Automatic Generation of Models of Microarchitectures

Dissertation

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

von
Andreas Abel

Saarbrücken
2020

Tag des Kolloquiums: 12. Juni 2020

Dekan: Prof. Dr. Thomas Schuster

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Thorsten Herfet
Berichterstatter:	Prof. Dr. Jan Reineke Prof. Dr. Wolfgang J. Paul Dr. Boris Köpf
Akademischer Mitarbeiter:	Dr. Roland Leißa

Abstract

Detailed microarchitectural models are necessary to predict, explain, or optimize the performance of software running on modern microprocessors. Building such models often requires a significant manual effort, as the documentation provided by hardware manufacturers is typically not precise enough. The goal of this thesis is to develop techniques for generating microarchitectural models automatically.

In the first part, we focus on recent x86 microarchitectures. We implement a tool to accurately evaluate small microbenchmarks using hardware performance counters. We then describe techniques to automatically generate microbenchmarks for measuring the performance of individual instructions and for characterizing cache architectures. We apply our implementations to more than a dozen different microarchitectures.

In the second part of the thesis, we study more general techniques to obtain models of hardware components. In particular, we propose the concept of gray-box learning, and we develop a learning algorithm for Mealy machines that exploits prior knowledge about the system to be learned.

Finally, we show how this algorithm can be adapted to minimize incompletely specified Mealy machines—a well-known NP-complete problem. Our implementation outperforms existing exact minimization techniques by several orders of magnitude on a number of hard benchmarks; it is even competitive with state-of-the-art heuristic approaches.

Zusammenfassung

Zur Vorhersage, Erklärung oder Optimierung der Leistung von Software auf modernen Mikroprozessoren werden detaillierte Modelle der verwendeten Mikroarchitekturen benötigt. Das Erstellen derartiger Modelle ist oft mit einem hohen Aufwand verbunden, da die erforderlichen Informationen von den Prozessorherstellern typischerweise nicht zur Verfügung gestellt werden. Das Ziel der vorliegenden Arbeit ist es, Techniken zu entwickeln, um derartige Modelle automatisch zu erzeugen.

Im ersten Teil beschäftigen wir uns mit aktuellen x86-Mikroarchitekturen. Wir entwickeln zuerst ein Tool, das kleine Microbenchmarks mithilfe von Performance Countern auswerten kann. Danach beschreiben wir Techniken, um automatisch Microbenchmarks zu erzeugen, mit denen die Leistung einzelner Instruktionen gemessen sowie die Cache-Architektur charakterisiert werden kann.

Im zweiten Teil der Arbeit betrachten wir allgemeinere Techniken, um Hardwaremodelle zu erzeugen. Wir schlagen das Konzept des "Gray-Box Learning" vor, und wir entwickeln einen Lernalgorithmus für Mealy-Maschinen, der bekannte Informationen über das zu lernende System berücksichtigt.

Zum Abschluss zeigen wir, wie dieser Algorithmus auf das Problem der Minimierung unvollständig spezifizierter Mealy-Maschinen übertragen werden kann. Hierbei handelt es sich um ein bekanntes NP-vollständiges Problem. Unsere Implementierung ist in mehreren Benchmarks um Größenordnungen schneller als vorherige Ansätze.

Acknowledgements

First and foremost, I would like to thank my advisor, Prof. Jan Reineke. He gave me the freedom to explore my own ideas and was always available for discussions and to provide guidance. I'm looking forward to continuing working with him!

I would also like to thank Prof. Wolfgang Paul and Dr. Boris Köpf for reviewing my thesis, and Prof. Thorsten Herfet for acting as the chair of the examination board.

Finally, I would like to thank my current and former colleagues at the Real-Time and Embedded Systems Lab and the Compiler Design Lab. In particular, I would like to thank Dr. Roland Leißa for serving as the academic assistant on my examination board.

Contents

1 Introduction **13**
1.1 Contributions and Structure of This Thesis 14
1.2 Publications 19

2 nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems **21**
2.1 Introduction 21
2.2 Background 23
2.2.1 Hardware Performance Counters 23
2.2.2 Assembler Instructions 25
2.3 Features 25
2.3.1 Example 26
2.3.2 Generated Code 27
2.3.3 Running the Generated Code 27
2.3.4 Kernel/User Mode 29
2.3.5 Interface 29
2.3.6 Loops vs. Unrolling 29
2.3.7 Accessing Memory 30
2.3.8 Warm-Up Runs 30
2.3.9 noMem Mode 30
2.3.10 Performance Counter Configurations 31
2.3.11 Execution Time of nanoBench 31
2.3.12 Supported Platforms 32
2.4 Implementation 32
2.4.1 Accurate Performance Counter Measurements 32
2.4.2 Generating Code 33
2.4.3 Kernel Module 34
2.4.4 Allocating Physically-Contiguous Memory 34
2.5 Related Work 35
2.6 Conclusions and Future Work 36

3 uops.info: Characterizing the Latency, Throughput, and Port Usage of Instructions on x86 Microarchitectures **39**
3.1 Introduction 40
3.2 Related Work 42
3.2.1 Information Provided by the Manufacturers 42
3.2.2 Measurement-Based Approaches 43
3.3 Background 44

3.3.1 Pipeline of Intel Core CPUs . . . 44
3.3.2 Pipeline of AMD Ryzen CPUs . . . 46
3.4 Definitions . . . 46
3.4.1 Latency . . . 47
3.4.2 Throughput . . . 47
3.4.3 Port Usage . . . 48
3.5 Algorithms . . . 49
3.5.1 Port Usage . . . 49
3.5.2 Latency . . . 52
3.5.3 Throughput . . . 59
3.6 Implementation . . . 61
3.6.1 Details of the x86 Instruction Set . . . 61
3.6.2 Measurements on the Hardware . . . 62
3.6.3 Analysis Using IACA . . . 63
3.6.4 Machine-Readable Output . . . 63
3.7 Evaluation . . . 63
3.7.1 Experimental Setup . . . 64
3.7.2 Hardware Measurements vs. Documentation . . . 64
3.7.3 Hardware Measurements vs. IACA . . . 67
3.7.4 Interesting Results . . . 69
3.8 Limitations . . . 76
3.9 Conclusions and Future Work . . . 77

4 Characterizing Cache Architectures 79
4.1 Introduction . . . 79
4.2 Background . . . 81
4.2.1 Cache Organization . . . 81
4.2.2 Replacement Policies . . . 82
4.3 Cache-Characterization Tools . . . 86
4.3.1 CacheInfo . . . 86
4.3.2 CacheSeq . . . 88
4.3.3 Replacement Policies . . . 90
4.3.4 Age Graphs . . . 92
4.3.5 Test for Adaptive Policies . . . 92
4.4 Results . . . 94
4.4.1 L1 Data Caches . . . 94
4.4.2 L2 Caches . . . 96
4.4.3 L3 Caches . . . 104
4.4.4 Resetting the Replacement Policy State . . . 109
4.4.5 Implementation Costs . . . 111
4.5 Related Work . . . 111

4.5.1 Microbenchmark-Based Cache Analysis . . . 111
4.5.2 Influence of the Replacement Policy on Performance Prediction Accuracy . . . 113
4.5.3 Security Aspects of Replacement Policies . . . 114
4.6 Conclusions and Future Work . . . 115

5 Gray-Box Learning of Serial Compositions of Mealy Machines 117
5.1 Introduction . . . 118
5.2 Problem Statement . . . 119
5.2.1 Basic Notions . . . 119
5.2.2 The Gray-Box Learning Problem . . . 120
5.3 Preliminaries . . . 121
5.4 Approach . . . 122
5.4.1 Observation Tables . . . 123
5.4.2 Inference Algorithm . . . 126
5.5 Implementation . . . 128
5.5.1 Computing the Partitions . . . 128
5.5.2 Reachability of the Error State . . . 131
5.5.3 Checking if Two Machines are Right-Equivalent . . . 131
5.5.4 Handling Counterexamples . . . 132
5.6 Evaluation . . . 132
5.7 Related Work . . . 134
5.8 Conclusions and Future Work . . . 135
5.A Appendix: Proofs for Chapter 5 . . . 136

6 MeMin: SAT-Based Exact Minimization of Incompletely Specified Mealy Machines 139
6.1 Introduction . . . 139
6.1.1 Outline . . . 141
6.2 Definitions . . . 141
6.2.1 Basic Definitions . . . 141
6.2.2 Problem Statement . . . 143
6.2.3 General Approach . . . 143
6.3 Related Work . . . 144
6.4 Approach . . . 146
6.4.1 Incompatibility Matrix . . . 147
6.4.2 Encoding as a SAT Problem . . . 147
6.4.3 Computing a Partial Solution . . . 149
6.5 Implementation . . . 149
6.5.1 Dealing with Partially Specified Outputs . . . 149

6.5.2 Dealing with Partially Specified Inputs 150
6.5.3 Undefined Reset States 150
6.6 Evaluation 150
6.6.1 Benchmarks 151
6.6.2 Evaluation of MeMin 155
6.6.3 Other Tools 155
6.6.4 Experimental Setup 158
6.7 Conclusions and Future Work 158
6.A Appendix: Complete Benchmark Results 159

7 Summary, Conclusions, and Future Work 165
7.1 Summary and Conclusions 165
7.1.1 Models of Recent Microarchitectures 165
7.1.2 General Models 166
7.2 Future Work 167

Bibliography 169

Index 197

1 Introduction

Modern microprocessors are among the most complex man-made systems. As a consequence, it is becoming increasingly difficult to predict, explain, or optimize the performance of software running on such microprocessors. As a basis, one needs detailed models of their microarchitectures.

Such models are, for example, required to build optimizing compilers, worst-case execution time (WCET) analyzers, cycle-accurate simulators, or self-optimizing software systems. Similarly, such models are necessary to show the presence or absence of microarchitectural security issues, such as Spectre and Meltdown [KHF+19, LSG+18]. Finally, detailed knowledge of microarchitectural details is also helpful when manually fine-tuning a piece of code for a specific processor.

Unfortunately, the documentation provided by hardware manufacturers is usually not detailed enough. To build microarchitectural models, engineers are thus often forced to perform measurements using microbenchmarks. Existing approaches typically require a significant amount of manual effort, and the results are not always accurate and precise.

The goal of this thesis therefore is to develop techniques for generating detailed models of microarchitectures automatically.

In the first part of the thesis, we focus on recent x86 microarchitectures. In particular, we develop techniques to automatically generate models for two properties that have a strong influence on the performance of software on a specific microarchitecture: the cache architecture and the latencies, throughputs, and port usages of individual instructions.

In the second part of the thesis, we look at more general techniques for obtaining models of hardware components. Specifically, we introduce the problem of *gray-box learning*, in which learning algorithms may exploit known information about the system to be learned.

1.1 Contributions and Structure of This Thesis

We now describe the addressed problems, and our approaches and contributions in more detail.

nanoBench

In Chapter 2, we develop *nanoBench*, a tool for evaluating small microbenchmarks using hardware performance counters on x86 systems. Hardware performance counters are special-purpose registers that store the counts of various hardware-related events.

In contrast to previous tools, *nanoBench* can execute microbenchmarks directly in kernel space. This makes it possible to benchmark privileged instructions, and it allows for more accurate measurements by disabling interrupts and preemptions. Furthermore, this makes it also possible to directly access certain performance counters that are only available in kernel space; previous tools require expensive system calls to access such counters.

nanoBench provides the option to temporarily pause performance counting during specific parts of a microbenchmark. Furthermore, the reading of the performance counters is implemented with minimal overhead, avoiding function calls and branches. As a consequence, *nanoBench* is precise enough to measure, e.g., whether individual memory accesses result in cache hits or misses.

We use *nanoBench* to evaluate the microbenchmarks generated by the tools described in the following paragraphs.

Instruction Characterizations

An aspect that has a relatively strong influence on performance is how ISA instructions decompose into micro-operations (µops), which ports these µops may be executed on, and what their latencies and throughputs are.

However, these properties are typically only poorly documented. Intel's processor manuals [Int12, Int19b], for example, only contain latency and throughput data for a number of "commonly-used instructions." They do not contain information on the decomposition of individual instructions into µops, nor do they state the execution ports that these µops can use.

The only way to obtain accurate instruction characterizations for many recent microarchitectures is thus to perform measurements using microbenchmarks.

However, existing approaches, such as [Fog19], require significant manual effort to create suitable microbenchmarks and to interpret their results. Furthermore, the results are not always accurate and precise.

In Chapter 3, we develop a new approach that can automatically generate microbenchmarks in order to characterize the latency, throughput, and port usage of instructions on x86 CPUs in an accurate and precise manner.

Before describing our algorithms and their implementation, we first discuss common notions of instruction latency, throughput, and port usage. For the latency, we propose a new definition that, in contrast to previous definitions, considers dependencies between different pairs of input and output operands, which enables more accurate performance predictions.

We then develop algorithms that generate assembler code for microbenchmarks to measure the properties of interest for most x86 instructions. Our algorithms take into account explicit and implicit dependencies, such as, e.g., dependencies on status flags. Therefore, they require detailed information on the x86 instruction set. To this end, we create a machine-readable XML representation of the x86 instruction set that contains all the information needed to automatically generate assembler code for every instruction. The relevant information is automatically extracted from the configuration files of Intel's *x86 Encoder Decoder (XED)* [Intc] library.

We have implemented our algorithms in a tool that we have successfully applied to 16 different Intel and AMD microarchitectures. The output of our tool is available both in the form of a human-readable, interactive HTML table and as a machine-readable XML file, so that it can be easily used to implement, e.g., simulators, performance prediction tools, or optimizing compilers.

Finally, we discuss several interesting insights obtained by comparing the results from our measurements with previously published data. Our precise latency data, for example, uncovered previously undocumented differences between different microarchitectures. It also explains discrepancies between previously published information. Apart from that, we uncovered various errors in Intel's IACA tool [Inta], and inaccuracies in the manuals.

Characterizing Cache Architectures

To bridge the increasing latency gap between the processor and main memory, modern microarchitectures employ memory hierarchies with multiple levels of cache memory. These caches are small but fast memories that make use

of temporal and spatial locality. Typically, they have a big impact on the execution time of computer programs; the penalty of a miss in the last-level cache can be more than 200 cycles.

In Chapter 4, we develop techniques for creating models of cache architectures. We focus, in particular, on cache replacement policies, which are typically undocumented for recent microarchitectures.

To this end, we develop several tools for determining cache parameters. The first tool, *cacheInfo*, provides details on the *structure* of the caches, such as the sizes, the associativities, the number of cache sets, or the number of C-Boxes and slices.

Our second tool, *cacheSeq*, makes it possible to analyze the *behavior* of the caches by measuring the number of cache hits and misses when executing an access sequence in one or more cache sets; the access sequence is supplied as a parameter to the tool and can be specified using a convenient syntax. To perform these measurements, the tool generates suitable microbenchmarks that are evaluated using *nanoBench*.

Based on *cacheSeq*, we then develop several tools for identifying the replacement policy. In particular, we implement a tool that can automatically infer permutation policies, and a tool that can automatically determine whether the policy belongs to a set of more than 300 variants of commonly used policies, including policies like MRU and QLRU, which are not permutation policies. These tools are precise enough to determine the policies used in individual cache sets. In addition to that, we develop a tool that can find out whether the cache uses an adaptive replacement policy. Furthermore, we propose a tool that creates *age graphs*, which are helpful for analyzing caches with nondeterministic replacement policies.

We have applied our tools to 13 different Intel microarchitectures, and we provide detailed models of their replacement policies. We have discovered several previously undocumented replacement policies.

Gray-Box Learning

The algorithms we developed for Chapters 3 and 4 can be seen as instances of *active learning* approaches. *Active learning* (also called *query learning*) refers to a class of machine-learning techniques in which the learning algorithm is able to interact with the system to be learned.

However, the algorithms we developed for Chapters 3 and 4 are heavily targeted at the specific problems. In Chapter 5, we look at more general

techniques. Specifically, we consider approaches for learning finite state machines, which are, in principle, suitable abstractions for modeling the behavior of microarchitectural components.

In active learning approaches, one commonly assumes an oracle, or teacher, that admits two kinds of queries about the system: output queries return the result of the system for a specific input, and equivalence queries check whether a conjectured model is consistent with the system to be learned and return a counterexample if not. Based on this setup, Angluin introduced the L^* algorithm [Ang87] for learning deterministic finite automata. L^* has since been extended to other modeling formalisms, such as Mealy machines [SG09], register automata [HSJC12, BHLM13, AFBKV15], or symbolic automata [MM14].

As the system is usually treated as a black box, no information about the internal structure of the system can be taken into account by most existing learning algorithms. In practice, however, systems are often composed of subcomponents, for some of which models might be available, but it is not possible to access the known and the unknown parts separately from the outside.

While it is in theory possible to learn a model of the entire system using existing black-box approaches, this is often not viable in practice because the state space is too large. A problem, which has received very little attention in the literature so far, is how to use the available information about the system to focus the learning algorithm on those parts that are unknown. We call this problem *gray-box learning*.

As a first step toward solving this problem, we study one specific instance. We consider the serial composition of two Mealy machines A and B, where A is known and B is unknown, and we assume that we can only perform queries on the composed machine.

While output queries can often be realized cheaply by measurements on the actual system, equivalence queries can usually only be approximated by a large number of such measurements. Our primary focus is thus to minimize the number of equivalence queries. We introduce an algorithm to exactly learn B in the context of A that performs at most $|B|$ equivalence queries, where $|B|$ denotes the number of states of B.

We evaluate our approach on compositions of randomly-generated machines against an implementation of the classic L^* algorithm in LearnLib [IHS15]. We show that our approach requires significantly fewer output and equivalence queries on most benchmarks.

Minimization of Incompletely Specified Mealy Machines

It is generally a desirable property of models to be as small as possible. In Chapter 6, we take a fresh look at the problem of minimizing incompletely specified Mealy machines, i.e., machines where one or more outputs or next states might not be specified. While the minimization problem is efficiently solvable for fully specified machines [Hop71], it is NP-complete for incompletely specified ones [Pfl73].

It turns out that this problem is quite closely related to the gray-box learning instance we studied in Chapter 5. A central algorithmic idea of our learning technique is a reduction to a Boolean satisfiability (SAT) problem. In Chapter 6, we show how a relatively straightforward adaptation of this reduction approach can be used to minimize incompletely specified Mealy machines.

The minimization problem has been extensively studied before, and a number of exact and heuristic approaches have been proposed. We compare an implementation of our exact approach to several other approaches on two sets of standard benchmarks.

Our approach outperforms the other exact approaches significantly, in particular on a number of hard benchmarks. In some cases, it is faster than existing approaches by several orders of magnitude.

On most benchmarks, our approach is also competitive with state-of-the-art heuristic methods. There are only two benchmarks on which a heuristic approach is significantly faster. However, in these two cases, this heuristic approach is not able to find the minimal result.

1.2 Publications

Key parts of this thesis have been published in the following papers.

- [AR14]: Andreas Abel and Jan Reineke. **Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation.** In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, March 23–25, 2014,* pages 141–142. © 2014 IEEE
- [AR15]: Andreas Abel and Jan Reineke. **MeMin: SAT-based exact minimization of incompletely specified Mealy machines.** In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, November 2–6, 2015,* pages 94–101. © 2015 IEEE
- [AR16]: Andreas Abel and Jan Reineke. **Gray-box learning of serial compositions of Mealy machines.** In *NASA Formal Methods—Proceedings of the 8th International Symposium, Minneapolis, MN, USA, June 7–9, 2016,* pages 272–287. Springer-Verlag, 2016.
- [AR19]: Andreas Abel and Jan Reineke. **uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures.** In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Providence, RI, USA, April 13–17, 2019,* pages 673–686. ACM, 2019.
- [AR20]: Andreas Abel and Jan Reineke. **nanoBench: A low-overhead tool for running microbenchmarks on x86 systems.** In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, USA, August 23–25, 2020.* To appear. © 2020 IEEE

2

nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems

In this chapter, we present *nanoBench*, a tool for evaluating small microbenchmarks using hardware performance counters on Intel and AMD x86 systems. Most existing tools and libraries are intended to either benchmark entire programs, or program segments in the context of their execution within a larger program. In contrast, *nanoBench* is specifically designed to evaluate small, isolated pieces of code. Such microbenchmarks are commonly used for analyzing undocumented hardware properties.

Unlike previous tools, *nanoBench* can execute microbenchmarks directly in kernel space. This allows to benchmark privileged instructions, and it enables more accurate measurements. The reading of the performance counters is implemented with minimal overhead, avoiding function calls and branches. As a consequence, *nanoBench* is precise enough to measure individual memory accesses.

Most of the work presented in this chapter has been published in [AR20].

2.1 Introduction

Benchmarking small pieces of code using hardware performance counters is often useful for analyzing the performance of software on a specific microprocessor, as well as for analyzing performance characteristics of the microprocessor itself.

Such microbenchmarks can, e.g., be helpful in identifying bottlenecks in loop kernels. To this end, modern x86 CPUs provide many performance events that can be measured, such as cache hits and misses in different levels of the memory hierarchy, the pressure on execution ports, mispredicted branches, etc.

Low-level aspects of microarchitectures are typically only poorly documented. Thus, the only way to obtain detailed information is often through microbenchmarks using hardware performance counters. This includes, for example, the latency, throughput, and port usage of individual instructions (see Section 3.2.2). Microbenchmarks have also been used to infer properties of the memory hierarchy (see Section 4.5.1). In addition to that, such benchmarks have been used to identify microarchitectural properties that can lead to security issues, such as Spectre [KHF+19] and Meltdown [LSG+18].

Often, such microbenchmarks consist of two parts: The main part, and an initialization phase that, for example, sets registers or memory locations to specific values or tries to establish a specific microarchitectural state, e.g., by flushing the caches. Ideally, the performance counters should only be active during the main part.

To facilitate the use of hardware performance counters, a number of tools and libraries have been proposed. Most of the existing tools fall into one of two categories. First, there are tools that benchmark entire programs, such as *perf* [per], or profilers like Intel's VTune Amplifier [Intb]. Tools in the second category are intended to benchmark program segments that are executed in the context of a larger program. They usually provide functions to start and stop the performance counters that can be called before and after the code segment of interest. Such tools are, for example, PAPI [TJYD10] and libpfc [Bil].

Tools from both categories are not particularly well suited for microbenchmarks of the kind described above. For tools from the first category, one obvious reason is that it is not possible to measure only parts of the code. Another reason is overhead. Just running a C program with an empty main function, compiled with a recent version of *gcc*, leads to the execution of more than 500,000 instructions and about 100,000 branches. Moreover, this number varies significantly from one run to another.

Overhead can also be a concern for tools from the second category. In PAPI, for example, the calls to start and stop the counters involve several hundred memory accesses, more than 150 branches, and for some counters even expensive system calls. This leads to unpredictable execution times and might, e.g., destroy the cache state that was established in the initialization part of the microbenchmark. Moreover, these calls will modify general-purpose

registers, so it is not possible to set the registers to specific values in the initialization part and use these values in the main part.

For several reasons, microbenchmarks often need to be run multiple times. One reason is the possibility of interference due to interrupts, preemptions or contention on shared resources that are also used by programs on other cores. Another reason are issues such as cold caches that impact the performance on the first runs. A third reason is that there are more performance events than there are programmable counters, so the measurements may need to be repeated with different counter configurations. Also, the code to be benchmarked itself often needs to be repeated several times. This is typically done by executing it in a loop or by unrolling it multiple times, or by a combination of both. All of this leads to a significant engineering effort that needs to be repeated over and over again.

In this chapter, we present *nanoBench*[1], an open-source tool that makes it very easy to evaluate small microbenchmarks on recent x86 CPUs. In Chapters 3 and 4, we describe techniques to automatically generate such microbenchmarks for characterizing the performance of instructions and for analyzing properties caches.

There are two variants of our tool: A user-space implementation and a kernel-space version. The kernel-space version makes it possible to directly benchmark privileged instructions, in contrast to any previous tool we are aware of. Furthermore, it allows for more accurate measurements than existing tools by disabling interrupts and preemptions. The tool is precise enough to measure, e.g., whether individual memory accesses result in cache hits or misses.

Microbenchmarks may use and modify any general-purpose and vector registers, including the stack pointer. After executing the microbenchmark, *nanoBench* automatically resets them to their previous values. The loop and unroll counts, as well as the number of repetitions and the aggregate function to be applied to the measurement results, can be specified via parameters.

2.2 Background

2.2.1 Hardware Performance Counters

Recent Intel and AMD processors are equipped with different types of hardware performance counters, i.e., special-purpose registers that store the counts of various hardware-related events. All of these counters can be read using

[1] `https://github.com/andreas-abel/nanoBench`

the *RDMSR*[2] instruction; many of them can also be read using the *RDPMC*[3] instruction. The *RDMSR* instruction is a privileged instruction and can, thus, only be used in kernel space. The *RDPMC* instruction, on the other hand, is faster than the *RDMSR* instruction, and it can be directly accessed in user space if a specific flag in a control register is set.

Core Performance Counters

Each logical core has a private performance monitoring unit with multiple performance counters. There are two types of core performance counters:

- **Fixed-Function Performance Counters**

  Recent Intel CPUs have three fixed-function performance counters that can be read with the *RDPMC* instruction. They count the number of retired instructions, the number of core cycles, and the number of reference cycles.

  In addition to that, there are two fixed-function counters that are available both on recent Intel CPUs, as well as on AMD family 17h CPUs: the *APERF* counter, which counts core clock cycles, and the *MPERF* counter, which counts reference cycles. These two counters can only be accessed with the *RDMSR* instruction and are, thus, only available in kernel space.

- **Programmable Performance Counters**

  Recent Intel CPUs have between two and eight, and AMD family 17h CPUs have six programmable performance counters. They can be programmed with a large number of different performance events (more than 200 on some CPUs), such as the number of µops that use a specific port, the number of cache misses in different levels of the memory hierarchy, the number of mispredicted branches, etc. These counters can be read with the *RDPMC* instruction.

Uncore/L3 Performance Counters

In addition to the per-core performance counters described above, recent processors also have a number of programmable global performance counters that can, in particular, count events related to the shared L3 caches. On Intel CPUs, these counters can only be read in kernel space.

[2] "Read from model specific register"
[3] "Read performance-monitoring counters"

2.2.2 Assembler Instructions

Throughout this thesis, we use assembler instructions in Intel syntax. They have the following form:

$$\texttt{mnemonic op}_1\texttt{, op}_2\texttt{, ...}$$

The mnemonic identifies the operation, e.g., *ADD* or *XOR*. The first operand $\texttt{op}_1$ is typically the destination operand, and the other operands are the source operands (an operand can also be both a source and destination operand). Operands can be registers, memory locations, or immediates. Memory operands use the syntax

$$\texttt{[}R_{base}\texttt{+}R_{index}\texttt{*}scale\texttt{+}disp\texttt{]},$$

where R_{base} and R_{index} are general-purpose registers, *disp* is an integer, and *scale* is 1, 2, 4, or 8. All of these components are optional and can be omitted.

In addition to these explicit operands, an instruction can also have implicit operands. As an example, consider the following instruction:

```
ADD RAX, [RBX]
```

This instruction computes the sum of the general-purpose register `RAX` and the memory at the address of register `RBX`, and stores the result in `RAX`. We refer to `RAX` and `[RBX]` as *explicit* operands. In addition to that, the instruction updates the status flags (e.g., the carry flag) according to the result. The status flags are *implicit* operands of the *ADD* instruction.

There are often multiple variants of an instruction with different operand types and/or widths.

Note that there is not always a one-to-one correspondence between assembler code and machine code. Sometimes, there are multiple possible encodings for the same assembler instruction. It is, in general, not possible to control which of these encodings the assembler selects. Thus, some machine instructions cannot be generated using assembler code.

2.3 Features

In this section, we first give a high-level overview by looking at a simple example that shows how *nanoBench* can be used. We then describe various features of *nanoBench* in more detail.

2.3.1 Example

The following example shows how *nanoBench* can be used to measure the latency of the L1 data cache on a Skylake-based system.

```
./nanoBench.sh -asm "mov R14, [R14]"
               -asm_init "mov [R14], R14"
               -config cfg_Skylake.txt
```

The tool will first execute the instruction

```
mov [R14], R14,
```

which copies the value of register `R14` to the memory location that `R14` points to. *nanoBench* always initializes `R14` (and a number of other registers) to point into a dedicated memory area that can be freely modified by microbenchmarks; this is described in more detail in Section 2.3.7.

nanoBench then starts the performance counters and executes the instruction

```
mov R14, [R14]
```

multiple times. The number of repetitions can be controlled via parameters; for more information see Section 2.3.6. The instruction loads the value at the address in `R14` into `R14`. Thus, the execution time of this instruction corresponds to the L1 data cache latency. Afterwards, *nanoBench* stops the performance counters. The entire benchmark is then repeated multiple times to obtain stable results.

The output of *nanoBench* will be similar to the following:

```
Instructions retired: 1.00
Core cycles: 4.00
Reference cycles: 3.52
UOPS_ISSUED.ANY: 1.00
UOPS_DISPATCHED_PORT.PORT_0: 0.00
UOPS_DISPATCHED_PORT.PORT_1: 0.00
UOPS_DISPATCHED_PORT.PORT_2: 0.50
UOPS_DISPATCHED_PORT.PORT_3: 0.50
UOPS_DISPATCHED_PORT.PORT_4: 0.00
UOPS_DISPATCHED_PORT.PORT_5: 0.00
MEM_LOAD_RETIRED.L1_HIT: 1.00
MEM_LOAD_RETIRED.L2_HIT: 0.00
MEM_LOAD_RETIRED.L3_HIT: 0.00
MEM_LOAD_RETIRED.L3_MISS: 0.00
```

The first three lines show the result of the fixed-function performance counters. The remaining lines correspond to the performance events specified in the `cfg_Skylake.txt` configuration file that was supplied as a parameter in the *nanoBench* call shown above; details on configuration files are described in Section 2.3.10.

From the results, we can conclude that the L1 data cache latency is 4 cycles. This agrees with the documentation in Intel's optimization manual [Int19b].

2.3.2 Generated Code

To execute a microbenchmark, *nanoBench* first generates code for a function similar to the pseudocode shown in Algorithm 2.1.

In line 2, the generated code first saves the current values of the registers to the memory and initializes certain registers to point to specific memory locations (see Section 2.3.7). Then, the initialization part of the microbenchmark is executed (line 3). In the next line (line 4), the performance counters are read. Unless the noMem option (see Section 2.3.9) is used, this step does not modify the values in any general-purpose or vector registers that were set by the initialization code (technically, it does modify certain registers temporarily, but it resets them to their previous value before the next line is executed).

Lines 5 to 9 contain the code for the main part of the microbenchmark. The code is unrolled multiple times (this can be configured via a parameter, see Section 2.3.6). If the parameter *loopCount* is larger than 0, the code for a for-loop is inserted in line 5; in this case, the code of the microbenchmark must not modify register `R15`, which is used to store the loop counter.

Afterwards, the performance counters are read a second time (line 10), and in line 11, the registers are restored to the values that were saved in line 2. Finally, the difference between the two performance counter values, divided by the number of repetitions, is returned.

2.3.3 Running the Generated Code

Algorithm 2.2 shows how the generated code is run. The code is run a configurable number of times. At the end, an aggregate function is applied to the measurement results, which can be either the minimum, the median, or the arithmetic mean (excluding the top and bottom 20% of the values). A configurable number of runs in the beginning can be excluded from the result; this is described in more detail in Section 2.3.8.

By default, *nanoBench* generates and runs two versions of the code: the first one with *localUnrollCount* set to the specified *unrollCount*, and the second time with *localUnrollCount* set to two times the specified *unrollCount*. The reported result is the difference between the two runs. This removes the overhead of the measurement instructions from the result, as well as anomalies that might be caused by the serialization instructions that are needed before and after reading the performance counters (see also Section 2.4.1).

nanoBench also provides an option that uses a *localUnrollCount* of 0 for one of the runs instead (i.e., there are no instructions between line 4 and line 10 in this case).

Algorithm 2.1: Generated code for a microbenchmark

```
1  Function generatedCode()
2      saveRegs
3      codeInit
4      m1 ← readPerfCtrs            // stores results in memory,
                                    // does not modify registers
5      for j ← 0 to loopCount do        // this line is omitted
                                        // if loopCount=0
6          code // copy #1
7          code // copy #2
8          ⋮
9          code // copy #localUnrollCount
10     m2 ← readPerfCtrs
11     restoreRegs
12     r ← (m2 − m1)/(max(1, loopCount) ∗ localUnrollCount)
13     return r
```

Algorithm 2.2: Running a microbenchmark

```
1  Function run(code)
2      for i ← −warmUpCount to nMeasurements do
3          m ← code()
4          if i ≥ 0 then                    // ignore warm-up runs
5              measurements[i] ← m
6      return agg(measurements)      // apply aggregate function
```

2.3.4 Kernel/User Mode

nanoBench is available in two versions: A user-space and a kernel-space version. The kernel-space version has several advantages over the user-space version:

- It makes it possible to benchmark privileged instructions.
- It can allow for more accurate measurement results as it disables interrupts and preemptions during measurements.
- It can use several performance counters that are not accessible from user space, like the uncore counters on Intel CPUs, or the APERF and MPERF counters.
- It can allocate physically-contiguous memory. See also Section 2.3.7.

On the other hand, executing microbenchmarks in kernel space can lead to potential data loss and security problems if the microbenchmarks contain bugs. It is thus recommended to use the kernel-space version only on dedicated test machines.

2.3.5 Interface

We provide a unified interface to the user-space and the kernel-space version in the form of two shell scripts, `nanoBench.sh` and `kernel-nanoBench.sh`, that have mostly the same command-line options.

In addition to that, we also provide a Python interface for the kernel-space version. This interface is used for the tools described in Chapters 3 and 4.

With all interfaces, the code of the microbenchmarks can be specified either as an assembler code sequence in Intel syntax (like in the example in Section 2.3.1), or by the name of a binary file containing x86 machine code.

2.3.6 Loops vs. Unrolling

For microbenchmarks that have code that needs to be repeated several times to obtain meaningful results, there is a trade-off between unrolling the code (i.e., creating multiple copies of it) and executing the code in a loop.

Using a loop has the advantage of keeping the code size small, so that it will fit into the cache. On the other hand, the loop introduces an additional overhead, which can be significant if the body of the loop is small.

Whether unrolling or a loop should be used, depends on the particular benchmark. For benchmarks that measure, e.g., the number of data cache misses, a loop is the better choice, as it does not introduce any overhead in terms of memory accesses. On the other, for a benchmark that measures the port usage of an instruction, using only unrolling is better, as otherwise, the µops of the loop code compete for ports with the µops of the benchmark. For some benchmarks, a combination of both a loop and unrolling yields the best results.

nanoBench provides two parameters, *loopCount* and *unrollCount*, that control the number of loop iterations, and how often the code is unrolled.

2.3.7 Accessing Memory

nanoBench initializes the registers `RSP` (i.e., the stack pointer), `RBP` (i.e., the base pointer), `RDI`, `RSI`, and `R14` to point into dedicated memory areas (of 1 MB each) that can be freely modified by the microbenchmarks.

Furthermore, for microbenchmarks needing a larger memory area, like benchmarks for determining cache parameters, the kernel version of *nanoBench* provides an option for reserving a physically-contiguous memory area of a configurable size (see also Section 2.4.4).

2.3.8 Warm-Up Runs

nanoBench provides the option of performing a configurable number of initial benchmark runs that are excluded from the results. This can, for example, be useful to make sure that the code and other accessed memory locations are in the cache. It can also be used to train the branch predictor to reduce the number of mispredicted branches. Furthermore, there are some instructions that require a warm-up period after having not been used for a while before they can execute at full speed again, like AVX2 instructions on some microarchitectures.

