land \big(|a_{in}| = |c_{in}|\big) \land \big(\forall 1 \leq k \leq |a_{in}|.\ a_{in}[k] \sqsubseteq_{in} c_{in}[k]\big) \\
&\Rightarrow (a_{in}, a_{al}) \sqsubseteq_{\mathbb{A}} (c_{in}, c_{al})
\end{aligned}
$$

2. *The domain contains no infinite ascending chains.* None of the feature domains contains infinite ascending chains by construction:

   - The Singletons domain has only ascending chains with up to two different elements.
   - For the Edit Distances and the Log Sizes domains, the length of ascending chains depends on the chosen value for the parameter K. In both, the longest ascending chains can only have $K + 2$ elements (one for each number in $[0, K]$ plus the $\top$ element).
   - The Subset-or-None domain allows arbitrarily long finite ascending chains: A finite set value M can be expanded $|M| + 1$ times until the $\top$ element is reached. Since we only use finite sets as values in this domain, no infinite ascending chains occur.

   The instruction abstraction is the product order of a configurable but fixed number of feature domains, therefore it does not contain infinite ascending chains either. Neither does the alias abstraction: Let $a_{al} \in \mathbb{A}_{al}$ be a value of the alias abstraction. In the domain construction, we required that any element of the alias abstraction contains only finitely many non-$\top$ entries. An ascending chain starting from a_{al} will set one of the (finitely many) non-$\top$ entries of a_{al} to $\top$ in each step. It can therefore also only be finite.

   An abstract block $a \in \mathbb{A}$ consists of a finite list of instruction abstractions and some abstract aliasing information. In an ascending chain starting from a, each step must increase one of the instruction abstractions or the alias abstraction, each of which can only be done a finite number of times. Every such ascending chain is therefore finite.

3. *The expansion functions are strictly ascending.* Expansion functions for the instruction abstraction (Equation (8.12)) are strictly ascending: They are defined such that one component of the instruction abstraction is replaced by a strictly larger one whereas all other components remain unchanged.

   Similarly, expansion functions E_{al} for the alias abstraction (Equation (8.17)) replace a non-$\top$ entry in the mapping with a $\top$ entry while leaving all other entries unchanged. Therefore, for any $h \in \mathrm{dom}(E_{al})$, $h \sqsubseteq_{al} E_{al}(h)$ and $h \neq E_{al}(h)$ holds.

Expansion functions for abstract basic blocks can have one of two forms (Equation (8.7)):

- $\lambda(a_{in}, a_{al}).\ (a_{in}[k \mapsto E_{in}(a_{in}[k])], a_{al})$ if $k \leq |a_{in}| \wedge a_{in}[k] \in \mathrm{dom}(E_{in})$ for some $k \in \mathbb{N}$ and $E_{in} \in Exps_{in}$ – an instruction abstraction is expanded and all other components are left unchanged.

- $\lambda(a_{in}, a_{al}).\ (a_{in}, E_{al}(a_{al}))$ if $a_{al} \in \mathrm{dom}(E_{al})$ for some $E_{al} \in Exps_{al}$ – the alias abstraction is expanded and all other components are left unchanged.

Since all expansion functions for the instruction and alias abstractions are strictly expanding, basic block expansion functions of either kind are also strictly expanding.

4. *For each abstract block, the number of applicable expansion functions is finite.* Expansions functions that are applicable for an abstract block (a_{in}, a_{al}) can either be expansions for one of the (finitely many) instruction abstractions or for the aliasing abstraction.

   For the aliasing abstraction, one expansion function per non-$\top$ entry in a_{al} is applicable. Since any aliasing abstraction can only have finitely many non-$\top$ entries, the aliasing abstraction can only contribute finitely many applicable expansion functions for the abstract block.

   For an instruction abstraction, there is an applicable expansion function for each successor of a non-$\top$ entry in a feature component. Since an instantiation of the abstract domain uses a finite number of feature components, what remains to be shown is that all elements of each feature domain only have finitely many direct successors in the domain order.

   - For the Singletons, Edit Distances, and Log Sizes domain, each element has either exactly one or zero (for $\top$) direct successors.

   - In the Subset-or-None domain, $\top$ also has no successors, DefNone and {} have one successor, and every other set has one direct successor per element. Since we only use finite sets, they all have a finite number of successors.

5. *For each abstract block, every direct successor in $\sqsubseteq_{\mathbb{A}}$ is reachable via an expansion function.* Let $a, b \in \mathbb{A}$ such that $b = (b_{in}, b_{al})$ is a direct successor of $a = (a_{in}, a_{al})$ in $\sqsubseteq_{\mathbb{A}}$.

   As $a \sqsubseteq b$, the following holds by Equation (8.4):

   $$x_{al} \sqsubseteq_{al} y_{al} \wedge |x_{in}| = |y_{in}| \wedge (\forall 1 \leq k \leq |x_{in}|.\ x_{in}[k] \sqsubseteq_{in} y_{in}[k])$$

   Since they are direct successors, exactly one of the $\sqsubseteq_{al}$ or $\sqsubseteq_{in}$ inequalities is not tight. If they were all tight, a and b would be equal, and if more than two were not tight, adjusting b to tighten one of them would give us an abstract block c between a and b in the order $\sqsubseteq_{\mathbb{A}}$.

   If they differ in the alias abstraction, there needs to be exactly one index pair $x \in Idx \times Idx$ such that $a_{al}(x) \neq b_{al}(x) = \top$ – otherwise, b would not be a direct successor of a, by an

analogous argument as in the previous paragraph. Then, the set *Exps* contains an expansion that expands a to b, based on Equations (8.7) and (8.17):

$$\lambda(a_{in}, a_{al}).\ (a_{in}, E(a_{al})) \text{ if } a_{al} \in \mathrm{dom}(E)$$
$$\text{with } E := \lambda h.\ h[x \mapsto \top] \text{ if } h(x) \neq \top$$

If a and b do not differ in the alias abstraction, they differ in the kth instruction abstraction. Then, they need to differ in the domain for exactly one feature f, such that $b_f := b_{in}[k][f]$ succeeds $a_f := a_{in}[k][f]$ in $\sqsubseteq_{\mathbb{A}_F[f]}$ – otherwise, b would not be a direct successor to a, by an analogous argument as in the previous paragraphs. For this case, the set *Exps* also contains an expansion that expands a to b, based on Equations (8.7) and (8.12):

$$\lambda(a_{in}, a_{al}).\ (a_{in}[k \mapsto E(a_{in}[k])], a_{al}) \text{ if } k \leq |a_{in}| \wedge a_{in}[k] \in \mathrm{dom}(E)$$
$$\text{with } E := \lambda ai.\ ai[f \mapsto b_f] \text{ if } ai[f] = a_f$$

Thus, every possible direct successor of a is covered by an expansion function.

6. *The expansion functions are monotone.* Let E be an expansion function from the basic block abstraction and let $(x_{in}, x_{al}), (y_{in}, y_{al}) \in \mathrm{dom}(E)$ such that $(x_{in}, x_{al}) \sqsubseteq_{\mathbb{A}} (y_{in}, y_{al})$. Let further $(x'_{in}, x'_{al}) := E((x_{in}, x_{al}))$ and $(y'_{in}, y'_{al}) := E((y_{in}, y_{al}))$. From the definition of the ordering relation (Equation (8.4)) follows:

$$x_{al} \sqsubseteq_{al} y_{al} \wedge |x_{in}| = |y_{in}| \wedge (\forall 1 \leq k \leq |x_{in}|.\ x_{in}[k] \sqsubseteq_{in} y_{in}[k])$$

The expansion function E can either be derived from an expansion function for the aliasing abstraction or from one for an instruction abstraction.

- If it concerns the aliasing abstraction, E only affects the aliasing information $x_{al}(i)$ and $y_{al}(i)$ for an index pair $i \in Idx \times Idx$ and is applicable if $x_{al}(i) \neq \top$ and $y_{al}(i) \neq \top$. The resulting expanded alias abstractions are $x'_{al} := x_{al}[i \mapsto \top]$ and $y'_{al} := y_{al}[i \mapsto \top]$. Only the entries for the index pair i are replaced with $\top$ values, all other entries remain the same. Therefore, $x'_{al} \sqsubseteq_{al} y'_{al}$ holds. As the instruction abstractions $x'_{in} = x_{in}$ and $y'_{in} = y_{in}$ are unchanged, $(x'_{in}, x'_{al}) \sqsubseteq_{\mathbb{A}} (y'_{in}, y'_{al})$ holds.
- If E concerns the instruction abstraction, it does not affect the aliasing abstraction, therefore $x'_{al} = x_{al} \sqsubseteq_{al} y_{al} = y'_{al}$ holds. Since instruction expansions do not affect the length of the list of instruction abstractions, $|x'_{in}| = |x_{in}| = |y_{in}| = |y'_{in}|$ also holds. E can only affect the instruction abstractions x_{in} and y_{in} at a single common position k^*, all others stay unchanged:

  $$\forall k \in \{1, \ldots, |x_{in}|\} \setminus \{k^*\}.\ x'_{in}[k] = x_{in}[k] \sqsubseteq_{in} y_{in}[k] = y'_{in}[k]$$

  What remains to be shown is that the affected entries $x'_{in}[k^*]$ and $y'_{in}[k^*]$ remain properly ordered: $x'_{in}[k^*] \sqsubseteq_{in} y'_{in}[k^*]$.

  By construction (Equation (8.12)), E only affects a single feature f in the instruction abstractions by applying an expansion function of the feature domain. As each such

expansion function only applies to a specific value v in the feature domain and replaces it with a specific direct successor v', we can conclude that $x_{in}[k^*](f) = v = y_{in}[k^*](f)$ and $x'_{in}[k^*](f) = v' = y'_{in}[k^*](f)$. Therefore, $x'_{in}[k^*] = y'_{in}[k^*]$ holds, which implies $x'_{in}[k^*] \sqsubseteq_{in} y'_{in}[k^*]$.

7. *The concretization function is monotone.* Let $(x_{in}, x_{al}), (y_{in}, y_{al}) \in \mathbb{A}$ such that $(x_{in}, x_{al}) \sqsubseteq_{\mathbb{A}} (y_{in}, y_{al})$. From the definition of the ordering relation (Equation (8.4)) follows:

$$x_{al} \sqsubseteq_{al} y_{al} \wedge |x_{in}| = |y_{in}| \wedge (\forall 1 \leq k \leq |x_{in}|.\ x_{in}[k] \sqsubseteq_{in} y_{in}[k]) \tag{A.12}$$

Let further $X := \gamma_{\mathbb{A}}((x_{in}, x_{al}))$ and $Y := \gamma_{\mathbb{A}}((y_{in}, y_{al}))$. We need to show that $X \subseteq Y$, i.e., any basic block $b \in X$ is also in Y.

Let $b \in X$. By the definition of $\gamma_{\mathbb{A}}$ (Equation (8.5)), the following holds:

$$b \in \gamma_{al}(x_{al}) \wedge |x_{in}| = |b| \wedge (\forall 1 \leq k \leq |x_{in}|.\ b[k] \in \gamma_{in}(x_{in}[k])) \tag{A.13}$$

We need to show that

$$b \in \gamma_{al}(y_{al}) \wedge |y_{in}| = |b| \wedge (\forall 1 \leq k \leq |y_{in}|.\ b[k] \in \gamma_{in}(y_{in}[k])) \tag{A.14}$$

We consider each conjunct individually:

- Since $b \in \gamma_{al}(x_{al})$ by Equation (A.13), Equation (8.15) ensures the following:

$$\bigwedge_{((i_1,i_2)\mapsto v)\in x_{al}} \Big(P(b, i_i, i_2) \Rightarrow \Big(v = \top \vee Q(b, i_1, i_2, v)\Big)\Big)$$

  with

$$P(b, i_i, i_2) := \big(b[i_1] \text{ and } b[i_2] \text{ exist and match}\big)$$

  and

$$\begin{aligned} Q(b, i_1, i_2, v) := &\big(v = \texttt{must} \wedge \big(b[i_1] \text{ and } b[i_2] \text{ alias}\big)\big) \\ &\vee \big(v = \texttt{mustnot} \wedge \big(b[i_1] \text{ and } b[i_2] \text{ do not alias}\big)\big) \end{aligned}$$

  As $x_{al} \sqsubseteq_{al} y_{al}$, we know that $((i_1, i_2) \mapsto v) \in x_{al}$ implies that $y_{al}((i_1, i_2)) \in \{v, \top\}$. Therefore, whenever $(v = \top \vee Q(b, i_1, i_2, v))$ holds for an entry $((i_1, i_2) \mapsto v) \in x_{al}$, then $\big(v' = \top \vee Q(b, i_1, i_2, v')\big)$ with $v' := y_{al}((i_1, i_2))$ also holds. Hence, the constraint from the definition of γ_{al} (Equation (8.15)) is fulfilled for y_{al}:

$$\bigwedge_{((i_1,i_2)\mapsto v)\in y_{al}} \Big(P(b, i_i, i_2) \Rightarrow \Big(v = \top \vee Q(b, i_1, i_2, v)\Big)\Big)$$

  Consequently, $b \in \gamma_{al}(y_{al})$.

- From Equation (A.12) and Equation (A.13), we learn that $|y_{in}| = |x_{in}| = |b|$.

- Let $k \in \{1, ..., |y_{in}|\}$. Equation (A.12) implies that $x_{in}[k] \sqsubseteq_{in} y_{in}[k]$. From Equation (A.13), we further know that the kth instruction $b[k]$ of b is in $\gamma_{in}(x_{in}[k])$. We need to show that $b \in \gamma_{in}(y_{in}[k])$. Unfolding the definitions of the instruction concretization γ_{in} (Equation (8.10)) and the corresponding order $\sqsubseteq_{in}$ (Equation (8.9)) reduces the proof obligation to showing that the concretization function of each feature domain is monotone. Let av_1, av_2 be a pair of elements of a feature domain X such that $av_1 \sqsubseteq_F av_2$. We first consider two cases independent of the specific feature domain:

  - If $av_1 = av_2$, $\gamma_X^{(f)}(av_1) = \gamma_X^{(f)}(av_2)$ needs to hold as well since $\gamma_X^{(f)}$ is a function.
  - If $av_2 = \top$, $\gamma_X^{(f)}(av_1) \subseteq \gamma_X^{(f)}(av_2) = \mathbf{I}$ necessarily holds.

  Otherwise:

  Singletons: No case other than the above two is possible.

  Edit Distances: The values need to follow the form $av_1 = (B : s, d : n)$ and $av_2 = (B : s, d : m)$ with editing distances $n < m$ and a common base string s.
  Let $i \in \gamma_X^{(f)}(av_1) = \{i \mid dist(f(i), s) \leq n\}$. Then, the following holds:

  $$dist(f(i), s) \leq n < m$$

  Therefore, $i \in \{i \mid dist(f(i), s) \leq m\} = \gamma_X^{(f)}(av_2)$. Thus, $\gamma_X^{(f)}(av_1) \subseteq \gamma_X^{(f)}(av_2)$ holds.

  Log Sizes: With the above two cases eliminated, $av_1 < av_2$ must hold.
  Let $i \in \gamma_X^{(f)}(av_1) = \{i \mid |f(i)| < 2^{av_1}\}$. Then, $|f(i)| < 2^{av_1} < 2^{av_2}$ must hold.
  Therefore, $i \in \{i \mid |f(i)| < 2^{av_2}\} = \gamma_X^{(f)}(av_2)$. Thus, $\gamma_X^{(f)}(av_1) \subseteq \gamma_X^{(f)}(av_2)$ holds.

  Subset-or-None: Any case involving DefNone or $\top$ is already captured by the above two cases. Therefore, $av_1 \supsetneq av_2$ needs to hold.
  Let $i \in \gamma_X^{(f)}(av_1) = \{i \mid f(i) \supseteq av_1\}$. Then, $f(i) \supseteq av_1 \supsetneq av_2$ must holds.
  Therefore, $i \in \{i \mid f(i) \supseteq av_2\} = \gamma_X^{(f)}(av_2)$. Thus, $\gamma_X^{(f)}(av_1) \subseteq \gamma_X^{(f)}(av_2)$ holds.

  The instruction concretization function is therefore monotone.

  Overall, this shows that the concretization function $\gamma_{\mathbb{A}}$ is monotone.

8. *Concretization and representation functions satisfy the soundness constraint.* Let $b \in \mathbb{B}$ be a basic block. We need to show that $b \in \gamma_{\mathbb{A}}(\beta_{\mathbb{A}}(b))$. By the definitions of $\gamma_{\mathbb{A}}$ (Equation (8.5)) and $\beta_{\mathbb{A}}$ (Equation (8.6)), the following therefore needs to hold:

   $$b \in \gamma_{al}(\beta_{al}(b)) \wedge |b| = |b| \wedge \left(\forall 1 \leq k \leq |b|.\ b[k] \in \gamma_{in}(\beta_{in}(b[k]))\right) \tag{A.15}$$

   The length constraint on the lists of instructions and instruction abstractions is trivially fulfilled. This leaves us with two proof obligations:

- $b \in \gamma_{al}(\beta_{al}(b))$: By the definition of γ_{al} (Equation (8.15)), this is equivalent to:

$$\begin{aligned}\bigwedge_{((i_1,i_2)\mapsto x)\in\beta_{al}(b)} \Big(&\big(b[i_1] \text{ and } b[i_2] \text{ exist and match}\big) \\ &\Rightarrow \Big(x = \top \vee \big(x = \texttt{must} \wedge \big(b[i_1] \text{ and } b[i_2] \text{ alias}\big)\big) \\ &\qquad \vee \big(x = \texttt{mustnot} \wedge \big(b[i_1] \text{ and } b[i_2] \text{ do not alias}\big)\big)\Big)\Big)\end{aligned} \tag{A.16}$$

  $\beta_{al}(b)$ is defined as follows by Equation (8.16):

$$\beta_{al}(b) := \lambda(i_1, i_2).\begin{cases}\texttt{must} & \text{if } b[i_1], b[i_2] \text{ exist, match, and alias} \\ \texttt{mustnot} & \text{if } b[i_1], b[i_2] \text{ exist, match, and do not alias} \\ \top & \text{otherwise}\end{cases}$$

  We consider the conjuncts of Equation (A.16) individually, let $((i_1, i_2) \mapsto x) \in \beta_{al}(b)$.