2.3.9 noMem Mode

By default, the code to read the performance counters writes the results to the memory. After a warm-up run, this memory location is usually in the cache, and thus, the time for these memory operations is constant.

However, for microbenchmarks that contain many memory accesses to different addresses that map to the same cache set, writing the performance counter

results to the memory can be problematic. One reason for this is that the memory accesses in line 4 may change a cache state that was established by the initialization part of the benchmark. Another reason is that the microbenchmark code may evict the block that stores the performance counter results, which would lead to additional cache misses.

To avoid these problems, *nanoBench* has a special mode that stores all performance counter measurements in registers instead of in memory. If this mode is used, certain general-purpose registers must not be modified by the microbenchmark.

Moreover, if this mode is used, *nanoBench* also provides a feature to temporarily pause performance counting. This feature can be used by including special magic byte sequences in the microbenchmark code for stopping and resuming performance counting. Using this feature incurs a certain timing overhead, so it is in particular useful for benchmarks that do not measure the time but, e.g., the number of cache hits or misses.

2.3.10 Performance Counter Configurations

The performance events to be measured are specified in a configuration file. The file uses a simple syntax to define the events. Unlike in some previous tools, like libpfc [Bil], the events are not hard-coded, which makes it easy to adapt *nanoBench* to future CPUs, as only a new configuration file has to be created.

If the configuration file contains more events than there are programmable performance counters, the benchmarks are automatically executed multiple times with different counter configurations.

We provide configuration files with all events for all recent Intel microarchitectures and the AMD Zen microarchitecture.

2.3.11 Execution Time of nanoBench

Evaluating microbenchmarks with *nanoBench* is very fast. As an example, we consider a benchmark consisting of a single *NOP* instruction, that is run with $unrollCount = 100$, $loopCount = 0$, $nMeasurements = 10$, and a configuration file with four events. On an Intel Core i7-8700K, running *nanoBench* with these parameters takes about 15 ms for the kernel version (assuming that the kernel module is already loaded), and about 50 ms for the user-space version.

2.3.12 Supported Platforms

We have successfully used *nanoBench* on processors from most generations of Intel's Core microarchitecture and with AMD Ryzen CPUs. All experiments were performed under Ubuntu 18.04, but *nanoBench* should be compatible with any Linux distribution that uses a recent kernel version.

2.4 Implementation

2.4.1 Accurate Performance Counter Measurements

Serializing Instruction Execution

As described in Section 2.2, performance counters can be read with the *RDPMC* or the *RDMSR* instruction. These instructions are not serializing instructions. Thus, due to out-of-order execution (see Section 3.3), they may be reordered with earlier or later instructions by the processor. For obtaining meaningful measurement results, it is therefore important to add instructions that serialize the instruction stream both before and after any instructions that read performance counters.

Previous approaches (e.g., [Fog]) often use the *CPUID* instruction for that purpose. However, for benchmarking short code segments, this is problematic. One reason for this is that the *CPUID* instruction has a variable latency and µop count. Paoloni [Pao10] observed that the execution time of the *CPUID* can differ by hundreds of cycles from run to run. The variable µop count can be eliminated by setting the register `RAX` to a fixed value before each execution of the *CPUID* instruction; this also reduces the variance in the execution time, but does not fully eliminate it. Moreover, for an instruction sequence of the form

```
A; CPUID; B,
```

the serialization property of the *CPUID* instruction only guarantees that all µops of `A` have completed before `B` is fetched and executed. It does not guarantee that all µops of `A` have completed before the first µop of the *CPUID* instruction is executed, and it does also not guarantee that all µops of the *CPUID* instruction have completed before the first µop of `B` is executed.

We propose to use the *LFENCE* instruction instead. This instruction is not fully serializing: it does not guarantee that earlier stores have become globally visible, and subsequent instructions may be fetched from memory before *LFENCE* completes. However, on Intel CPUs it does guarantee that

"*LFENCE* does not execute until all prior instructions have completed locally, and no later instruction begins execution until *LFENCE* completes" [Int19c]. For our purposes, this is sufficient, and the guarantee is even somewhat stronger than that for the *CPUID* instruction, as it also orders the *LFENCE* instruction itself with respect to the preceding and succeeding instructions. On AMD CPUs, the *LFENCE* provides similar guarantees if Spectre mitigations are enabled. Using the *LFENCE* instruction for measurements of short durations was also recently recommended by McCalpin [McC18].

Reducing Interference

In the kernel-space version, we disable preemptions and hard interrupts during measurements, as they can perturb the measurement results [WM08, WTM13]. This is not possible for the user-space version; however, we do pin the process to a specific core in this case to avoid the cost of process switches between cores.

Furthermore, for obtaining unperturbed measurement results, we recommend disabling hyper-threading. When using performance counters for resources shared by multiple cores, such as L3 caches, we furthermore recommend disabling all cores that share these resources. We provide shell scripts for this in our repository.

For microbenchmarks that measure properties of caches, such as the benchmarks described in Chapter 4, it can be helpful to disable cache prefetching. On Intel CPUs, this can be achieved by setting specific bits in a model-specific register (MSR). Details on how to do this are available in the documentation of *nanoBench*.

2.4.2 Generating Code

As described in Section 2.3, *nanoBench* runs microbenchmarks by generating a function that contains the code of the microbenchmark, as well as setup and measurement instructions. This is implemented by first allocating a large enough memory area and marking it as executable. Then, the corresponding machine code is written to this memory area, including *unrollCount* many copies of the code of the microbenchmark. If this code contains the magic byte sequences for pausing performance counting as described in Section 2.3.9, they are replaced by corresponding machine code for reading performance counters.

Generating the code for executing the microbenchmarks at runtime in this way makes it possible to access the performance counters without having to execute any function calls or branches.

2.4.3 Kernel Module

The kernel-space version of *nanoBench* is implemented as a kernel module. While the module is loaded, it provides a set of virtual files that are used to configure and run microbenchmarks. For example, setting the loop count, or the code of the microbenchmark is done by writing the corresponding values to specific files under `/sys/nb/`. Reading the file `/proc/nanoBench` generates the code for running the benchmark (as described in Section 2.4.2), runs the benchmark (possibly multiple times, depending on the configuration), and returns the result of the benchmark.

Note that it is usually not necessary to access these virtual files directly, as we provide convenient interfaces that perform these accesses automatically (see Section 2.3.5).

2.4.4 Allocating Physically-Contiguous Memory

In Linux kernel code, the *kmalloc* function can be used to allocate physically-contiguous memory. However, with recent kernel versions, this is limited to at most 4 MB.

Some of the microbenchmarks for determining properties of the L3 caches that we describe in Chapter 4 require larger memory areas. We are not aware of a way to directly allocate larger physically-contiguous memory areas. However, we noticed that in many cases, subsequent calls to *kmalloc* yield adjacent memory areas. This is, in particular, the case if the system was rebooted recently. Moreover, the corresponding virtual addresses are also adjacent.

Based on this observation, we implemented a greedy algorithm that tries to find a physically-contiguous memory area of the requested size by performing multiple calls to *kmalloc*. If this does not succeed, the tool proposes a reboot. Note that allocating memory is only necessary once when the kernel module is loaded, and not before each microbenchmark run.

2.5 Related Work

Perf [per] and Intel's *VTune Amplifier* [Intb] are two examples of tools that are targeted at analyzing whole programs using hardware performance counters. Tools from this category can often display performance statistics at different levels of granularity, sometimes for individual source code lines. However, this data is usually obtained via sampling and, thus, not precise. Such tools are commonly used for identifying the parts of a program that would most benefit from further optimizations.

PAPI [TJYD10] is a widely used tool for accessing performance counters. It provides C and Fortran interfaces that provide functions for configuring and reading performance counters. It can be used for measuring the performance of smaller code segments in the context of a larger program. However, reading the performance counters leads to multiple function calls, branches, and memory accesses. Therefore, it is not suitable for the class of microbenchmarks considered in this thesis.

LIKWID [THW10] is a tool suite providing multiple performance analysis tools. It can both benchmark whole programs, as well as, similar to *PAPI*, specific code regions of a larger program. Reading the performance counters requires expensive system calls [RTHW14].

libpfc [Bil] is a library that was designed in a way to make it possible to use performance counters with a very low overhead. It provides macros with inline assembler code for reading the performance counters. Thus, it does not require function calls or branches. Like our tool, it uses the *LFENCE* instruction to serialize the instruction stream. In fact, a very early version of our tool was based on *libpfc*. However, *libpfc* only supports Haswell CPUs, and it does not support accessing uncore performance counters.

Agner Fog [Fog] provides a framework for running microbenchmarks similar to the microbenchmarks considered in this thesis. The code of the microbenchmark, which is not allowed to use all registers, must be inserted into specific places in a file provided by the framework. The overhead for reading performance counters is relatively small; it does not require function calls or branches. However, the tool uses the *CPUID* instruction for serialization, which can be problematic for short microbenchmarks, as described in Section 2.4.1. The tool only supports a relatively small number of performance events, and it only supports performance counters that can be read with the *RDPMC* instruction (i.e., it does not support uncore counters on Intel CPUs, or the APERF/MPERF counters).

In concurrent work, Chen et al. [CBM+19] present a tool for benchmarking basic blocks using the core cycles, and the L1 data and instruction cache performance counters. Unlike similar tools, Chen et al.'s tool supports microbenchmarks that can make arbitrary memory accesses; this is implemented by automatically mapping all accessed virtual memory pages to a single physical page. The tool was used to train Ithemal [MRAC19], which is a basic block throughput predictor that is based on a neural network. Chen et al. also propose a benchmark suite, called *BHive*, that consists of more than 300,000 basic blocks, and they use their tool to obtain throughput measurements for these basic blocks on CPUs with the Ivy Bridge, Haswell, and Skylake microarchitectures. The code that reads the performance counters contains branches, and it uses the *CPUID* instruction for serialization; however, it lacks a serialization instruction after reading the core cycles counter for the first time. As a consequence, the measurement results are relatively noisy. For Skylake, for example, we found in the *BHive* benchmark suite about 20,000 basic blocks that have instructions with memory operands, but a measured throughput value smaller than 0.5; for more than 2,200 of these blocks, the measured throughput value was even smaller than 0.45[4]. These throughput values are obviously incorrect, since Skylake can execute at most two instructions with memory operands per cycle. Chen et al. compare their measurement results with predictions from Ithemal and several other throughput prediction tools, including Intel's IACA (see also Section 3.2.1). As the average deviation of Ithemal's predictions from the measured throughputs is smaller than the average deviations of the other tools, the authors conclude that Ithemal outperforms the other tools.

None of the existing tools that we are aware of allows for executing benchmarks directly in kernel space.

2.6 Conclusions and Future Work

We have presented a new tool that significantly reduces the engineering effort required for evaluating small microbenchmarks in an accurate and precise way.

We demonstrate the usefulness of our tool in the following two chapters. In Chapter 3, we show how it can be used to characterize the latency, throughput, and port usage of x86 instructions. In Chapter 4, we use *nanoBench* to evaluate microbenchmarks for analyzing cache properties.

[4] https://github.com/ithemal/bhive/issues/1

Future Work

On Intel CPUs, the performance counters can be configured in way such that when one of the counters overflows, all counters stop counting. Recently, Brandon Falk [Fal19] proposed an approach that uses this feature in a creative way to get cycle-by-cycle performance data. The main idea is to set the value of the core cycles counter to N cycles below overflow before the measurements, and to repeat the measurements multiple times with different values for N. Falk implemented this technique in a custom CPU research kernel, called *Sushi Roll*, that is, unfortunately, not publicly available. We plan to add a similar functionality to *nanoBench*.

Furthermore, we would also like to adapt *nanoBench* to non-x86 architectures, such as ARM, MIPS, or RISC-V [WLPA14].

3

uops.info: Characterizing the Latency, Throughput, and Port Usage of Instructions on x86 Microarchitectures

In this chapter, we present the design and implementation of a tool to construct faithful models of the latency, throughput, and port usage of x86 instructions.

To this end, we first discuss common notions of instruction throughput and port usage, and introduce a more precise definition of latency that, in contrast to previous definitions, considers dependencies between different pairs of input and output operands.

We then develop novel algorithms to infer the latency, throughput, and port usage based on automatically-generated microbenchmarks that are more accurate and precise than existing work. The microbenchmarks are evaluated using *nanoBench* (see Chapter 2).

The output of our tool is provided both in the form of a human-readable, interactive HTML table and as a machine-readable XML file.

We provide experimental results for many recent microarchitectures and discuss various cases where the output of our tool differs considerably from prior work.

This chapter is an extended version of [AR19]. We analyze seven additional microarchitectures (including two from AMD) and more than $10,000$ *additional instruction variants. Furthermore, we provide a more detailed description of our algorithms and an enhanced evaluation.*

3.1 Introduction

Developing tools that predict, explain, or even optimize the performance of software is challenging due to the complexity of today's microarchitectures. Unfortunately, this challenge is exacerbated by the lack of a precise documentation of their behavior.

While the high-level structure of modern microarchitectures is well-known and stable across multiple generations, lower-level aspects may differ considerably between microarchitecture generations and are generally not as well documented. An important aspect with a relatively strong influence on performance is how ISA instructions decompose into micro-operations (µops), which ports these µops may be executed on, and what their latencies are.

Knowledge of this aspect is required, for instance, to build performance-analysis tools like CQA [CRON+14], Kerncraft [HHEW15], UFS [PWKJ16], llvm-mca [Bia18], or OSACA [LHH+18, LHHW19]. It is also useful when configuring cycle-accurate simulators like Zesto [LSX09], gem5 [BBB+11], McSim+ [ALOJ13], or ZSim [SK13]. Optimizing compilers, such as GCC [GCC] or LLVM [LA04], can profit from detailed instruction characterizations to generate efficient code for a specific microarchitecture. Similarly, such knowledge can be helpful when manually fine-tuning a piece of code for a specific processor.

Unfortunately, information about the port usage, latency, and throughput of individual instructions at the required level of detail is hard to come by. Intel's processor manuals [Int12, Int19b] only contain latency and throughput data for a number of "commonly-used instructions." They do not contain information on the decomposition of individual instructions into µops, nor do they state the execution ports that these µops can use.

The only way to obtain accurate instruction characterizations for many recent microarchitectures is thus to perform measurements using microbenchmarks. Such measurements are aided by the availability of performance counters that provide precise information on the number of elapsed cycles and the cumulative port usage of instruction sequences. A relatively large body of work (see Section 4.5.1) uses microbenchmarks to infer properties of the memory hierarchy. Another line of work [JEJI08, GJB+10, GJ11, BBG+12] uses automatically generated microbenchmarks to characterize the energy consumption of microprocessors. Comparably little work [CRON+14, Fog19, Gra17, Ins] is targeted at instruction characterizations. Furthermore, existing approaches, such as [Fog19], require significant manual effort to create the

microbenchmarks and to interpret the results of the experiments. Furthermore, its results are not always accurate and precise, as we will show later.

In this chapter, we develop a new approach that can automatically generate microbenchmarks in order to characterize the latency, throughput, and port usage of instructions on x86 CPUs in an accurate and precise manner.

Before describing our algorithms and their implementation, we first discuss common notions of instruction latency, throughput, and port usage. For the latency, we propose a new definition that, in contrast to previous definitions, considers dependencies between different pairs of input and output operands, which enables more accurate performance predictions.

We then develop algorithms that generate assembler code for microbenchmarks to measure the properties of interest for most x86 instructions. Our algorithms take into account explicit and implicit dependencies, such as, e.g., dependencies on status flags. Therefore, they require detailed information on the x86 instruction set. We create a machine-readable XML representation of the x86 instruction set that contains all the information needed to automatically generate assembler code for every instruction. The relevant information is automatically extracted from the configuration files of Intel's *x86 Encoder Decoder (XED)* [Intc] library.

We have implemented our algorithms in a tool that we have successfully applied to 16 different Intel and AMD microarchitectures. In addition to running the generated microbenchmarks on the actual hardware using *nanoBench* (see Chapter 2), we have also implemented a variant of our tool that runs them on top of IACA [Inta]. IACA is a closed-source tool published by Intel that can statically analyze the performance of loop kernels on different Intel microarchitectures. It is, however, not updated anymore and its results are not always accurate, as we will show later.

The output of our tool is available both in the form of a human-readable, interactive HTML table and as a machine-readable XML file, so that it can be easily used to implement, e.g., simulators, performance prediction tools, or optimizing compilers.

Finally, we discuss several interesting insights obtained by comparing the results from our measurements with previously published data. Our precise latency data, for example, uncovered previously undocumented differences between different microarchitectures. It also explains discrepancies between previously published information. Apart from that, we uncovered various errors in IACA, and inaccuracies in the manuals.

3.2 Related Work

In this section, we describe existing sources of detailed instruction data for recent Intel and AMD microarchitectures. We first consider information provided by the manufacturers directly and then look at measurement-based approaches.

3.2.1 Information Provided by the Manufacturers

Intel

Intel's *Optimization Reference Manuals* [Int12, Int19b] contain a set of tables with latency and throughput data for "commonly-used instructions." The tables are not complete; for some instructions, only throughput information is provided. The manuals do not contain detailed information about the port usage of individual instructions.

For the most recent microarchitecture (Ice Lake), Intel provides a machine-readable file with latency and throughput numbers for a relatively large number of instruction variants [Int19a]. However, the file does not contain information on the port usage of these instruction variants.

IACA [Inta] is a closed-source tool developed by Intel that can statically analyze the performance of loop kernels on several microarchitectures (which can be different from the system that the tool is run on). The tool generates a report which includes throughput and port usage data of the analyzed loop kernel. By considering loop kernels with only one instruction, it is possible to obtain the throughput of the corresponding instruction. However, it is, in general, not possible to determine the port bindings of the individual µops this way. Early versions of IACA were also able to analyze the latency of loop kernels; however, support for this was dropped in version 2.2. As of April 2019, IACA has reached its "End Of Life"[1].

The instruction scheduling components of LLVM [LA04] for Sandy Bridge, Haswell, Broadwell, and Skylake were recently updated and extended with latency and port usage information that was, according to the commit message[2], provided by the architects of these microarchitectures. The resulting models are available in the LLVM repository.

[1] https://software.intel.com/en-us/articles/intel-architecture-code-analyzer
[2] `https://reviews.llvm.org/rL307529`

AMD

AMD provides a spreadsheet with latency, throughput, and port usage data for Family 17h processors [AMD17]. This data "is based on estimates and is subject to change." The document was last updated in 2017, when Zen was AMD's current microarchitecture. It is unclear in how far the data also applies to CPUs with the Zen+ or the Zen 2 microarchitecture, which are also Family 17h processors.

3.2.2 Measurement-Based Approaches

Agner Fog [Fog19] provides lists of instruction latency, throughput, and port usage data for different x86 microarchitectures. The data in the lists is not complete; for example, latency data for instructions with memory operands is often missing. The port usage information is sometimes inaccurate or imprecise; we discuss reasons for this in Section 3.5.1. The data is obtained using a set of test scripts developed by the author. These test scripts generate microbenchmarks that are evaluated using Fog's measurement framework (see Section 2.5). The outputs of the microbenchmarks have to be interpreted manually to build the instruction tables.

CQA [CRON+14] is a performance analysis tool for x86 code that requires latency, throughput, and port usage data to build a performance model of a microarchitecture. It includes a microbenchmark module that supports measuring the latency and throughput of many x86 instructions. For non-supported instructions, the authors use Agner Fog's instruction tables [Fog19]. The paper briefly mentions that the module can also measure the number of µops that are dispatched to execution ports using performance counters, but no further details are provided.

EXEgesis [Goo] is a project that can create a machine-readable list of instructions by parsing the PDF representation of Intel's *Software Developer's Manual* [Int19c]. One of the goals of the project is also to infer latencies and µop scheduling information for different instruction and microarchitecture pairs.

Granlund [Gra17] presents measured latency and throughput data for different x86 processors. He considers only a relatively small subset of the x86 instruction set.

AIDA64 [Fin] is a commercial, closed-source system information tool that can perform throughput and latency measurements of x86 instructions. Results for many processors obtained using AIDA64 are available at [Ins].

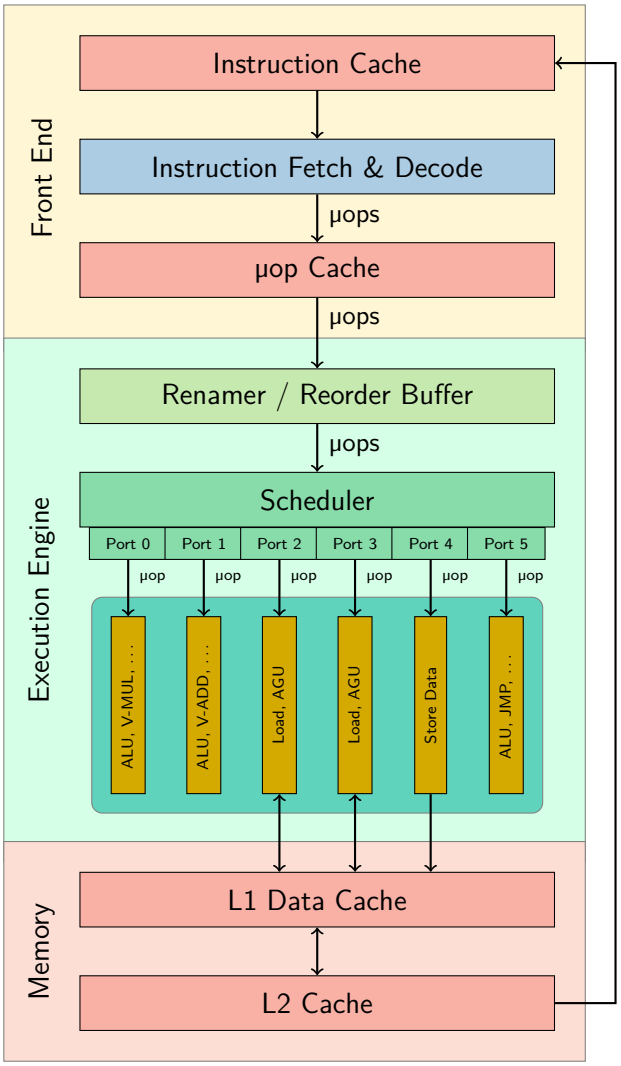

Figure 3.1: Pipeline of Intel Core CPUs (simplified)

3.3 Background

In this section, we describe the pipelines of recent Intel and AMD x86 processors.

3.3.1 Pipeline of Intel Core CPUs

Figure 3.1 shows the general structure of the pipeline of Intel Core CPUs [Int19b, Wika]. The pipeline consists of the front end, the execution engine (back end), and the memory subsystem.

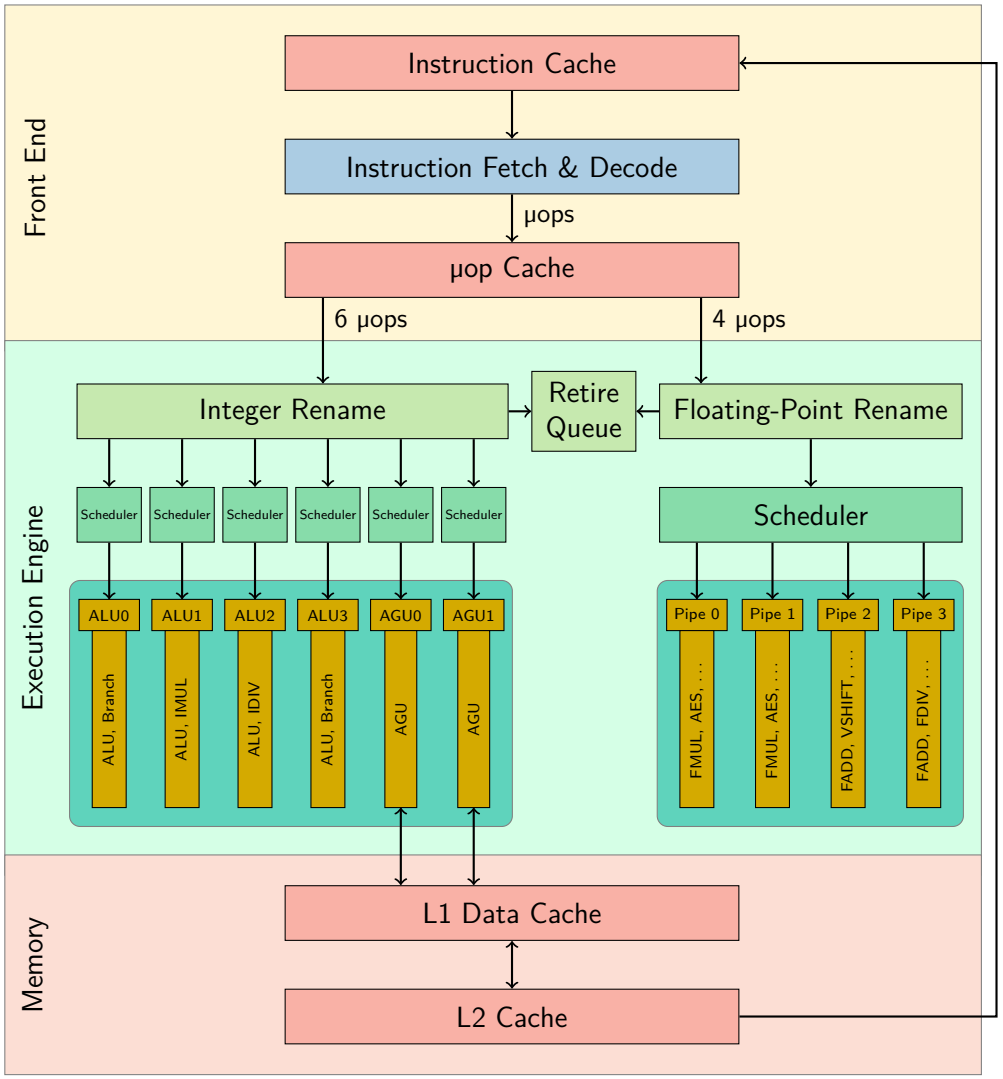

Figure 3.2: Pipeline of AMD Ryzen CPUs (simplified)

The front end is responsible for fetching instructions from the memory and for decoding them into a sequence of micro-operations (µops). Decoded µops are stored in a µop cache.

The *reorder buffer* stores the µops *in order* until they are retired. The renamer is responsible for register allocation (i.e., mapping the architectural registers to physical registers), and for eliminating false dependencies among µops. On some microarchitectures, it can also directly execute certain special µops, including zero idioms (e.g., an *XOR* of a register with itself), and register-to-register moves (*"move elimination"*).

The remaining µops are then forwarded to the *scheduler* (also known as the *reservation station*), which queues the µops until all their source operands are ready. Once the operands of a µop are ready, it is dispatched through an execution port. Due to *out-of-order* execution, µops are not necessarily dispatched in program order.

Each port (Intel Core microarchitectures have 6, 8, or 10 of them) is connected to a set of different functional units, such as an ALU, an address-generation unit (AGU), or a unit for vector multiplications.

Each port can accept at most one µop in every cycle. However, as most functional units are fully pipelined, a port can typically accept a new µop in every cycle, even though the corresponding functional unit might not have finished executing a previous µop. An exception to this are the divider units, which are not fully pipelined.

All recent Intel processors have performance counters for the number of µops that are executed on each port.

3.3.2 Pipeline of AMD Ryzen CPUs

Figure 3.2 shows the general structure of the pipelines of recent AMD processors [AMD17, Wikb]. A main difference to Intel's pipelines is that the execution engine is split into two parts: The left part handles integer and memory operations, while the right part handles floating-point and SIMD operations. The *retire queue* (which is similar to the *reorder buffer* in Intel processors) is shared between both parts.

The functional units are grouped into so-called *pipes*. Each pipe can start executing at most one µop in every cycle. As most functional units are fully pipelined, each pipe can typically accept a new µop in every cycle. For the purpose of the discussion in this chapter, pipes can thus be considered to be similar to ports on Intel CPUs, and we will use the two terms interchangeably in the remainder of this chapter.

Recent AMD CPUs have performance counters for each of the four floating-point pipes. However, there are no individual counters for the pipes on the integer side.

3.4 Definitions

In this section, we define the microarchitectural properties we want to infer, i.e., latency, throughput, and port usage.

3.4.1 Latency

The latency of an instruction is commonly [Int19b] defined as the "number of clock cycles that are required for the execution core to complete the execution of all of the µops that form an instruction" (assuming that there are no other instructions that compete for execution resources). Thus, it denotes the time from when the operands of the instruction are ready and the instruction can begin execution to when the results of the instruction are ready.

This definition ignores the fact that different operands of an instruction may be read and/or written by different µops. Thus, a µop of an instruction I might already begin execution before all source operands of I are ready, and a subsequent instruction I' that depends on some (but not all) results of I might begin execution before all results of I have been produced.

To take this into account, we propose the following definition for latency instead. Let $S = \{s_1, ..., s_m\}$ be the source operands, and $D = \{d_1, ..., d_m\}$ be the destination operands of an instruction. We define the latency of the instruction to be the mapping

$$lat : S \times D \to \mathbb{N}$$

such that $lat(s_i, d_j)$ denotes the time from when source operand s_i becomes ready until the result d_j is ready (assuming all other dependencies are not on the critical path). Thus, if t_{s_i} denotes the time at which source operand s_i becomes ready, then destination operand d_j is ready at time

$$t_{d_j} = max\{t_{s_i} + lat(s_i, d_j) \mid s_i \in S\}.$$

With the usual approach of using a single value lat as the latency of an instruction, this value would be

$$t_{d_j} = max\{t_{s_i} \mid s_i \in S\} + lat,$$

which might be significantly greater than what would be observed in practice.

3.4.2 Throughput

When comparing throughput data from different publications, it is important to note that these publications do not all use the same definition of throughput. Intel defines throughput in its manuals [Int12, Int19b] as follows.

Definition 3.1 (Throughput—Intel)**.** The number of clock cycles required to wait before the issue ports are free to accept the same instruction again.

On the other hand, Agner Fog [Fog19] uses the following definition for (reciprocal) throughput:

Definition 3.2 (Throughput—Fog)**.** The average number of core clock cycles per instruction for a series of independent instructions of the same kind in the same thread.

Granlund [Gra17] uses a similar definition as Fog.

These two definitions are not equivalent, as there can be factors other than contention for the issue ports that may reduce the rate at which instructions can be executed (e.g., the front end, or the memory subsystem). Moreover, it is not always possible to find instructions of the same kind that are truly independent, as many instructions have implicit dependencies on certain registers or flags. Hence, the second definition may yield higher throughput values (corresponding to a lower throughput) than the first definition for the same instruction.

Some publications [Fog19, Gra17] use the term *throughput* to denote *instructions per cycle*, while others [Int19b, Inta] use it to denote *cycles per instruction*. In this chapter, we will use the term with the latter meaning.

3.4.3 Port Usage

Let P be the set of ports of a CPU and U the set of µops of an instruction *instr*. Let $ports : U \to 2^P$ be a mapping such that $ports(u)$ is the set of ports which have a functional unit that can execute the µop u.

We define the port usage $pu : 2^P \to \mathbb{N}$ of *instr* to be a mapping such that

$$pu(pc) = |\{u \in U \mid ports(u) = pc\}|,$$

i.e., $pu(pc)$ denotes the number of µops of *instr* whose functional units are at the ports in pc (we will call the set pc a *port combination*). Note that, e.g., for a 1-µop instruction with a µop u such that $ports(u) = \{0, 1\}$, we have that $pu(\{0, 1\}) = 1$, but $pu(\{0\}) = pu(\{1\}) = 0$.

For, e.g., an instruction with $pu(\{0, 1, 5\}) = 3$, $pu(\{2, 3\}) = 1$, and $pu(pc) = 0$ for all other port combinations pc, we will use the notation $3 * p015 + 1 * p23$ to denote the port usage. In other words, the instruction consists of three µops that may each be executed on ports 0, 1, and 5, and one µop that may be executed on ports 2 and 3.

3.5 Algorithms

In this section, we describe the algorithms that we developed to infer the port usage, the latency, and the throughput.

3.5.1 Port Usage

The existing approach by Agner Fog [Fog19] to determine the port usage measures the number of µops on each port when executing the instruction repeatedly in isolation. If the result of such a measurement is, e.g., that there is, on average, one µop on port 0 and one µop on port 5, the author would conclude that the port usage is $1 * p0 + 1 * p5$.

However, this might be incorrect: A port usage of $2 * p05$ could lead to exactly the same measurement result when the instruction is run in isolation, but to a very different result when run together with an instruction that can only use port 0 (the *PBLENDVB*[3] instruction on the Nehalem microarchitecture is an example for such a case).

In another example, if the measurement result is that there are, on average, 0.5 µops on each of port 0, 1, 5, and 6, the author would conclude that the port usage is $2 * p0156$, whereas the actual usage might, for example, be $1 * p0156 + 1 * p06$ (this is, e.g., the case for the *ADC*[4] instruction on the Haswell microarchitecture).

In this section, we propose an algorithm that can automatically infer an accurate model of the port usage. Our algorithm is based on the notion of a *blocking instruction*: We define a *blocking instruction* for a port combination *pc* to be an instruction whose µops can use all the ports in *pc*, but no other port that has the same functional unit as a port in *pc*. Blocking instructions are interesting because they can be used to determine whether or not an instruction can only be executed on a given set of ports, the set of ports blocked by the blocking instruction.

Before describing our algorithm to infer a model of the port usage we will first describe how to find a suitable set of blocking instructions. Let FU be the set of types of functional units that the CPU uses, and let $ports : FU \rightarrow 2^P$ be a mapping from the functional unit types to the set of ports P that are connected to a functional unit of the given type. The set of port combinations for which we need to find blocking instructions is the set $\{ports(fu) \mid fu \in FU\}$.

[3] "Variable blend packed bytes"
[4] "Add with carry"

We assume that for each of these port combinations (except for the ports that are connected to the *store data* and *store address* units), there is a 1-µop instruction that can use exactly the ports in the combination. This assumption holds on all recent microarchitectures.

Our algorithm first groups all 1-µop instructions based on the ports they use when run in isolation. We exclude several classes of instructions, for example system instructions, serializing instructions, zero-latency instructions, the *PAUSE* instruction, and instructions that can change the control flow based on the value of a register. From the remaining instructions, the algorithm chooses from each group an instruction with the highest measured throughput (see Section 3.5.3) as the blocking instruction for the port combination corresponding to this group; here, we consider only throughput measurements obtained by unrolling (see Section 3.6.2) to ensure that blocking instructions do not lead to bottlenecks in the front end.

As blocking instructions for the port combinations for the ports that are connected to the *store data* and *store address* units, we use the *MOV* instruction (from a general-purpose register to the memory). This instruction is a 2-µop instruction; one of its µops uses the *store data* unit, and the other a *store address* unit.

To avoid SSE-AVX transition penalties when characterizing SSE or AVX instructions, our algorithm determines two separate sets of blocking instructions for these two types of instructions. For SSE instructions, the blocking instructions should not contain AVX instructions, and vice versa.

Port Usage Algorithm

We use Algorithm 3.1 to infer the port usage of an instruction *instr*. The algorithm first sorts the set of port combinations by the size of its elements. This ensures that, when iterating over the port combinations, combinations that are a subset of another port combination are processed first.