  - If $b[i_1], b[i_2]$ do not exist or match, the premise of the conjunct's implication is not satisfied, the conjunct is True.
  - If $b[i_1], b[i_2]$ exist, match, and alias, $\beta_{al}(b)(i_1, i_2)$ is "`must`". Then, the second disjunct of the implication's consequent is satisfied.
  - Otherwise, $b[i_1], b[i_2]$ exist, match, but do not alias and $\beta_{al}(b)(i_1, i_2)$ is "`mustnot`". Then, the third disjunct of the implication's consequent is satisfied.

  All conjuncts of Equation (A.16) are therefore satisfied: $b \in \gamma_{al}(\beta_{al}(b))$ holds.

- $\big(\forall 1 \le k \le |b|.\ b[k] \in \gamma_{in}(\beta_{in}(b[k]))\big)$: Let $k \in \{1, \ldots, |b|\}$. Using the definitions for γ_{in} and β_{in} (Equations (8.10) and (8.11)) transforms our proof obligation as follows:

$$b[k] \in \bigcap_{f \in Features} \gamma^{(f)}_{\mathbb{A}_F[f]}\left(\beta^{(f)}_{\mathbb{A}_F[f]}(b[k])\right)$$

  We therefore need to show that every feature domain X is sound for any instruction feature f, i.e.,

$$\forall i \in \mathbf{I}.\ i \in \gamma^{(f)}_X\left(\beta^{(f)}_X(i)\right)$$

  Let $i \in \mathbf{I}$ be an instruction scheme and let $f(i)$ be its feature value for the feature f.

  Singletons:

$$\begin{aligned} & i \in \gamma^{(f)}_X\left(\beta^{(f)}_X(i)\right) & \\ \Leftrightarrow\ & i \in \gamma^{(f)}_X\left(f(i)\right) & \text{Def. } \beta^{(f)}_X\text{, Table 8.2} \\ \Leftrightarrow\ & i \in \left\{j \;\middle|\; f(j) = f(i)\right\} & \text{Def. } \gamma^{(f)}_X\text{, Table 8.2} \\ \Leftrightarrow\ & \text{True} & \end{aligned}$$

Edit Distances:

$$
\begin{aligned}
& i \in \gamma_X^{(f)}\left(\beta_X^{(f)}(i)\right) \\
\Leftrightarrow\quad & i \in \gamma_X^{(f)}\left((B : f(i), d : 0)\right) && \text{Def. } \beta_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow\quad & i \in \left\{j \;\middle|\; dist(f(j), f(i)) \leq 0\right\} && \text{Def. } \gamma_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow\quad & i \in \left\{j \;\middle|\; f(j) = f(i)\right\} \\
\Leftrightarrow\quad & \text{True}
\end{aligned}
$$

Log Sizes: If $f(i) = \emptyset$:

$$
\begin{aligned}
& i \in \gamma_X^{(f)}\left(\beta_X^{(f)}(i)\right) \\
\Leftrightarrow\quad & i \in \gamma_X^{(f)}\left(0\right) && \text{Def. } \beta_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow\quad & i \in \left\{j \;\middle|\; |f(j)| < 2^0\right\} && \text{Def. } \gamma_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow\quad & i \in \left\{j \;\middle|\; |f(j)| < 1\right\} \\
\Leftrightarrow\quad & i \in \left\{j \;\middle|\; f(j) = \emptyset\right\} \\
\Leftrightarrow\quad & \text{True} && \text{by assumption}
\end{aligned}
$$

If $|f(i)| \geq 2^K$:

$$
\begin{aligned}
& i \in \gamma_X^{(f)}\left(\beta_X^{(f)}(i)\right) \\
\Leftrightarrow\quad & i \in \gamma_X^{(f)}\left(\top\right) && \text{Def. } \beta_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow\quad & i \in \mathbf{I} && \text{Def. } \gamma_X^{(f)} \\
\Leftrightarrow\quad & \text{True}
\end{aligned}
$$

Otherwise:

$$
\begin{aligned}
& i \in \gamma_X^{(f)}\left(\beta_X^{(f)}(i)\right) \\
\Leftrightarrow\quad & i \in \gamma_X^{(f)}\left(\left\lfloor \log_2(|f(i)|) + 1 \right\rfloor\right) && \text{Def. } \beta_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow\quad & i \in \left\{j \;\middle|\; |f(j)| < 2^{\left\lfloor \log_2(|f(i)|)+1 \right\rfloor}\right\} && \text{Def. } \gamma_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow\quad & |f(i)| < 2^{\left\lfloor \log_2(|f(i)|)+1 \right\rfloor} \\
\Leftrightarrow\quad & \log_2(|f(i)|) < \left\lfloor \log_2(|f(i)|) + 1 \right\rfloor && \log_2 \text{ is strictly increasing, } f(i) \neq \emptyset \\
\Leftrightarrow\quad & \text{True} && \text{property of } \lfloor \cdot \rfloor
\end{aligned}
$$

Subset-or-None: If $f(i) = \emptyset$:

$$
\begin{aligned}
& i \in \gamma_X^{(f)}\left(\beta_X^{(f)}(i)\right) & \\
\Leftrightarrow \quad & i \in \gamma_X^{(f)}\,(\mathsf{DefNone}) & \text{Def. } \beta_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow \quad & i \in \left\{j \;\middle|\; f(j) = \emptyset\right\} & \text{Def. } \gamma_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow \quad & f(i) = \emptyset & \\
\Leftrightarrow \quad & \mathsf{True} & \text{by assumption}
\end{aligned}
$$

If $f(i) \neq \emptyset$:

$$
\begin{aligned}
& i \in \gamma_X^{(f)}\left(\beta_X^{(f)}(i)\right) & \\
\Leftrightarrow \quad & i \in \gamma_X^{(f)}\,(f(i)) & \text{Def. } \beta_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow \quad & i \in \left\{j \;\middle|\; f(j) \supseteq f(i)\right\} & \text{Def. } \gamma_X^{(f)}\text{, Table 8.2} \\
\Leftrightarrow \quad & \mathsf{True} &
\end{aligned}
$$

Therefore, all feature domains and, consequently, the instruction abstraction are sound.

□

Appendix B

SMT and (I)LP Solving Terminology

Satisfiability modulo theories and (integer) linear programming are important concepts for this dissertation. This appendix collects the related terminology that we use throughout the thesis. We refer to the textbooks by Bertsimas and Tsitsiklis (1997) and Biere *et al.* (2009) for more background on linear optimization and satisfiability modulo theories.

Linear Programming

A *linear program* – or short: *LP* – consists of an *objective function* and a set of *constraints*. Both are formulated in terms of real-valued *variables*. Variables may only occur linearly in the objective and the constraints. Therefore, each involved term is only a sum of constants and variables with a constant coefficient.

For instance, consider the following linear program:

$$\begin{aligned} \text{minimize} \quad & x + y \\ \text{subject to} \quad & 3 \cdot x - 4 \geq y \\ & y \geq 0 \end{aligned}$$

The objective of this LP is to minimize the value of the sum $x + y$ under the constraints $3 \cdot x - 4 \geq y$ and $y \geq 0$ with the variables x and y.

A *feasible solution* to a linear program assigns values to the variables that do not violate any of the LP's constraints. An *optimal solution* to a linear program is a feasible solution for which the value of the objective function – the *objective value* – is minimal (or maximal, depending on the objective formulation). For the above example, $\{x \mapsto 3, y \mapsto 1\}$ is a feasible solution with the objective value 4. A better, and indeed optimal, solution with the objective value $4/3$ is $\{x \mapsto 4/3, y \mapsto 0\}$. We refer to the value that a solution s assigns to a variable v as $s[v]$.

In general, linear programs could have one or more optimal solutions, their objective value could be unbounded, or they may be infeasible, i.e., have no feasible solutions at all. An *LP solver* is a program that determines which of these cases applies for a given linear program and that finds an optimal solution if one exists. There are polynomial-time algorithms (Bertsimas and Tsitsiklis, 1997, Chapter 8) and practical implementations for solving linear programs.

Integer Linear Programming

An *integer linear program* – or short: *ILP* – differs from a linear program in the requirement that the variables need to be integer-valued. If we interpret the above example linear program

as an ILP, $\{x \mapsto 4/3, y \mapsto 0\}$ is no longer a feasible solution since the value for x is not an integer. An optimal solution to this ILP is $\{x \mapsto 2, y \mapsto 0\}$. In a *mixed-integer linear program* (*MILP*), only a subset of the variables is required to be integer-valued.

Solving ILPs and MILPs is NP-complete – a simplified variant is one of Karp's NP-complete problems (Karp, 1972) – but modern (M)ILP solvers can nevertheless solve instances of considerable size.

Satisfiability Modulo Theories

Satisfiability modulo theories – or short: *SMT* – is the problem of determining if a *formula* in first-order logic that involves some predicates and functions with defined meanings is satisfiable. These predicates and functions are grouped in *theories*. The constraints of a linear program can be considered as a conjunction of atomic formulas in the *linear real arithmetic* (*LRA*) theory. The constraints of mixed-integer linear programs require the *linear integer and real arithmetic* (*LIRA*) theory.

SMT formulas can further involve boolean variables and arbitrary boolean operators. Typically, there is no objective function for optimization in SMT solving.[1]

For instance, the following formula constrains a variable y depending on another variable x:

$$\big((x \geq 0) \rightarrow (y = x + 1)\big) \\ \wedge\big(\neg(x \geq 0) \rightarrow (y = 0)\big)$$

If the value of x is greater than or equal to 0, $y = x + 1$ must hold. Otherwise, $y = 0$ must hold.

The terminology of satisfiability modulo theories differs from the one used for integer linear programming: If an assignment of variables to values exists that does not render the formula contradictory, the formula is *satisfiable*. Such an assignment is called a *model*. Otherwise, the formula is *unsatisfiable*.

The above example formula is satisfiable, for instance with the models $\{x \mapsto 1, y \mapsto 2\}$ or $\{x \mapsto -3, y \mapsto 0\}$. If we extend the example formula with the conjunct $\wedge(y \leq x)$, the resulting formula is unsatisfiable.

The computational complexity and decidability of satisfiability modulo theories depends on the involved theories and the use of quantifiers. If only boolean variables and operators are involved, checking for satisfiability is NP-complete (Garey and Johnson, 1990, Section 2.6). With linear theories and quantifiers, the problem is at least exponential (Fischer and Rabin, 1998), and with non-linear integer theories, it is undecidable (Matiyasevich, 1993). For decidable theories, SMT solver implementations are available that check if a given formula is satisfiable and that construct a model if this is the case.

[1] The Z3 SMT solver (de Moura and Bjørner, 2008) does however support optimization objectives.

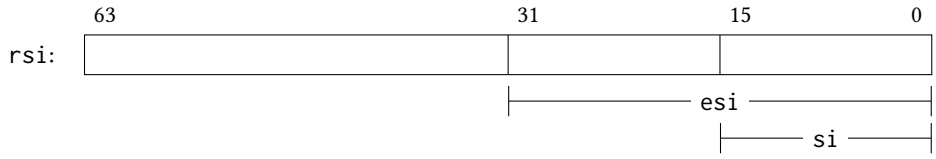
Appendix C

x86-64 Register Reference

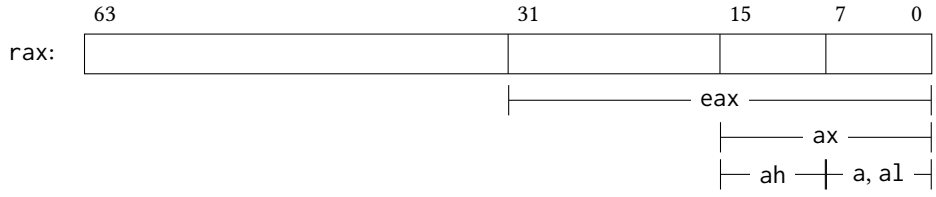
This reference summarizes the registers and the structure of memory operands in the x86-64 instruction set architecture that we use in examples throughout the thesis. It is based on Sections 3.4, 3.7, and 14.1 of the Intel 64 and IA-32 Architectures Software Developer's Manual (Intel, 2023b). Intricacies that are not relevant for this thesis are omitted.

General-Purpose Registers

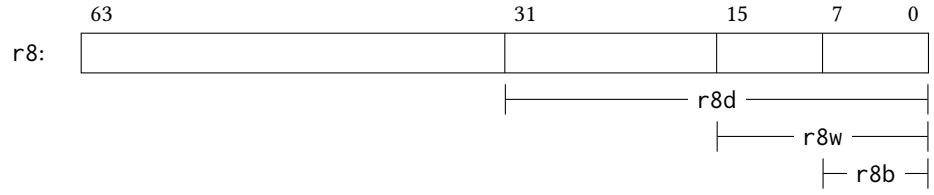
The x86-64 ISA provides 16 general-purpose registers (GPRs) with a width of 64 bits: `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rbp`, `rsp`, and `r8` – `r15`. Instructions may operate on smaller parts of these 64-bit registers through names of subregisters. For `rsi`, `rdi`, `rbp`, and `rsp`, the names for the least significant 32 and 16 bits follow the following scheme:



The `rax`, `rbx`, `rcx`, and `rdx` registers provide additional names for the two least significant bytes:



For `r8` – `r15`, the least significant 32, 16, and 8 bits can be accessed individually:



Vector and Floating-Point Registers

The x86-64 ISA includes a variety of instruction set extensions and related registers for vector and floating-point operations. For this thesis, we consider only the AVX and AVX 2 ISA extensions. They provide 16 registers that are 256 bits wide, named `ymm0` – `ymm15`. The 128-bit-wide lower halves can be accessed via the names `xmm0` – `xmm15`.

Flag Registers

x86-64 further includes a range of 1-bit flag registers that are set by arithmetic operations and that are, e.g., read by conditional branch and move instructions:

CF: The carry flag is set in case an operation's result is out of range if it is treated as unsigned integer.

OF: The overflow flag is set in case an operation's result is out of range if it is treated as signed integer.

AF: The auxiliary carry flag is set in case an operation's result is out of range if it is treated as binary-coded decimal.

ZF: The zero flag is set if an operation produces zero as its result.

SF: The sign flag is set to the most significant bit of an operation's result, i.e., the sign bit of a signed integer.

PF: The parity flag indicates if the least significant byte of an operation's result contains an even number of set bits.

The direction flag register **DF** also counts towards the flag registers, but plays a different role. Its value controls if string instructions increment or decrement their address register.

Memory Operands

The x86-64 ISA provides complex addressing modes for references to the system's main memory. In this thesis we consider memory operands of the following form:

$$[segment : base + scale * index + displacement]$$

There are restrictions on each component:

- *base* and *index* are general-purpose registers.
- *scale* can be one of the constants 2, 4, or 8.
- *displacement* is a constant.
- *segment* is a segment register mainly used in legacy execution modes.

Most of these components are optional.

In the Intel assembly syntax that we use for the examples in this thesis, memory operands are prefixed with descriptors that indicate their access width:

- `byte ptr` for 8-bit accesses (1 byte)
- `word ptr` for 16-bit accesses (2 bytes)
- `dword ptr` for 32-bit accesses (4 bytes)
- `qword ptr` for 64-bit accesses (8 bytes)
- `xmmword ptr` for 128-bit accesses (16 bytes)
- `ymmword ptr` for 256-bit accesses (32 bytes)

Consider the following examples:

- The following instruction instance reads and writes 64 bits (= 8 bytes) from the memory location determined by the *base* register `rbx`.

```
add qword ptr [rbx], 42
```

- The following instruction instance reads 128 bits (= 16 bytes) from the memory location determined by the sum of the *base* register `rax`, the *index* register `rdx`, which is multiplied by the *scale* factor 4, and the constant *displacement* 8.

```
vaddpd xmm3, xmm2, xmmword ptr [rax + 4*rdx + 8]
```

Appendix D

Measuring Basic Block Throughput

Accurate throughput measurements for basic blocks are important for our port mapping inference methods. For this purpose, we designed a throughput measurement infrastructure with a focus on portability, building upon the measurement method used for PMEvo (Ritter and Hack, 2020) and the Bachelor's thesis of Weis (2019).

As laid out in Section 3.2.1, our benchmarking tool emits the to-be-benchmarked instruction sequence within a loop as inline assembly in a C program frame. The resulting C program is compiled into a binary that performs several repetitions of the benchmark when executed. It reports the observed execution rates based on time measurements and the processor's clock frequency.

Alternative approaches like nanoBench (Abel and Reineke, 2020) and the BHive test harness (Chen *et al.*, 2019) consist of a C program with assembly components that is compiled once and modifies its own code when executed to insert instructions that are passed via command line parameters. Compared to them, our approach is slower, since every new benchmark requires invoking a C compiler. It is however also more easily portable and maintainable as no complex self-modifying binaries are involved and we do not directly use hardware-specific performance counters. Since our framework only uses widely available Linux and POSIX features, it is applicable to a wide variety of hardware platforms.

Figure D.1 shows a simplified version of the C benchmarking frame. The measurement tool instantiates this frame with the investigated assembly instructions (inserted in line 8) and values for configurable parameters (represented as identifiers in capital letters in Figure D.1) for each benchmark. The resulting program is compiled with a C compiler for the platform under test and executed.

The core measurement procedure (lines 1–12) wraps the benchmarked code in a loop that runs for a given number `num_iters` of iterations. Before each measurement, a clean memory and register state is established (lines 2, 5) to allow reproducible measurements. The loop itself uses a general-purpose register (`r15`) as a loop counter that may not be used by the benchmarked basic block. Two calls to the POSIX `clock_gettime` function (Linux Programmer's Manual, 2020) with a monotonic clock wrap the loop to measure its execution time.