For each port combination *pc*, the algorithm determines the number of *µops* that may use all of the ports in *pc* but no others. To determine this set, the algorithm concatenates *blockRep* copies of the corresponding blocking instruction with the instruction that we want to analyze (line 5). *blockRep* is the product of the maximum latency of the instruction (see Section 3.5.2), i.e., the maximum over the latencies for all input/output pairs, and the maximum number of ports. This ensures that there is always a sufficient number of instructions available that can block the ports of the combination. The operands of the copies of the blocking instructions are chosen such that they

Algorithm 3.1: Port usage

```
1  portCombinationsList ← sort(portCombinations)
2  µopsForCombination ← []                          // list of pairs
3  foreach pc in portCombinationsList do
4      blockRep ← 10 · maxLatency(instr)
5      code ← copy(blockingInstr(pc), blockRep) + ";" + instr
6      µops ← measureUopsOnPorts(code, pc)
7      µops ← µops − blockRep
8      foreach (pc′, µops′) in µopsForCombination do
9          if pc′ ⊂ pc then
10             µops ← µops − µops′
11     if µops > 0 then
12         µopsForCombination.add((pc, µops))
13 return µopsForCombination
```

are independent from the operands of *instr* and independent from subsequent instances of the blocking instruction.

When executing the concatenation, instruction *instr* will only be executed on one of the blocked ports if there is no other port that it can be executed on. The algorithm thus measures the number of *µops* that use the ports in the combination when executing the concatenation (line 6). From this value, it subtracts the number of µops, *blockRep*, of the blocking instructions (line 7). The remaining number of *µops* can only be executed on the ports in *pc*, otherwise they would have been executed on other ports.

However, it may have been determined previously for a strict subset pc' of *pc* that some or even all of these *µops* can only be executed on that subset pc'. Thus, the number of $\mu ops'$ on subsets pc' of the port combination *pc*, which have been determined in previous iterations of the loop, are subtracted from *µops* (line 10). The remaining number of *µops* can be executed on all ports in *pc* but on no other ports.

The algorithm can be optimized by first measuring which ports are used when running the instruction in isolation. The loop then does not need to iterate over all port combinations, but only over those whose ports are also used when the instruction is run in isolation. Furthermore, we can exit the loop early when the sum of the *µops* in the *µopsForCombination* list reaches the total number of *µops* of the instruction.

3.5.2 Latency

Let I be an instruction with source operands S and destination operands D. We use the following general approach to determine the latency $lat(s, d)$ for some $s \in S$ and $d \in D$.

Let us first consider the simplest possible case:

1. The type of the source operand s is the same as the type of the destination operand d.
2. All instruction operands are *explicit* register operands, and no register operand is both read from *and* written to by I.
3. The µops of I do not compete for execution ports.

Then we can create a *dependency chain* of copies of I, such that the register for the destination operand of an instance of I is the register used for the source operand of the next instance of I. The other registers should be chosen such that no additional dependencies are introduced.

Given such a chain c of sufficient length, we can determine $lat(s, d)$ by measuring the run time of the chain and dividing it by the length of c.

Let us now consider the case that the types of the source operand s and the destination operand d are different. Then it is impossible to create a dependency chain consisting only of instances of the instruction I. To create a chain we need an instruction C that has a source operand s_C with the same type as d, and a destination operand d_C with the same type as s. We call such an instruction a *chain instruction*. Given a chain instruction C, we can create a chain by concatenating instances of I and C, such that the destination operand of I uses the same register as the source operand of C and vice versa. Assuming we already know the latency $lat_C(s_C, d_C)$ of C, we can determine the latency $lat(s, d)$ by measuring the chain's run time, dividing it by the number of occurrences of I, and by subtracting $lat_C(s_C, d_C)$. Chain instructions should be instructions that have as few as possible other operands, they should ideally be one-µop instructions that can use multiple ports, and their latency should either be known or easy to determine in isolation.

Let us now assume there are *implicit* operands or register operands that are both read from and written to by the instruction I. Such operands are a challenge as they may introduce *additional* dependencies: This is the case if I has implicit operands that are both read from and written to (such as, e.g., status flags), or if $s \neq d$, and s or d are both read from and written to.

In such cases, the run time of a chain involving I may be determined by the latency of the additional dependency, rather than the latency from s to d, which we would like to determine.

Similar issues can occur if there are µops of I that are not needed for computing the output in d based on the input in s, but that compete for execution ports with the µops needed for this dependency.

We handle these issues by adding sufficiently many additional chain instructions to the dependency chain, such that the execution time of the dependency chain exceeds the latencies of any additional dependencies, and µops that are not on the critical path can be executed in parallel to the dependency chain. The number of additional chain instructions is determined based the execution time of running instruction I in isolation (i.e., without any chain instructions).

In the following subsections, we first describe the most interesting cases of how we create dependency chains for different types of source/destination operands. Then, we briefly describe how we determine the latencies of the chain instructions themselves.

Register → Register

Both registers are general-purpose registers If both registers are general-purpose registers, we use the *MOVSX*[5] instruction to create a dependency chain.

We do not use the *MOV* or *MOVZX*[6] instructions for this purpose, as these can be zero-latency instructions on some microarchitectures in some cases, which can be executed by the renamer (*"move elimination"*, see Section 3.3.1). However, *move elimination* is not always successful (in our experiments, we found that in a chain consisting of only (dependent) *MOV* instruction, about one third of the instructions were actually eliminated). Using the *MOVSX* instruction avoids this uncertainty.

Moreover, because the *MOVSX* instruction supports source and destination registers of different sizes, this also avoids problems with *partial register stalls* (see Section 3.5.2.4 of Intel's Optimization Manual [Int19b]). A *partial register stall* occurs when an instruction writes an 8 or 16-bit portion of a general-purpose register, and a subsequent instruction reads a larger part of the register.

[5] "Move with sign-extension"
[6] "Move with zero-extend"

If at least one of the two register operands is not an implicit operand, we could also use the same register for both operands instead of using a chain instruction for measuring the latency. However, if one of the operands is both read and written, it would not be possible to measure the latencies between the two operands separately. Moreover, there are some instructions that behave differently if the same register is used for multiple operands. For example, some instructions with two register operands (like *XOR* and *SUB*) are *zero idioms* that always set the register to zero (independent of the actual register content) if the same register is used for both operands. In all recent microarchitectures, these instructions break the dependency on the register that is used; on some microarchitectures, they can in some cases be executed by the renamer and do not use any execution ports (see Section 3.5.1.7 of Intel's Optimization Manual [Int19b]). There are also other instructions that behave differently on some microarchitectures when the same register is used, for example the *SHLD* instruction (for details see Section 3.7.4).

To be able to detect such a behavior, our algorithm therefore creates microbenchmarks for both scenarios (i.e., using a separate chain instruction, and using the same register for different operands).

A third option would be to chain the instruction with itself by reversing the order of the two operands (i.e., the destination operand of one instruction would use the same register as the source operand of the next instruction). However, as this would not work for instructions with implicit register operands, we do not pursue this alternative.

Both registers are MMX registers In this case, we use the *MOVQ*[7] instruction, which is not a zero-latency instruction, to create a dependency chain. We also perform a separate experiment where the same register is used for different operands, as described above for general-purpose registers.

Both registers are SIMD registers Since all MOV instructions for SIMD registers (i.e., XMM, YMM, and ZMM registers) can be zero-latency instructions on some microarchitectures, we use shuffle instructions instead.

SIMD instructions can perform floating-point or integer operations. If a source for a floating-point operation comes from an integer operation (or vice-versa), a *bypass delay* can occur (see Sections 3.5.1.10 and 3.5.2.3 of Intel's Optimization Manual [Int19b]). To capture such cases, we perform measurements with both a floating-point and an integer shuffle instruction

[7] "Move quadword"

as chain instructions. In particular, we use the *SHUFPD*[8] and *PSHUFD*[9] instructions (with immediate operand 0). For AVX instructions, we use the corresponding VEX encoded instructions (*VSHUFPD* and *VPSHUFD*) to avoid AVX-SSE transition penalties.

As described in the previous paragraphs, we also consider the case where different operands use the same register. However, this is not possible for AVX2 gather instructions, as they require all arguments to be pairwise different.

The registers have different types If the registers have different types (e.g., one is a vector register, and the other a general-purpose register), then it is, in general, not possible to find a chain instruction whose latency could be determined in isolation. Instead, we separately measure and report the execution times for compositions of the instruction with all meaningful chain instructions with the corresponding types. Note that these times might be higher than the sum of the latencies of the instruction and the chain instruction due to bypass delays. If we then take the minimum of these times and subtract 1, we can obtain an upper bound on the latency of the instruction (there are no zero-latency instructions between registers of different types).

Memory → Register

To measure the latency of a *MOV* instruction from the memory to a general-purpose register, we can use a chain of

```
MOV RAX, [RAX]
```

instructions, where we assume that register `RAX` contains the address of a memory location that stores its own address. As its address depends on the result of the previous load, the next load can only begin after the previous load has completed.

However, this simple approach would not work for most other instructions, as they usually do not just copy a value from the memory to a register. Instead, we generalize the approach as follows: Let R_a be the register that contains the memory address, and let R_d be the destination register. We use

`XOR` R_a`,` R_d`;` `XOR` R_a`,` R_d

to create a dependency from R_d to R_a. Note that the double *XOR* effectively leaves R_a unchanged. However, since *XOR* also modifies the status flags,

[8] "Packed interleaved shuffle of pairs of double-precision floating-point values"
[9] "Shuffle packed doublewords"

we additionally add a dependency-breaking instruction for the flags to the chain (i.e., an instruction that writes the flags but does not read them so that subsequent instructions that read the flags do not have to wait until *XOR* completes before they can begin execution). Furthermore, if R_d is an 8 or 16-bit register, we prepend a *MOVSX* instruction to the chain to avoid partial register stalls.

The base register of a memory operand is always a general-purpose register. If the destination register of the instruction is not a general-purpose register, we combine the double *XOR* technique with the approach for registers described in the previous section to obtain an upper bound on the latency.

Status Flags → Register

As there are no instructions that read the status flag register and write a vector register, we only need to consider general-purpose registers here.

To create a dependency from a general-purpose register R to the status flags register, we use the instruction

$$\texttt{TEST}\ R,\ R$$

This instruction reads both register operands (we use for both operands the same register), and writes all status flags (except the `AF` flag). It has no other dependencies.

Register → Memory

It is not directly possible to measure the latency of a store to memory, i.e., the time until the data has been written to the L1 cache. We can, however, measure the execution time of a chain with a load instruction. For the *MOV* instruction, we could, e.g., measure the execution time of the sequence

```
MOV [RAX], RBX; MOV RBX, [RAX].
```

However, the execution time of this sequence might be lower than the sum of the times for a load and for a store. One reason for this is "store to load forwarding", i.e., the load obtains the data directly from the store buffer instead of through the cache. The second reason is that the address of the load does not depend on the preceding store, and thus the address computation might already begin before the store.

While the time of such a sequence does not directly correspond to the latency, it still might provide valuable insights. We therefore measure the execution

time in a chain with a suitable load instruction for all instructions that read a register and store data to the memory.

In particular, we use the *MOV* instruction for general-purpose registers, the *MOVQ* instruction for MMX registers, and the *(V)MOVSS*[10], *(V)MOVD*[11], *(V)MOVSD*[12], *(V)MOVQ*[13], *(V)MOVUPD*[14], and *(V)MOVDQU*[15] instructions for SSE and AVX registers.

Status Flags → Memory

If the source is a status flag, we use the *TEST* instruction instead of the *MOV* instruction. The *TEST* instruction reads from a memory location and writes all status flags (except the `AF` flag).

Memory → Memory

Almost all instructions that both read from and write to memory do so using the same operand (and thus, the same address). Therefore, we can just chain them with themselves. For instructions that use different operands, we make sure that both operands use the same address.

Address → Memory

For instructions that store to memory, we can additionally consider the dependency from when the address of the store is ready to when the data is stored in memory. We do this by combining the approach described under "Register → Memory" with the double *XOR* technique described under "Memory → Register".

Register → Status Flags

If the source register is an 8-bit general-purpose register, we use the corresponding variant of the *SETcc*[16] instruction for the status flag to create a dependency chain. If the source is a general-purpose register that is wider than 8 bits, we use the *CMOVcc*[17] instruction with the source register as first

[10] "Move or merge scalar single-precision floating-point value"
[11] "Move doubleword"
[12] "Move or merge scalar double-precision floating-point value"
[13] "Move quadword"
[14] "Move unaligned packed double-precision floating-point values"
[15] "Move unaligned packed integer values"
[16] "Set byte on condition"
[17] "Conditional move"

operand and an otherwise unused register as second operand. This distinction is necessary to avoid partial register stalls.

If the source register is a vector register, we can obtain an upper bound on the latency by taking the minimum execution time over the combination of the *CMOVcc* instruction with all instructions that read a general-purpose register and write to a vector register of the given type. Note that there is no instruction that reads a status flag and writes to a vector register.

Memory → Status Flags

To create a dependency chain, we use the *CMOVcc* instruction using the base register of the memory address for both operands.

Status Flags → Status Flags

If the instruction reads and writes the carry flag, we use the *CMC*[18] as chain instruction. We currently do not consider dependencies between other status flags.

Branch Instructions

For instructions that change the control flow (i.e., they write to the RIP register), it would be possible to create a dependency chain by using the *LEA*[19] instruction, which can directly read the `RIP` register in 64-bit mode. However, due to speculative execution, this would not be a true dependency. We therefore do not consider such dependencies (we do, however, consider any other dependencies such an instruction might have).

Divisions

For instructions that use the divider units, it is known that their latency depends on the contents of their register (and memory, where applicable) operands. We test these instructions both with values that lead to a high latency, and with values that lead to a low latency (we obtained those values from Agner Fog's [Fog19] test scripts). As most of these instructions use one operand both as input and output operand, and the output of a division with a value that leads to a high latency is often a value that leads to a lower latency, the techniques described in the previous sections for automatically creating dependency chains cannot be used in this case. We therefore handle

[18] "Complement carry flag"
[19] "Load effective address"

these instructions separately. If, e.g., R is a register that is both a source and a destination register, and R_c contains a value that leads to a high latency, we can use

$$\texttt{AND } R, R_c\texttt{; OR } R, R_c,$$

or the corresponding vector instructions, to create a dependency chain that always sets R to the same value.

Latencies of the Chain Instructions

As chain instructions, we generally use instructions whose latencies can be determined in isolation, i.e., without needing other chain instructions. For example, the *MOVSX* instruction has only one input and one output operand, and we can determine its latency by, e.g., executing

```
MOVSX RAX, EAX
```

multiple times and dividing the execution time by the number of repetitions (here, `EAX` refers to the lower 32 bits of the `RAX` register).

There are a few exceptions. An example are the *TEST* and *SETcc* instructions that we use to create a dependency chain from a general-purpose register to the status flags and vice versa. While we cannot measure the latencies of these two instructions in isolation, we can measure the time for a combination of the two, for example with the code sequence

```
TEST AL, AL; SETC AL
```

On all CPUs we tested, this combination needs two cycles, and thus we know that each of the instructions needs one cycle. If our test program measures a different execution time for this combination, it would abort with an error message and we would need to find a different approach for this case.

3.5.3 Throughput

As mentioned in Section 3.4.2, there are different ways of defining throughput. We will now first describe how we can measure the throughput according to Definition 3.2. Then, we will show how the throughput according to Definition 3.1 can be computed from the port usage.

Measuring Throughput

To measure the throughput of an instruction, we first try to find a sequence of independent instances of the instruction that avoids read-after-write dependencies as much as possible. To this end, we select registers and memory

locations in a way such that they are not written by one instruction of the sequence and read by a subsequent instruction. This is, however, not possible for implicit operands that are both read and written.

We then measure the execution time over several repetitions of this sequence and obtain the throughput by dividing this time by the total number of instructions that have been executed.

We observed that sometimes longer sequences of independent instruction instances can lead to higher execution times per instruction than shorter sequences, in particular, when they use many different memory locations or registers. We therefore perform measurements for sequences of different lengths (we consider sequences with 1, 2, 4, and 8 elements).

For instructions with implicit operands that are both read and written, we additionally consider sequences with dependency-breaking instructions. However, as the dependency-breaking instructions also consume execution resources, this does not necessarily lead to a lower execution time of the sequence in all cases.

For instructions that use the divider units, the throughput can depend on the value of their operands. We test these instructions both with values that lead to a high throughput, and with values that lead to a low throughput. For this, we use the same values that we used to measure the latencies of such instructions.

Computing Throughput from Port Usage

Intel's definition of throughput (Definition 3.1) assumes that the ports are the only resource that limits the number of instructions that can be executed per cycle, and that there are no implicit dependencies.

If we execute an instruction for which these requirements hold repeatedly, then the average wait time until the next instruction can be executed corresponds to the average usage (per instruction) of the port with the highest usage, and the number of µops on this port will be equal to the execution time (however, this is not true for instructions that use the divider unit, which is not fully pipelined).

For instructions, for which the above requirements do not hold, it is not possible to directly measure the throughput according to Intel's definition.

However, for instructions that do not use the divider unit, it can be computed from the port usage measured in Section 3.5.1. For 1-µop instructions, the

throughput is $\frac{1}{|P|}$, where P is the set of ports that the µop can use. More generally, the throughput is the solution of the following optimization problem, where PU is the result from Algorithm 3.1, and $f(p, pc)$ are variables:

$$\begin{array}{lll} \text{Minimize} & \max\limits_{p \in Ports} \sum\limits_{(pc,\mu) \in PU} f(p, pc) & \\ \text{Subject to} & f(p, pc) = 0 & p \notin pc \\ & \sum\limits_{p \in Ports} f(p, pc) = \mu & (pc, \mu) \in PU \end{array}$$

The variable $f(p, pc)$ captures the share of the µops that map to the port combination pc that are scheduled on port p. A schedule maximizing the throughput will minimize the maximum port load $\max_{p \in Ports} \sum_{(pc,\mu) \in PU} f(p, pc)$.

We can reduce this optimization problem to a linear program by replacing the maximum in the objective with a new variable z, and adding constraints of the form

$$\sum_{(pc,\mu) \in PU} f(p, pc) \leq z$$

for all $p \in Ports$. The linear program can be solved using standard LP solvers.

3.6 Implementation

In this section, we describe various aspects of our implementation of the algorithms developed in Section 3.5.

3.6.1 Details of the x86 Instruction Set

The algorithms described in Section 3.5 require detailed information on the x86 instruction set, including, e.g., the types and widths of (implicit and explicit) operands. While this information is available in Intel's *Software Developer's Manual* [Int19c], there was, until recently, no sufficiently precise machine-readable description of the instruction set.

Fortunately, Intel recently published the source code of their *x86 Encoder Decoder (XED)* [Intc] library. The build process of this library uses a set of configuration files that contain a complete description of the syntax of the x86 instruction set. While this description is very concise, it is not well documented and quite complex to parse (collecting the information for a single instruction requires reading multiple files). It also contains a lot of low-level details, e.g., regarding the encoding, that are not needed for our purposes.

We therefore convert this format to a simpler XML representation that contains enough information for automatically generating assembler code for each instruction variant, and that also includes information on the operand types and on whether they are explicit or implicit.

We then use this representation for the implementation of the algorithms described in Section 3.5. The representation contains, for example, all the information that is necessary to automatically pick the corresponding chain instructions for each instruction variant from the list in Section 3.5.

3.6.2 Measurements on the Hardware

We use *nanoBench* (see Chapter 2) for evaluating the generated microbenchmarks using hardware performance counters. In particular, we measure the number of µops on each port and the number of core clock cycles (which can be different from reference cycles due to, e.g., frequency scaling). We use the kernel-space version of *nanoBench* (see Section 2.3.4), which makes it possible to benchmark privileged instructions, allows for more accurate measurements, and provides access to the APERF counter on AMD CPUs for counting core clock cycles.

We configure *nanoBench* to use unrolling (see Section 2.3.6). The unroll count is chosen such that the code is small enough to fit in the instruction cache, but large enough to allow for accurate results.

For measuring the throughput, we additionally perform an experiment with a loop and a small unroll count, such that the code fits into the µop cache (see Section 3.3). While the code for the loop introduces an additional overhead, this can nonetheless lead to a higher throughput, as there are instructions for which decoding is a bottleneck when only unrolling is used. On Intel CPUs, this is for example the case for instructions that have a "Length-Changing Prefix" (LCP, see Section 3.4.2.3 of [Int19b]); with such instructions, the CPU uses a slower length-decoding algorithm. For measuring the port usage, this is not an issue, as we do not use such instructions as blocking instructions. For measuring latencies, this is also not an issue, as the chain instructions introduce a sufficient amount of delay to hide such bottlenecks.

Some instructions require that registers or memory locations contain valid values of a given type. Floating-point instructions, for example, require that operands represent valid floating-point numbers (that are not denormal); otherwise, delays and/or exceptions can occur. Before executing SSE instructions on systems that support AVX (i.e., where parts of the XMM registers are shared with YMM registers), the upper bits of the registers need to zeroed

(using the *VZEROUPPER*[20] or the *VZEROALL*[21] instructions) to avoid false dependencies. We implemented this using the initialization sequence feature of *nanoBench* (see Section 2.3).

3.6.3 Analysis Using IACA

In addition to running the code sequences generated by our algorithms on the actual hardware, we also implemented a variant of our tool that automatically analyzes them with Intel's IACA tool (see also Section 3.2.1). IACA treats the code sequences as the body of a loop, and reports average throughput and port usage values for multiple iterations of this loop. Thus, the results should correspond to the measurements on the actual hardware, which are also averages over executing the code sequences multiple times.

We consider the IACA versions 2.1, 2.2, 2.3, and 3.0. Intel added support for more recent microarchitectures in the newer versions, but at the same time dropped support for older ones. For microarchitectures that are supported by multiple versions, we analyze the code sequences with all of these versions, as we observed (undocumented) differences between the results from different versions of IACA for the same instructions.

3.6.4 Machine-Readable Output

We store the results of our algorithms in a machine-readable XML file. The file contains the results for all tested microarchitectures, both as measured on the actual hardware and as obtained from running our microbenchmarks on top of IACA.

3.7 Evaluation

In this section, we first describe the platforms on which we ran our tool.

Then, we compare our latency and throughput results for the Ice Lake microarchitecture with Intel's documentation. After that, we compare the results obtained by running our microbenchmarks on the actual hardware with those obtained by analyzing them with IACA.

Finally, we discuss several insights that we obtained from the measurement results.

[20] "Zero upper bits of YMM and ZMM registers"
[21] "Zero XMM, YMM and ZMM registers"

Table 3.1: Tested microarchitectures and number of instruction variants for which measurement results were obtained

Microarchitecture	Processor	Release Date	# Instr.
Conroe	Intel Core 2 Duo E6750	Q3'07	1946
Wolfdale	Intel Core 2 Duo E8400	Q1'08	2042
Nehalem	Intel Core i5-750	Q3'09	2076
Westmere	Intel Core i5-650	Q1'10	2090
Sandy Bridge	Intel Core i7-2600	Q1'11	2797
Ivy Bridge	Intel Core i5-3470	Q2'12	2808
Haswell	Intel Xeon E3-1225 v3	Q2'13	3366
Broadwell	Intel Core i5-5200U	Q1'15	3377
Skylake	Intel Core i7-6500U	Q3'15	3420
Kaby Lake	Intel Core i7-7700	Q1'17	3420
Skylake-X	Intel Core i9-7900X	Q2'17	12523
Coffee Lake	Intel Core i7-8700K	Q4'17	3420
Cannon Lake	Intel Core i3-8121U	Q2'18	12673
Ice Lake	Intel Core i5-1035G1	Q3'19	13326
Zen+	AMD Ryzen 5 2600	Q2'18	3399
Zen 2	AMD Ryzen 7 3700X	Q3'19	3401

3.7.1 Experimental Setup

We ran our tool on the platforms described in Table 3.1, which includes processors from most Intel microarchitectures released in the last 13 years, as well as AMD Zen+ and Zen 2 processors. The last column in Table 3.1 shows the number of instruction variants for which we obtained measurement results. The numbers are higher for newer microarchitectures due to their larger instruction sets. Figure 3.3 shows a subset of the actual machines we used.

All experiments were performed using Ubuntu 18.04. To reduce the risk of interference, we disabled hyper-threading. The total run time of our tool ranges from around 80 minutes (on the Coffee Lake system), to around 20 hours (on the Cannon Lake system).

3.7.2 Hardware Measurements vs. Documentation

For the Ice Lake microarchitecture, Intel has published a machine-readable csv file with latency and throughput data for a relatively large number of instruction variants [Int19a]. Moreover, the naming of the instruction variants

Figure 3.3: Experimental setup

is based on Intel's *XED* library (see Section 3.6.1), which makes a comparison with our data relatively straightforward. However, we use a somewhat more fine-grained definition of *instruction variant*, which means that in some cases, an entry in Intel's file corresponds to multiple variants in our file. For example, we treat AVX-512 instructions with and without a zeroing modifier as two separate variants, whereas Intel's file considers them to be the same variant.

For the throughput, Intel provides data for 9789 instruction variants (according to our definition). For 9723 of these variants (i.e., 99.33%), our throughput results (as computed from the port usage according to Section 3.5.3) agree with Intel's data. For 16 of the remaining instruction variants, the differences are due to rounding (0.67 vs 0.66). 24 of the remaining instruction variants are *move* instructions that do not need execution ports (*move elimination*, see Section 3.3.1), and hence, our approach could not compute a throughput from the port usage. For 12 variants of the *V(P)BLENDV** instructions, we obtained a port usage of *2*p015*, and thus, a throughput of 0.67; however, Intel reports a throughput of 1. The remaining 14 instruction variants are division instructions, for which the differences between our results and Intel's values are probably due to them not being fully pipelined (see Section 3.3.1).

We now compare the throughput in Intel's file with the measured throughput (see Section 3.5.3). This comparison might appear to not be very meaningful, as the data in Intel's file is based on Definition 3.1, whereas the measured throughput data is based on Definition 3.2. However, the comparison can provide some insights into how big the difference between the two definitions actually is in practice. In 9594 cases (i.e., 98%), the throughput values are the same. In 9762 cases (i.e., 99.72%), the throughput values differ by at most 0.1 cycles. Interestingly, for the *V(P)BLENDV** and the division instruction variants described above, the measured throughput does agree with Intel's throughput values.

If we compare the measured throughput values with the throughput values computed from the port usage for all 13326 instruction variants, for which we obtained measurement data, then in 7.03% of the cases, the values differ by more than 0.1 cycles. This includes, in particular, division instruction, *GATHER/SCATTER* instructions, instructions with a *LOCK* prefix, instructions that modify the control flow, and several system instructions; for these instructions, factors other than the contention on execution ports, seem to limit the measured throughput. For most of these instructions, no throughput data is available in Intel's file.

Finally, we compare our measured latency values with the data in Intel's file. The file contains latency data for 7655 instruction variants. This data is in the

Table 3.2: Comparison of measurement results with IACA

Microarchitecture	IACA	# Instr.	µops	µops_RL	Ports
Nehalem	2.1–2.2	1702	85.31%	92.49%	98.42%
Westmere	2.1–2.2	1714	85.30%	92.42%	98.02%
Sandy Bridge	2.1–2.3	2439	87.41%	93.46%	98.26%
Ivy Bridge	2.1–2.3	2447	85.82%	91.74%	96.62%
Haswell	2.1–3.0	3297	88.96%	93.18%	95.29%
Broadwell	2.2–3.0	3196	89.61%	92.39%	91.67%
Skylake	2.3–3.0	3188	89.05%	92.15%	90.21%
Skylake-X	2.3–3.0	12156	96.41%	97.26%	95.76%

form of a single value per instruction variant, whereas our approach obtains a separate value for each pair of input/output operands. In the following, we compare the value from Intel's file with the maximum of the values that we obtained with our tool.

As described in Section 3.5.2, in some cases, our approach cannot determine exact values for the latency, but only upper bounds. For 3476 of the instruction variants for which Intel's file provides latency data, our approach was able to determine exact values; for 3472 of these variants (i.e., 99.88%), our numbers agree with Intel's data. For the remaining 4179 instruction variants, the maxima of the values obtained by our tool are upper bounds; in all of these cases, our numbers are exactly one cycle larger than the values in Intel's file.

3.7.3 Hardware Measurements vs. IACA

IACA supports the microarchitectures shown in Table 3.2; the second column in this table shows which versions of IACA support the corresponding microarchitecture. The third column shows the number of instruction variants for which we obtained results both from hardware measurements and by executing our microbenchmarks using IACA.

For between 85.30% (Westmere) and 96.41% (Skylake-X) of these instruction variants, IACA reports the same µop count as our measurements on the hardware (see the "µops" column); we consider IACA to report the same count if at least one IACA version reports this count. If we ignore instruction variants with a *REP* prefix (which can have a variable number of µops), and instructions with a *LOCK* prefix (for which IACA in most cases reports a µop count that is different from our measurements), then the µop counts are the same for the percentages in the "µops_RL" column of Table 3.2.

If we consider only the instruction variants for which IACA and our tool report the same µop count, then in between 90.21% (Skylake) and 98.42% (Nehalem) of the cases, the port usages as obtained from measurements on the hardware and as obtained from running our microbenchmarks on top of IACA, are also the same. The percentages for each microarchitecture are shown in the last column of Table 3.2.

Differences Between Hardware Measurements and IACA While some of the discrepancies might be due to measurement errors on the hardware, in many cases we were able to conclude that the output of IACA was incorrect.

There are, for instance, several instructions that read from memory, but that do not have a µop that can use a port with a load unit (e.g., the *IMUL*[22] instruction on Nehalem). On the other hand, there are instructions (like the *TEST mem, R* instruction on Nehalem) that have a *store data* and a *store address* µop in IACA, even though they do not write to the memory. We also found several cases where IACA is not aware that different variants of an instruction have a different port usage. On the actual hardware, the 32-bit variant of the *BSWAP*[23] instruction on Skylake, for example, has just one µop, whereas the 64-bit variant has two µops. In IACA, both variants have two µops. In a number of cases, the sum of the µops on each of the ports does not add up to the total number of µops reported by IACA. An example for this is the *VHADDPD*[24] instruction on Skylake. According to our measurements on the hardware, the port usage of this instruction is $1 * p01 + 2 * p5$. IACA also reports that the instruction has three µops in total. However, the detailed (per port) view only shows one µop.

Differences Between Different IACA Versions We found a number of cases where different IACA versions reported different port usages for the same instructions on the same microarchitecture.

Often, the results from the newer versions correspond to our measurements on the hardware, so in these cases, the differences seem to be due to fixes of (undocumented) bugs in earlier versions of IACA. One example for this is the *VMINPS*[25] instruction on the Skylake microarchitecture. In IACA 2.3, this instruction can use the ports 0, 1, and 5, whereas in IACA 3.0 and on the actual hardware, the instruction can only use ports 0 and 1. On the other

[22] "Signed multiply"
[23] "Byte swap"
[24] "Packed double-FP horizontal add"
[25] "Minimum of packed single-precision floating-point values"

hand, we also found a few cases where the results of an older version of IACA correspond to the measurements on the hardware. An example for this is the *SAHF*[26] instruction on the Haswell microarchitecture. On the actual hardware and in IACA 2.1, this instruction can use the ports 0 and 6. In IACA 2.2, 2.3, and 3.0, however, the instruction can additionally use the ports 1 and 5.

Latency/Throughput In many cases, it was not possible to obtain accurate latency and throughput data from IACA. One reason for this is that IACA ignores several dependencies between instructions. IACA 3.0, for instance, ignores dependencies on status flags. The *CMC* instruction, for example, which complements the carry flag, is reported to have a throughput of 0.25 cycles by IACA, which is impossible in practice due to the dependency on the carry flag; on the actual hardware, we measured a throughput of 1 cycle. IACA also completely ignores memory dependencies. For example, the sequence

```
mov [RAX], RBX; mov RBX, [RAX]
```

is reported to have a throughput of 1 cycle; the measured throughput of this sequence on the microarchitectures supported by IACA is at least 3 cycles. Furthermore, based on our observations, IACA does not seem to model latency differences between different pairs of input and output operands.

3.7.4 Interesting Results

AES Instructions

We first look at an example where our new approach for determining the latencies of an instruction revealed undocumented performance differences between successive microarchitectures. According to the manual [Int12], the

`AESDEC` XMM_1, XMM_2[27]

instruction has a latency of 8 cycles on the Sandy Bridge architecture. Agner Fog [Fog19] and AIDA64 [Ins] report the same latency. According to IACA 2.1 and the LLVM model, the latency is 7 cycles.

The instruction reads and writes the first operand, and reads the second operand. Based on our measurements on the Sandy Bridge system, the latency $lat(XMM_1, XMM_1)$ is 8 cycles, while $lat(XMM_2, XMM_1)$ is only 1 cycle. The instruction uses 2 µops.

[26] "Store AH into flags"

[27] "Perform one round of an AES decryption flow"

According to Intel's instruction set reference [Int19c], the instruction performs the following operations:

```
1 STATE ← XMM1
2 RoundKey ← XMM2
3 STATE ← InvShiftRows(STATE)
4 STATE ← InvSubBytes(STATE)
5 STATE ← InvMixColumns(STATE)
6 DEST[127:0] ← STATE XOR RoundKey
```

We can see that the second operand is only needed in the last step (line 6). So, our latency measurements suggest that one of the two µops probably computes the XOR operation in the last step (which has a latency of 1 cycle).

We obtained the same result on the Ivy Bridge system (i.e., Sandy Bridge's successor). On the Haswell system (i.e., Ivy Bridge's successor), on the other hand, the instruction has just one µop, and the measured latency values for both cases are 7 cycles. The same latency is reported in Intel's manual, by IACA, by the LLVM model, and by Agner Fog.

On the Westmere microarchitecture (i.e., Sandy Bridge's predecessor), which was the first microarchitecture to support the AES instruction set, the instruction has 3 µops, and we measured a latency of 6 cycles for both operand pairs. The same latency is reported in the 2012 version of Intel's manual [Int12] (the current version contains no data for Westmere), by IACA 2.1, and by AIDA64. Agner Fog did not analyze a Westmere system; there is also no LLVM model for Westmere.

We observed the same behavior for the *AESDECLAST*, *AESENC*, and *AESENCLAST* instructions. To the best of our knowledge, the behavior on Sandy Bridge and Ivy Bridge has not been documented before.

There are also variants of these instructions where the second operand is a memory operand instead of a register operand.

For these variants, our tool reports for the Sandy Bridge system a latency of 8 cycles for the register-to-register dependency (as before), and an upper bound on the memory-to-register latency of 7 cycles. According to IACA 2.1 and the LLVM model, the latency is 13 cycles (this value was probably obtained by just adding the load latency to the latency of the register-to-register variants of these instructions). Agner Fog and AIDA64 do not report the latency of the memory variants.

SHLD

We will now see an example that shows that our approach can explain differences among previously published data for the same instruction on the same microarchitecture.

According to the manual [Int12], as well as Granlund [Gra17], IACA, and AIDA64, the

$$\texttt{SHLD}\ R_1\texttt{,}\ R_2\texttt{,}\ imm^{28}$$

instruction has a latency of 4 cycles on the Nehalem microarchitecture. Agner Fog reports a latency of 3 cycles.

The instruction reads and writes the first operand, and reads the second operand. According to our measurements, $lat(R_1, R_1)$ is 3 cycles, whereas $lat(R_2, R_1)$ is 4 cycles. Thus, $lat(R_1, R_1)$ corresponds to Fog's result, while $lat(R_2, R_1)$ corresponds to the latency the other approaches report.

On the Skylake microarchitecture, the same instruction is reported to have a latency of 3 cycles by the manual [Int19b], by the LLVM model, and by Agner Fog. According to Granlund and AIDA64, the latency is 1 cycle.

According to our results for the Skylake system, the latency is 3 cycles if different registers are used for the two operands, but only 1 cycle if the same register is used for both operands (the Nehalem system does not exhibit this behavior). This suggests that Granlund and AIDA64 test the latency by using the same register for both operands, while Fog uses different registers for both operands and, thus, considers only the implicit dependency on the first operand.

MOVQ2DQ

Next, we will show an example where the port usage is modeled inaccurately by existing work.

According to Agner Fog's instruction tables, the *MOVQ2DQ*[29] instruction has a port usage of $1 * p0 + 1 * p15$. This is probably based on the observation that if one executes the instruction repeatedly on its own, then, on average, there is 1 µop on port 0, and there are about 0.5 µops on port 1 and 0.5 µops on port 5.

However, our approach shows that the second µop can actually use port 0, port 1, and port 5. If we execute the instruction together with a blocking

[28] "Double precision shift left"

[29] "Move quadword from MMX technology to XMM register"

instruction for port 1 and port 5, then all µops of the *MOVQ2DQ* instruction will use port 0. According to IACA and to the LLVM model, both µops of this instruction can only use port 5.

MOVDQ2Q

The following example shows a case where existing work reports an inaccurate port usage on one microarchitecture and an imprecise usage on another microarchitecture for the same instruction.

On Haswell, the *MOVDQ2Q*[30] instruction has, according to our results, a port usage of $1 * p015 + 1 * p5$. IACA 2.1 reports the same result. However, according to IACA 2.2, 2.3, 3.0, and the LLVM model, the port usage of this instruction is $1 * p01 + 1 * p015$. According to Agner Fog, the port usage is $1 * p01 + 1 * p5$.

On Sandy Bridge, our measurements agree with both IACA and the LLVM model for the same instruction ($1 * p015 + 1 * p5$). Agner Fog reports the usage as $2 * p015$.

Instructions with Multiple Latencies

Apart from the examples already described, we also found latency differences for different pairs of input and output operands for a number of other instructions. This includes most instructions that have a memory operand and another input operand, where such differences can be expected. We also found differences for the non-memory variants of the following instructions: *ADC*, *CMOV(N)BE*, *(I)MUL*, *PSHUFB*, *ROL*, *ROR*, *SAR*, *SBB*, *SHL*, *SHR*, *(V)MPSADBW*, *VPBLENDV(B/PD/PS)*, *(V)PSLL(D/Q/W)*, *(V)PSRA(D/W)*, *(V)PSRL(D/Q/W)*, *XADD*, and *XCHG*. For the *(I)MUL*, *ROL*, and *ROR* instructions, this behavior is described in [Int19b]; for the *ADC* and *SBB* instructions, the behavior has been observed by [Gra17]. For the remaining instructions, the differences have, to the best of our knowledge, so far been undocumented.

Zero Idioms

According to our results, the *(V)PCMPGT(B/D/Q/W)*[31] instructions are also dependency-breaking idioms, even though they are not in the list of dependency-breaking idioms in Intel's Optimization Manual [Int19b].

[30] "Move Quadword from XMM to MMX Technology Register"
[31] "Compare packed data for greater than"

Shift Instructions on AMD CPUs

Consider the following code sequence, which is from one of microbenchmarks used to measure the latency of the *SHL*[32] instruction:

```
1 SHL RBX, CL
2 CMOVC RBX, RAX
```

The first line shifts the RBX register to the left by the number of bits specified in the CL register. If the CL register is not 0, the instruction also modifies the status flags; otherwise, it leaves them unchanged. The second line contains a conditional move instruction that performs a move operation from the RAX to the RBX register if the carry flag is set.

On both of our AMD CPUs, this code sequence has a (measured) throughput of 2 cycles if the value in the CL register is not 0. However, if the CL register is 0, the code needs more than 26 cycles. So there seems to be a significant penalty for accessing a status flag that was left unchanged by the *SHL* instruction. We observed the same behavior also with other shift instructions and with other instructions that read status flags. On the Intel CPUs we tested, the execution time of these instructions does not depend on the value of the CL register.

To better understand this phenomenon, we performed several additional experiments.

First, we considered a microbenchmark with only the first line. The throughput of this benchmark is 1 cycle, independent of the value in CL.

Then, we added an instruction that writes (but does not read) the status flags, and that has a dependency on the register output of the shift instruction:

```
1 SHL RBX, CL
2 TEST RBX, RBX
3 CMOVC RBX, RAX
```

This code sequence takes 3 cycles, independently of the value in CL. So the delay only occurs when there is an instruction that reads the status flags, and the last instruction that wrote the flags was executed before the shift instruction.

[32] "Shift left"

Next, we added an instruction that breaks the dependencies on the flags and the RBX register, which makes subsequent copies of the code independent of each other:

```
1 XOR RBX, RBX
2 SHL RBX, CL
3 CMOVC RBX, RAX
```

The throughput is now about 0.75 cycles if CL is not 0, and more than 27 cycles otherwise. This suggests that reading a flag that was left unchanged by the shift instruction does not just lead to an increase in the latency, but it rather seems to stall the entire pipeline for several cycles.

If we add additional *CMOVC* instructions at the end, there are no additional stalls. So the stall only occurs upon the first read of a flag that was left unchanged by the shift instruction.

To analyze at which point in time the delay occurs, we performed the following experiment. We inserted a sequence of 1000

```
MOVSX RBX, EBX
```

instructions between the shift and the conditional move instruction. The length of the sequence is far larger than any of the buffers in the back end. This ensures that by the time the shift instruction leaves the pipeline, the processor does not know yet whether there will be an instruction that reads the status flags at some point in the future.

The throughput of this sequence is 1002 cycles if CL is not 0, and over 1027 cycles otherwise. If we omit the *CMOVC* instruction, the execution time is 1001 cycles, independent of the value in CL. This shows that the stall occurs when reading the status flags.

Next, we would like to test if the condition that triggers the stalls survives serializing instructions. If we insert the *CPUID* or the *MFENCE* instruction (which is a serializing instruction on AMD CPUs [AMD19], but not on Intel CPUs [Int19c]) between the shift and the conditional move instruction, the execution time is independent of the value in CL. On the other hand, if we insert a move instruction to a debug register instead (which is also a serializing instruction), the execution time is more than 34 cycles larger if CL is 0.

Finally, we would like to test if branches have an influence on the condition that triggers the stalls. To do this, we use the following microbenchmark.

```
1 SHL RBX, CL
2 MOV RDX, RCX
3 MOV RCX, <loopCount>
4 l:  LOOP l
5 MOV RCX, RDX
6 CMOVC RBX, RAX
```

The *LOOP* instruction in line 4 decrements the `RCX` register, and jumps to the label `l` if `RCX` is not 0; it does not modify status flags.

If the loop count is at least 10, but smaller than 27 on ZEN+ and smaller than 92 on ZEN2, the execution time is about 4 cycles larger than the loop count if `CL` is not 0, and about 30 cycles larger than the loop count otherwise. If the loop count is larger, the execution times are independent of the initial value in `CL`; however, they are at least about 25 cycles larger than the loop count. A further analysis using performance counters showed that if the loop counts are smaller than 27 (on ZEN+) and 92 (on ZEN2), there are no mispredicted branches. If the loop count is larger, there is one mispredicted branch (presumably the last branch of the loop). So, the handling of a mispredicted branch seems to also clear the condition that leads to the delay we observed. Moreover, since the penalty for a mispredicted branch is similar to this delay, we think that it is possible that the processor might do something similar in both cases.

Covert Channels On processors that support hyper-threading, the described effect can be used to create a covert channel, i.e., a channel that is not intended for information transfer and that is used by a trojan process to deliberately leak sensitive information [Lam73, GYCH18].

The basic idea is as follows. In one thread, the sender would store the secret in the `CL` register, execute a shift instruction, followed by an instruction that reads the status flags, and multiple instances of an instruction I that can only use a specific port B. In the other thread, the receiver would measure the time for executing multiple instances of I. If the time is smaller than a certain threshold, this would mean that there was less competition for port B, which happens if the sender is stalled after executing a shift with `CL` set to 0. On the other hand, if the measured time is larger than the threshold, this means that there was competition for port B, which implies that `CL` is 1.

Initial experiments show that creating a covert channel on this basis is indeed possible. However, a detailed analysis of this channel, for example with respect

to the bandwidth that can be achieved, is left as future work. It is also left as future work to investigate whether existing software uses instructions that read flags that were written by a shift instruction while the `CL` register contained some secret. If this is the case, this might make a side channel attack possible. In contrast to a covert channel, a side channel refers "to the accidental leakage of sensitive data" [GYCH18].

Covert channels based on contention of functional units on processors with hyper-threading were also described by [WL06, CV14, ABu+19, BSN+19]. In contrast to the covert channel proposed in this paragraph, their work requires branches that execute different instructions based on the secret.

3.8 Limitations

Our tool currently has the following limitations:

- We only support instructions that can be used in 64-bit mode.
- We do not support the mostly obsolete x87 floating-point instruction set.
- A number of system instructions are not supported. This includes, e.g., instructions that write to segment or control registers, instructions that trigger interrupts, the VT-x instructions for virtual machines, and instructions that use I/O ports. It also includes instructions like the undefined instruction (*UD*) or the halt instruction (*HLT*), which cannot be measured in a meaningful way.
- Except for the division instructions, we currently do not consider performance differences that might be due to different values in registers or different immediate values. We do, however, consider immediates of different lengths, e.g., 16-bit and 32-bit immediates.
- We do not consider differences due to different memory addressing modes, e.g., with scale and offset. We only test instructions that only use the base register.
- For the AMD CPUs, the port usage data is currently limited to one-µop floating-point instructions, as there are no performance counters for the integer pipes, and the performance counters for the floating-point pipes returned incorrect results in several cases when combining different floating-point instructions.

3.9 Conclusions and Future Work

We have presented novel algorithms and their implementations to characterize the latency, throughput, and port usage of instructions on recent x86 microarchitectures, which we believe will prove useful to predict, explain, and optimize the performance of software running on these microarchitectures, e.g., in performance-analysis tools like CQA [CRON+14], Kerncraft [HHEW15], or llvm-mca [Bia18]. The experimental evaluation demonstrates that the obtained instruction characterizations are both more accurate and more precise than those obtained by prior work.

Our results are available on our website[33] both in the form of of a human-readable, interactive HTML table and as a machine-readable XML file.

In his Bachelor thesis [Mee18], Hendrik Meerkamp implemented SUACA, a performance-prediction tool similar to Intel's IACA, exploiting the results obtained in the present work.

Future Work

Future work includes addressing the limitations described in the previous section.

We would also like to extend our approach to characterize other undocumented performance-relevant aspects of the pipeline, e.g., regarding micro and macro fusion, or whether instructions use the simple decoder, the complex decoder, or the Microcode-ROM.

Our approach currently infers the number of µops for each port combination, but it does not provide information on the relative order in which they are executed. It would be possible to obtain such information with the extension to *nanoBench* proposed in Section 2.6 that provides cycle-by-cycle performance data.

[33] `https://www.uops.info`

4

Characterizing Cache Architectures

In this chapter, we develop several tools that generate microbenchmarks to precisely characterize the cache architectures of recent processors. We focus, in particular, on cache replacement policies, which are typically undocumented. The generated microbenchmarks are evaluated using *nanoBench*.

We apply these tools to Intel CPUs with 13 different microarchitectures, uncovering several previously undocumented replacement policy variants.

Parts of the material presented in this chapter have been published in [AR14] and [AR20].

4.1 Introduction

To bridge the increasing latency gap between the processor and main memory, modern microarchitectures employ memory hierarchies with multiple levels of cache memory. These caches are small but fast memories that make use of temporal and spatial locality. Typically, they have a big impact on the execution time of computer programs; the penalty of a miss in the last-level cache can be more than 200 cycles.

Detailed cache models are, for example, necessary to build cycle-accurate simulators such as Zesto [LSX09], gem5 [BBB+11], McSim+ [ALOJ13], or ZSim [SK13]. Similarly, such models are an essential part of worst-case execution time (WCET) analyzers for real-time systems [W+08, Rei08, LGR+16]. Information on cache properties is also required by self-optimizing software systems like ATLAS [CWPD01], PHiPAC [BACD97], or FFTW [FJ05], as well as platform-aware compilers, such as PACE [C+10]. Furthermore, cache models are needed for devising or mitigating certain covert-channel and side-channel attacks (see Section 4.5.3).

In this chapter, we develop techniques for creating such cache models. We focus, in particular, on cache replacement policies, which are typically undocumented for recent microarchitectures.

To this end, we first introduce the necessary background regarding caches and replacement policies (Section 4.2). This includes a new naming scheme for referring to different variants of the QLRU replacement policy, which is used in many recent CPUs.

In Section 4.3, we propose several tools for determining cache parameters. The first tool, *cacheInfo*, provides details on the *structure* of the caches, such as the sizes, the associativities, the number of cache sets, or the number of C-Boxes and slices.

The second tool, *cacheSeq*, makes it possible to analyze the *behavior* of the caches by measuring the number of cache hits and misses when executing an access sequence in one or more cache sets; the access sequence is supplied as a parameter to the tool and can be specified using a convenient syntax. To perform these measurements, the tool generates suitable microbenchmarks that are evaluated using *nanoBench* (see Chapter 2).

Based on *cacheSeq*, we then develop several tools for identifying the replacement policy. In particular, we implement a tool that can automatically infer permutation policies, and a tool that can automatically determine whether the policy belongs to a set of more than 300 variants of commonly used policies, including policies like MRU and QLRU, which are not permutation policies. These tools are precise enough to determine the policies used in individual cache sets. In addition to that, we develop a tool that can find out whether the cache uses an adaptive replacement policy. Furthermore, we propose a tool that creates *age graphs*, which are helpful for analyzing caches with nondeterministic replacement policies.

We have applied our tools to 13 different Intel microarchitectures, and we provide detailed models of their replacement policies.

We have discovered several previously undocumented replacement policies. In particular, we have discovered a new approximation to LRU that is used by the L1 cache of the Ice Lake microarchitecture. In addition to that, we have identified multiple previously undocumented variants of QLRU replacement that are used by the L2 and L3 caches of several recent microarchitectures. Furthermore, we have discovered two previously unknown randomized variants of PLRU replacement that are used by the L2 caches of the Core 2 Duo E6750 and E8400 processors.

4.2 Background

4.2.1 Cache Organization

Caches are small but fast memories that store a subset of the main memory's contents to bridge the latency gap between the CPU and main memory.

To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* of a specific *block size*. Blocks are cached as a whole in cache lines of the same size. Usually, the block size is a power of two (on all recent x86 microarchitectures, the block size is 64 Bytes). This way, the block number is determined by the most significant bits of a memory address, i.e., $block(addr) = \lfloor addr / blockSize \rfloor$; the remaining bits are known as the *offset*.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache ("cache hit") or not ("cache miss"). To enable an efficient look-up, each block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into a number of equally-sized *cache sets*. The size of a cache set is called the *associativity* A of the cache. A cache with associativity A is often called A-*way* set-associative. It consists of A *ways*, each of which consists of one cache line in each cache set. In the context of a cache set, the term *way* thus refers to a single cache line. Usually, also the number of cache sets is a power of two such that the set number, also called index, is determined by the least significant bits of the block number, i.e., $index(addr) = block(addr) \bmod nSets$. The remaining bits of the block number are known as the *tag*. Tags are stored along with the data to decide whether and where a block is cached within a set.

In Intel microarchitectures, starting with Sandy Bridge, the last-level cache is divided into multiple slices. A hash function H is used to map block numbers to slices, i.e. $slice(addr) = H(block(addr))$. Each of the slices has *nSetsPerSlice* many cache sets and is organized as described in the previous paragraph; in particular, the set index is determined by $index(addr) = block(addr) \bmod nSetsPerSlice$. The slices are managed by so-called C-Boxes, which provide the interface between the core and the last-level cache, and which are responsible for maintaining cache coherence. Usually, there is one C-Box per physical core. The first microarchitectures that used sliced last-level caches (Sandy Bridge, Ivy Bridge, and Haswell) had one slice per C-Box [HWH13, IES15, LYG+15, MSN+15, YGL+15, IGI+16, KAGPJ16]. Skylake and more recent microarchitectures can have multiple slices per C-Box [DKPT17, FRMK19]; all C-Boxes of a given processor manage the

same number of slices. Each C-Box has several performance counters that can, for example, count the number of lookup events for the corresponding part of the last-level cache. These counters belong to the class of *uncore performance counters* (see also Section 2.2.1).

The hash function that maps block numbers to slices is undocumented. However, several paper have reverse-engineered this hash function for Sandy Bridge, Ivy Bridge, and Haswell CPUs [HWH13, IES15, MSN+15, YGL+15, IGI+16, KAGPJ16].

Addresses that map to the same cache set and to the same slice are commonly called *congruent*.

4.2.2 Replacement Policies

Since the number of congruent memory blocks is usually far greater than the associativity of the cache, a *replacement policy* must decide which memory block to replace upon a cache miss. Most replacement policies try to exploit temporal locality and base their decisions on the history of memory accesses. Usually, cache sets are treated independently of each other such that accesses to one cache set do not influence replacement decisions in other sets.

Permutation Policies

Many commonly used policies can be modeled as so-called *permutation policies* [Abe12, AR13]. These policies have in common that

1. they maintain a total order of the elements in the cache,
2. upon a cache hit, the order is updated; the new order only depends on the position of the accessed element in the order, and
3. upon a cache miss, the smallest element in the order is replaced.

Permutation policies can thus be fully specified by a permutation vector

$$\Pi = \langle \Pi_0, \ldots, \Pi_{A-1}, \Pi_{\text{miss}} \rangle.$$

The permutation Π_i determines how to update the order upon an access to the i^{th} element of the order. Π_{miss} determines how to update the order upon a cache miss; it is typically fixed to be $(A-1, 0, 1, \ldots, A-2)$ [Abe12, AR13].

Among the replacement policies that can be modeled as permutation policies are, for example, "first-in first-out" (FIFO), "least-recently used" (LRU), and "tree-based pseudo-LRU" (PLRU). PLRU is an approximation to LRU that

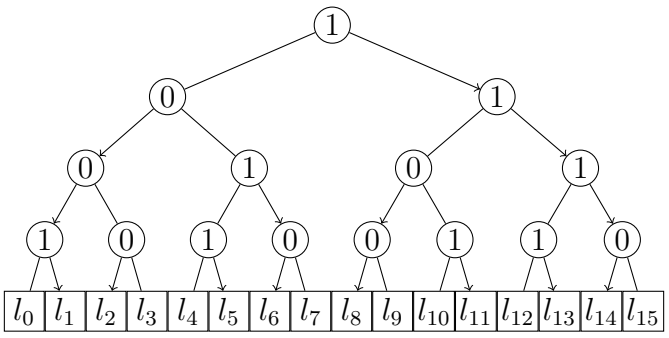

Figure 4.1: Possible PLRU state after an access to l_4

maintains a binary search tree for each cache set. Upon a cache miss, the element that the tree bits currently point to is replaced; however, if the cache is not full, the new element is typically inserted into the leftmost empty line instead. After each access to an element, all the bits on the path from the root of the tree to the leaf that corresponds to the accessed element are set to point away from this path. Figure 4.1 illustrates this policy. The policy requires the associativity to be a power of two. Figure 4.2 shows the permutation vectors for LRU, FIFO, and PLRU at associativity 8.

Permutation policies were introduced by [Abe12, AR13], along with an efficient algorithm for inferring them automatically. This algorithm determines the $A+1$ permutations of a permutation policy one at a time. This is achieved by first establishing a particular cache state, accessing the element corresponding to the permutation we want to determine, and finally reading out the resulting cache state. Reading out a cache state means finding the position of each element in the cache by determining the minimal number of additional misses before the element gets replaced.

MRU/QLRU

However, not all popular policies can be modeled as permutation policies. One example is the MRU policy [RGBW07]. This policy stores one status bit for each cache line. Upon an access to a line, the corresponding bit is set to zero; if it was the last bit that was set to one before, the bits for all other lines are set to one. Upon a cache miss, the leftmost element whose bit is set to one gets replaced. This policy is sometimes also called “bit-PLRU” [PJ15] or “PLRUm” [AZMM04]. A variant of this replacement policy that only checks upon a cache miss whether there is still a line whose status bit is one is called “not-recently-used” (NRU) [JTSE10].

$$\Pi_0^{LRU} = (0,1,2,3,4,5,6,7)$$
$$\Pi_1^{LRU} = (1,0,2,3,4,5,6,7)$$
$$\Pi_2^{LRU} = (2,0,1,3,4,5,6,7)$$
$$\Pi_3^{LRU} = (3,0,1,2,4,5,6,7)$$
$$\Pi_4^{LRU} = (4,0,1,2,3,5,6,7)$$
$$\Pi_5^{LRU} = (5,0,1,2,3,4,6,7)$$
$$\Pi_6^{LRU} = (6,0,1,2,3,4,5,7)$$
$$\Pi_7^{LRU} = (7,0,1,2,3,4,5,6)$$

(a) LRU

$$\Pi_0^{PLRU} = (0,1,2,3,4,5,6,7)$$
$$\Pi_1^{PLRU} = (1,0,3,2,5,4,7,6)$$
$$\Pi_2^{PLRU} = (2,1,0,3,6,5,4,7)$$
$$\Pi_3^{PLRU} = (3,0,1,2,7,4,5,6)$$
$$\Pi_4^{PLRU} = (4,1,2,3,0,5,6,7)$$
$$\Pi_5^{PLRU} = (5,0,3,2,1,4,7,6)$$
$$\Pi_6^{PLRU} = (6,1,0,3,2,5,4,7)$$
$$\Pi_7^{PLRU} = (7,0,1,2,3,4,5,6)$$

(b) PLRU

$$\Pi_0^{FIFO} = (0,1,2,3,4,5,6,7)$$
$$\Pi_1^{FIFO} = (0,1,2,3,4,5,6,7)$$
$$\Pi_2^{FIFO} = (0,1,2,3,4,5,6,7)$$
$$\Pi_3^{FIFO} = (0,1,2,3,4,5,6,7)$$
$$\Pi_4^{FIFO} = (0,1,2,3,4,5,6,7)$$
$$\Pi_5^{FIFO} = (0,1,2,3,4,5,6,7)$$
$$\Pi_6^{FIFO} = (0,1,2,3,4,5,6,7)$$
$$\Pi_7^{FIFO} = (0,1,2,3,4,5,6,7)$$

(c) FIFO

Figure 4.2: Permutation vectors for LRU, PLRU & FIFO at associativity 8 (Source: [Abe12])

Generalizations of this policy that use two status bits per cache line are called “Quad-Age LRU” (QLRU) [JGSW12, BMME19] or “2-bit Re-reference Interval Prediction” (RRIP) [JTSE10]. The two bits are supposed to represent the age of a block.

During our experiments, we found out that some recent Intel CPUs use variants of this policy that were not described in the literature so far. In particular, the variants differ from each other in the *hit promotion policy*, in the *insertion age*, in the location in the cache where a block is inserted upon a miss, in how the bits are updated if there is no more block with age 3, and in whether this update occurs only on a miss or also on a hit.

In the following, we will describe these parameters in detail, and we propose a naming scheme for referring to the different variants.

The *hit promotion policy* describes how the age of a block is updated upon a hit. We assume that the age is always reduced, unless it is already 0. Thus, the hit promotion policy can be modeled by one of the following functions.

Let $x \in \{0, 1, 2\}$, and $y \in \{0, 1\}$.

$$Hxy(a) := \begin{cases} x, & \text{if } a = 3 \\ y, & \text{if } a = 2 \\ 0, & \text{otherwise} \end{cases}$$

The *insertion age* is the age that will be assigned to a block upon a miss. For $x \in \{0, 1, 2, 3\}$, we will use Mx to denote that the insertion age is x. Furthermore, we will use $MR_p x$ to denote a policy that inserts new blocks with age x with probability $\frac{1}{p}$, and with age 3 otherwise. Note that the insertion age might be different if blocks are brought into the cache by prefetching. We currently do not consider this scenario.

We consider the following three variants as to where a block will be inserted upon a miss.

- *R0:* If the cache is not yet full (after executing the *WBINVD* instruction), insert the new block in the leftmost empty location. Otherwise, replace the block in the leftmost location whose status bits are 3. If there is no such block, the behavior is undefined.
- *R1:* Like *R0*, but if there is no location whose status bits are 3, always replace the leftmost block, independently of its status bits.
- *R2:* Like *R0*, but insert blocks in the rightmost empty location if the cache is not yet full.

If after an access, there is no more block whose age is 3, the status bits of potentially all blocks will be updated. Let i denote the location of the block that was accessed. Let $age(b)$ be the current age of block b, and let $age'(b)$ be the new age (after the update). Let M be the maximum (current) age of any block. We consider the following variants for age':

- *U0:* $age'(b) := age(b) + (3 - M)$
- *U1:* $age'(b) := \begin{cases} age(b), & \text{if } b = i \\ age(b) + (3 - M), & \text{otherwise} \end{cases}$
- *U2:* $age'(b) := age(b) + 1$
- *U3:* $age'(b) := \begin{cases} age(b), & \text{if } b = i \\ age(b) + 1, & \text{otherwise} \end{cases}$

We will use a name of the form QLRU_H11_M1_R1_U2 to refer to the corresponding variant.

Some variants do not check after each access whether there is still a block with age 3, as described above, but only upon a miss, before selecting the block to replace. We will refer to such variants by adding the suffix UMO ("update on miss only") to the name.

Note that not all combinations are possible. For example, *R0* cannot be combined with *U2* or *U3*, as it always requires at least one block with age 3. Also, some combinations are observationally equivalent; this is, e.g., the case for *R0* and *R1* in combination with *U0*.

The 2-bit SRRIP-HP policy proposed by Jaleel et al. [JTSE10] would be named QLRU_H00_M2_R0_U0_UMO according to our naming scheme. The corresponding "bimodal RRIP" (BRRIP) policy from the same paper would be named QLRU_H00_MR_p2_R0_U0_UMO.

Adaptive Policies

Some caches in recent CPUs use adaptive replacement policies that can dynamically switch between two different policies. This can, for example, be implemented via *set dueling* [QJP+07, JTSE10, Won13]: A number of sets are dedicated to each policy, and the remaining sets are *follower sets* that use the policy that is currently performing better.

4.3 Cache-Characterization Tools

Based on *nanoBench* (see Chapter 2), we have developed a set of tools for analyzing undocumented properties of caches.

4.3.1 CacheInfo

This tool reports details on the structure of the caches of the system that it is run on. This includes the sizes, the associativities, the number of sets, the block sizes, and, for L3 caches, the number of C-Boxes and the number of slices.

All of these properties, except the last two, are obtained with the *CPUID* instruction. Alternatively, it would also be possible to measure these properties using microbenchmarks [SS95, MS96, LT98, TY00, BC00, CD01, DMM+04, Man04, JB07, YPS05, BT09, GDTF+10, CS11, AR12, Abe12, DLM+13, CX18]; however, this approach is not necessary for the processors that we target.

The number of C-Boxes is obtained by reading the model-specific register (MSR) 0x396.

For obtaining the number of slices per C-Box, we use the microbenchmark-based approach shown in Algorithm 4.1. The algorithm first searches for a maximal set of addresses with the same set index that map to the same C-Box and that do not cause cache misses when accessing them repeatedly. The number of slices per C-Box then corresponds to the size of this set divided by the associativity of the L3 cache.

Algorithm 4.2 shows the approach we use for finding such a maximal set. The algorithm maintains a set of addresses (for a given index and C-Box) that do not cause cache misses when accessing them repeatedly. In each iteration of the loop, it tries to extend this set with an additional address. The algorithm continues until it did not find any additional addresses for *maxNotAdded* many steps. In practice, we set *maxNotAdded* to the associativity of the L3 cache.

Addresses that are separated by a multiple of $nSetsPerSlice \cdot blockSize$ have the same set index (see Section 4.2.1). Let $nAllL3Sets = (nSetsPerSlice \cdot nSlices)$ be the sum of the number of cache sets of all slices, as reported by the *CPUID* instruction. We have that

$$\begin{aligned}\frac{nAllL3Sets}{nCBoxes} &= \frac{nSetsPerSlice \cdot nSlices}{nCBoxes} \\ &= \frac{nSetsPerSlice \cdot nCBoxes \cdot nSlicesPerCBox}{nCBoxes} \\ &= nSetsPerSlice \cdot nSlicesPerCBox.\end{aligned}$$

Thus, $\frac{nAllL3Sets}{nCBoxes} \cdot blockSize$ is a multiple of $nSetsPerSlice \cdot blockSize$, and therefore, addresses that are separated by this stride have the same set index. Note that we could also, for example, use a stride of $nAllL3Sets \cdot blockSize$ (or any other multiple of $nSetsPerSlice \cdot blockSize$); however, since we do not know *nSetsPerSlice* at this point, $\frac{nAllL3Sets}{nCBoxes} \cdot blockSize$ is the smallest possible stride that we can be sure to be a multiple of $nSetsPerSlice \cdot blockSize$.

The function call *getCBoxOfAddress(a)* first generates a microbenchmark that flushes address *a* from all caches (using the *CLFLUSH* instruction). To determine the C-Box of address *a*, this microbenchmark is then evaluated with the kernel-space version of *nanoBench*, measuring the number of accesses to each C-Box using the corresponding uncore performance counters (see Section 2.2.1). We use the option described in Section 2.4.4 to allocate a large enough physically-contiguous memory area; *baseAddr* points to the beginning of this area. Using the *CLFLUSH* instruction instead of loads or stores has the advantage that the accesses are guaranteed to reach the L3 cache.

Algorithm 4.1: Measuring the number of slices per C-Box

```
1 Function getNumberofSlices()
2     return ⌈|findMaxNonEvictingL3Addresses(0, 0)|/L3Assoc⌉
```

Algorithm 4.2: Finding a maximal set of addresses (for a specific C-Box and cache set) that do not cause evictions in the L3 cache

```
1  Function findMaxNonEvictingL3Addresses(cBox, cacheSet)
2      stride ← (nAllL3Sets/nCBoxes) ∗ blockSize
3      addresses ← ∅
4      curAddr ← baseAddr + cacheSet ∗ blockSize
5      notAdded ← 0
6      while notAdded < maxNotAdded do
7          curAddr ← curAddr + stride
8          if getCBoxOfAddress(curAddr) = cBox then
9              newAddresses ← addresses ∪ {curAddr}
10             if hasL3Conflicts(newAddresses) then
11                 notAdded ← notAdded + 1
12             else
13                 addresses ← newAddresses
14                 notAdded ← 0
15     return addresses
```

The function call *hasL3Conflicts(addresses)* creates and runs a microbenchmark that checks whether accessing the supplied addresses multiple times leads to cache misses. To make sure that the accesses reach the L3 cache, the generated microbenchmark additionally contains accesses to addresses that map to the same cache set (and hence, also to the same sets in higher-level caches), but to different C-Boxes; these additional accesses are excluded from the measurement results.

4.3.2 CacheSeq

This tool can be used to measure how many cache hits and misses executing an access sequence (i.e., a sequence of congruent addresses) generates. To this end, *cacheSeq* automatically generates a suitable microbenchmark that is then evaluated using the kernel-space version of *nanoBench*.

Access sequences can be specified using strings of the following form:

"A ⟨wbinvd⟩ B_0 B_1 B_2 B_3 B_0? B_1! X A?"

Elements of the sequence that end with a "?" will be included in the performance counter measurements. The other elements will be accessed, but the number of hits and misses that they generate will not be recorded; this is implemented using the feature described in Section 2.3.9, that makes it possible to temporarily pause performance counting. Elements that end with a "!" will be flushed (using the *CLFLUSH* instruction) instead of being accessed. "⟨wbinvd⟩" means that the *WBINVD*[1] instruction will be executed at the corresponding location in the access sequence. This instruction, which is a privileged instruction, flushes all caches.

The following parameters can be specified via command-line options:

- The cache level in which the sequence should be accessed.
- The cache sets in which the sequence should be accessed. This can be a list or a range of sets.
- The C-Box in which the sequence should be accessed. If there are multiple slices per C-Box, all accesses will be to the same slice.
- The access sequence can be executed a configurable number of times in a loop. Additionally, it is possible to specify an *initialization sequence* that is executed once in the beginning.
- To make sure that the memory accesses reach the selected cache level, the tool can automatically add additional accesses to higher-level caches that evict the elements of the access sequence from these caches. These additional accesses are excluded from the performance counter measurements. The tool provides two options as to how the additional addresses are selected. If the number of sets of the selected cache level is larger than the number of sets of the higher-level caches, the tool can choose addresses that map to the same set in the higher-level caches, but to different sets in the selected level. For sliced caches, the tool additionally provides the option to choose addresses that map to the same set, but to a different slice.
- The number of times the generated microbenchmark is executed and the aggregate function that is applied to the measurement results. These two parameters are passed to *nanoBench* (see Section 2.3.3).

[1] "Write back and invalidate cache"

The syntax for specifying access sequences was inspired by an early version of Vila et al.'s *MemBlockLang* (MBL) language [VGGK20].

We now describe how *CacheSeq* finds suitable addresses for the elements of an access sequence, i.e., addresses that are congruent for the selected cache level.

To this end, the tool first runs *CacheInfo* (Section 4.3.1) to obtain the basic parameters of the caches. As in the implementation of *CacheInfo*, we use *nanoBench* with the option of allocating physically-contiguous memory. Thus, for non-sliced caches, finding congruent addresses is straightforward.

For sliced caches, we use the approach shown in Algorithm 4.3 to find a set of n congruent addresses for a specific C-Box and cache set. Our approach does not require knowledge of the undocumented hash function mapping addresses to slices, which, thus far, has only been reverse-engineered for some of the CPUs that we target. The algorithm first finds a minimal eviction set, i.e., a set of associativity many congruent addresses. Then, it searches for further addresses that have conflicts with the addresses in the eviction set, using the *hasL3Conflicts()* function already described in Section 4.3.1.

For finding a minimal eviction set, we use the approach shown in Algorithm 4.4. In the first step, the algorithm searches greedily for a set of addresses (for the given index and C-Box) that have conflicts in the L3 cache. In the second step, the algorithm removes all addresses from this set that are not actually necessary for the conflict to occur. Vila et al. [VKM19] describe more sophisticated techniques for finding eviction sets that also work if the memory area is not physically-contiguous and if L3 performance counters are not accessible. However, in our setting these techniques would likely not provide a significant benefit over the proposed approach.

The tools described in the following sections are all based on *cacheSeq*.

4.3.3 Replacement Policies

We implemented two tools for automatically inferring deterministic replacement policies.

The first tool implements the algorithm proposed in [Abe12, AR13] for inferring permutation policies. In contrast to the previous implementation described in [Abe12, AR13], our current implementation is able to determine the policy in individual cache sets. The previous implementation assumed that all cache sets use the same policy.