The benchmarking process needs to handle arbitrary instruction sequences of varying length fully automatically. For any benchmarked instruction sequence, the number of iterations of the benchmarking loop needs to be high enough for stable measurements, but not excessively high to ensure reasonable overall benchmarking times. For this reason and in contrast to nanoBench and the BHive test harness, our implementation is parameterized by a target time

```
1  double measure(int num_iters) {
2    initialize_memory();
3    double start_time = gettime();
4    asm volatile(
5      ... // initialize registers, align loop
6      "mov r15, [num_iters]\n"
7      "loop:\n"
8      ... // benchmarked code is inserted HERE
9      "dec r15; jnz loop\n"
10     : ... ); // clobber all registers and memory
11   return gettime() - start_time;
12 }
13
14 int main(void) {
15   allocate_memory();
16   for (int rep = 0; rep < NUM_SAMPLES; ++rep) {
17     // determine number of iterations and clock frequency
18     double prerun_time = measure(PRERUN_ITERATIONS);
19     int num_iters = (PRERUN_ITERATIONS * TARGET_TIME) / prerun_time;
20     double freq_before = getfreq();
21     // actual measurements
22     double time = measure(2 * num_iters) - measure(num_iters);
23     // check if the clock frequency was unstable
24     double freq_after = getfreq();
25     double freq = (freq_before + freq_after) / 2;
26     if (abs(freq_before - freq_after) > FREQ_THRESHOLD * freq) {
27       print("measurement discarded");
28     } else {
29       print((time * freq) / num_iters); // return cycles per iteration
30     }
31   }
32 }
```

Figure D.1. Schematic benchmarking frame, in C with inline assembly.

for which the benchmarked code should approximately run (configured via the `TARGET_TIME` parameter), rather than the number of loop iterations. Our tool automatically estimates a suitable number `num_iters` of loop iterations based on the time required for a short pre-run (lines 18, 19) with a fixed number of iterations (configured via `PRERUN_ITERATIONS`, typically 1,000). In experiments with the processors evaluated for this thesis, we obtained stable results with 1 ms as target for the individual benchmarking time. This parameter value is therefore used in the benchmarks for this thesis.

For the actual measurements, we follow an approach similar to nanoBench and measure the difference between `num_iters` and $2 \cdot$ `num_iters` iterations. The assumption is that non-steady-state parts of the execution of the benchmark occur in the beginning and the end of both runs in equal magnitude, so that these parts cancel out in the difference of the execution times.

We need to determine the processor's clock frequency to convert the measured execution time into clock cycles. Therefore, the clock frequency is read directly before and after the benchmark code via the Linux kernel's CPUFreq subsystem (Wysocki, 2017). If the difference between the frequency readings exceeds a threshold (we typically use 1% of the arithmetic mean of the readings), the results are discarded (lines 26–27). Otherwise, we consider the arithmetic mean of the readings as the processor's clock frequency.

Frequency scaling features of the processor should be disabled as far as possible to obtain reliable results. For systems with unreliable frequencies like AMD's Zen processors, we add additional `measure(num_iters)` calls before the first frequency measurement (line 20) and before the actual measurements (line 22) as warm-up runs.

Lastly, we compute the measured inverse throughput $tp^{-1}(e)$ of the experiment with the following formula:

$$tp^{-1}(e) = \frac{\text{measured time} \times \text{frequency}}{\#\text{executed copies of } e}$$

The benchmarking binary outputs the observed inverse throughput for multiple (configured via the `NUM_SAMPLES` parameter) such measurements. We take the median of the non-discarded measurement samples to accommodate for occasional fluctuations in the processor's clock frequency.

This measurement method is most effective when applied to unrolled basic blocks as described in Section 3.2.1. Longer instruction sequences in the loop reduce the relative overhead of the loop's decrement and branch operations. We typically perform measurements with a range of unroll factors that yield approximately 40, 80, and 200 instructions in the code inserted into the program frame. Among these, we select the minimal observed number of cycles per experiment instance as the inverse throughput.

This measurement method comes with limitations:

- The benchmarked code may not use the `r15` register, since it is reserved for the loop iteration counter. The instruction scheme instantiation strategy in Section 3.2.1 can ensure this in the benchmarks we perform for port mapping inference.
- Only a fixed range of memory addresses may be accessed. Unless the benchmarked code is crafted very carefully, this precludes memory accesses with addresses that are computed in the benchmarked instruction sequence. The instantiation strategy in Section 3.2.1 ensures that our benchmarks fulfill this requirement as long as no (unbalanced) `push`/`pop` operations or other instructions with hard-coded or changing memory operands are involved. The framework could be extended to automatically record faulting addresses in a page fault handler and map them into valid address ranges, similar to the BHive test harness.
- System instructions are not supported since our tool does not operate in kernel mode. Our tool could support such instructions with an extension that transfers the benchmarking binary to a custom kernel module for execution, similar to nanoBench's kernel module for the same purpose. We leave this extension for future work.

Appendix E

Supplementary Examples

E.1. Number of Blocking Instructions in the uops.info Algorithm

The uops.info port mapping inference algorithm, as described by Abel and Reineke (2019, Algorithm 1), combines an instruction *instr* with $8 \cdot \mathit{maxLatency}(\mathit{instr})$ blocking instructions B (where 8 is the maximal number of ports in any considered microarchitecture). The intention is that, when executing the resulting experiment with an optimal scheduler, µops of *instr* are only executed on ports that are blocked by B if they cannot be executed anywhere else.[1] Here, we give a counter example consisting of a (contrived) microarchitecture and experiments that shows that this number of blocking instructions is insufficient in general. This counter example was confirmed by the authors of uops.info. Their software implementation does not share this issue.

Consider a microarchitecture for five instructions, `I`, `B1`, `B01`, `B12`, `B02` and three ports `P0`, `P1`, `P2`. It uses the following port mapping:

`I`: $1 \times \{\mathtt{P1}\} + 1 \times \{\mathtt{P0}, \mathtt{P1}\} + 4 \times \{\mathtt{P0}, \mathtt{P2}\}$

`B1`: $1 \times \{\mathtt{P1}\}$

`B01`: $1 \times \{\mathtt{P0}, \mathtt{P1}\}$

`B02`: $1 \times \{\mathtt{P0}, \mathtt{P2}\}$

`B12`: $1 \times \{\mathtt{P1}, \mathtt{P2}\}$

All `B` instructions are blocking instructions, each with a latency of 1 cycle. Instruction `I` has a latency of 2 cycles, utilizing all ports for 2 cycles if it is executed on its own as shown in Figure E.1.

Since our microarchitecture has three ports and the maximum latency of the instructions is two cycles, the experiments for the uops.info algorithm contain six blocking instructions. For the blocking instruction `B1`, this leads to an execution like the one shown in Figure E.2. Since we can observe that seven µops are executed on the blocked port `P1`, the algorithm correctly concludes that $1 \times \{\mathtt{P1}\}$ is part of the port usage of `I`. With the blocking instruction `B12`, we obtain executions like the one displayed in Figure E.3. Here, the performance counters observe a total of eight µops on the blocked ports `P1` and `P2`, leading to the wrong conclusion that a $\{\mathtt{P1}, \mathtt{P2}\}$ µop is part of the port usage. This happens because one $\{\mathtt{P0}, \mathtt{P2}\}$ µop is executed on the blocked port `P2` even though it could be executed on `P2`. The six instances of the blocking instruction are however not enough to force it to this unblocked port.

[1] See the paper or Section 6.1 of this thesis for more background.

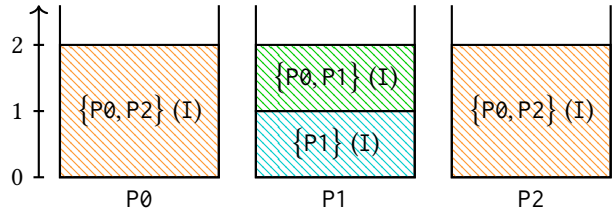


Figure E.1. An optimal port utilization when executing the experiment $\{\texttt{I} \mapsto 1\}$.

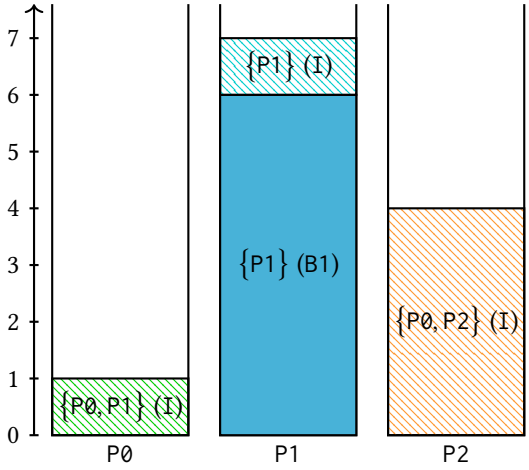


Figure E.2. An optimal port utilization when executing the experiment $\{\texttt{I} \mapsto 1, \texttt{B1} \mapsto 6\}$.