Algorithm 4.3: Finding a set of n congruent L3 addresses

```
1  Function findCongruentL3Addresses(n, cBox, cacheSet)
2      stride ← (nAllL3Sets/nCBoxes) * blockSize
3      minEvictionSet ← findMinimalL3EvictionSet(cBox, cacheSet)
4      addresses ← ∅
5      curAddr ← max(minEvictionSet)
6      while |addresses| + |minEvictionSet| < n do
7          curAddr ← curAddr + stride
8          if hasL3Conflicts(minEvictionSet ∪ {curAddr}) then
9              addresses ← addresses ∪ {curAddr}
10     return minEvictionSet ∪ addresses
```

Algorithm 4.4: Finding a minimal L3 eviction set

```
1  Function findMinimalL3EvictionSet(cBox, cacheSet)
2      stride ← (nAllL3Sets/nCBoxes) * blockSize
3      minEvictionSet ← ∅
4      curAddr ← baseAddr + cacheSet * blockSize
5      while ¬hasL3Conflicts(minEvictionSet) do
6          curAddr ← curAddr + stride
7          if getCBoxOfAddress(curAddr) = cBox then
8              minEvictionSet ← minEvictionSet ∪ {curAddr}
9      for a ∈ minEvictionSet do
10         if hasL3Conflicts(minEvictionSet \ {a}) then
11             minEvictionSet ← minEvictionSet \ {a}
12     minEvictionSet ← minEvictionSet \ max(minEvictionSet)
13     return minEvictionSet
```

The second tool generates random access sequences and compares the number of hits obtained by executing them with *cacheSeq* with the number of hits in a simulation of different replacement policies, including common policies like LRU, PLRU, and FIFO, as well as all meaningful QLRU variants, as introduced in Section 4.2.2.

By default, the tool generates random sequences of length 50 (which is significantly larger than the associativities of the caches that we consider). The sequences are generated as follows. For position i of the sequence, the tool chooses with probability 50% a fresh element and with probability 50%

an element that already occurs in the sequence. We found that typically a relatively small number of such sequences suffices for identifying a policy that is likely the correct one. In our experiments, evaluating 250 sequences always produced multiple counterexamples for all but at most one policy.

Both of our tools for determining replacement policies clear the caches (using the *WBINVD* instruction) at the start of each access sequence. However, for some microarchitectures we tested, the behavior of the replacement policies appeared to be nondeterministic after executing the *WBINVD* instruction; this was also observed by Vila et al. [VGGK20]. We therefore added an option to our tools that allows to specify a *reset sequence*, i.e., an access sequence that is executed after the *WBINVD* instruction and that establishes a fixed replacement policy state; the blocks occurring in the reset sequence are not used again afterwards. We discuss suitable reset sequences and further insights in Section 4.4.4.

4.3.4 Age Graphs

This tool takes as input an access sequence *Seq*, and list or range of cache sets. For each block B that occurs in *Seq*, the tool generates a graph that shows the number of cache hits for executing the access sequence

$$Seq + \text{“}F_0 \ \ldots \ F_n \ B?\text{”}$$

in the selected cache sets for increasing values of n. The blocks F_i are fresh blocks that do not occur in *Seq*. Furthermore, we require that there is no “?” in *Seq*.

These graphs can be considered to illustrate the “age” of a block, as they show how many fresh blocks need to be accessed to evict the block from the cache. The graphs are, in particular, useful for analyzing caches with replacement policies that are nondeterministic and, thus, cannot be inferred with the tools described in the previous section. Examples of such graphs are discussed in Section 4.4.3.

4.3.5 Test for Adaptive Policies

For caches with adaptive policies, we implemented a tool that can identify the cache sets that use a fixed policy and the sets that use a varying policy.

Algorithm 4.5 illustrates how our approach works. The algorithm iterates repeatedly, in random order, over the cache sets of all slices. For each set, it uses *cacheSeq* (see Section 4.3.2) to measure the number of L3 hits on

Algorithm 4.5: Test for adaptive policies

```
1  allSets ← [0, 1, ..., nL3SetsPerSlice − 1]
2  for sl ∈ slices do
3      sets[sl] ← copy(allSets)
4      for s ∈ allSets do
5          L3Hits[sl][s] ← ∅
6
7  testSeq ← "B1? B2? ... B_{L3Assoc+1}?"
8  hitSeq ← "B1? B2? ... B_{L3Assoc}?"
9  missSeq ← "B1? B2? ... B_{3·L3Assoc}?"
10
11 notChanged ← 0
12 while notChanged < maxNotChanged do
13     notChanged ← notChanged + 1
14     for hmSeq ∈ {hitSeq, missSeq} do
15         for sl ∈ slices do
16             for s ∈ random.shuffle(sets[sl]) do
17                 m ← cacheSeq(testSeq, sl, s, loopCount)
18                 L3Hits[sl][s] ← L3Hits[sl][s] ∪ m
19                 if max(L3Hits[sl][s]) − min(L3Hits[sl][s]) > 1 then
20                     sets[sl] ← sets[sl] \ {s}
21                     notChanged ← 0
22                 else
23                     cacheSeq(hmSeq, sl, s, 100)
24 return (sets, L3Hits)
```

a sequence consisting of (*associativity* + 1) many different blocks that is repeated a configurable number of times, using the *loop* option of *cacheSeq* (by default, we use a loop count of 10). This sequence has a so-called *thrashing access pattern*, i.e., a cyclic access pattern that is longer than the associativity of the cache [JTSE10]. In caches with adaptive policies, usually one of the policies is more thrashing-resistant than the other, and thus, the number of hits on this sequence can be used to distinguish the policies.

Whenever the algorithm finds a cache set for which the maximum of the number of L3 hits measured so far differs from the minimum, it assumes that this is a set with a varying replacement policy and excludes it from further iterations. Otherwise, it assumes that the set is potentially a set with a fixed

policy. The algorithm then runs *cacheSeq* again on this set using either (in even iterations) a sequence that generates mostly hits, or (in odd iterations) a sequence that generates mostly misses. For cache sets with a fixed policy, we expect such a sequence to be able to influence the policy used in the remaining cache sets.

All of these steps are repeated until the algorithm did not find any additional cache sets with a varying policy for *maxNotChanged* many iterations (we use 10 as the default value for *maxNotChanged*). At the end, the variable *sets* contains only the cache sets with a fixed policy.

Furthermore, our tool generates a graph that shows the maximum and minimum number of L3 hits for all sets in all slices; we discuss examples of such graphs in Section 4.4.3.

4.4 Results

We have applied our tools for determining the cache parameters to 13 out of the 16 CPUs that we used for the evaluation in Chapter 3 (see Table 3.1). On these CPUs, we disabled all but one core, and we also disabled cache prefetching. We did not consider the two AMD CPUs, as we could not find a way to disable their cache prefetchers. We also did not consider the server variant of Skylake (Skylake-X), for which the uncore performance counter configuration is significantly different from the other CPUs.

Our results for the L1, L2, and L3 caches are summarized in Tables 4.1 – 4.3. We describe the most interesting results in Sections 4.4.1, 4.4.2, and 4.4.3, respectively. In Section 4.4.4, we discuss how to reset the replacement policy state. Finally, in Section 4.4.5, we analyze the implementation costs of the discovered policies in terms of the number of status bits they require.

4.4.1 L1 Data Caches

The L1 data caches of all processor we considered, except for the Ice Lake CPU, are 8-way set-associative and use the PLRU replacement policy.

The L1 data cache of the Ice Lake CPU is 12-way set-associative. It uses a permutation policy; the corresponding permutation vectors are shown in Figure 4.3. A possible intuitive explanation of this policy is illustrated in Figure 4.4. It uses three PLRU trees with 4 elements each; the trees are ordered by the recency of the last access to one of their elements. Upon a cache miss, the element that the bits of the least-recently accessed tree point

Table 4.1: L1 data cache results

CPU (Microarchitecture)	Size	Assoc.	Policy
Core 2 Duo E6750 (Conroe)	32 kB	8	PLRU
Core 2 Duo E8400 (Wolfdale)	32 kB	8	PLRU
Core i5-750 (Nehalem)	32 kB	8	PLRU
Core i5-650 (Westmere)	32 kB	8	PLRU
Core i7-2600 (Sandy Bridge)	32 kB	8	PLRU
Core i5-3470 (Ivy Bridge)	32 kB	8	PLRU
Xeon E3-1225 v3 (Haswell)	32 kB	8	PLRU
Core i5-5200U (Broadwell)	32 kB	8	PLRU
Core i7-6500U (Skylake)	32 kB	8	PLRU
Core i7-7700 (Kaby Lake)	32 kB	8	PLRU
Core i7-8700K (Coffee Lake)	32 kB	8	PLRU
Core i3-8121U (Cannon Lake)	32 kB	8	PLRU
Core i5-1035G1 (Ice Lake)	48 kB	12	LRU_3PLRU_4

Table 4.2: L2 cache results

CPU (Microarchitecture)	Size	Assoc.	Policy
Core 2 Duo E6750 (Conroe)	4 MB	16	$PLRU_8Rand_2$
Core 2 Duo E8400 (Wolfdale)	6 MB	24	$Rand_3PLRU_8$
Core i5-750 (Nehalem)	256 kB	8	PLRU
Core i5-650 (Westmere)	256 kB	8	PLRU
Core i7-2600 (Sandy Bridge)	256 kB	8	PLRU
Core i5-3470 (Ivy Bridge)	256 kB	8	PLRU
Xeon E3-1225 v3 (Haswell)	256 kB	8	PLRU
Core i5-5200U (Broadwell)	256 kB	8	PLRU
Core i7-6500U (Skylake)	256 kB	4	QLRU_H00_M1_R2_U1
Core i7-7700 (Kaby Lake)	256 kB	4	QLRU_H00_M1_R2_U1
Core i7-8700K (Coffee Lake)	256 kB	4	QLRU_H00_M1_R2_U1
Core i3-8121U (Cannon Lake)	256 kB	4	QLRU_H00_M1_R0_U1
Core i5-1035G1 (Ice Lake)	512 kB	8	QLRU_H00_M1_R0_U1

Table 4.3: L3 cache results

CPU (Microarchitecture)	Size	Assoc.	C-Boxes	Slices	Policy
Core i5-750 (Nehalem)	8 MB	16	-	-	MRU
Core i5-650 (Westmere)	4 MB	16	-	-	MRU
Core i7-2600 (Sandy Bridge)	8 MB	16	4	4	MRU*
Core i5-3470 (Ivy Bridge)	6 MB	12	4	4	see Section 4.4.3
Xeon E3-1225 v3 (Haswell)	8 MB	16	4	4	see Section 4.4.3
Core i5-5200U (Broadwell)	3 MB	12	2	2	see Section 4.4.3
Core i7-6500U (Skylake)	4 MB	16	2	4	see Section 4.4.3
Core i7-7700 (Kaby Lake)	8 MB	16	4	8	see Section 4.4.3
Core i7-8700K (Coffee Lake)	8 MB	16	6	12	see Section 4.4.3
Core i3-8121U (Cannon Lake)	4 MB	16	2	4	see Section 4.4.3
Core i5-1035G1 (Ice Lake)	6 MB	12	4	8	see Section 4.4.3

$$
\begin{aligned}
\Pi_0 &= (0,1,2,3,4,5,6,7,8,9,10,11)\\
\Pi_1 &= (1,0,2,4,3,5,7,6,8,10,9,11)\\
\Pi_2 &= (2,0,1,5,3,4,8,6,7,11,9,10)\\
\Pi_3 &= (3,1,2,0,4,5,9,7,8,6,10,11)\\
\Pi_4 &= (4,0,2,1,3,5,10,6,8,7,9,11)\\
\Pi_5 &= (5,0,1,2,3,4,11,6,7,8,9,10)\\
\Pi_6 &= (6,1,2,3,4,5,0,7,8,9,10,11)\\
\Pi_7 &= (7,0,2,4,3,5,1,6,8,10,9,11)\\
\Pi_8 &= (8,0,1,5,3,4,2,6,7,11,9,10)\\
\Pi_9 &= (9,1,2,0,4,5,3,7,8,6,10,11)\\
\Pi_{10} &= (10,0,2,1,3,5,4,6,8,7,9,11)\\
\Pi_{11} &= (11,0,1,2,3,4,5,6,7,8,9,10)
\end{aligned}
$$

Figure 4.3: Permutation vectors for the Ice Lake L1 policy, © 2020 IEEE

to is replaced. In the example in Figure 4.4, the next element to be replaced would be l_1. In the following, we will call this policy $\text{LRU}_3\text{PLRU}_4$. We are unaware of any previous descriptions of this policy. However, it can be seen as a generalization of the policy used by the 6-way set-associative L1 cache of the Intel Atom D525, which we described in [AR13].

4.4.2 L2 Caches

The L2 caches of CPUs with the Nehalem, Westmere, Sandy Bridge, Ivy Bridge, Haswell, and Broadwell microarchitectures use the PLRU policy. The more recent generations use two variants of QLRU replacement.

Core 2 Duo E6750

Figure 4.7 shows an age graph for the access sequence "$\langle$wbinvd$\rangle$ B_0 ... B_{15}" on a Core 2 Duo E6750 (i.e., an access sequence with associativity many blocks). The sequence is accessed in all 4096 cache sets of this CPU, starting with an empty cache. The graph shows that an access to B_i leads to hits in all sets if fewer than 8 additional blocks (in all sets) have been accessed since the previous access to B_i. If between 8 and 15 additional blocks have been accessed, an access to B_i leads to hits in about 50% of the cache sets. For each additional sequence of 8 blocks, the number of hits decreases by about 50%.

Figure 4.8 shows an age graph for the sequence "$\langle$wbinvd$\rangle$ B_0 ... B_{15} B_{10}". We can see that the additional access to B_{10} changes the order in which the blocks are evicted from the cache. More specifically, for the eight blocks

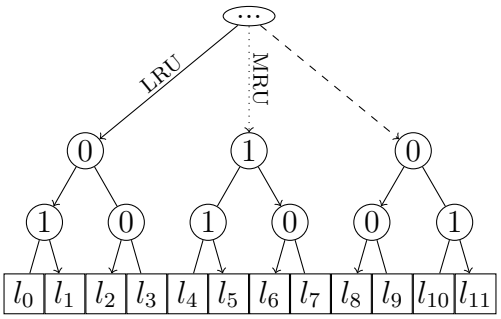

Figure 4.4: Possible $\mathrm{LRU}_3\mathrm{PLRU}_4$ state after an access to l_4, © 2020 IEEE

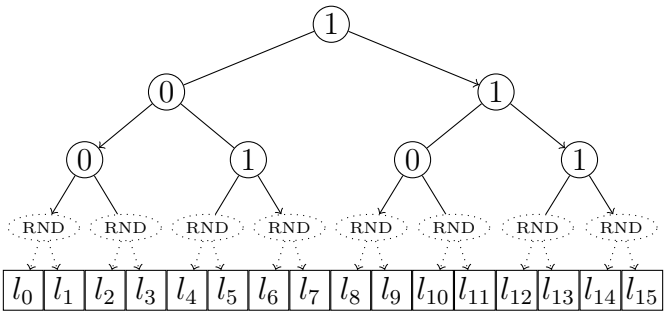

Figure 4.5: Possible $\mathrm{PLRU}_8\mathrm{Rand}_2$ state after an access to l_4, © 2014 IEEE

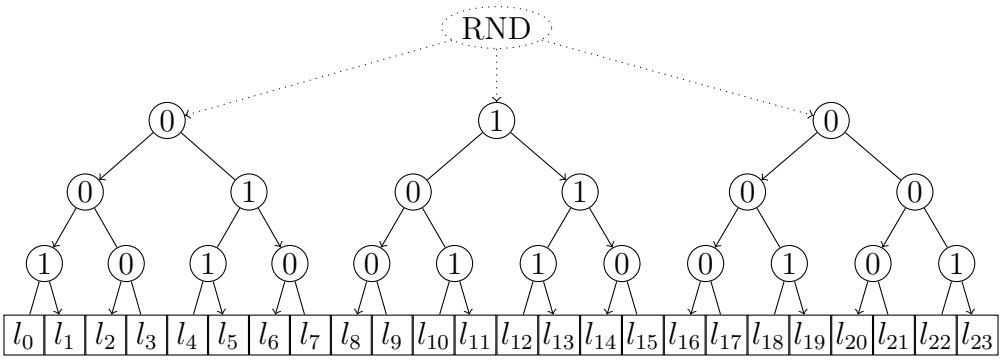

Figure 4.6: Possible $\mathrm{Rand}_3\mathrm{PLRU}_8$ state after an access to l_4, © 2014 IEEE

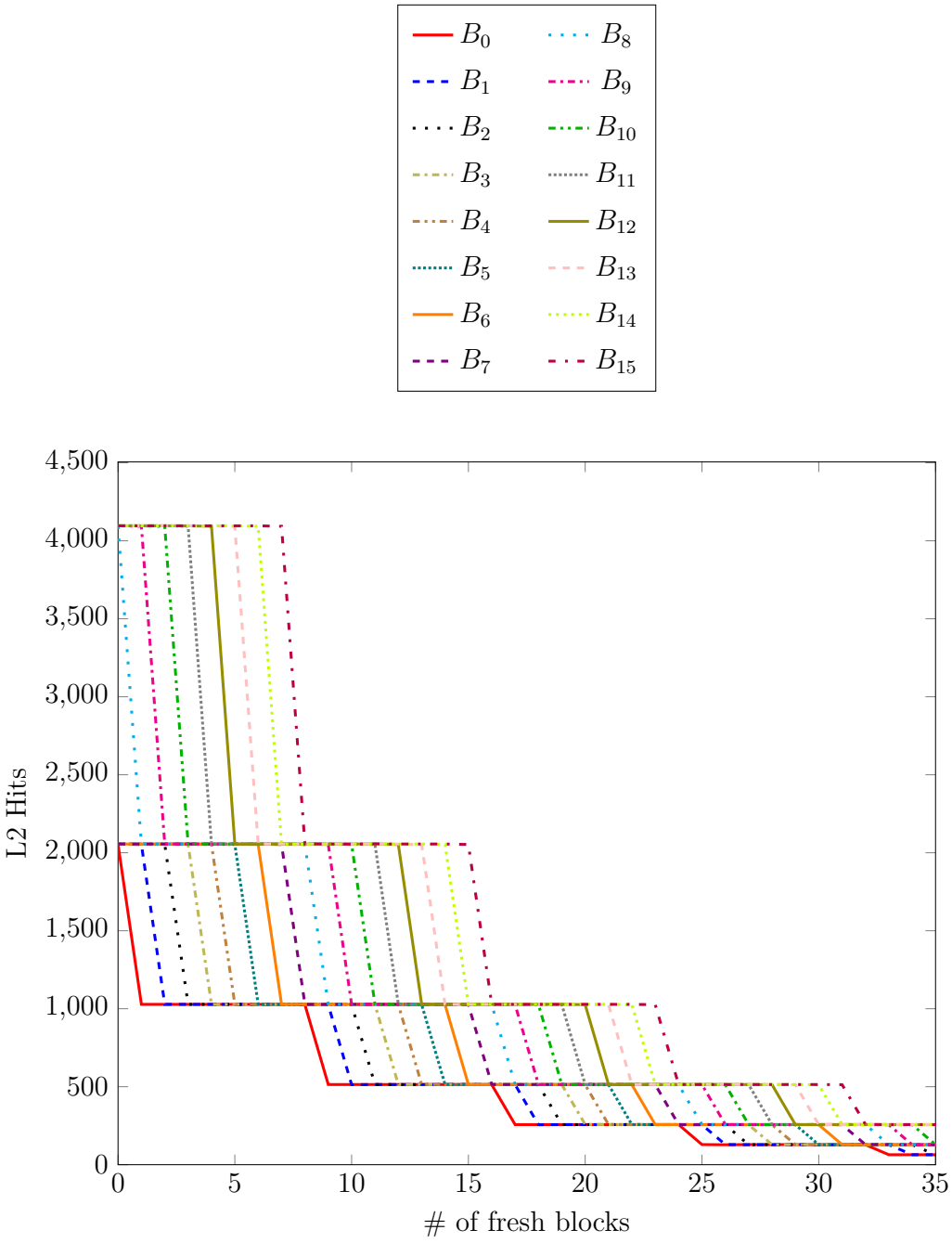

Figure 4.7: Core 2 Duo E6750 age graph for the access sequence "⟨wbinvd⟩ B_0 ... B_{15}"

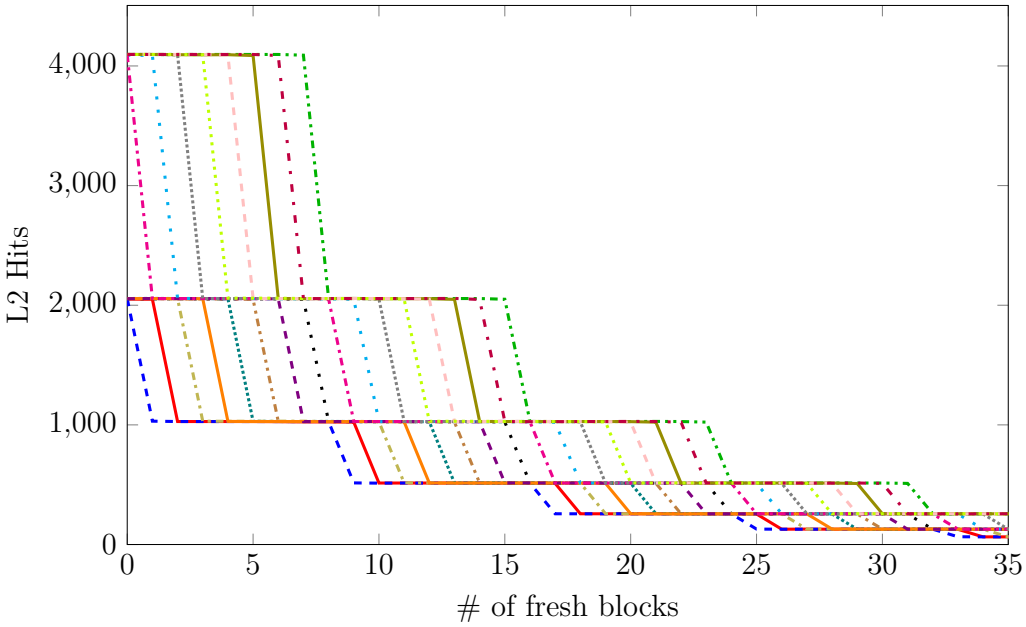

Figure 4.8: Core 2 Duo E6750 age graph for the access sequence "⟨wbinvd⟩ B_0 ... B_{15} B_{10}"

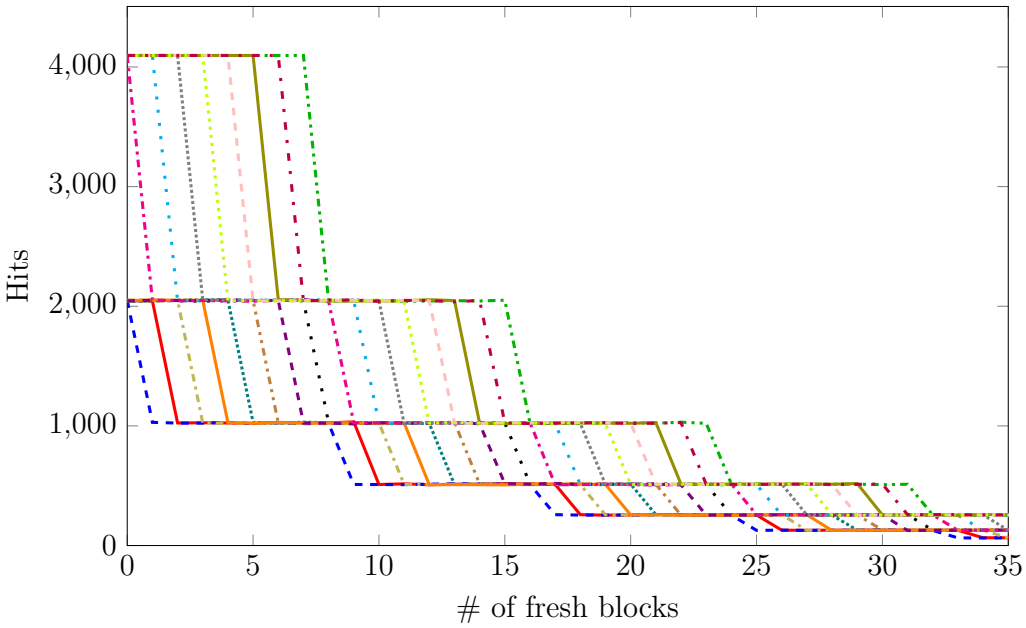

Figure 4.9: Simulated age graph for $\text{PLRU}_8\text{Rand}_2$ for the access sequence "⟨wbinvd⟩ B_0 ... B_{15} B_{10}"

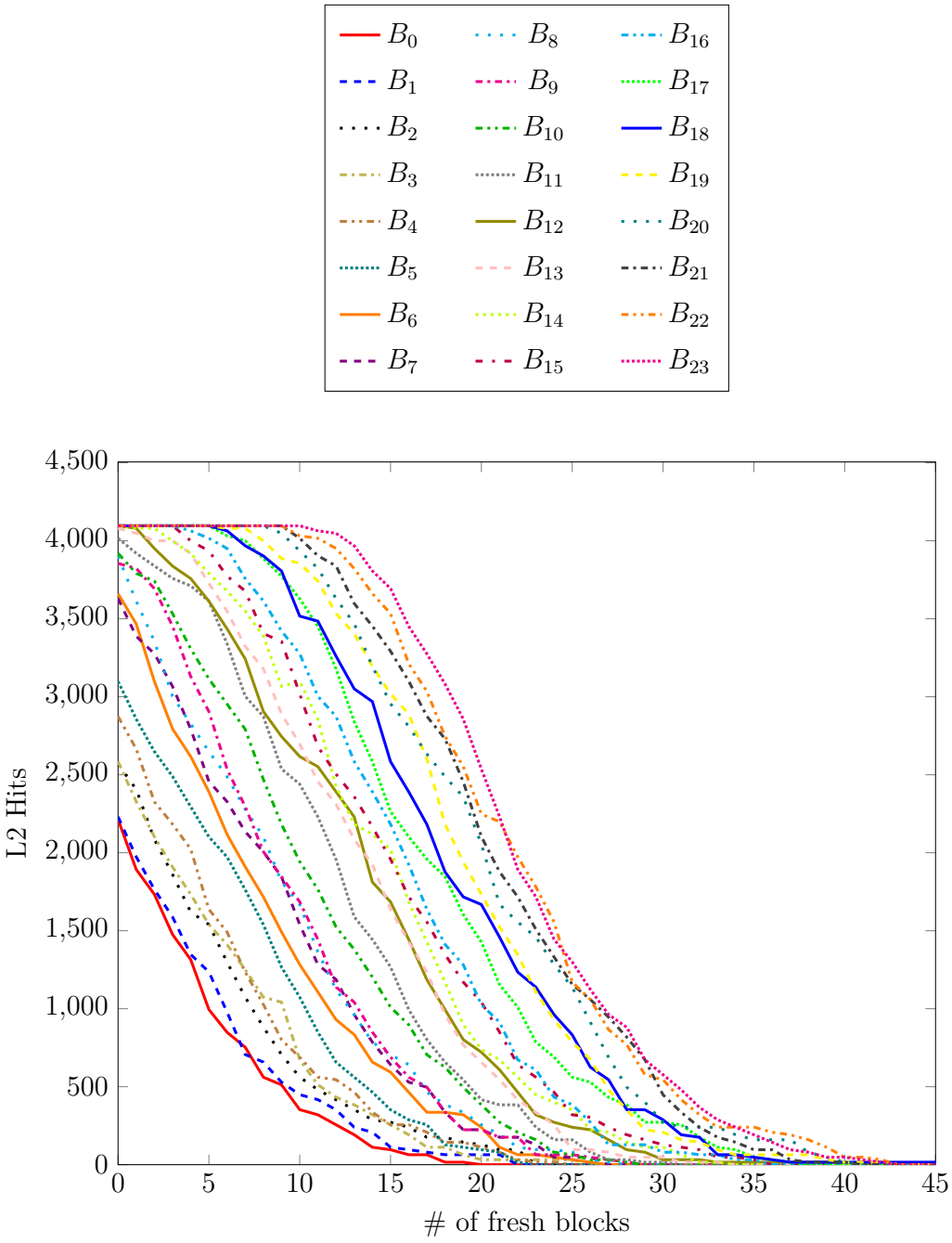

Figure 4.10: Core 2 Duo E8400 age graph for the access sequence "⟨wbinvd⟩ B_0 ... B_{23}"

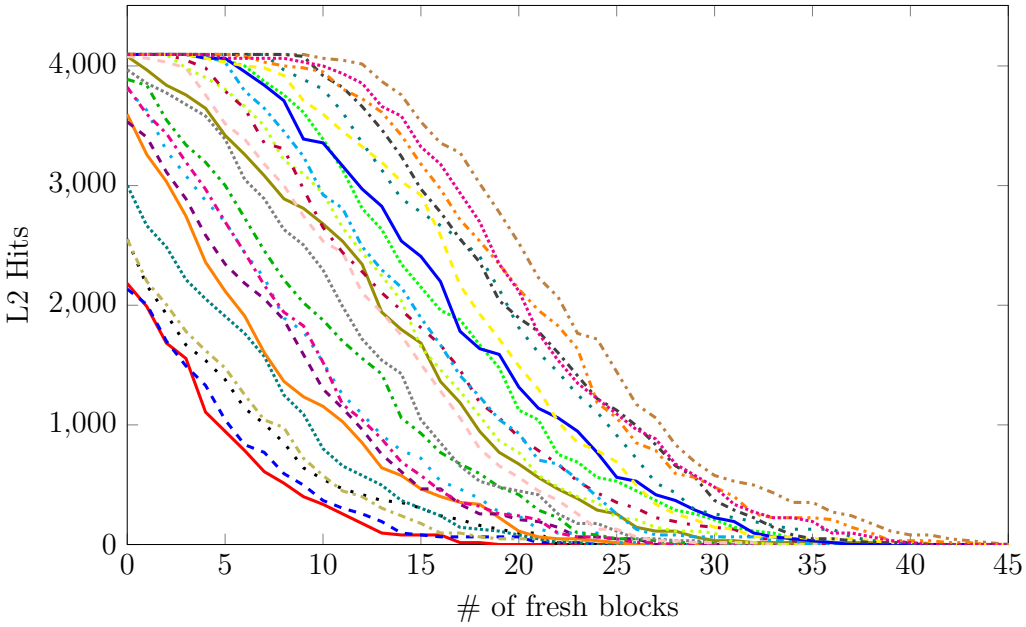

Figure 4.11: Core 2 Duo E8400 age graph for the access sequence "⟨wbinvd⟩ B_0 ... B_{23} B_4"

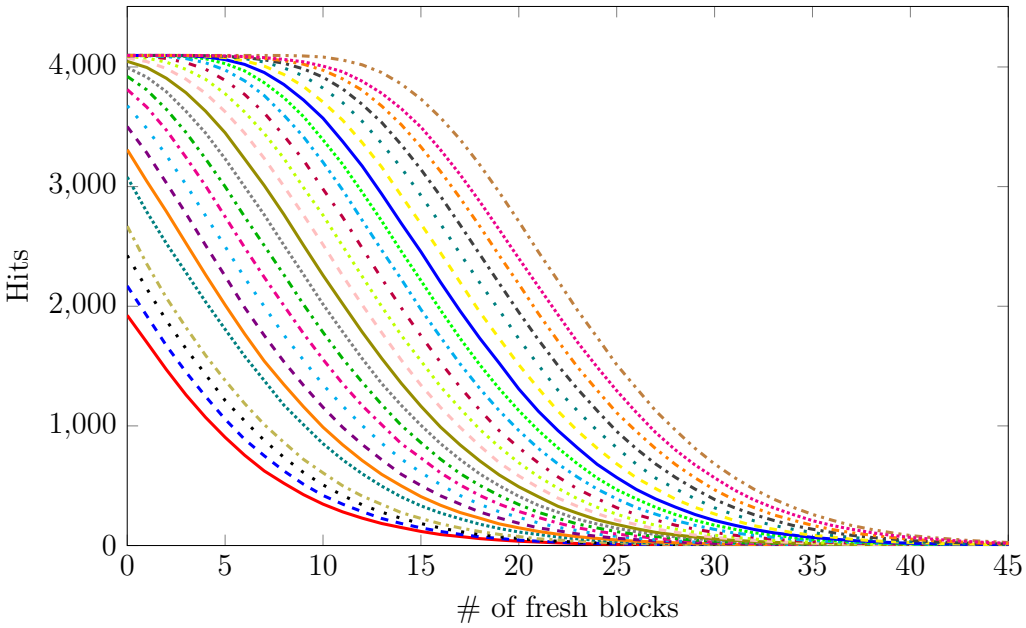

Figure 4.12: Simulated age graph for $\text{Rand}_3\text{PLRU}_8$ for the access sequence "⟨wbinvd⟩ B_0 ... B_{23} B_4"

$B_8, \ldots, B_{15}$, their relative order changes in the same way as in an 8-way set-associative cache with the PLRU policy (this corresponds to permutation Π_5^{PLRU} in Figure 4.2). Furthermore, we can see that the access to B_{10} changes the relative order of the eight blocks $B_0, \ldots, B_7$ in the same manner.

The following model agrees with these observations. Consider a PLRU-like policy in which the lowest bits (i.e., the bits that are closest to the leaves) are replaced by (pseudo-)randomness. In the following, we will call this policy $\text{PLRU}_8\text{Rand}_2$; Figure 4.5 illustrates the policy. Under this policy, one of the two elements to which the tree bits point is replaced with a probability of 50%. Furthermore, after every eight subsequent misses the tree bits point to the same subtree.

Figure 4.9 shows the result of a simulation of the $\text{PLRU}_8\text{Rand}_2$ policy on the same access sequence that was used for Figure 4.8. We can see that the simulation matches the actual measurements very closely.

Core 2 Duo E8400

Figure 4.10 shows an age graph for the access sequence "$\langle$wbinvd$\rangle$ B_0 ... B_{23}" on a Core 2 Duo E8400, which has a 24-way set-associative L2 cache. Similarly as with the E6750, an access to B_i leads to hits in all sets if fewer than 8 additional blocks have been accessed since the previous access to B_i. However, unlike with the E6750, the number of hits decreases more continuously if more additional blocks are accessed.

Figure 4.11 shows the effect of an additional access to block B_4. This block is now the block that gets evicted last. The relative order of the other blocks does not change significantly.

A possible model that leads to a similar behavior is the following. Consider a PLRU-like policy in which the root node is replaced by (pseudo-)randomness, as illustrated in Figure 4.6; in the following, we will call this policy $\text{Rand}_3\text{PLRU}_8$. Under this policy, the elements are separated into three groups with eight elements each; within each group they are managed by the PLRU policy. Upon a miss, one of the groups is chosen randomly.

Figure 4.12 shows the result of a simulation of the $\text{Rand}_3\text{PLRU}_8$ policy on the same access sequence that was used for Figure 4.11. The results are quite similar; however, the graph of the simulation has a more regular structure. A possible cause for the differences could be that the simulation uses a different random number generator than the actual hardware.