E.2. Facile's Port Mapping Simulation Algorithm

Abel *et al.* (2023) describe a variation of the bottleneck simulation algorithm from Section 5.1.4 (first presented in our work on PMEvo (Ritter and Hack, 2020)) as a component of their fast throughput predictor Facile. Reconsider the characterization of an experiment e's inverse throughput with a port mapping M that our algorithm computes:

$$tp_M^{-1}(e) = \max_{Q \subseteq \mathbf{P}} \frac{\sum\{e(i) \mid M[i] \subseteq Q\}}{|Q|}$$

Their approach differs from ours as follows:

> Rather than considering every port combination that some subset of a benchmark's µops can be dispatched to, we heuristically consider only port combinations required by pairs of µops.
>
> — Abel *et al.* (2023, Section 4.8)

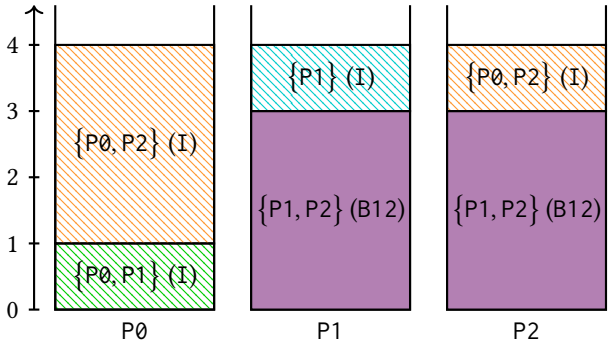
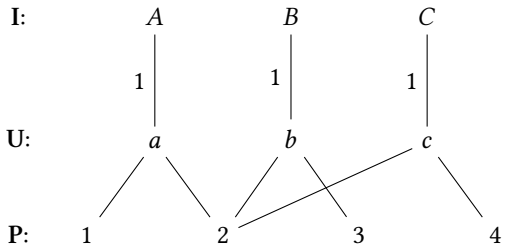
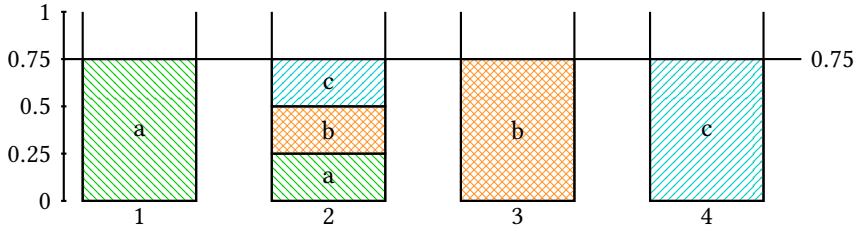


Figure E.3. An optimal port utilization for the experiment $\{\texttt{I} \mapsto 1, \texttt{B12} \mapsto 6\}$.

While this simplification is likely to produce accurate results for practical port mappings, it does not compute the correct inverse throughput (according to the linear program from Theorem 3.12) in general. Consider the following example: Let $\mathbf{I} = \{A, B, C\}$, $\mathbf{U} = \{a, b, c\}$, and $\mathbf{P} = \{1, 2, 3, 4\}$ with the following port mapping:



Consider the experiment $\{A \mapsto 1, B \mapsto 1, C \mapsto 1\}$. An optimal execution achieves an inverse throughput of 0.75 cycles and may distribute the µops as follows:



However, the Facile algorithm computes the maximum of the following values:

$$M[a] \cup M[a] = \{1, 2\} : \frac{1}{2}$$

$$M[b] \cup M[b] = \{2, 3\} : \frac{1}{2}$$

$$M[c] \cup M[c] = \{2, 4\} : \frac{1}{2}$$

$$M[a] \cup M[b] = \{1, 2, 3\} : \frac{2}{3}$$

$$M[a] \cup M[c] = \{1, 2, 4\} : \frac{2}{3}$$

$$M[b] \cup M[c] = \{2, 3, 4\} : \frac{2}{3}$$

The result, $\frac{2}{3}$, only underapproximates the correct inverse throughput $\frac{3}{4}$.

List of Algorithms

List of Figures

List of Tables

Bibliography

Andreas Abel (2020). Automatic Generation of Models of Microarchitectures. Ph.D. thesis, Saarland University, Saarbrücken, Germany, URL https://dx.doi.org/10.22028/D291-31299.

Andreas Abel (2022). DiffTune Revisited: A Simple Baseline for Evaluating Learned llvm-mca Parameters. In Machine Learning for Computer Architecture and Systems 2022, URL https://openreview.net/forum?id=dw4evoj6AE.

Andreas Abel and Jan Reineke (2019). uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019 (edited by Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck), pp. 673–686, ACM, doi: 10.1145/3297858.3304062, URL https://doi.org/10.1145/3297858.3304062.

Andreas Abel and Jan Reineke (2020). nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020, pp. 34–46, IEEE, doi: 10.1109/ISPASS48437.2020.00014, URL https://doi.org/10.1109/ISPASS48437.2020.00014.

Andreas Abel and Jan Reineke (2022). uiCA: Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures. In ICS '22: 2022 International Conference on Supercomputing, Virtual Event, June 28 - 30, 2022 (edited by Lawrence Rauchwerger, Kirk W. Cameron, Dimitrios S. Nikolopoulos, and Dionisios N. Pnevmatikatos), pp. 33:1–33:14, ACM, doi: 10.1145/3524059.3532396, URL https://doi.org/10.1145/3524059.3532396.

Andreas Abel, Shrey Sharma, and Jan Reineke (2023). Facile: Fast, Accurate, and Interpretable Basic-Block Throughput Prediction. In IEEE International Symposium on Workload Characterization, IISWC 2023, Ghent, Belgium, October 1-3, 2023, pp. 87–99, IEEE, doi: 10.1109/IISWC59245.2023.00023, URL https://doi.org/10.1109/IISWC59245.2023.00023.

Abderaouf N. Amalou, Élisa Fromont, and Isabelle Puaut (2022). CATREEN: Context-Aware Code Timing Estimation with Stacked Recurrent Networks. In 34th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2022, Macao, China, October 31 - November 2, 2022 (edited by Marek Z. Reformat, Du Zhang, and Nikolaos G. Bourbakis), pp. 571–576, IEEE, doi: 10.1109/ICTAI56018.2022.00090, URL https://doi.org/10.1109/ICTAI56018.2022.00090.

AMD (2019). Processor Programming Reference for AMD Family 17h Models 01h,08h, Revision B2 Processors. AMD, URL https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/54945_3_03_ppr_ZP_B2_pub.zip. Revision 3.03, Accessed: November 14, 2023.

AMD (2020). Software Optimization Guide for AMD EPYC 7003 Processors. AMD, URL https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/software-optimization-guides/56665.zip. Revision 3.00, Accessed: November 14, 2023.

AMD (2021a). Processor Programming Reference (PPR) for AMD Family 19h Model 21h, Revision B0 Processors. AMD, URL https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/56214-B0-PUB.zip. Revision 3.05, Accessed: November 14, 2023.

AMD (2021b). Software Optimization Guide for AMD Family 17h Processors. AMD, URL https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/software-optimization-guides/55723_3_01_0.zip. Revision 3.01, Accessed: November 14, 2023.

ARM (2015). Cortex-A72 Software Optimization Guide. ARM, URL https://developer.arm.com/documentation/uan0016/a/. ARM UAN: 0016A, Accessed: November 14, 2023.

ARM (2016). ARM Cortex-A72 MPCore Processor Technical Reference Manual. ARM, URL https://developer.arm.com/documentation/100095/0003/. Revision r0p3, Accessed: November 14, 2023.

Sanjeev Arora and Boaz Barak (2009). Computational Complexity: A Modern Approach. Cambridge University Press, ISBN 978-0-511-80409-0, URL https://doi.org/10.1017/CBO9780511804090.

Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu (2010). An adaptive performance modeling tool for GPU architectures. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010 (edited by R. Govindarajan, David A. Padua, and Mary W. Hall), pp. 105–114, ACM, doi: 10.1145/1693453.1693470, URL https://doi.org/10.1145/1693453.1693470.

Dimitris Bertsimas and John Tsitsiklis (1997). Introduction to Linear Optimization. Athena Scientific, 1st edition, ISBN 978-1-886529-19-9.

Lucas Biehl (2021). Characterising Input-Dependent Instruction Timing via Measurements. Master's thesis, Saarland University, URL https://embedded.cs.uni-saarland.de/publications/theses/thesis_cs_msc_Biehl_Lucas_Alexander.pdf.

Armin Biere, Marijn Heule, and Hans van Maaren (2009). Handbook of Satisfiability, chapter 26: Satisfiability Modulo Theories. IOS press.

Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):pp. 1–7, doi: 10.1145/2024716.2024718, URL https://doi.org/10.1145/2024716.2024718.

Igor Böhm, Björn Franke, and Nigel P. Topham (2010). Cycle-Accurate Performance Modelling in an Ultra-Fast Just-in-Time Dynamic Binary Translation Instruction Set Simulator. In Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2010), Samos, Greece, July 19-22, 2010 (edited by Fadi J. Kurdahi and Jarmo Takala), pp. 1–10, IEEE, doi: 10.1109/ICSAMOS.2010.5642102, URL https://doi.org/10.1109/ICSAMOS.2010.5642102.

Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov (2014). Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014, pp. 114–129, IEEE Computer Society, doi: 10.1109/SP.2014.15, URL https://doi.org/10.1109/SP.2014.15.

James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski (2018). SPEC CPU2017: Next-Generation Compute Benchmark. In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018 (edited by Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar), pp. 41–42, ACM, doi: 10.1145/3185768.3185771, URL https://doi.org/10.1145/3185768.3185771.

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein (1981). Register Allocation Via Coloring. *Comput. Lang.*, 6(1):pp. 47–57, doi: 10.1016/0096-0551(81)90048-5, URL https://doi.org/10.1016/0096-0551(81)90048-5.

Guillaume Chatelet (2018). llvm-exegesis: Automatic Measurement of Instruction Latency/Uops. llvm-dev Mailing List, URL https://lists.llvm.org/pipermail/llvm-dev/2018-March/121814.html. Accessed: November 14, 2023.

Isha Chaudhary, Alex Renda, Charith Mendis, and Gagandeep Singh (2023). COMET: X86 Cost Model Explanation Framework. doi: 10.48550/arXiv.2302.06836, URL https://doi.org/10.48550/arXiv.2302.06836.

Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondrej Sýkora, Saman P. Amarasinghe, and Michael Carbin (2019). BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. In IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019, pp. 167–177, IEEE, doi: 10.1109/IISWC47752.2019.9042166, URL https://doi.org/10.1109/IISWC47752.2019.9042166.

Yuting Chen and Zhendong Su (2015). Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015 (edited by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans), pp. 793–804, ACM, doi: 10.1145/2786805.2786835, URL https://doi.org/10.1145/2786805.2786835.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith (2000). Counterexample-Guided Abstraction Refinement. In Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings, volume 1855 of *Lecture Notes in Computer Science* (edited by E. Allen Emerson and A. Prasad Sistla), pp. 154–169, Springer, doi: 10.1007/10722167_15, URL https://doi.org/10.1007/10722167_15.

Patrick Cousot and Radhia Cousot (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977 (edited by Robert M. Graham, Michael A. Harrison, and Ravi Sethi), pp. 238–252, ACM, doi: 10.1145/512950.512973, URL https://doi.org/10.1145/512950.512973.

Kenneth A De Jong (2006). Evolutionary Computation: A Unified Approach. MIT Press, ISBN 978-0-262-25598-1.

Leonardo Mendonça de Moura and Nikolaj S. Bjørner (2008). Z3: An Efficient SMT Solver. In Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4963 of *Lecture Notes in Computer Science* (edited by C. R. Ramakrishnan and Jakob Rehof), pp. 337–340, Springer, doi: 10.1007/978-3-540-78800-3_24, URL https://doi.org/10.1007/978-3-540-78800-3_24.

Nicolas Derumigny, Théophile Bastian, Fabian Gruber, Guillaume Iooss, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello (2022a). PALMED: Throughput Characterization for Superscalar Architectures. In IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022 (edited by Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman), pp. 106–117, IEEE, doi: 10.1109/CGO53902.2022.9741289, URL https://doi.org/10.1109/CGO53902.2022.9741289.

Nicolas Derumigny, Fabian Gruber, Théophile Bastian, Guillaume Iooss, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello (2022b). PALMED: Throughput Characterization for Superscalar Architectures – Extended Version. URL https://arxiv.org/abs/2012.11473.

Andrea Di Biagio (2018). llvm-mca: A Static Performance Analysis Tool. llvm-dev Mailing List, URL https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html. Accessed: November 14, 2023.

Andrea Di Biagio (2020). [llvm-mca] Resource consumption of ProcResGroups. llvm-dev Mailing List, URL https://lists.llvm.org/pipermail/llvm-dev/2020-May/141486.html. Accessed: November 14, 2023.

Rafael Dutra, Jonathan Bachrach, and Koushik Sen (2019). GUIDEDSAMPLER: Coverage-guided Sampling of SMT Solutions. In 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019 (edited by Clark W. Barrett and Jin Yang), pp. 203–211, IEEE, doi: 10.23919/FMCAD.2019.8894251, URL https://doi.org/10.23919/FMCAD.2019.8894251.

Michael J. Fischer and Michael O. Rabin (1998). Super-Exponential Complexity of Presburger Arithmetic. In Quantifier Elimination and Cylindrical Algebraic Decomposition, pp. 122–135, Springer, ISBN 978-3-7091-9459-1, doi: 10.1007/978-3-7091-9459-1_5.

Agner Fog (2022). Instruction Tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. URL https://www.agner.org/optimize/instruction_tables.pdf. Accessed: November 14, 2023.

Agner Fog (2023). The microarchitecture of Intel, AMD, and VIA CPUs. URL https://www.agner.org/optimize/microarchitecture.pdf. Accessed: November 14, 2023.

Michael R. Garey and David S. Johnson (1990). Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., USA, ISBN 978-0-7167-1045-5.

GCC (2023). Scheduling Model for AMD Zen Microarchitectures. URL https://github.com/gcc-mirror/gcc/blob/15b83b69ca99d97643075776ba94f2dd1f05b46e/gcc/config/i386/znver.md. Accessed: November 14, 2023.

Google (2018). EXEgesis: Automatic Measurement of Instruction Latency/Uops. URL https://github.com/google/EXEgesis. Accessed: November 14, 2023.

Torbjörn Granlund (2019). Instruction latencies and throughput for AMD and Intel x86 processors. URL https://gmplib.org/~tege/x86-timing.pdf. Accessed: November 14, 2023.

Thomas Gruber, Jan Eitzinger, Georg Hager, and Gerhard Wellein (2023). LIKWID. doi: 10.5281/zenodo.10105559, URL https://doi.org/10.5281/zenodo.10105559.

Gurobi Optimization, LLC (2023). Gurobi Optimizer Reference Manual. URL https://www.gurobi.com.

John L Hennessy and David A Patterson (2017). Computer Architecture: A Quantitative Approach – 6th Edition. Elsevier, ISBN 978-0-12-811906-8.

Eddie Hung, Steven J. E. Wilton, Haile Yu, Thomas C. P. Chau, and Philip Heng Wai Leong (2009). A detailed delay path model for FPGAs. In Proceedings of the 2009 International Conference on Field-Programmable Technology, FPT 2009, Sydney, Australia, December 9-11, 2009 (edited by Neil W. Bergmann, Oliver Diessel, and Lesley Shannon), pp. 96–103,

IEEE Computer Society, doi: 10.1109/FPT.2009.5377673, URL https://doi.org/10.1109/FPT.2009.5377673.

Intel (2012). Intel Architecture Code Analyzer. URL https://software.intel.com/en-us/articles/intel-architecture-code-analyzer. Accessed: November 14, 2023.

Intel (2023a). Intel 64 and IA-32 Architectures Optimization Reference Manual: Volume 1. Intel, URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Order Number: 248966-048, Accessed: November 14, 2023.

Intel (2023b). Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Intel, URL https://www.intel.com/content/www/us/en/content-details/789583/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html. Order Number: 325462-081US, Accessed: December 5, 2023.

Nathan Jay and Barton P. Miller (2018). Structured random differential testing of instruction decoders. In 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018 (edited by Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd), pp. 84–94, IEEE Computer Society, doi: 10.1109/SANER.2018.8330199, URL https://doi.org/10.1109/SANER.2018.8330199.

Dougall Johnson (2021). Apple M1 Microarchitecture Research. URL https://dougallj.github.io/applecpu/firestorm.html. Accessed: November 14, 2023.

Richard M. Karp (1972). Reducibility Among Combinatorial Problems. In Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA (edited by Raymond E. Miller and James W. Thatcher), The IBM Research Symposia Series, pp. 85–103, Plenum Press, New York, doi: 10.1007/978-1-4684-2001-2_9, URL https://doi.org/10.1007/978-1-4684-2001-2_9.

Maurice G. Kendall (1938). A New Measure of Rank Correlation. *Biometrika*, 30(1/2):pp. 81–93, ISSN 00063444, URL http://www.jstor.org/stable/2332226.

Maurice G. Kendall (1945). The Treatment of Ties in Ranking Problems. *Biometrika*, 33(3):pp. 239–251, ISSN 00063444, URL http://www.jstor.org/stable/2332303.

Chris Lattner and Vikram S. Adve (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA, pp. 75–88, IEEE Computer Society, doi: 10.1109/CGO.2004.1281665, URL https://doi.org/10.1109/CGO.2004.1281665.

Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein (2018). 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@SC 2018, Dallas, TX, USA, November 12, 2018, pp. 121–131, IEEE, doi: 10.1109/PMBS.2018.8641578, URL https://doi.org/10.1109/PMBS.2018.8641578.

Lingda Li, Thomas Flynn, and Adolfy Hoisie (2023). Learning Independent Program and Architecture Representations for Generalizable Performance Modeling. *CoRR*, abs/2310.16792, doi: 10.48550/ARXIV.2310.16792, URL https://doi.org/10.48550/arXiv.2310.16792.

Linux Programmer's Manual (2020). clock_gettime(3) — Linux manual page. URL https://man7.org/linux/man-pages/man3/clock_gettime.3.html. Accessed: November 14, 2023.

Zhengyang Liu, Stefan Mada, and John Regehr (2023). Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. *CoRR*, abs/2306.00229, doi: 10.48550/ARXIV.2306.00229, URL https://doi.org/10.48550/arXiv.2306.00229.

LLVM (2021). LLVM Version 13 Scheduling Model for AMD Zen/Zen+ CPUs. URL https://github.com/llvm/llvm-project/blob/release/13.x/llvm/lib/Target/X86/X86ScheduleZnver1.td. Accessed: November 14, 2023.