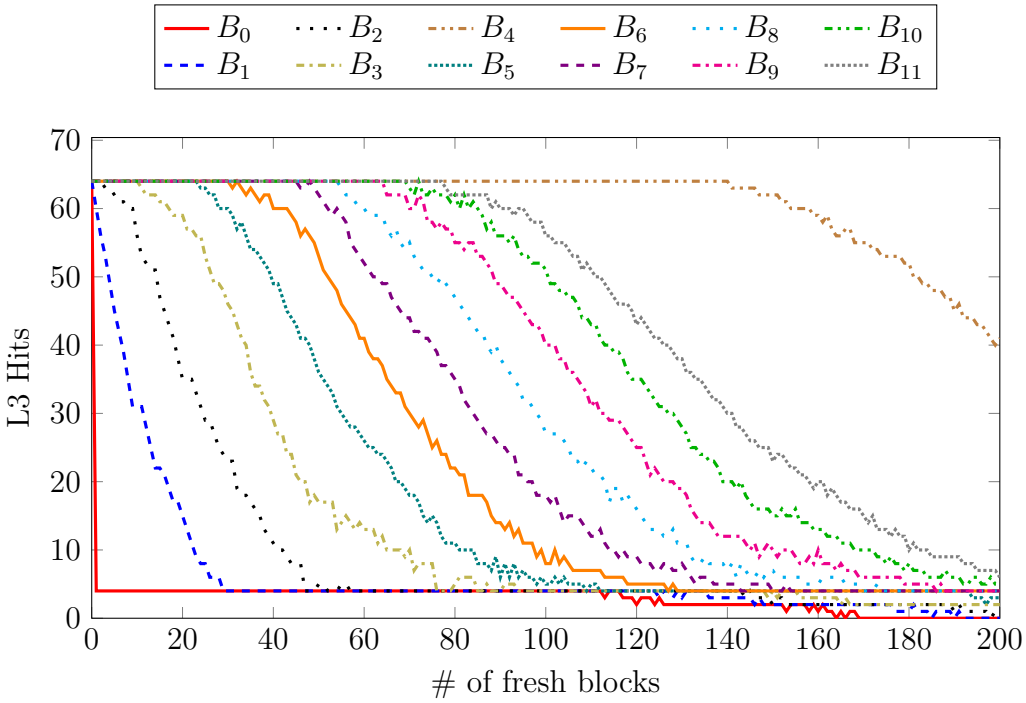

Figure 4.13: Ivy Bridge age graph for the access sequence "⟨wbinvd⟩ B_0 ... B_{11} B_4", © 2020 IEEE

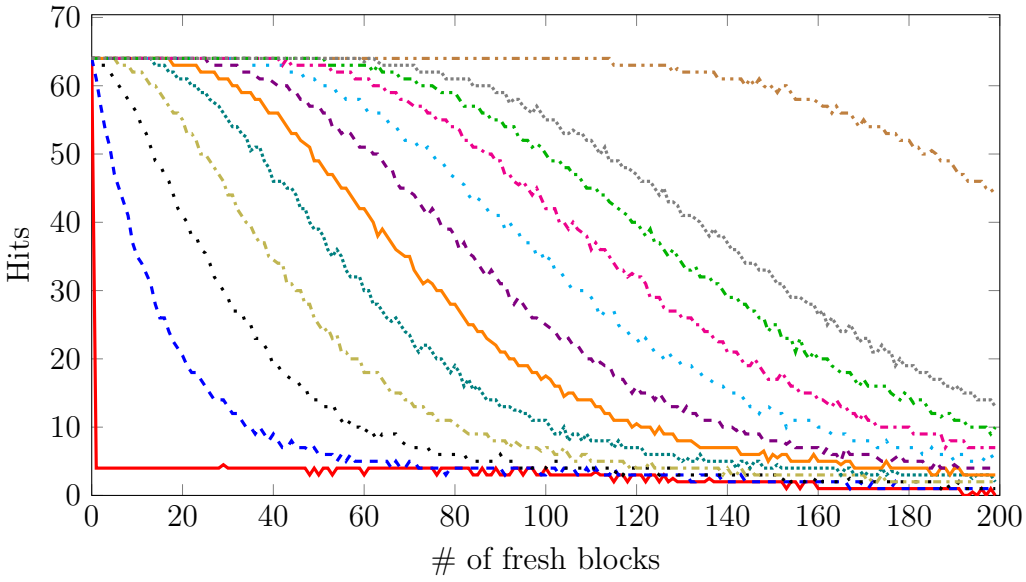

Figure 4.14: Simulated age graph for QLRU_H11_MR_{16}1_R1_U2 for the access sequence "⟨wbinvd⟩ B_0 ... B_{11} B_4", © 2020 IEEE

4.4.3 L3 Caches

The Nehalem and Westmere CPUs use the MRU replacement policy in their L3 caches. This was also reported by [ENBSH11]. The Sandy Bridge CPU uses a variant of this policy that, upon clearing the cache with the *WBINVD* instruction, sets all status bits to one, and does not update the status bits until all lines are filled.

The more recent generations use adaptive policies with different variants of QLRU replacement.

Ivy Bridge

For the Ivy Bridge machine, we found that the cache sets 512–575 and the cache sets 768–831 (in all C-Boxes) use a fixed policy, whereas the other sets are follower sets. Figure 4.15 shows the corresponding graph that was generated by the tool for analyzing adaptive policies described in Section 4.3.5.

According to our results, the sets 512–575 use the QLRU_H11_M1_R1_U2 policy.

The policy used by the sets 768–831 appears to be nondeterministic. Figure 4.13 shows an age graph for the access sequence "$\langle$wbinvd$\rangle$ B_0 ... B_{11} B_4" (note that the associativity of the cache is 12); the accesses of the sequence were performed in all of these 64 sets.

We can see that the curves for B_i and B_{i+1} ($i > 0$) are similar, but shifted by about 16 (except for B_4, which is accessed twice in the sequence). Furthermore, for B_0, about $\frac{15}{16}$ of the blocks are evicted immediately when the first fresh block is accessed, while the remaining $\frac{1}{16}$ of the blocks remain in the cache relatively long.

A policy that has a similar behavior for B_0, and for which the corresponding curves are also separated by about 16, is the QLRU_H11_MR$_{16}$1_R1_U2 policy, i.e., a variant of the policy used in sets 512–575 that inserts new blocks with age 1 in $\frac{1}{16}th$ of the cases, and with age 3 otherwise.

Figure 4.14 shows an age graph for a simulation of this policy. The graph is similar to Figure 4.13, though not identical. One explanation for the differences could be that the hardware actually uses a different policy variant. However, an alternative explanation could be that the differences are due to the hardware and the simulation using different random number generators. A more thorough investigation of the differences is left as future work.

Haswell, Broadwell

The Haswell and Broadwell CPUs use the same cache sets as the Ivy Bridge processor as dedicated sets, but only in C-Box 0. All other cache sets are follower sets.

Both CPUs use the QLRU_H11_M1_R0_U0 replacement policy in sets 512–575 in C-Box 0. The policy in sets 768–831 in C-Box 0 might be the QLRU_H11_MR_{16}1_R0_U0 policy.

Skylake, Kaby Lake, Coffee Lake, Cannon Lake

The Skylake, Kaby Lake, Coffee Lake, and Cannon Lake CPUs use the QLRU_H11_M1_R0_U0 policy in 16 fixed cache sets[2]. These sets were first discovered by Vila et al. [VKM19, VGGK20].

Figure 4.16 shows a graph created on the Kaby Lake machine using the tool for analyzing adaptive policies described in Section 4.3.5. This graph was created using *cacheSeq's* option of not adding additional accesses for clearing higher cache levels (see Section 4.3.2). The associativity of the L3 cache on these CPUs is higher than the associativities of the higher-level caches; we verified that the sequences used for generating the graph do not lead to hits in the higher-level caches.

The graph shows that for the 16 sets mentioned above, the test sequences always lead to 0 hits in the L3 cache. The policy in the remaining sets changes, depending on the number of hits and misses in the sets with the fixed policy, between this policy and a policy that is slightly more thrashing-resistant; unlike with the adaptive policies used by previous generations, there appear to be no sets that can only use the second policy.

We were not able to identify the second policy. Unlike the tool for analyzing adaptive policies, the tools for determining the replacement policy (see Section 4.3.3) require using cacheSeq's default option that adds accesses to other slices to clear higher-level caches on these machines. However, when using this option, we did not observe any differences between the two policies. In particular, for the thrashing sequence with $(associativity + 1)$ many elements that the tool for analyzing adaptive policies uses, we always observed 0 hits in all sets when using this option. A further investigation of this phenomenon is left as future work.

[2]Sets 0, 33, 132, 165, 264, 297, 396, 429, 528, 561, 660, 693, 792, 825, 924, and 957

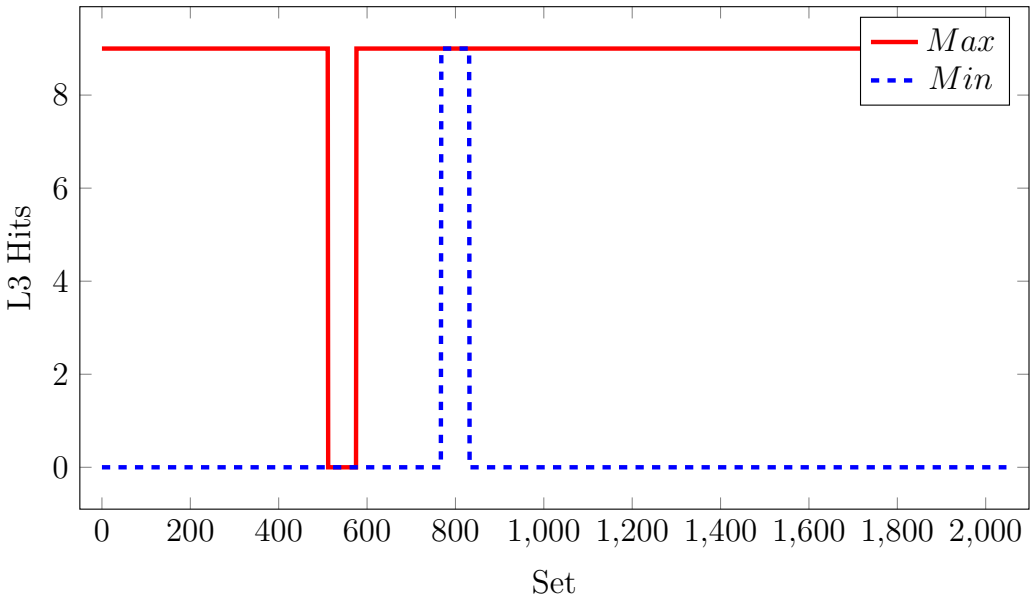

Figure 4.15: Test for adaptive policies on Ivy Bridge

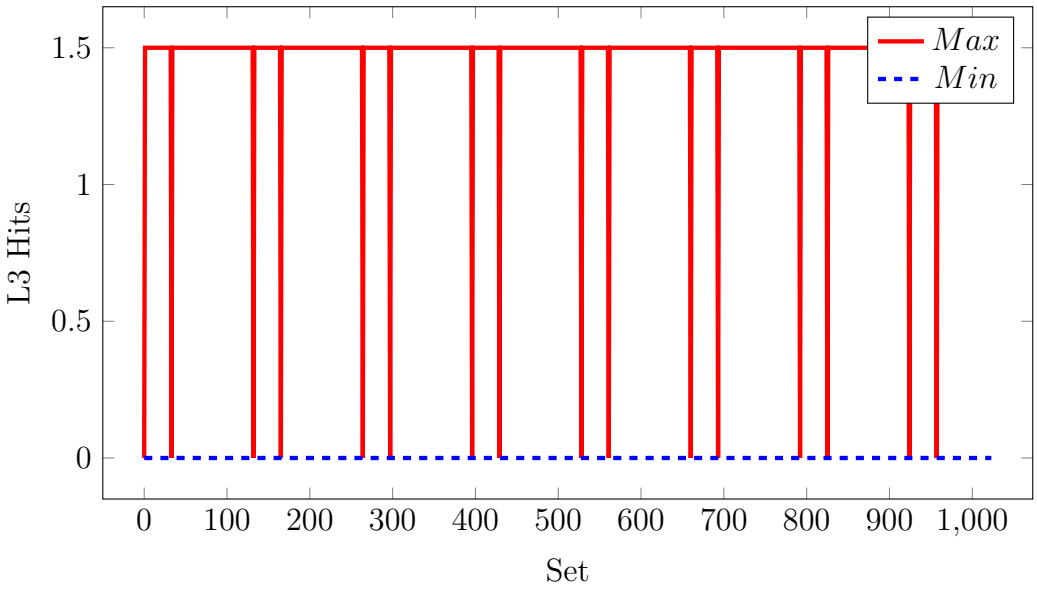

Figure 4.16: Test for adaptive policies on Kaby Lake

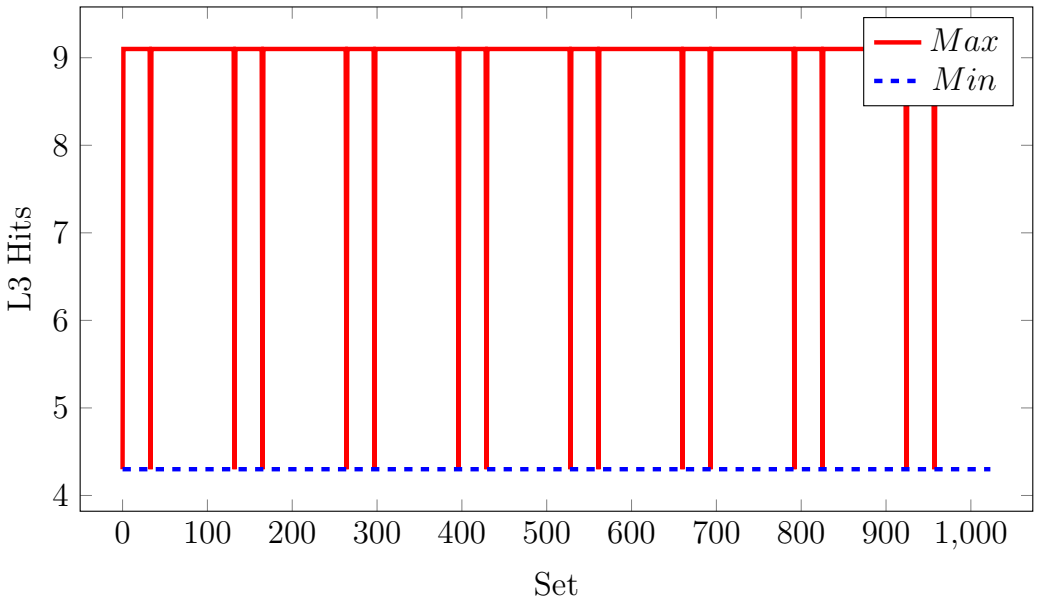

Figure 4.17: Test for adaptive policies on Ice Lake

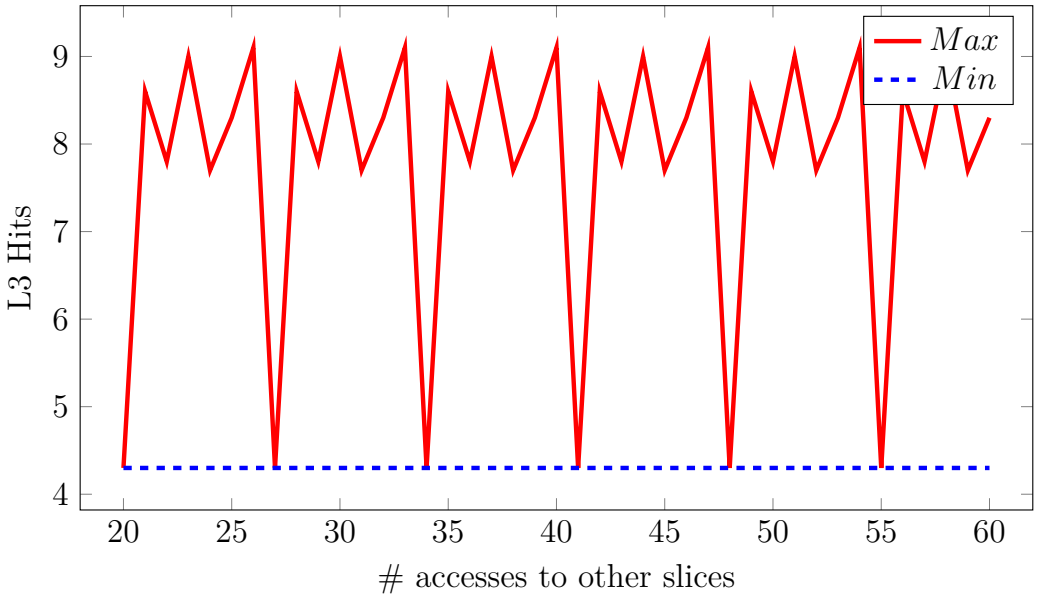

Figure 4.18: Minimum and maximum number of L3 hits on Ice Lake in cache set 1, depending on the number of accesses to other slices

Ice Lake

On Ice Lake, the same 16 cache sets described in the previous section use a fixed replacement policy: they use the QLRU_H00_M1_R0_U1 policy (i.e., the same policy as the L2 cache).

Figure 4.17 shows a graph created using the tool for analyzing adaptive policies described in Section 4.3.5. Unlike the graph in the previous section, this graph was created with *cacheSeq's* default option that adds accesses to other slices to clear higher-level caches (this was also necessary here, as the associativity of the L3 cache is not larger than the associativity of the L1 cache).

At first sight, the graph looks similar to the graph in Figure 4.16. However, the minima and maxima of the number of L3 hits are significantly higher than in the previous graph. Thus, the policy used in the fixed sets has some thrashing resistance, unlike the policies in the fixed sets of previous microarchitecture generations. The remaining sets can switch between this policy and a second policy that is even more thrashing-resistant; which of the two policies is used depends on the recent number of hits and misses in the fixed sets.

However, further experiments showed that the exact behavior of the second policy depends on the number of accesses to other slices.

Figure 4.18 shows a graph that was generated by varying the number n of accesses to other slices; the y-axis shows the number of L3 hits in cache set 1 for the same access sequence used for generating Figure 4.17 (i.e., an access sequence with $(associativity + 1)$ many elements that is repeated 10 times).

For values of n such that $n + 1$ is a multiple of 7, the number of L3 hits is the same for both replacement policies that set 1 can use. The tool described in Section 4.3.3 shows that the two policies are in fact the same for these values of n (for this experiment, we used special initialization sequences that produce a lot of hits and misses in the fixed sets, respectively, to trigger the corresponding replacement policy in the remaining sets). For other values of n, the second replacement policy is none of the variants that our tool considers. However, the graph suggests that the second policy might be a variant of the policy used in the fixed sets that inserts new blocks with age 3 instead of age 1 on every 7th access, where the corresponding access counter takes all slices into account. A further investigation of this hypothesis is left as future work.

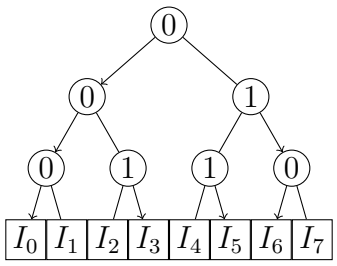

Figure 4.19: Possible PLRU state after accessing "I_0 ... I_7 I_1 I_2 I_4"

4.4.4 Resetting the Replacement Policy State

As described in Section 4.3.3, the behavior after clearing the caches with the *WBINVD* instruction appeared to be nondeterministic on several microarchitectures. In particular, this was the case for the L1 caches of the Core 2 Duos, the Sandy Bridge, Ivy Bridge, Haswell, and Broadwell CPUs, and the L2 caches of the Skylake, Kaby Lake, and Coffee Lake CPUs.

For the Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, Kaby Lake, and Coffee Lake CPUs, we found out that the apparent nondeterminism is actually due to a dependency on the previous replacement policy state (i.e., the state before the *WBINVD* instruction was executed). For Haswell and Skylake, this was also discovered independently by Pepe Vila [Vil19].

For the L1 caches of the Sandy Bridge, Ivy Bridge, Haswell, and Broadwell CPUs (which are 8-way set-associative and use the PLRU policy), the access sequence

$$\text{"}I_0 \ \ldots \ I_7 \ I_1 \ I_2 \ I_4 \ \langle\text{wbinvd}\rangle \ B_0 \ \ldots \ B_7 \ B_8 \ B_0?\text{"}$$

leads to a cache miss, whereas the sequence

$$\text{"}I_0 \ \ldots \ I_7 \ I_0 \ \langle\text{wbinvd}\rangle \ B_0 \ \ldots \ B_7 \ B_8 \ B_0?\text{"}$$

leads to a cache hit.

This suggests that the PLRU tree bits are not updated when blocks are inserted into an empty cache. If blocks are inserted from left to right into an empty cache, then after executing the sequence "I_0 ... I_7 I_1 I_2 I_4", the tree bits will point to I_0. This is illustrated in Figure 4.19. If the blocks after the *WBINVD* instruction are inserted in the same order and the tree bits are not updated, the access to B_8 will evict B_0, and hence, we get a cache miss when accessing B_0 again. On the other hand, after executing "I_0 ... I_7 I_0", the tree bits will point away from I_0. Thus, the access to B_8 will not evict B_0.

For the L2 caches of the Skylake, Kaby Lake, and Coffee Lake CPUs (which are 4-way set-associative and use the QLRU_H00_M1_R2_U1 policy), the access sequence

"$I0\ I1\ I2\ I3\ I0\ I1\ I2\ I0\ I1\ I2\ \langle\text{wbinvd}\rangle\ B0\ B1\ B2\ B3\ B4\ B2?$"

leads to a cache miss, whereas the sequence

"$I0\ I1\ I2\ I3\ I0\ I1\ I3\ I0\ I1\ I3\ \langle\text{wbinvd}\rangle\ B0\ B1\ B2\ B3\ B4\ B2?$"

leads to a cache hit.

A possible explanation for this behavior is that the check whether there is still a line with age 3 after an access uses the previous ages for lines that are currently empty.

We observed the same differences when using *CLFLUSH* instructions instead of the *WBINVD* instruction.

Reset Sequences

Based on these results, we identified the following reset sequences (see Section 4.3.3) to established a fixed replacement policy state.

For the L1 caches of the CPUs mentioned above, we use the reset sequence

"$I_0\ \ldots\ I_7$",

and for the L2 caches, we use the reset sequence

"$I_0\ I_1\ I_2\ I_3\ I_0\ I_1\ I_2\ I_0\ I_1\ I_2$".

After executing these reset sequences, the subsequent behavior was always deterministic (note that the blocks of a reset sequence are not used again later).

Covert/Side Channels

Clearing the cache using the *WBINVD* or *CLFLUSH* instructions is sometimes recommended as a technique for mitigating covert channels and side channels [ZBW17, GYLH16]. Ge et al. claim that after executing the *WBINVD* instruction, "the caches are in a defined state" [GYLH16]. The experiments in this section show that this is not the case for several recent processors. Using the obtained insights to implement and evaluate covert channels is left as future work.

Table 4.4: Number of status bits required for different replacement policies, in terms of the associativity A.

Policy	Number of Status Bits	A=4	A=8	A=12	A=16	A=24
FIFO	$\lceil log_2(A) \rceil$	2	3	4	4	5
LRU	$\lceil log_2(A!) \rceil$	5	16	29	45	80
PLRU	$A - 1$	3	7	-	15	-
LRU_3PLRU_4	A	-	-	12	-	24
$PLRU_8Rand_2$	$\frac{A}{2} - 1$	1	3	-	7	-
$Rand_3PLRU_8$	$A - 3$	-	-	9	-	21
MRU	A	4	8	12	16	24
QLRU	$2 \cdot A$	8	16	24	32	48

4.4.5 Implementation Costs

The cost of implementing a particular replacement policy can be quantified in terms of the minimal number of status bits that an implementation requires [Rei08]. Table 4.4 shows these costs for different policies, including the policies that we uncovered in this chapter.

4.5 Related Work

4.5.1 Microbenchmark-Based Cache Analysis

A number of papers have proposed microbenchmark-based techniques for determining parameters of the memory hierarchy like the cache size, the associativity, the block size, or the latency [SS95, MS96, LT98, TY00, BC00, CD01, DMM+04, Man04, JB07, YPS05, YJS+06, BT09, MHSM09, WPSAM10, GDTF+10, CS11, AR12, Abe12, DLM+13, HKP15, MC17, CX18].

While some of these approaches make assumptions as to the underlying replacement policy (e.g. [SS95] and [TY00] assume that LRU is used), only a few publications have also tried to determine the replacement policy.

The approaches described in [CD01] and [BC00] are able to detect LRU-based policies but treat all other policies as random. John and Baumgartl's [JB07] approach is able to distinguish between LRU and several of its derivatives.

In [Abe12, AR13], we proposed an algorithm that can automatically infer *permutation policies* (see Section 4.2.2). In the present work, we developed an improved implementation of this algorithm that can infer the policies in

individual cache sets; the implementation in [Abe12, AR13] was based on the assumption that all cache sets use the same policy.

Henry Wong [Won13] discovered that Ivy Bridge CPUs use set dueling to switch between two different replacement policies. He identified the sets that use fixed policies; however, he was not able to determine which two policies are actually used. Similar work was described by Zhang et al. [ZGY14].

Briongos et al. [BMME19] present an approach for analyzing the replacement policies used by the L3 caches in recent Intel CPUs. Similar to the technique described in Section 4.3.3, their approach generates random access sequences and compares their behavior to simulations of different policies. However, unlike in our approach, they do not measure the total number of hits that the sequence generates. Instead, they only determine the first element to be evicted upon the first miss after executing the access sequence. Furthermore, unlike in our approach, they rely on timing measurements instead of using performance counters.

Briongos et al. applied their technique to CPUs with the Haswell, Broadwell, Skylake, and Kaby Lake microarchitectures. Our results for these microarchitectures disagree with their results. The policies they describe would be the QLRU_H21_M2_R0_U0_UMO and QLRU_H21_M3_R0_U0_UMO variants according to our naming scheme. Our tool found several counterexamples for these policies on all of the tested CPUs. Briongos et al. also stated that the two policies did not agree with all of their observations; however, they assumed that "the errors were due to noise". Furthermore, according to Briongos et al., the dedicated sets on the Haswell and Broadwell CPUs are distributed over different slices; according to our results, they are all in the same slice. As, according to the paper, they use an approach from [LYG+15], we assume that they also rely on a statement from that paper that "when the number of cores in the processor is a power of two, the set index bits are not used for determining the LLC slice." This was, however, shown to be incorrect in later work [MSN+15]. Thus, their observations rather seem to be an artifact of the hash function used for determining the cache slices. Briongos et al. did not find the sets with a varying policy on Skylake and Kaby Lake, and thus incorrectly concluded that these CPUs do not use an adaptive policy.

Rueda [RC13] developed a technique for learning replacement policies using register automata. He was able to learn the FIFO and LRU policies for caches with an associativity of at most 5, and the PLRU and MRU policies for caches with an associativity of at most 4. He did not successfully apply his technique to actual hardware.

In concurrent work, Vila et al. [VGGK20] describe an approach for inferring replacement policies using automata learning, and an approach for automatically generating human-readable representations of the learned policies. For software-simulated caches, they were able to learn FIFO and PLRU up to associativity 16, MRU up to associativity 12, and several other policies up to associativity 6. Furthermore, Vila et al. also applied their techniques to actual hardware with the Haswell, Skylake, and Kaby Lake microarchitectures. They successfully learned the policies used by the L1 and L2 caches of these three processors, as well as the policy used by the leader sets of the L3 caches on Skylake and Kaby Lake. Their results agree with our results. Vila et al.'s approach was, however, not able to learn the policies used by the L3 cache of the Haswell CPU, as the associativity was too high for their approach, and one of the policies is nondeterministic.

Vila et al.'s approach relies on a tool called *CacheQuery*, that is quite similar to the *CacheSeq* tool proposed in Section 4.3.2. The main differences are the following:

1. *CacheQuery* uses a more expressive syntax,
2. *CacheQuery* is based on timing measurements, whereas *CacheSeq* uses performance counters, and
3. *CacheQuery* requires the parameters of the caches, such as the associativities or the number of cache sets, to be specified manually, whereas *CacheSeq* determines them automatically using *CacheInfo*.

4.5.2 Influence of the Replacement Policy on Performance Prediction Accuracy

Multiple works have identified the cache replacement policy as an important factor for the accuracy of performance prediction techniques.

Grund et al. [GR08] describe an analytical method to estimate miss ratios for a given cache configuration. They consider the replacement policy to be an important component that "can have a significant influence on the cache performance." Guo et al. [GS06] present an analytical model for cache replacement policy performance and claim that the "replacement policy is a factor that should be taken into account in designing a cache and modeling cache performance." Tam et al. [TASS09] model the performance of a system in terms of miss rate curves (MRCs) that plot the miss rate of an access sequence as a function of the cache size. They state that "the MRC of a Least Recently Used (LRU) policy may be significantly different from that of

a Most Recently Used (MRU) policy for the same memory access sequence." Furthermore, according to Kegley et al. [KPD+11], the replacement policy "can substantially influence the performance of a system."

Unfortunately, current performance modeling approaches often do not consider different policies. Tam et al.'s approach [TASS09], for instance, only supports LRU as they believe that "it is the most commonly used replacement policy." Similarly, Fraguela et al. [FDTZ04] claim that LRU "nowadays is by far the most common" policy. As our work shows, this is not true for most current microarchitectures.

4.5.3 Security Aspects of Replacement Policies

Caches are at the center of many covert-channel and side-channel attacks [OST06, GBK11, YF14, LYG+15, GMWM16, DKPT17, MWK17, TLM18, GYCH18, DXS19, vSMK+20, LHS+20].

Several papers show that knowledge of the replacement policy can be important for devising or mitigating such attacks.

In [KMO12], Köpf et al. propose a technique for automatically deriving upper bounds on the amount of information leakage through cache side channels. They focus on the LRU replacement policy. CacheAudit [DKMR15, BKMO14, DK17, CnKR17] improves upon this work by considering more types of adversaries, by providing tighter bounds, and by considering different replacement policies. In particular, their current implementation supports caches that use the FIFO, LRU, and PLRU policies.

In [CnKR19], Cañones et al. present techniques for comparing caches with different replacement policies with respect to their vulnerability to side-channel attacks.

Accessing memory locations with a high frequency can lead to bit flips in neighboring DRAM rows [KDK+14]; this phenomenon is commonly called the *Rowhammer bug*. Gruss et al. [GMM16] describe a JavaScript-based attack that exploits this bug to obtain unrestricted access to the systems of website visitors. To this end, they require an efficient *eviction strategy*, i.e., an access sequence that can be used instead of the *CLFLUSH* instruction, which is not available in JavaScript. Whether an eviction strategy is efficient on a specific CPU depends on the replacement policy. As Gruss et al. do not know the replacement policies used in recent Intel CPUs, they go to great lengths to find suitable eviction strategies; overall, they evaluate more than 18,000 possible strategies.

Kim et al. describe StealthMem [KPMR12], a technique that manages a set of locked lines which are never evicted from the last-level cache. Their implementation is based on the assumption that the cache has the k-LRU property, which means that the replacement policy will never evict the k most-recently accessed blocks. To find a value for k for which their processor with the Nehalem microarchitecture has the k-LRU property, they propose the following experiment. They use access sequences of the form "$B_0\ B_1\ \ldots\ B_{k'}\ B_0$?" and evaluate them with increasing values for k' between 1 and 16 (which corresponds to the associativity). They start seeing L3 misses at $k' = 15$ and conclude that the cache has the 14-LRU property. This is incorrect. According to the results in this chapter, and also according to Eklov et al. [ENBSH11], the L3 caches of Nehalem-based processors use the MRU replacement policy. Caches with this policy do not have the k-LRU property for any $k > 1$. As an example, consider the access sequence "$\langle$wbinvd$\rangle\ B_0\ \ldots\ B_{14}\ B_0\ B_{15}\ B_{16}\ B_0$?". The access to B_{15} will flip the status bits of all other blocks. Consequently, the access to B_{16} will evict B_0, and thus, the final access to B_0 will lead to a cache miss. We verified that this is indeed the case on our Nehalem system.

Kiriansky et al. [KLA+18] propose hardware modifications to defend against cache timing attacks. In particular, they propose a cache partitioning technique that, unlike previous partitioning techniques, takes the state of the replacement policy into account to ensure isolation. They describe implementations of this technique for PLRU, MRU, and SRRIP.

Briongos et al. [BMME19] present an attack that exploits the state of the L3 replacement policy to extract information from a victim; their attack is undetectable by countermeasures that rely on monitoring cache misses. Xiong and Szefer [XS20] describe a related attack that uses the state of the replacement policy in the L1 cache to leak information.

4.6 Conclusions and Future Work

We developed several tools that generate microbenchmarks for analyzing properties of caches, focusing, in particular, on cache replacement policies. We applied our tools to 13 recent Intel microarchitectures, and we identified several previously undocumented replacement policies. Furthermore, we discovered that, contrary to popular belief, flushing the cache does not necessarily reset the state of the replacement policy; this might be used to devise new covert-channel and side-channel attacks.

Future Work

There are multiple directions for future work. One goal is to increase the class of policies that can be inferred fully automatically. Currently, our tools are able to infer all permutation policies, as well as around 300 deterministic policies that are not permutation policies. A promising approach is to use automata learning techniques, as shown by Rueda [RC13] and Vila et al. [VGGK20]. Future work would include enhancing these approaches to support larger associativities. Furthermore, it would be interesting to develop techniques that can identify nondeterministic policies automatically.

Another direction for future work would be to add support for non-Intel CPUs. For AMD CPUs, it would likely be sufficient to make our algorithms robust against noise introduced by the prefetchers, which cannot be disabled on these CPUs. A related goal would be to infer models of the prefetchers themselves. To add support for non-x86 CPUs, we would need to extend *nanoBench*, as proposed in Section 2.6.

Furthermore, we would like to extend our tools to also support instruction and µop caches (see Section 3.3), which are typically only poorly documented.

5

Gray-Box Learning of Serial Compositions of Mealy Machines

The techniques we proposed in the previous chapters can be seen as instances of *active learning* approaches. *Active learning* (also called *query learning*) refers to a class of machine-learning techniques in which the learning algorithm is able to interact with the system to be learned.

The techniques described in the previous chapters were heavily targeted at the specific problems. In this chapter, we look at more general techniques. Specifically, we consider approaches for learning finite state machines, which are, in principle, suitable abstractions for modeling the behavior of microarchitectural components.

Most existing learning algorithms for finite state machines treat the system to be learned as a black box. These algorithms often do not scale to complex realistic systems, as the state space of such systems is typically too large.

In this chapter, we propose the concept of *gray-box learning* for dealing with such systems. The main idea hereby is to use available information about the system to focus the learning algorithms on the parts that are unknown.

As a first step toward solving this problem, we study one specific instance. We consider the serial composition of two Mealy machines A and B, where A is known and B is unknown, and we assume that we can only perform queries on the composed machine. For this scenario, we develop an efficient algorithm that learns a model of the unknown machine B.

This chapter is an extended version of [AR16]. We provide more details on the SAT reduction (in Section 5.5.1), and proofs for the lemmas and theorems (in Appendix 5.A), which were omitted from the original publication due to space reasons.

5.1 Introduction

Tools to analyze software or hardware systems, such as static analyzers or model checkers, require accurate models as input. Third-party components, however, are rarely specified at the level of detail required by such tools.

One approach to automatically obtain formal models of systems is active learning [Set11]. Here, one commonly assumes an oracle, or teacher, that admits two kinds of queries about the system: output queries return the result of the system for a specific input, and equivalence queries check whether a conjectured model is consistent with the system to be learned and return a counterexample if not. Based on this setup, Angluin introduced the L^* algorithm [Ang87] for learning deterministic finite automata. L^* has since been extended to other modeling formalisms, such as Mealy machines [SG09], register automata [HSJC12, BHLM13, AFBKV15], or symbolic automata [MM14]. It is also at the heart of several model checking approaches, including [CGP03, GPY02, VSVA05].

As the system is treated as a black box, no information about the internal structure of the system can be taken into account by most existing learning algorithms. In practice, however, systems are often composed of subcomponents, for some of which models might be available, but it is not possible to access the known and the unknown parts separately from the outside. Partial information about the inner workings of a system may be inferred from manuals or conjectured from similar, yet better documented systems. This scenario is depicted in Figure 5.1.

While it is in theory possible to learn a model of the entire system using existing black-box approaches, this is often not viable in practice because the state space is too large. A problem, which has received little attention in the literature so far, is how to use the available information about the system to focus the learning algorithm on those parts that are unknown. This problem could be termed *gray-box learning*.

In this chapter, as a first step toward solving this problem, we study one specific instance: We assume that the system C is the serial composition of two Mealy machines A and B, and that we have a model for the left machine (A) and want to learn the right machine (B). We further assume that we can perform output and equivalence queries only on C as a whole. This scenario is shown in Figure 5.2.

While output queries can often be realized cheaply by measurements on the actual system, equivalence queries can usually only be approximated by a

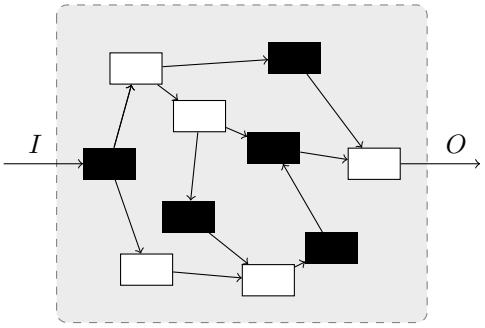

Figure 5.1: Mealy machine network

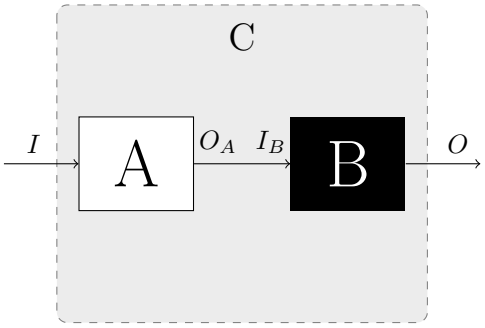

Figure 5.2: Serial composition

large number of such measurements. Our primary focus is thus to minimize the number of equivalence queries. We introduce an algorithm to exactly learn B in the context of A that performs at most $|B|$ equivalence queries, where $|B|$ denotes the number of states of B.

We evaluate our approach on compositions of randomly-generated machines against an implementation of the classic L^* algorithm in LearnLib [IHS15]. We show that our approach requires significantly fewer output and equivalence queries on most benchmarks.

5.2 Problem Statement

In this section, we first formally define several concepts used throughout this chapter. Then, we give a precise description of the problem that we address.

5.2.1 Basic Notions

Definition 5.1 (Completely Specified Mealy Machine)**.** A completely specified Mealy machine is a tuple (Q, I, O, δ, q_r), where

- $Q \neq \emptyset$ is a finite set of states
- $I \neq \emptyset$ is a finite set of input symbols
- $O \neq \emptyset$ is a finite set of output symbols
- $\delta : Q \times I \to Q \times O$ is the transition function
- $q_r \in Q$ is the initial (reset) state.

In this chapter, we only consider completely specified Mealy machines; incompletely specified Mealy machines are introduced in Chapter 6.

We extend δ to sequences in the usual way. We use ϵ to denote the empty sequence. Further, we use $M(x)$ to denote the output sequence of a Mealy machine M when reading x, and $M_L(x)$ to denote the last output of M when reading x.

Given two Mealy machines A and B, we can compose them to a serial Mealy machine C by using the output of A as the input for B. Formally:

Definition 5.2 ((Synchronous) Serial Composition of Mealy Machines)**.** Let $A = (Q_A, I_A, O_A, \delta_A, q_{r,A})$ and $B = (Q_B, I_B, O_B, \delta_B, q_{r,B})$ be two Mealy machines such that $O_A \subseteq I_B$. The serial composition of A and B is a Mealy machine $C = (Q, I, O, \delta, q_r)$, where

- $Q := Q_A \times Q_B$
- $I := I_A$
- $O := O_B$
- $\delta((q_A, q_B), i) := ((q'_A, q'_B), o)$, where $(q'_A, o_A) := \delta_A(q_A, i)$, and $(q'_B, o) := \delta_B(q_B, o_A)$
- $q_r := (q_{r,A}, q_{r,B})$

Given a composition of two Mealy machines A and B, we define a machine B' to be right-equivalent to B in the context of A if the composition of A and B describes a machine that is equivalent to the composition of A and B'. Formally:

Definition 5.3 (Right-Equivalence)**.** Let A, B, and B' be Mealy machines. Then, B' is *right-equivalent* to B in the context of A iff

$$\forall x \in I^* : B(A(x)) = B'(A(x)).$$

5.2.2 The Gray-Box Learning Problem

In this chapter, we address the following problem. We assume that we have a serial composition C of two Mealy machines A and B. Further, we assume that we have a model of A, but B is unknown. While we do not have a model of C, we assume that we can determine the output of C on any input by an *output query*, and we can test whether a machine is equivalent to C by an *equivalence query*.

Using existing techniques, like Angluin's L^* algorithm [Ang87], one could consider C to be a black box and learn a model of C. Such an approach would

in the worst case employ a polynomial number of output and equivalence queries in the size of C, which can be up to $|A| \cdot |B|$.

Instead, our goal is to exploit the knowledge we have about A, and to learn a model of a minimum-size machine B', such that B' is right-equivalent to B in the context of A. In particular, as we consider equivalence queries to be more expensive than output queries, we want the number of equivalence queries to be polynomial in the number of states of B', independently of the size of A.

5.3 Preliminaries

Existing active learning approaches for Mealy machines (and related machine types) are usually based on a Myhill-Nerode-like equivalence relation that partitions the set of input words into classes such that the words that are in the same class cannot be distinguished with respect to different suffixes:

Definition 5.4 (Equivalence of Input Words)**.** Given a function $F : I^* \to O$, two words $x, y \in I^*$ are equivalent, $x \sim y$, iff

$$\forall z \in I^* : F(x \cdot z) = F(y \cdot z).$$

F can be modeled by a Mealy machine iff this relation has finitely many equivalence classes. One can then construct a minimum-size Mealy machine whose states are the equivalence classes of this relation. Existing approaches compute the equivalence relation in a co-inductive fashion. In the beginning, they consider all words to be equivalent. Then, in each round, this hypothesis is refined by identifying at least one new equivalence class, until the equivalence relation is fully determined.

If we consider the machine B in the serial composition with A, then it is possible that not all input sequences for B can be produced by A. Let $tr(A) = \{A(x) \mid x \in I^*\}$ be the set of output sequences that A can produce. For each output sequence $x \in tr(A)$, there might be multiple input sequences that produce this output. Let $A^{-1} : tr(A) \to I^*$ be a function such that $A^{-1}(x)$ returns one of these input sequences. In the following, it will not be important which of the possibly multiple sequences is actually returned.

We have that every right-equivalent Mealy machine B' for B in the context of A has to agree with the partial function $F_P : I_B^* \rightharpoonup O$ such that

$$\forall x \in tr(A) : F_P(x) = B_L(x).$$

Note that while we do not have immediate access to B, we can use output queries on C to access B, as for all $x \in tr(A)$, $B_L(x) = C_L(A^{-1}(x))$.

Similarly to Definition 5.4, we define two words to be *right-compatible* in the context of A iff they cannot be distinguished with respect to different suffixes.

Definition 5.5 (Right-Compatibility)**.** Two words $x, y \in I_B^*$ are *right-compatible* in the context of A, $x \sim_A y$, iff

$$\forall z \in I_B^* : (xz \notin tr(A) \vee yz \notin tr(A) \vee B_L(xz) = B_L(yz)).$$

Otherwise, x and y are incompatible, $x \nsim_A y$.

However, right-compatibility is, unlike equivalence, not transitive. Thus, it is not an equivalence relation, which means we cannot directly use the construction sketched above to build a minimum-size machine.

To see this, consider three output symbols $a, b, c \in O_A$ with

$$\forall z \in I_B^* : az, bz \in tr(A) \wedge B_L(az) = 0 \wedge B_L(bz) = 1$$

and

$$\forall z \in I_B^* : cz \notin tr(A).$$

So B always outputs 0 if the first output of A was a, it always outputs 1 if the first output of A was b, and A never outputs c as the first output.

This means that $a \sim_A c$ and $b \sim_A c$, but $a \nsim_A b$. For this example, we can build a machine with three states that is right-equivalent to B. From the start state, a transition with c can go to any state. This also shows that there can be multiple machines with the minimum number of states that are right-equivalent to B.

5.4 Approach

Equivalence queries are typically assumed to be more expensive than output queries. Many existing active learning techniques therefore focus on keeping the number of required equivalence queries low.

At a high level, Angluin's L^* algorithm for instance, can be described as follows. In each round, the algorithm first performs a sequence of output queries in a systematic way, until there is exactly one machine of minimum size that is consistent with the results from all output queries performed so far. Only then, the algorithm performs an equivalence query. If this query returns a counterexample, this implies that the correct machine must have at least one additional state. Thus, Angluin's algorithm performs at most n equivalence queries, where n is the size of the minimal correct machine.

Unlike in Angluin's setting, in general no unique machine of minimum size that is consistent with a set of observations exists. The basic idea behind our approach is to perform output queries until *all* machines of minimum size that are consistent with these queries are right-equivalent in the context of A. We then perform an equivalence query for one of these machines. If this query results in a counterexample, this counterexample witnesses that all of these machines are incorrect, and thus, the correct machine must have at least one additional state.

One challenge is to find a suitable sequence of output queries that is guaranteed to reduce the number of machines that are consistent with all queries performed so far. The basic idea is to iteratively construct all machines of minimum size that agree with all of the previous queries. We can then check whether each pair of these machines is right-equivalent. If they are not, we use a distinguishing sequence as a counterexample, without performing an equivalence query.

However, applying this approach naively would not be viable in many cases because there can be an exponential number of machines of the same size that are consistent with a set of observations, in particular in the beginning, when only a small number of queries have been performed. Thus, we identify a number of necessary conditions for candidate machines to be right-equivalent which can be efficiently determined on observation tables. Some of these conditions correspond to notions from Angluin's algorithm, such as consistency and closedness, while others, like input-completeness, are special to our particular setting.

In the rest of this section, we describe our proposed algorithm in detail and introduce the necessary theoretical concepts. In particular, we describe in detail which output queries our algorithm performs to systematically reduce the number of machines that are consistent with the observations made so far.

5.4.1 Observation Tables

The main data structure used in our approach is an *observation table*. The rows of the table are indexed by a set of prefixes, the columns by a set of suffixes, and the entries of the table store the last output symbol of an output query for the concatenation of the corresponding prefix and suffix. If this concatenation is not a possible output sequence of the left machine A, we do not perform an output query but store $\perp$ in this cell instead. In contrast to most previous definitions, our observation tables *do not* consist of two explicitly distinguished parts.

Definition 5.6 (Observation Table)**.** An observation table $T = (S, E, Q)$ consists of

- a finite non-empty prefix-closed set of prefixes $S \subseteq tr(A)$
- a finite suffix-closed set of suffixes $E \subseteq I_B^*$ (such that $I_B \subseteq E$, and $\epsilon \notin E$)
- a function $Q : (S, E) \rightarrow O_B$ such that $Q(x, e) = C_L(A^{-1}(xe))$ iff $xe \in tr(A)$ and $Q(x, e) = \perp$ otherwise.

For a set $R \subseteq S$ and $a \in I_B$, let $Succ_T(R, a) := \{xa \mid x \in R \wedge xa \in S\}$, i.e., $Succ_T(R, a)$ is the set of successor rows for elements of R that are in the table.

In the following, we will use the term *row* both for the prefixes and for the entries of a row, when it is clear what is meant from the context.

We call two rows compatible if all columns that are not $\perp$ are the same in both rows.

Definition 5.7 (Compatibility)**.** The rows for two prefixes $x, y \in S$ are compatible iff $\forall e \in E : Q(x, e) = \perp \vee Q(y, e) = \perp \vee Q(x, e) = Q(y, e)$.

We call an observation table consistent if whenever two rows are compatible, their successors are also compatible.

Definition 5.8 (Consistency)**.** An observation table T is consistent iff for all prefixes $x, y \in S$ such that the rows for x and y are compatible, for all $a \in I_B$ all rows in $Succ_T(\{x, y\}, a)$ are compatible.

If there is a suffix $e \in E$ that shows that the successors of x and y under an input a are not compatible, then ae is a suffix that shows that the rows for x and y are also not compatible. Thus, we can add ae to E to resolve this inconsistency.

We define a partition of the set of rows as follows.

Definition 5.9 (Partition)**.** A partition for an observation table T=(S, E, Q) is a partition $P = \{P_1, ..., P_k\}$ of S, such that

- for all $x, y \in P_i$: the rows for x and y are compatible,
- for each P_i and for all $a \in I_B$, there is a P_j such that $Succ_T(P_i, a) \subseteq P_j$.

Note that if $Succ_T(P_i, a) \neq \emptyset$, then there is only one such P_j since all classes of the partition are disjoint.

We will later show how we can use partitions to build candidate machines that are consistent with the observations made so far. The words in the same class of a partition will then lead to the same states in these candidate machines.

We call a partition closed if for each class of the partition and each input symbol a, the observation table contains a successor row (under a) for at least one word of this class, if we know from the observations made so far that such a successor must exist. Our inference algorithm uses closedness as a way to determine which additional rows should be added to the table.

Definition 5.10 (Closedness for Partitions)**.** Let $P = \{P_1, ..., P_k\}$ be a partition for $T = (S, E, Q)$. P is closed if for all $P_i \in P$: if there is some $x \in P_i$ and some sequence $az \in E$ with $a \in I_B$ and $z \in I_B^*$ such that $Q(x, az) \neq \perp$, then there must be some $y \in P_i$ for which $Q(y, az) \neq \perp$, and $ya \in S$.

Given an observation table T, let $\Pi(T, n)$ be the set of all partitions of size n. Let $\Pi_{min}(T)$ be the set of partitions of minimum size for an observation table T, i.e., $\Pi_{min}(T) = \Pi(T, m)$ where $m = min\{n \mid \Pi(T, n) \neq \emptyset\}$.

Definition 5.11 (Closedness)**.** An observation table $T = (S, E, Q)$ is closed if all minimum-size partitions $P \in \Pi_{min}(T)$ are closed.

Definition 5.12 (Partial Closedness)**.** An observation table T is partially closed (p-closed) iff for all prefixes $x \in S$ and all sequences $az \in E$ such that $Q(x, az) \neq \perp$, there is a prefix $y \in S$ such that the rows for x and y are compatible, $Q(y, az) \neq \perp$ and $ya \in S$.

If a table is not p-closed, then no partition can be closed.

Definition 5.13 (Agreement)**.** A Mealy machine M agrees with an observation table $T = (S, E, Q)$ if for all $x \in S$ and $e \in E$, $Q(x, e) = \perp \vee Q(x, e) = M_L(xe)$.

For any closed partition $P = \{P_1, ..., P_k\}$ in $\Pi_{min}(T)$, we can build the following Mealy machine $M_P = (Q, I, O, \delta, q_r)$ with $k + 1$ states:

- $Q := P \cup \{error\}$
- $I := I_B$
- $O := O_B \cup \perp$
- $\delta(P_i, a) := (error, \perp)$ if $Succ_T(P_i, a) = \emptyset$, otherwise: $\delta(P_i, a) := (P_j, b)$ such that for some $x \in P_i$: $Q(x, a) = b \neq \perp$ and $Succ_T(P_i, a) \subseteq P_j$
- $q_r := P_i$ such that $\epsilon \in P_i$

This machine enters a special error state if there is a class of the partition for which the successor class is not defined.

In the following, we will use the notation $\pi_i(t)$ to denote the i-th component of a tuple t, e.g., $\pi_2(q_r, a) = a$.

Lemma 5.1. *Let P be a closed partition of an observation table T=(S, E, Q), and let $M_P = (Q, I, O, \delta, q_r)$ be the Mealy machine constructed as described above. Then for all words $x \in S$, $x \in \pi_1(\delta^*(q_r, x))$.*

Theorem 5.1. *For a closed partition P of an observation table T, the machine M_P agrees with T.*

Definition 5.14. Let $\gamma(M_P)$ be the set of machines with k states that can be obtained from M_P by removing the error state and replacing the transitions to the error state by transitions with arbitrary outputs and successor states.

Theorem 5.2. *Let T be a closed observation table. Then every minimum-size machine M that agrees with T is isomorphic to an element of $\gamma(M_P)$ for some $P \in \Pi_{min}(T)$.*

Theorem 5.3. *If for a closed partition P the error state is not reachable in a composition of A with M_P, then all machines in $\gamma(M_P)$ are right-equivalent.*

If the error state is reachable, we can use an input sequence that leads to the error state to extend the observation table.

Definition 5.15 (Input-Completeness)**.** An observation table $T = (S, E, Q)$ is input-complete if for all minimum-size partitions $P \in \Pi_{min}(T)$, the error state is not reachable in a composition of A with M_P.

Definition 5.16 (Uniqueness)**.** An observation table $T = (S, E, Q)$ is unique if for all pairs of minimum-size partitions $P, P' \in \Pi_{min}(T)$, the machines M_P and $M_{P'}$ are right-equivalent in the context of A.

It follows that all machines of minimum-size size that agree with a consistent, closed, input-complete, and unique observation table are right-equivalent, and they can be obtained from the partitions.

5.4.2 Inference Algorithm

At a high level, our algorithm works as shown in Algorithm 5.1. In each iteration of the main loop, we first make sure that the observation table is consistent and p-closed (by adding additional rows and columns if necessary). Then, we successively determine the partitions of minimum size for the observation table. Whenever we find a partition that is not closed, we add

Algorithm 5.1: Main algorithm

```
   Input: Machine A, OutputQuery OQ, EquivalenceQuery EQ
 1 begin
 2     ObservationTable OT ← empty table
 3     addRow([ϵ])
 4     curSize ← 1
 5     while true do
 6         while ¬consistent ∨ ¬p-closed do
 7             makeConsistent()                                  // consistency
 8             makePClosed()                                     // p-closedness
 9         partitions ← ∅
10         prevMachine ← ⊥
11         while true do
12             partition ← findNextPartition(partitions, curSize)
13             if partition = ⊥ then
14                 if prevMachine = ⊥ then
15                     curSize ← curSize + 1
16                     continue
17                 else
18                     counterexample ← EQ(prevMachine)
19                     if counterexample = ⊥ then
20                         removeErrorState(prevMachine)
21                         return prevMachine
22                     else
23                         handleCounterexample(counterexample)
24                         break
25             if ¬isClosed(partition) then
26                 closePartition()                              // closedness
27                 break
28             machine ← getMachineForPartition(partition)
29             errorPath ← getPathToErrorStateInComposition(A,machine)
30             if errorPath ≠ ⊥ then
31                 handleCounterexample(errorPath)    // input-completeness
32                 break
33             if prevMachine ≠ ⊥ then
34                 distInput ← checkRightEquivalence(A, machine, prevMachine)
35                 if distInput ≠ ⊥ then
36                     handleCounterexample(distInput)           // uniqueness
37                     break
38             partitions ← partitions ∪ {partition}
39             prevMachine ← machine
```

new rows to the table such that the partition becomes closed, and we continue with the next iteration of the main loop. If we find a closed partition, we check whether the error state is reachable in a composition of the corresponding machine with A. If we find a sequence that leads to the error state, this means that the table is not input-complete. Thus, we add this sequence (and its prefixes) to the observation table and continue with the next iteration of the main loop. If we find more than one closed and input-complete partition in the same iteration of the main loop, we check whether the machines for these two partitions are right-equivalent in the context of A. If we find a distinguishing sequence, we extend the observation table accordingly, and continue with the next iteration of the main loop. If finally the table is consistent, closed, input-complete, and unique, we perform an equivalence query for the last machine we found (which is right-equivalent to all machines of minimum size that agree with the table). If the equivalence query is successful, we are done, otherwise, we get a counterexample that we add to the table.

5.5 Implementation

In this section, we describe how our algorithm can be implemented. We also propose some improvements that make the algorithm more usable in practice.

5.5.1 Computing the Partitions

We reduce the problem of finding the partitions for a given size n to a Boolean satisfiability (SAT) problem. A related reduction was used by Heule and Verwer [HV10] for finding DFAs that agree with a set of positive and negative input samples. However, in contrast to our approach, their approach directly computes one arbitrary machine that agrees with the samples. On the other hand, our approach computes a partition which, in general, corresponds to a machine, in which some transitions may be unspecified (if the partition is not closed or input-complete). Thus, we can exploit this additional information to determine which output queries should be performed.

SAT solvers typically require the problem to be in conjunctive normal form (CNF). We first give a high-level description of each subproblem, and then show how to translate it to CNF.

In the following, we will use literals of the form $r_{x,i} \in \mathbb{B}$ to denote that row x is in class i of the partition. We assume that the rows are numbered from 0 to $|S| - 1$.

Covering Condition

All rows of the table must be in at least one class. We therefore add, for all rows x, a clause of the form

$$r_{x,0} \vee r_{x,1} \vee \cdots \vee r_{x,n-1}.$$

Disjointness

No row must be in more than one class. This can be expressed by adding, for all rows x and classes i, the following implication:

$$r_{x,i} \implies \bigwedge_{j>i} \neg r_{x,j}$$

In CNF, this corresponds to the following set of clauses:

$$\bigwedge_{j>i} (\neg r_{x,i} \vee \neg r_{x,j})$$

Compatibility

All rows that are in the same class must be pairwise compatible. For a row x, let $Inc(x)$ be the set of rows that are incompatible to x. To ensure that no incompatible rows are in the same class, we add for each row x and each class i the implication

$$r_{x,i} \implies \bigwedge_{\substack{y \in Inc(x) \\ y>x}} \neg r_{y,i}.$$

In CNF, this corresponds to

$$\bigwedge_{\substack{y \in Inc(x) \\ y>x}} (\neg r_{x,i} \vee \neg r_{y,i}).$$

Common Successors

For all rows that are in the same class i and for which the observation table also contains their successor rows for a given input a, there must be a class j that contains all these successor rows. Thus, we have for each input symbol a and each class i a clause of the form

$$\exists j \colon \forall x \colon (r_{x,i} \implies r_{x',j}),$$

where x' is used to denote the successor row of x under input a. If the observation table does not contain this successor row, the corresponding implication is omitted.

In the above formula, we can represent the existential quantifier as a disjunction, and the universal quantifier as a conjunction. The formula is thus equivalent to

$$\bigvee_{0 \leq j < n} (\bigwedge_{x} (\neg r_{x,i} \vee r_{x',j})).$$

A direct conversion of this formula to CNF would lead to an exponential increase in its size. To obtain a more compact representation, we introduce an auxiliary literal for each input symbol and class of the form $Z_j^{a,i}$. The following formula is then equisatisfiable to the formula above:

$$(\bigvee_{0 \leq j < n} Z_j^{a,i}) \wedge \bigwedge_{0 \leq j < n} \bigwedge_{x} (\neg Z_j^{a,i} \vee \neg r_{x,i} \vee r_{x',j})$$

Finding a Partial Solution

Since we are only interested in sets of classes, the ordering of the classes does not matter. Thus, by assigning some rows to a fixed class, we can significantly reduce the number of symmetrical cases the SAT solver has to consider. We exploit this by precomputing a "partial solution". More specifically, we first compute a set $R = \{x_0, \ldots, x_k\}$ of pairwise incompatible rows. For all solutions of the SAT problem, each of these rows must be in a separate class. Thus, we can obtain an equisatisfiable formula by just assigning all elements of R to arbitrary, different classes. To this end, we replace the covering clauses of the rows in R by the clause

$$r_{x_0,0} \wedge r_{x_1,1} \wedge \cdots \wedge r_{x_k,k}.$$

Furthermore, for a row $x_i \in R$, we can remove all literals $r_{x_i,j}$ such that $j \neq i$ from the SAT formulation and simplify the other constraints accordingly.

Moreover, we can also use the cardinality of R as a lower bound for the required number of classes.

If we represent the compatibility relation on the rows as a graph (such that there is an edge whenever two rows are compatible), the problem of finding such a set R corresponds to finding an independent set in the graph. While the problem of finding a maximum independent set is NP-hard, a heuristic is sufficient for our purposes, since a non-maximal set would just lead to a smaller reduction of symmetries, and to a smaller lower bound, but it would still lead to a correct solution.

We use the following simple sequential greedy heuristic to find a set of pairwise incompatible rows. We first create a list of all rows that is sorted in reverse

order based on the number of incompatible rows for each row. The algorithm maintains a set R of pairwise incompatible rows, where R is initially empty. We then iterate over the sorted list of rows. Whenever we encounter a row that is incompatible to all rows in R, we add this row to R. A similar approach was used by [HV10].

Excluding Previously Discovered Partitions

Since our algorithm searches for multiple partitions for the same observation table, we add for each previously found partition a disjunction of the negated literals for this partition.

5.5.2 Reachability of the Error State

If the error state is reachable with an input a from a state in the composition of the hypothesis machine with the left machine A, this means for no prefix p in the observation table that leads to this state, the input pa is a possible output of the left machine, however, there is another possible output sequence that leads to the same state that has a corresponding successor. We can thus use this sequence as a counterexample.

A straightforward way to check the reachability would be to build the composition, and then to perform a breadth-first search on the composition. A necessary condition for the reachability of the error state in the composition is that the error state is reachable in the hypothesis machine. We have observed that in practice, if the error state is reachable in the hypothesis machine, then in many cases, it is also reachable in the composition. Thus, we use the following approach to find a corresponding sequence quickly: We first determine for each state of the hypothesis machine the distance of the shortest path to the error state. We then use this distance to guide the search in a modified breadth-first search in the composed machine.

5.5.3 Checking if Two Machines are Right-Equivalent

A straightforward way to check whether two hypothesis machines B and B' are equivalent in the context of A would be to compose both with A, and then check the two compositions for equivalence, for example using Hopcroft-Karp's near-linear algorithm. However, this can be computationally expensive when A is large compared to B and B', as it requires building the composition twice.

Therefore, we take the following alternative approach. We build a new machine D that outputs 1 iff the outputs of B and B' differ on (a prefix of) the corresponding input, and 0 otherwise. While the size of D can be quadratic in the size of B, we have observed that, after minimization, in practice the sizes are smaller or comparable to B. To check whether B and B' are right-equivalent we can then just check whether the composition of A with the minimized version of D can output 1, using the search algorithm described in the previous section.

5.5.4 Handling Counterexamples

Like in the original version of Angluin's L^* algorithm, we handle counterexamples by adding all prefixes of the counterexamples as rows to the table. Since, in general, the length of a counterexample can depend on $|C|$, the number of rows that are added (and hence the number of output queries that need to be performed to determine their entries) is not independent of $|A|$.

Rivest and Schapire [RS93] described an improved approach to handle counterexamples that needs to perform only a logarithmic number of membership queries (in the length of the counterexample). However, it is not possible to directly adapt this method to our setting, since it requires that there is always a suffix of the counterexample that is a distinguishing suffix for two compatible rows. It is future work to develop more advanced methods to deal with counterexamples in our setting.

5.6 Evaluation

In this section, we compare two variants of our approach with the Mealy machine version of Angluin's L^* algorithm. We use a set of randomly generated compositional Mealy machines with between 1,000 and 1,000,000 states, and an input and output alphabet of size 4.

The results are shown in Figure 5.3. *GBLearning* ("Gray-box Learning") is an implementation of the approach described in the previous sections. *GBLearning-Simple* is a variant of our approach that does neither check whether the error state is reachable, nor whether different machines that are consistent with the observation table are right-equivalent. Instead, it immediately performs an equivalence query upon finding a closed partition. Thus, the number of equivalence queries of this variant is *not* independent of the size of the right machine.

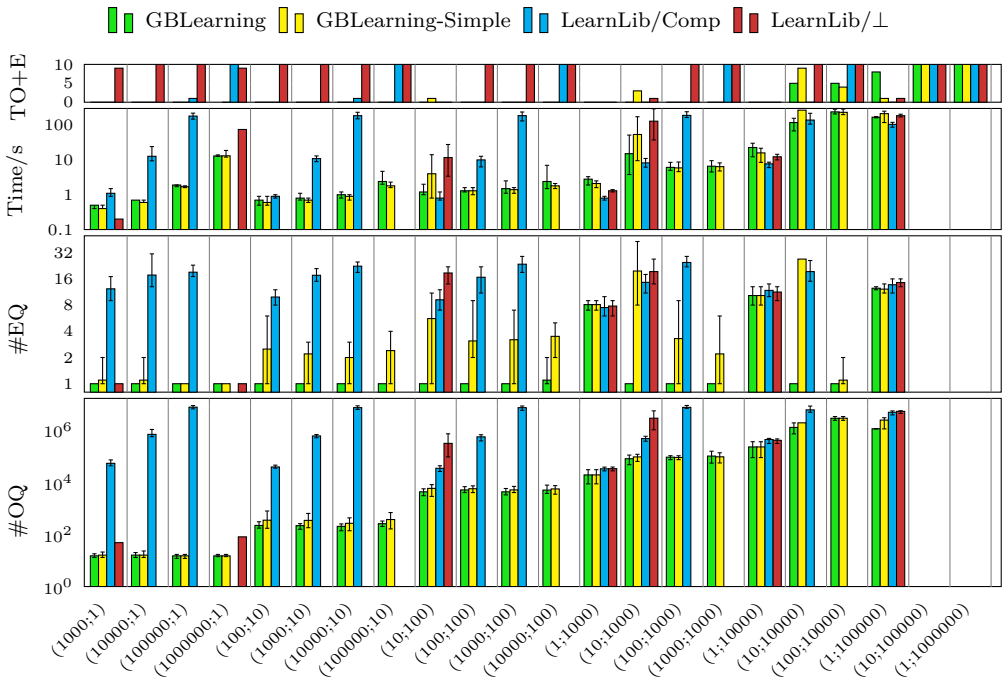

Figure 5.3: Evaluation on randomly-generated machines

We compare these implementations with two variants of Angluin's L^* algorithm, as implemented in LearnLib [IHS15] (*ExtensibleLStarMealy*). *LearnLib/Comp* treats the system as a black box and learns the composition. Furthermore, we modified LearnLib (*LearnLib/*$\perp$) such that it uses L^* on the right machine; impossible inputs are assumed to result in a special output symbol ($\perp$). Equivalence queries are performed by first composing the hypothesis for the right machine with the left machine. Note that this variant does not learn a minimum-size machine; in fact, the learned machine might even be larger than the composition.

The columns of Figure 5.3 show the sizes of the randomly-generated machines in the form $(|Q_A|; |Q_B|)$. The rows show the number of output queries (#OQ), equivalence queries (#EQ), and the execution time in seconds (averages, minima and maxima for the successful runs of 10 different randomly-generated machines of the same size). The row *TO+E* shows on how many of the 10 runs a timeout (5 minutes), or an error occurred. For *LearnLib/Comp* we observed one error, and for *LearnLib/*$\perp$ three errors due to an exception ("incompatible output symbols"). All other entries in this row were timeouts. We used the jar-Release of LearnLib in version *0.9.1-ase2013-tutorial-r1*. Both our tool and LearnLib use a query cache to avoid performing the same output query multiple times.

We observe that *LearnLib/*$\perp$ was only successful when A had 10 or fewer states, or when B had just one state. It performed slightly better than *LearnLib/Comp* in only a few cases where $|Q_A| = 1$ or $|Q_B| = 1$. *LearnLib/Comp* was successful on almost all benchmarks with up to 100,000 states; however, it could not solve any benchmark with more states.

The implementations of our tool could also handle composed machines of larger sizes, in particular when B is relatively small. *GBLearning* was successful on all benchmarks where B had up to 1,000 states, on several where B had 10,000 states, and on two where B had 100,000 states.

For those machines that our implementations and *LearnLib/Comp* could handle, the number of output queries was much smaller for our implementations if $|Q_A| > 1$. In this case, there was no significant difference in the number of output queries between the two variants of our approach. Also, for $|Q_A| > 1$, the number of output queries depends mainly on $|Q_B|$ for both variants.

For *GBLearning*, the number of equivalence queries was mostly 1 or 2 even for relatively large unknown machines; however, randomly-generated machines might not be representative in this regard. *GBLearning-Simple* needed significantly more equivalence queries than *GBLearning* for $|Q_B| > 1$, but significantly fewer than *LearnLib/Comp* for $|Q_A| > 10$.

5.7 Related Work

The concept of actively learning DFAs using membership and equivalence queries was introduced by Angluin in [Ang87]. Angluin developed a polynomial-time learning algorithm, called L^*, for fully-specified DFAs. Rivest and Schapire [RS93] later improved this algorithm and proposed a modification that does not require the system to have a reset state.

Multiple studies [LN12, GL06, GLP06, PO99, HL11] considered scenarios in which the teacher is unable to answer some output queries. In contrast to our setting, where the input language is a known regular language, these approaches assume that no information about the unspecified inputs is available a priori, so that whether a particular input is specified can only be determined by performing an output query. In this scenario, the best bound Leucker and Neider [LN12] could give for the number of required equivalence queries is $n^{\mathcal{O}(n)}$. Hsu and Lee [HL11] claimed that their approach is able learn a minimum-size model for an incompletely specified FSM in polynomial time. However, this approach is incorrect; it does, in general, not find a minimum-size machine [Lee15].

The term "gray-box" has been used in relation with Angluin's algorithm before, but in different contexts. Babic et al. [BBS12] describe an approach to learn an input-output relation for a program. They propose a symbolic version of L^* that is allowed to inspect the internal symbolic state of the program. Henkler et al. [H+10] consider real-time statecharts that have an additional interface for retrieving the current internal state. Elkind et al. introduce grey-box checking [EGPQ06]. A grey-box system consists of completely-specified (white boxes) and unknown components (black boxes). The goal of grey-box checking is then to check whether the system satisfies a property, given, e.g., by an LTL formula. The main problem studied by Elkind et al. is to learn a model of the entire system given the knowledge about the white boxes, which can then be used to model check the property. In contrast to our setting, they consider finite automata that synchronize on common letters in their alphabet, whereas we consider Mealy machines with explicit inputs and outputs. Furthermore, they only use output queries; equivalence queries are realized via a large number of output queries.

5.8 Conclusions and Future Work

We have introduced an algorithm for gray-box learning of serial compositions of Mealy machines. Experimental results confirm that taking into account prior knowledge about a system to be learned often yields significant performance gains.

There are plenty of open problems left for future work: In this chapter, we have considered the serial composition of two Mealy machines. In future work, we would like to extend our approach to arbitrary composition topologies.

While we can precisely bound the number of equivalence queries, we lack such knowledge about the number of output queries. More generally, we would like to better understand the computational complexity of the problem at hand.

In our experimental evaluation, we realized equivalence queries by automata-theoretic constructions, as we had precise knowledge of the system to be learned. In real application scenarios, such knowledge is not available. In those cases, it would be interesting to systematically perform measurements in a way that focuses on the unknown parts.

5.A Appendix: Proofs for Chapter 5

Lemma 5.1. *Let P be a closed partition of an observation table T=(S, E, Q), and let $M_P = (Q, I, O, \delta, q_r)$ be the Mealy machine constructed as described above. Then for all words $x \in S$, $x \in \pi_1(\delta^*(q_r, x))$.*

Proof. By induction on the length of x.

Base Case:
For $|x| = 0$, i.e., $x = \epsilon$, by construction $\epsilon \in q_r = \pi_1(\delta^*(q_r, \epsilon))$.