LLVM (2022). LLVM Scheduling Model for AMD Zen Microarchitectures. URL https://github.com/llvm/llvm-project/blob/4eb1f1fab35d0f386b458bf1da4396bbeb00b04f/llvm/lib/Target/X86/X86ScheduleZnver1.td. Accessed: November 14, 2023.

LLVM (2023a). llvm-exegesis - LLVM Machine Instruction Benchmark. URL https://llvm.org/docs/CommandGuide/llvm-exegesis.html. Accessed: November 14, 2023.

LLVM (2023b). llvm-mca - LLVM Machine Code Analyzer (Command Guide). URL https://llvm.org/docs/CommandGuide/llvm-mca.html. Accessed: November 14, 2023.

Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo (2021). The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Software Eng.*, 47(11):pp. 2312–2331, doi: 10.1109/TSE.2019.2946563, URL https://doi.org/10.1109/TSE.2019.2946563.

Yuri V Matiyasevich (1993). Hilbert's tenth problem. MIT Press, ISBN 978-0-262-13295-4.

William M. McKeeman (1998). Differential Testing for Software. *Digit. Tech. J.*, 10(1):pp. 100–107, URL https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf.

Charith Mendis, Alex Renda, Saman P. Amarasinghe, and Michael Carbin (2019). Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA, volume 97 of *Proceedings of Machine Learning Research* (edited by Kamalika Chaudhuri and Ruslan Salakhutdinov), pp. 4505–4515, PMLR, URL http://proceedings.mlr.press/v97/mendis19a.html.

Kaisa Miettinen (1998). Nonlinear Multiobjective Optimization. Springer Science & Business Media, ISBN 978-1-4615-5563-6, URL https://doi.org/10.1007/978-1-4615-5563-6.

Yang Ni, Yeseong Kim, Tajana Rosing, and Mohsen Imani (2022). Online Performance and Power Prediction for Edge TPU via Comprehensive Characterization. In 2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022, Antwerp, Belgium, March 14-23, 2022 (edited by Cristiana Bolchini, Ingrid Verbauwhede, and Ioana Vatajelu), pp. 612–615, IEEE, doi: 10.23919/DATE54114.2022.9774764, URL https://doi.org/10.23919/DATE54114.2022.9774764.

Georg Ofenbeck, Ruedi Steinmann, Victoria Caparrós Cabezas, Daniele G. Spampinato, and Markus Püschel (2014). Applying the Roofline Model. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014, pp. 76–85, IEEE Computer Society, doi: 10.1109/ISPASS.2014.6844463, URL https://doi.org/10.1109/ISPASS.2014.6844463.

Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein (2021). Revizor: Fuzzing for Leaks in Black-box CPUs. *CoRR*, abs/2105.06872, URL https://arxiv.org/abs/2105.06872.

Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi (2010). N-version disassembly: differential testing of x86 disassemblers. In Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010 (edited by Paolo Tonella and Alessandro Orso), pp. 265–274, ACM, doi: 10.1145/1831708.1831741, URL https://doi.org/10.1145/1831708.1831741.

Chris Parnin and Alessandro Orso (2011). Are automated debugging techniques actually helping programmers? In Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011 (edited by Matthew B. Dwyer and Frank Tip), pp. 199–209, ACM, doi: 10.1145/2001420.2001445, URL https://doi.org/10.1145/2001420.2001445.

Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana (2017). NEZHA: Efficient Domain-Independent Differential Testing. In 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017, pp. 615–632, IEEE Computer Society, doi: 10.1109/SP.2017.27, URL https://doi.org/10.1109/SP.2017.27.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati (2016). Scaling up Superoptimization. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016 (edited by Tom Conte and Yuanyuan Zhou), pp. 297–310, ACM, doi: 10.1145/2872362.2872387, URL https://doi.org/10.1145/2872362.2872387.

Michael L. Pinedo (2022). Scheduling: Theory, Algorithms, and Systems. Springer International Publishing, 6th edition, ISBN 978-3-031-05921-6, doi: 10.1007/978-3-031-05921-6, URL https://doi.org/10.1007/978-3-031-05921-6.

Angela Pohl, Biagio Cosenza, and Ben H. H. Juurlink (2019). Portable Cost Modeling for Auto-Vectorizers. In 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2019, Rennes, France, October 21-25, 2019, pp. 359–369, IEEE Computer Society, doi: 10.1109/MASCOTS.2019.00046, URL https://doi.org/10.1109/MASCOTS.2019.00046.

Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin (2020). DiffTune: Optimizing CPU Simulator Parameters with Learned Differentiable Surrogates. In 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020, pp. 442–455, IEEE, doi: 10.1109/MICRO50266.2020.00045, URL https://doi.org/10.1109/MICRO50266.2020.00045.

Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh (2004). Symbolic Implementation of the Best Transformer. In Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings, volume 2937 of *Lecture Notes in Computer Science* (edited by Bernhard Steffen and Giorgio Levi), pp. 252–266, Springer, doi: 10.1007/978-3-540-24622-0_21, URL https://doi.org/10.1007/978-3-540-24622-0_21.

Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin (2018). Anchors: High-Precision Model-Agnostic Explanations. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018 (edited by Sheila A. McIlraith and Kilian Q. Weinberger), pp. 1527–1535, AAAI Press, URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16982.

Fabian Ritter and Sebastian Hack (2020). PMEvo: Portable Inference of Port Mappings for Out-of-Order Processors by Evolutionary Optimization. In Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020 (edited by Alastair F. Donaldson and Emina Torlak), pp. 608–622, ACM, doi: 10.1145/3385412.3385995, URL https://doi.org/10.1145/3385412.3385995.

Fabian Ritter and Sebastian Hack (2022). AnICA: Analyzing Inconsistencies in Microarchitectural Code Analyzers. *Proc. ACM Program. Lang.*, 6(OOPSLA2):pp. 1–29, doi: 10.1145/3563288, URL https://doi.org/10.1145/3563288.

Fabian Ritter and Sebastian Hack (2024). Explainable Port Mapping Inference with Sparse Performance Counters for AMD's Zen Architectures. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024-1 May 2024 (edited by Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir), pp. 317–330, ACM, doi: 10.1145/3620666.3651363, URL https://doi.org/10.1145/3620666.3651363.

Andres Charif Rubial, Emmanuel Oseret, Jose Noudohouenou, William Jalby, and Ghislain Lartigue (2014). CQA: A Code Quality Analyzer Tool at Binary Level. In 21st International

Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014, pp. 1–10, IEEE Computer Society, doi: 10.1109/HiPC.2014.7116904, URL https://doi.org/10.1109/HiPC.2014.7116904.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat (2006). Combinatorial sketching for finite programs. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006 (edited by John Paul Shen and Margaret Martonosi), pp. 404–415, ACM, doi: 10.1145/1168857.1168907, URL https://doi.org/10.1145/1168857.1168907.

Ondrej Sýkora, Phitchaya Mangpo Phothilimthana, Charith Mendis, and Amir Yazdanbakhsh (2022). GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation. pp. 14–26, doi: 10.1109/IISWC55918.2022.00012, URL https://doi.org/10.1109/IISWC55918.2022.00012.

Shaojie Tan, Qingcai Jiang, Zhenwei Cao, Xiaoyu Hao, Junshi Chen, and Hong An (2023). Uncovering the performance bottleneck of modern HPC processor with static code analyzer: a case study on Kunpeng 920. *CCF Transactions on High Performance Computing*, pp. 1–22.

Jan Treibig, Georg Hager, and Gerhard Wellein (2010). LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In 39th International Conference on Parallel Processing, ICPP Workshops 2010, San Diego, California, USA, 13-16 September 2010 (edited by Wang-Chien Lee and Xin Yuan), pp. 207–216, IEEE Computer Society, doi: 10.1109/ICPPW.2010.38, URL https://doi.org/10.1109/ICPPW.2010.38.

Larry Wasserman (2004). All of Statistics: A Concise Course in Statistical Inference. Springer, ISBN 978-0-387-21736-9, URL https://doi.org/10.1007/978-0-387-21736-9.

Kallistos Weis (2019). Portable Instruction-Throughput Estimation for Port-Mapping Synthesis. Universität des Saarlandes, Saarbrücken. Bachelor's Thesis.

WikiChip (2023a). Zen - Microarchitectures - AMD. URL https://en.wikichip.org/wiki/amd/microarchitectures/zen. Accessed: November 14, 2023.

WikiChip (2023b). Zen+ - Microarchitectures - AMD. URL https://en.wikichip.org/wiki/amd/microarchitectures/zen%2B. Accessed: November 14, 2023.

Samuel Williams, Andrew Waterman, and David A. Patterson (2009). Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):pp. 65–76, doi: 10.1145/1498765.1498785, URL https://doi.org/10.1145/1498765.1498785.

William Woodruff, Niki Carroll, and Sebastiaan Peters (2021). Differential analysis of x86-64 instruction decoders. In IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021, pp. 152–161, IEEE, doi: 10.1109/SPW53761.2021.00029, URL https://doi.org/10.1109/SPW53761.2021.00029.

Rafael J. Wysocki (2017). CPU Performance Scaling. URL https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html. Accessed: November 14, 2023.