Induction step:
Let $ya := x$ with $a \in I_B$ and $y \in I_B^*$. Since S is prefix-closed, $y \in S$. We have that $\pi_1(\delta^*(q_r, x)) = \pi_1(\delta(\pi_1(\delta^*(q_r, y)), a))$. Let $P_i := \pi_1(\delta^*(q_r, y))$. By the induction hypothesis, $y \in P_i$. Thus, $ya \in Succ_T(P_i, a)$. Since by construction $Succ_T(P_i, a) \subseteq \pi_1(\delta(P_i, a))$, $x = ya \in \pi_1(\delta(P_i, a)) = \pi_1(\delta^*(q_r, ya)) = \pi_1(\delta^*(q_r, x))$. □

Theorem 5.1. *For a closed partition P of an observation table T, the machine M_P agrees with T.*

Proof. Let P be a closed partition of an observation table $T = (S, E, Q)$, and $M_P = (Q, I, O, \delta, q_r)$ the corresponding Mealy machine. Let $x \in S$, $e \in E$ and $Q(x, e) \neq \perp$. We show by induction on the length of e that $Q(x, e) = M_{P_L}(xe)$.

Base Case:
For $|e| = 1$, we have $M_{P_L}(xe) = \pi_2(\delta^*(q_r, xe)) = \pi_2(\delta(\pi_1(\delta^*(q_r, x)), e))$. Let $P_i := \pi_1(\delta^*(q_r, x))$. By Lemma 5.1, $x \in P_i$. Since $Q(x, e) \neq \perp$ and P is closed, $Succ_T(P_i, e) \neq \emptyset$. Thus, by the definition of M_P we have that $\pi_2(\delta(P_i, e)) = Q(y, e)$ for some $y \in P_i$. Since x and y are in the same class of the partition, they are compatible. Thus, $Q(y, e) = Q(x, e)$.

Induction step:
Let $az := e$ with $a \in I_B$ and $z \in I_B^+$, and let $P_i \in P$ s.t. $x \in P_i$. Since P is closed, $Q(x, e) = Q(x, az) = Q(y, az)$ for some $y \in P_i$, and since $ya \in S$ and E is suffix-closed, $Q(y, az) = Q(ya, z)$. By the induction hypothesis, we have that $A(ya, z) = M_{P_L}(yaz)$. By Lemma 5.1, $x \in \pi_1(\delta^*(q_r, x))$ and $y \in \pi_1(\delta^*(q_r, y))$. Since the classes of P are disjoint and $x, y \in P_i$, we have that $\pi_1(\delta^*(q_r, x)) = P_i = \pi_1(\delta^*(q_r, y))$. So the inputs x and y take the machine M_P to the same state, and thus $M_{P_L}(yaz) = M_{P_L}(xaz) = M_{P_L}(xe)$. □

Theorem 5.2. *Let T be a closed observation table. Then every minimum-size machine M that agrees with T is isomorphic to an element of $\gamma(M_P)$ for some $P \in \Pi_{min}(T)$.*

Proof.
Let $T = (S, E, Q)$ be a closed observation table. Let $M = (Q, I, O, \delta, q_r)$ be a minimum-size machine that agrees with T. We have to show that there is a $P \in \Pi_{min}(T)$ s.t. M is isomorphic to some machine in $\gamma(M_P)$.

Let $M' = (Q', I, O, \delta', q_r)$, with $Q' = Q \cup \{error\}$, be a machine obtained from M by replacing all transitions that are not used by any input from $(S \cup S \cdot E) \cap tr(A)$ with transitions to a new *error*-state with output $\perp$. This machine still agrees with T (but it has one state more than a minimum-size machine).

For each $q \in Q'$, let $P_q \subseteq S$ be the set of words s.t. M' reaches state q when reading a word from P_q, formally: $P_q = \{x \in S \mid \pi_1(\delta'^*(q_r, x)) = q\}$ (note that $P_{error} = \emptyset$).

We now show that $P := \{P_q \mid q \in Q'\} \setminus \{P_{error}\}$ is a closed minimum-size partition for T.

- Since M' is deterministic, all elements of P are disjoint.
- The rows for all words that are in the same class of P are pairwise compatible. We prove this by contradiction. Assume there are $x, y \in P_q$ s.t. x and y are not compatible. This means there is some $e \in E$, s.t. $Q(x, e) \neq \perp$, $Q(y, e) \neq \perp$, and $Q(x, e) \neq Q(y, e)$. But then, by the definition of agreement, $M'_L(xe) \neq M'_L(ye)$. This is a contradiction since x and y both lead to the same state $q \in Q'$ in M'.
- Let $P_q \in P$ and $a \in I_B$. We have to show that there is some $P_j \in P$ s.t. $Succ_T(P_q, a) \subseteq P_j$.

$$\begin{aligned} Succ_T(P_q, a) &= \{xa \in S \mid x \in P_q\} \\ &= \{xa \in S \mid x \in \{y \in S \mid \pi_1(\delta'^*(q_r, y)) = q\}\} \\ &= \{xa \in S \mid \pi_1(\delta'^*(q_r, x)) = q\} \\ &\subseteq \{xa \in S \mid \pi_1(\delta'(\pi_1(\delta'^*(q_r, x)), a)) = \pi_1(\delta'(q, a))\} \\ &= \{xa \in S \mid \pi_1(\delta'^*(q_r, xa)) = \pi_1(\delta'(q, a))\} \\ &\subseteq \{x \in S \mid \pi_1(\delta'^*(q_r, x)) = \pi_1(\delta'(q, a))\} \\ &= P_{\pi_1(\delta'(q,a))} \end{aligned}$$

  If $P_{\pi_1(\delta'(q,a))} \neq P_{error}$, then we can set $P_j := P_{\pi_1(\delta'(q,a))}$. Otherwise, $Succ_T(P_q, a) = \emptyset$, and thus any $P_j \in P$ satisfies $Succ_T(P_q, a) \subseteq P_j$.

- P is a minimum-size partition for T. We show this by contradiction. Assume that there is a minimum-size partition P' that is smaller than P. Since T is closed, P' is closed. The machines in $\gamma(M_{P'})$ agree with T and have size $|P'| < |P| = |M|$. Contradiction.
- Since P has minimum size and T is closed, P is closed.

We now show that the machine $M_P = (Q_P, I, O, \delta_P, q_{P,r})$ is isomorphic to M'. Let $f : Q_P \to Q'$ s.t. $f(P_q) = q$ and $f(error) = error$. f is an isomorphism between M_P and M':

- It is easy to see that f is bijective.
- $f(q_{P,r}) = q_r$, since $\epsilon \in P_{q_r}$.
- Assume that $\delta_P(P_i, a) = (error, \perp)$ for some class P_i and some input a. We have that $Succ(P_i, a) = \emptyset$. Since P is closed, there is no $x \in P_i$ s.t. $Q(x, a) \neq \perp$. Thus, there is no $x \in P_i$ s.t. $xa \in tr(A)$, and hence $\delta'(f(P_i), a) = (error, \perp)$.
- Otherwise, $\delta_P(P_i, a) = (P_j, b)$. Since for some $x \in P_i$, $Q(x, a) = b$, by agreement we have that $\pi_2(\delta'(f(P_i))) = b$. Further, since $Succ(P_i, a) \neq \perp$ there is some $x \in P_i$ s.t. $xa \in S$ and $xa \in P_j$. Thus, $\pi_1(\delta'(f(P_i))) = \pi_1(f(P_j))$.

Since M' differs from M only in the transitions that lead to the error state, M is isomorphic to some machine in $\gamma(M_P)$. □

Theorem 5.3. *If for a closed partition P the error state is not reachable in a composition of A with M_P, then all machines in $\gamma(M_P)$ are right-equivalent.*

Proof. If the error state is not reachable in a composition with A, then there is no $x \in tr(A)$ s.t. M_P takes any of the transitions to the error state when reading x. Thus, modifying any of these transition does not change the behavior of the machine for inputs from $tr(A)$. Since right-equivalence only requires two machines to behave in the same way for inputs from $tr(A)$, all machines in $\gamma(M_P)$ are right-equivalent. □

6

MeMin: SAT-Based Exact Minimization of Incompletely Specified Mealy Machines

In this chapter, we take a fresh look at a well-known NP-complete problem—the exact minimization of incompletely specified Mealy machines.

It turns out that this problem is closely related to the problem addressed in the previous chapter. We develop an approach to minimize incompletely specified Mealy machines by solving a series of Boolean satisfiability (SAT) problems, similar to the SAT reduction described in Section 5.5.1.

We evaluate our implementation on the same set of benchmarks used previously in the literature. On a number of hard benchmarks, our approach outperforms existing exact minimization techniques by several orders of magnitude; it is even competitive with state-of-the-art heuristic approaches on most benchmarks.

The work presented in this chapter has been published in [AR15].

6.1 Introduction

The minimization of Mealy machines is a fundamental problem with applications in many different areas. It is, for instance, an important part of many approaches for logic synthesis, as it can reduce the complexity of the resulting circuits. State reduction also plays an important role in fault-tolerant design of sequential machines [Ahm01]. Another application is in the area of Model-Based Testing (MBT), where an abstract model of a system can be used to

automatically derive test cases. Most approaches in this area require the models to be minimized [AS09]. State minimization also has applications in compiler design and language processing. Recently, it has been used as part of a watermarking technique for copyright protection of IP cores [EMRCS14], as it can reduce the threat of losing states from the signature during an attack.

While the problem of minimizing Mealy machines is efficiently solvable for fully specified machines [Hop71], it is NP-complete for incompletely specified machines, i.e., machines where one or more outputs or next states might not be specified [Pfl73]. Minimization of a machine M in this context means finding a machine M' with the minimal number of states that has the same input/output behavior on all input sequences, on which the behavior of M is defined (but M' might be defined on additional input sequences on which M is not defined). Unlike for fully specified machines, there is no canonical minimal machine.

The problem has been extensively studied before, and a number of exact and heuristic approaches have been proposed. The *standard* or *classic* technique is a two-step approach, that was originally proposed by Paull and Unger [PU59], and improved by Grasselli and Luccio [GL65]. With this approach, in a first step, a number of sets of compatible states (i.e., states for which no input sequence exists, such that their outputs are different) are determined. In the second step, a subset of these sets is selected such that certain closure and covering criteria are fulfilled.

Almost all exact methods and also many heuristic methods follow this two-step approach. The disadvantage of all of these approaches is that, in particular, the enumeration in the first step can be computationally expensive, as there can be an exponential number of compatible classes.

In this chapter, we propose a new approach that is not based on the standard approach, and that does not require the enumeration of a large number of sets of compatible states. Instead, we propose a formulation of the problem as a Boolean satisfiability (SAT) problem, similar to the SAT reduction described in Section 5.5.1. Even though the search space of our method is larger compared to methods based on the standard approach, we show that current SAT solvers are powerful enough to efficiently solve these types of problems.

More specifically, in a precomputation step, we determine a set of pairwise incompatible states that are part of any solution, i.e., we compute some form of a "partial solution". The size of this partial solution also constitutes a lower

bound on the number of states of the minimal machine. We then iteratively call a Boolean satisfiability solver to check whether the partial solution can be extended to a complete solution of the size of the lower bound. If this is not the case, we increase the lower bound by one. This is repeated until a solution is found. This solution is then guaranteed to correspond to a minimal machine that covers the original machine.

We compare our method to several other approaches on two sets of standard benchmarks: the ISM benchmarks, used by, e.g., [KVBSV94, HM96, PO99, GF07, AS13], and the MCNC benchmarks, used by, e.g., [KS91, PG93, RHSJ94, SGL95, HM96, AD01, HXB04, KS13]. These benchmarks come from several sources, e.g., logic synthesis, learning problems, and networks of interacting FSMs.

Our approach outperforms the other exact approaches significantly, in particular on a number of hard benchmarks. In some cases, it is faster than existing approaches by several orders of magnitude.

On most benchmarks, our approach is also competitive with state-of-the-art heuristic methods. There are only two benchmarks on which a heuristic approach is significantly faster. However, in these two cases, this heuristic approach is not able to find the minimal result.

6.1.1 Outline

In Section 6.2, we introduce basic definitions used throughout the chapter, and we formally define the problem addressed in this chapter. Section 6.3 introduces the most important related work. In Section 6.4, we describe our new approach in detail, and in Section 6.5, we describe details of our implementation. In Section 6.6, we evaluate our approach on a set of standard benchmarks. Finally, Section 6.7 concludes.

6.2 Definitions

In this section, we formally define several concepts used throughout this chapter.

6.2.1 Basic Definitions

We will use the following definition for *Mealy machines*, which is more general than the definition we used in the previous chapter, where we only considered completely specified *Mealy machines*.

Definition 6.1. A Mealy machine is a tuple $(I, O, Q, q_r, \delta, \lambda)$, where

- $I \neq \emptyset$ is a finite set of input symbols
- $O \neq \emptyset$ is a finite set of output symbols
- $Q \neq \emptyset$ is a finite set of states
- $q_r \in Q \cup \{\bot\}$ is the initial (reset) state
- $\delta : (Q, I) \to Q \cup \{\phi\}$ is the transition function
- $\lambda : (Q, I) \to O \cup \{\epsilon\}$ is the output function.

ϕ denotes an unspecified state, ϵ denotes an unspecified output, and $\bot$ denotes an unspecified initial state.

Regarding the initial state, previous definitions in the literature are not consistent. Some approaches assume that there is always a designated initial state, while others assume that any state can be the initial state. With respect to minimization this can make a difference if states are not reachable from the initial state. Our definition is a generalization of the previous definitions.

A machine M is called *completely specified* if for all states, all next states and outputs are defined, i.e., for all $q \in Q$ and $a \in I$, $\delta(q, a) \neq \phi$ and $\lambda(q, a) \neq \epsilon$. M is called *incompletely specified* if one or more next states or outputs may be unspecified.

With this definition, a completely specified machine is a special case of an incompletely specified machine. Previous approaches often defined an incompletely specified machine as a machine where at least one transition is unspecified. However, since our approach can also be used to minimize completely specified machines, we choose this definition.

In the following, we will without loss of generality only consider machines M that have no transition for which only the output is specified, i.e.,

$$\forall q \in Q, a \in I : (\delta(q, a) = \phi) \implies (\lambda(q, a) = \epsilon).$$

Note that any machine M can be transformed to such a machine M' by adding an additional target state with no outgoing transitions.

We extend δ and λ to sequences in the usual way (where $\delta(\phi, a) = \phi$ and $\lambda(\phi, a) = \epsilon$).

A state q is called a *predecessor state* of a state q' under input a if $\delta(q, a) = q'$.

An input sequence $a_0 \dots a_n \in I^*$ is called *applicable for a state* q if there is a sequence of states $q_0 \dots q_n$ with $q_0 = q$ s.t. $\delta(q_i, a_i) = q_{i+1}$ and $q_{i+1} \neq \phi$ for

all $0 \leq i < n$. An input sequence is called *applicable for a machine* M if it is applicable for its initial state. If the machine has no initial state ($q_r = \perp$), the sequence is applicable if it is applicable for at least one of the states of the machine.

Two *outputs* $o_1, o_2 \in O \cup \{\epsilon\}$ are *compatible* iff $o_1 = \epsilon$, or $o_2 = \epsilon$, or $o_1 = o_2$. Two output sequences $o = o_0 \dots o_n$ and $o' = o'_0 \dots o'_n$ are compatible if o_i and o'_i are compatible for all $0 \leq i \leq n$. An output sequence $o = o_0 \dots o_n$ *subsumes* a sequence $o' = o'_0 \dots o'_n$ if for all $0 \leq i \leq n$, $o'_i = \epsilon \vee o_i = o'_i$.

Two *states* are *compatible* if for all applicable input sequences for both states, the corresponding output sequences are compatible. Two states are *distinguishable* if they are not compatible. In this case, there is a distinguishing input sequence that is applicable for both states such that the corresponding output sequences differ.

A Mealy machine $M' = (I, O, Q', q'_r, \delta', \lambda')$ *covers* a Mealy machine $M = (I, O, Q, q_r, \delta, \lambda)$, iff for all applicable input sequences $s \in I^*$ for M, $\lambda'(q'_r, s)$ subsumes $\lambda(q_r, s)$. If the machine has no reset state, we require that for all states $q \in Q$ there is a state $q' \in Q'$ such that for all applicable input sequences $s \in I^*$ for M, $\lambda'(q', s)$ subsumes $\lambda(q, s)$.

We are now ready to formally define the problem addressed in this chapter.

6.2.2 Problem Statement

Given a Mealy Machine M, our goal is to find a Mealy machine M' with the minimum number of states, such that M' covers M.

6.2.3 General Approach

For a Mealy machine $M = (I, O, Q, q_r, \delta, \lambda)$, a *compatibility class* (also called a *compatible*) is a set $C \subseteq Q$, such that all elements of C are pairwise compatible.

For a compatibility class $C = \{q_0, \dots, q_n\}$ and an input a, we define a *successor* function

$$Succ(C, a) = \bigcup_{0 \leq j \leq n} \{\delta(q_j, a) \mid \delta(q_j, a) \neq \phi\}.$$

In previous work, the set $Succ(C, a)$ was often called the *implied set* of C under input a.

A set $S = \{C_1, \dots, C_n\}$ of compatibility classes is *closed* if for all $C_j \in S$ and all inputs $a \in I$, there exists a $C_k \in S$ such that $Succ(C_j) \subseteq C_k$.

A closed set of compatibility classes *covers* a Mealy machine M if every state of the machine is contained in at least one class of the set.

The following theorem is based on a theorem that was first proposed and proven by Paull and Unger [PU59].

Theorem 6.1. *From a closed set $S = \{C_1, \ldots, C_n\}$ of compatibility classes with the minimum number of classes that covers a Mealy machine $M = (I, O, Q, q_r, \delta, \lambda)$, one can derive a Mealy machine $M' = (I, O, Q', q'_0, \delta', \lambda')$ with the minimal number of states that covers M as follows:*

- $Q' := S$
- $q'_r := C_j$ *for some* j *s.t.* $q_r \in C_j$ *if* $q_r \neq \perp$, *and* $q'_r := \perp$ *otherwise*
- $\delta'(C_j, a) = \phi$ *if* $Succ(C_j, a) = \emptyset$, *and otherwise* $\delta'(C_j, a) = C_k$ *for some* k *s.t.* $Succ(C_j, a) \subseteq C_k$
- $\lambda'(C_j, a) = o$ *if* $\lambda(q, a) = o$ *for some* $q \in C_j$ *and* $\lambda'(C_j, a) = \epsilon$ *otherwise*

It is important to note that the elements of S can overlap, i.e., S is not necessarily a partition of Q.

6.3 Related Work

The concept of Mealy machines was introduced by G. H. Mealy in 1955 [Mea55]. While it was initially believed that incompletely specified machines could be minimized by similar techniques as completely specified machines [Auf58], Ginsburg [Gin59] discovered that there are examples for which this is not possible. Pfleeger [Pfl73] proved in 1973 that the problem of minimizing incompletely specified machines is NP-complete.

Paull and Unger [PU59] proposed an approach for exact state minimization based on their general theory that was introduced in the previous section. The approach consists of two steps:

1. Enumeration of all compatibility classes.
2. Solution of a covering problem, i.e., choosing a set of compatibility classes (from step 1) of minimum size that satisfies the closure and covering conditions.

Grasselli and Luccio [GL65] discovered that only some compatibility classes (prime compatibles) need to be considered. Prime compatibles are those compatibles that are not included in any larger compatible with the same or

fewer closure constraints. They proved that for any Mealy machine, there is at least one covering of minimal size that consists only of prime compatibles.

This approach is often called the "standard" [PO99] or "classic" [AS13] approach, and has also been described in textbooks [KVBSV10]. Almost all subsequently proposed exact minimization techniques are based on various improvements of this standard method. Rho et al. [RHSJ94] presented a tool called STAMINA which implements Grasselli and Lucio's method, as well as an extension of their method based on compatible pairs that was first proposed by Sarkar et al. [DSBC69]. Kam et al. developed an approach that represents prime compatibles implicitly as Binary Decision Diagrams (BDDs) and is, thus, able to handle significantly larger sets of prime compatibles. Several approaches proposed improvements of either the identification of compatibility classes [Ben71, HM95, Ahm01], or for solving the corresponding covering problem [PG93, LD97, VKBSV97].

The downside of all of these approaches is that the generation of all compatibility classes can be computationally very expensive, as the number of compatibility classes can be exponential in the number of states [Rub75]. This is also an issue for the implicit approaches, as not all sets of compatibility classes can be efficiently represented as BDDs.

Pena and Oliveira [PO99] presented the tool BICA, which implements a method that is not based on the classic approach. Instead, it is based on a modification of Angluin's algorithm [Ang87] for learning Mealy machines. They generate a sequence of tree-FSMs (i.e., an FSM for which the corresponding graph is a tree with the initial state as the root). Each TFSM is reduced to a minimal consistent Mealy machine using a variant of Bierman's search algorithm [BF72] until a solution that covers the original machine is found. Grinchtein and Leucker [GL06] later modified BICA by replacing Bierman's algorithm with a SAT-based approach. This lead to significantly better results on two benchmarks, for which the TFSM reduction time was the dominating factor before.

In addition to the exact approaches, a number of heuristic methods have been proposed. Some of these are also based on the classic two-step approach. Rho et al. [RHSJ94] described a method that uses a restricted subset of the prime compatibles, but solves the covering step exactly. On the other hand, Ahmad and Das [AD01] proposed a technique that requires the enumeration of all prime compatibles, but uses a heuristic for the second step. Several approaches use both a subset of the prime compatibles (typically the maximal compatibles), and heuristics for the covering step [BWL72, KS91, SGL95, HM96, HXB04].

A few heuristic techniques are not based on the classic approach. Avedillo et al. [AQH90] described a method that reduces the number of states of a machine through a sequence of transformations on a symbolic description of the machine. Gören and Ferguson [GF07] presented an approach that is based on a checking sequence generation technique. Klimowicz and Solov'ev [KS13] proposed a method based on merging pairs of compatible states. Alberto and Simao [AS13] described a technique that selects compatible states using maximum cliques on the distinction graph.

6.4 Approach

Unlike almost all previous exact minimization techniques, our technique does not require the enumeration of compatibility classes. Instead, we first compute a partial solution that is part of any solution. The size of this partial solution also corresponds to a lower bound on the number of states of the minimized machine. Then, we use a SAT solver to determine if there is a machine that covers the original machine with the size of the lower bound. If the SAT solver cannot find a satisfying assignment, we increase the lower bound by one.

So, at a high level, our algorithm works as follows.

Algorithm 6.1: Main algorithm

```
   Input: M = (I, O, Q, q_r, δ, λ)
1  begin
2      m ← computeIncompatibilityMatrix(M)
3      P ← computePartialSolution(m)
4      lowerBound ← |P|
5      for nClasses ← lowerBound to |Q| do
6          clauses ← createSATProblem(nClasses, M, m, P)
7          (satisfiable, model) ← runSATSolver(clauses)
8          if satisfiable then
9              return buildMachine(model, M)
```

In the following subsections, we describe our approach in more detail. We assume that we want to minimize a machine with $|Q|$ states, and that the states are numbered from 0 to $|Q| - 1$.

6.4.1 Incompatibility Matrix

The first step of our algorithm is to determine which pairs of states are compatible, and which pairs are incompatible. Similar to previous approaches, we store this information in a matrix m, such that $m[i][j] = 1$ if states i and j are incompatible, and $m[i][j] = 0$ if they are compatible.

However, to compute this matrix, we use a slightly different approach from the classic approach [GL65]. Our algorithm works as follows:

- Initialize all entries of the matrix with 0.
- Iterate over all state pairs (i, j). If $m[i][j]$ is not already set to 1, check whether there is an input symbol a such that the outputs of i and j differ. If this is the case, set this entry of the matrix to 1.
- Whenever an entry (i, j) of the matrix changes its value from 0 to 1, we set the value of all predecessor state pairs under the same inputs to 1 as well.

The last step is executed at most $|Q|^2$ times. To determine the predecessor state pairs efficiently, we can compute a list of predecessor state pairs for each state pair by iterating over all state pairs and inputs once. Thus, these $|Q|^2$ lists have at most $|Q|^2 \cdot |I|$ elements in total. Since the last step iterates over each of the lists at most once, the overall complexity of our algorithm is in $O(|Q|^2 \cdot |I|)$.

The classic algorithm does not perform the update upon a change. Instead, it repeatedly iterates over all state pairs (i, j) and sets $m[i][j]$ to 1 if one of their successor pairs is set to 1. The algorithm terminates upon reaching a fixed point. Thus, its complexity is in $O(|Q|^3 \cdot |I|)$.

6.4.2 Encoding as a SAT Problem

In this section, we describe, how we encode the problem *"Is there a closed set of compatibility classes of size n that covers the machine?"* as a Boolean satisfiability problem; this is similar to the encoding described in Section 5.5.1. It is straightforward to show that if the problem is unsatisfiable for n classes, but satisfiable for $n+1$ classes, a satisfying assignment for $n+1$ classes fulfills the conditions of Theorem 6.1.

SAT solvers typically require the problem to be in conjunctive normal form (CNF). In this section, we will provide high level-descriptions for each sub-problem; the translations to CNF are analogous to the translations described in Section 5.5.1.

In the following, we will use literals of the form $s_{x,i} \in \mathbb{B}$ to denote that state x is in compatibility class i.

Covering Condition

All states of the original machine must be in at least one compatibility class. We therefore add, for all states x, a clause of the form

$$s_{x,0} \vee s_{x,1} \vee \cdots \vee s_{x,n-1}.$$

Compatibility

All states that are in the same class must be pairwise compatible. For a state x, let $Inc(x) \subseteq Q$ be the set of states that are incompatible to x. To ensure that no incompatible states are in the same class, we add for each state x and each class i the implication

$$s_{x,i} \implies \bigwedge_{\substack{y \in Inc(x) \\ y > x}} \neg s_{y,i}.$$

Incompatibility is symmetric. However, it suffices to have one constraint for each pair of states. This is ensured by $y > x$.

Closure

For all states that are in the same class, there must be another class that contains all of their successor states. Thus, we have for each input symbol $a \in I$ and each class i a clause of the form

$$\exists j \colon \forall x \colon (s_{x,i} \implies s_{x',j}),$$

where x' is used to denote the successor state of x under input a. If the successor state is undefined, the corresponding implication is omitted.

Note that to fulfill the closure property, it is not sufficient to require that for all pairs of states that are in the same class, their successor pairs must be in the same class. This is because the classes are not disjoint. So it is possible that for three states s_1, s_2, s_3, there is a class i that contains the successors of s_1 and s_2, a class j that contains the successors of s_2 and s_3, and a class k that contains the successors of s_1 and s_3, but there might not be a single class that contains the successors of all three states.

6.4.3 Computing a Partial Solution

In this section, we present an approach to reduce the number of symmetrical cases the SAT solver has to consider by precomputing a "partial solution". More specifically, we first compute a set $S = \{x_0, \ldots, x_k\}$ of pairwise incompatible states, using the heuristic technique proposed in Section 5.5.1. For all solutions of the SAT problem, each of these states must be in at least one class, but no pair of these states may be in the same class. However, since we are only interested in sets of compatibility classes, the ordering of the classes does not matter. Thus, we can obtain an equisatisfiable formula by just assigning all elements of S to arbitrary, different classes. To this end, we add a clause of the form

$$s_{x_0,0} \wedge s_{x_1,1} \wedge \cdots \wedge s_{x_k,k}.$$

Moreover, we can use the cardinality of S as a lower bound for the required number of classes.

6.5 Implementation

We have implemented our approach in a tool called MeMin. MeMin is available for download from our website[1]. The tool is written in C++, and it uses MiniSat [ES04] as SAT solver. MiniSat is an open-source SAT solver that can be used as a library and that provides a simple API.

Like many previous tools, MeMin accepts inputs in the widely used KISS2 input format [SSL+92]. In this format, Mealy machines are described by a set of 4-tuples of the form (*input*, *currentState*, *nextState*, *output*). *input* and *output* are sequences of $\{0, 1, -\}$, where $-$ means that the corresponding bit is not specified. In the following paragraphs, we describe how we deal with some of the implications of using this specification format.

6.5.1 Dealing with Partially Specified Outputs

The KISS2 format allows for partially specified outputs, i.e., outputs in which a subset of the bits are undefined. A machine M covers such a partially specified machine M' iff the set of possible outputs of M is a (non-empty) subset of the set of possible outputs of M'.

We incorporate this generalization into our framework in a similar way as the authors of [PO99]. We define two outputs to be compatible if all of their

[1] `http://embedded.cs.uni-saarland.de/MeMin.php`

bits at the same position are either the same, or at least one of the bits is undefined. We then use this definition of the compatibility of outputs in the definition of the compatibility of states.

6.5.2 Dealing with Partially Specified Inputs

Partially specified inputs are used as a shorthand for sets of inputs. One straightforward way to deal with this would be to just add transitions for all concrete inputs that are described by a partially specified input. However, this approach is only viable when the number of unspecified bits is small.

Instead, we use the following approach. We first partition the set of states into equivalence classes such that two states are in the same class if they are transitively compatible. For each equivalence classes C we then compute a set of disjoint partially specified inputs D such that all intersections of inputs from C can be expressed by a combination of inputs from D. Finally, we replace all transitions of states in C with transitions that have the corresponding inputs from D.

Furthermore, for the closure constraints in Section 6.4.2, if there are multiple inputs that have exactly the same output/next state behavior, we only need to consider one of these inputs.

6.5.3 Undefined Reset States

The KISS2 format allows for the optional definition of a reset state. According to the specification [SSL+92], if no reset state is specified, the first state encountered in the transition list is implicitly assumed to be the reset state. However, several of the benchmarks we use in the evaluation would not make much sense with this specification, as for example in *rubin2250*, only three (out of 2250) states are reachable from the first state. We therefore added a command line parameter to our tool that controls whether the first state should be the reset state, or any state might be a reset state if no explicit reset state is specified.

6.6 Evaluation

In this section, we first evaluate our approach on two sets of standard benchmarks against two other exact and two heuristic techniques. We then investigate how the precomputation of a partial solution, as described in Section 6.4.3, influences the execution time.

6.6.1 Benchmarks

We compare the performance of our implementation with BICA [PO99], which is based on Angluin's learning algorithm, and STAMINA (exact mode) [RHSJ94], which is a popular implementation of the explicit version of the two-step standard approach. Furthermore, we also compare our tool with STAMINA (heuristic mode), and COSME [AS13], which is another, recently proposed, heuristic technique.

There are two standard sets of benchmarks that were used in the evaluations of previous techniques: The ISM benchmarks, used by, e.g., [KVBSV94, HM96, PO99, GF07, AS13], and the MCNC benchmarks, used by, e.g., [KS91, PG93, RHSJ94, SGL95, HM96, AD01, HXB04, KS13].

The ISM benchmarks contain several examples that exhibit a very large number of prime compatibles, and are therefore not solvable by techniques that are based on the explicit enumeration of prime compatibles, such as STAMINA. The machines after minimization are, however, rather small (at most 14 states), which makes them amenable to the learning-based approach.

Most benchmarks from the MCNC suite, on the other hand, can be easily solved by the standard approach. However, some of the minimized machines are significantly larger (up to 135 states), which makes these benchmarks harder for the learning-based technique, as well as for the technique based on implicit enumeration, where "the representation becomes inefficient", "when there are many states and few compatibles" [KVBSV94].

The scatter plots in Figure 6.1 and 6.2 show the results of the different tools on both sets of benchmarks. MeMin reported a correct result on all benchmarks. Cases where the other tool did not return a correct result are indicated with orange and red (for the exact approaches, a non-minimal result is considered to be incorrect). The reported runtimes are the averages over five runs; the timeout was set to five minutes. The standard deviations of the runtimes were in all cases smaller than 4.5%. Detailed tables with all results can be found in Appendix 6.A.

ISM Benchmarks

This set of benchmarks was compiled by the authors [KVBSV94] of the ISM tool. The benchmarks come from a variety of sources, including asynchronous synthesis procedures, FSMs constructed to be compatible with a given collection of examples of input/output behavior, FSMs that are part of a surrounding network of FSMs, FSMs contructed to have an exponential number (up to 2^{1500}) of prime compatibles, and randomly generated machines.

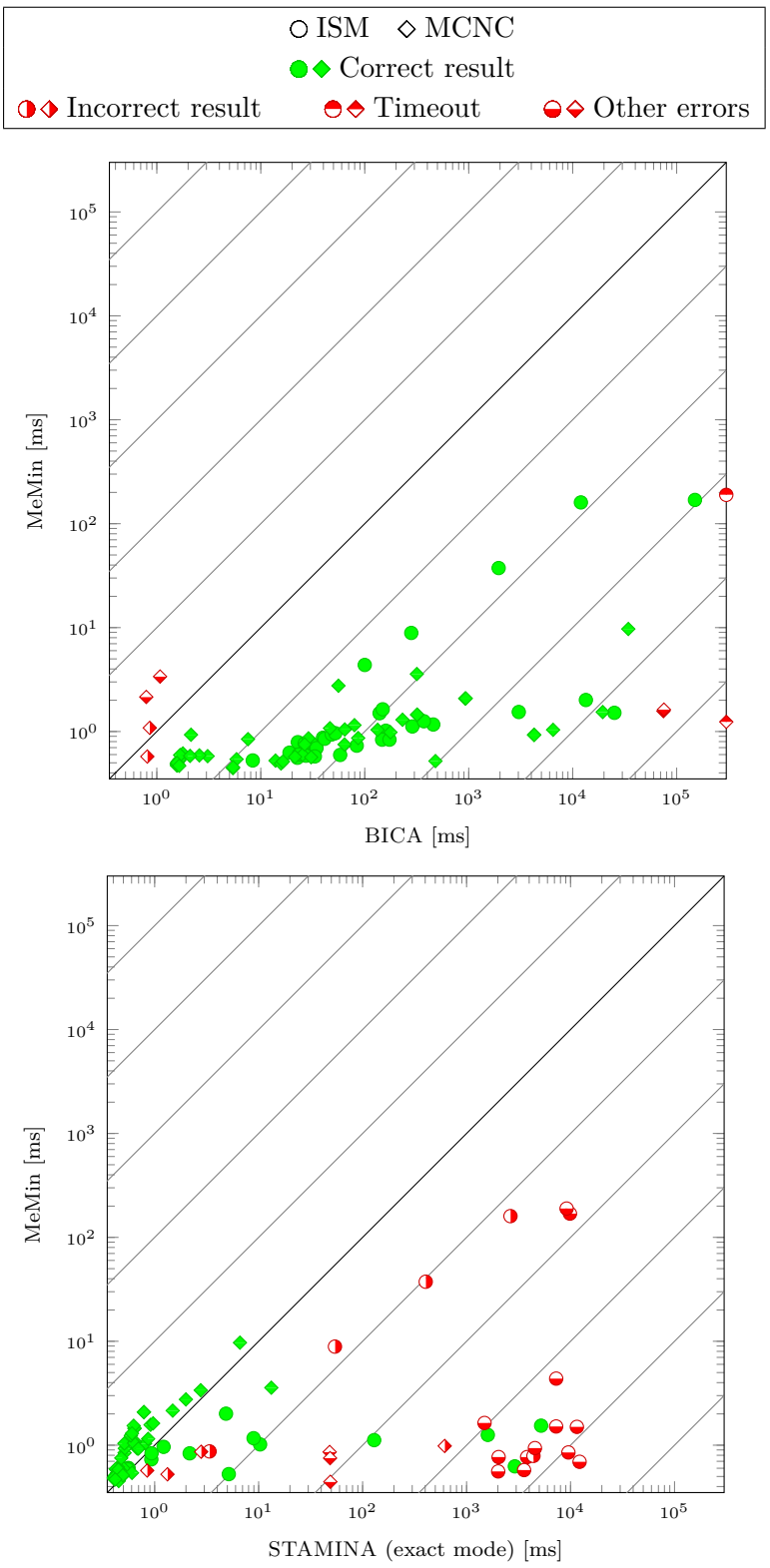

Figure 6.1: Benchmark results — exact approaches, © 2015 IEEE

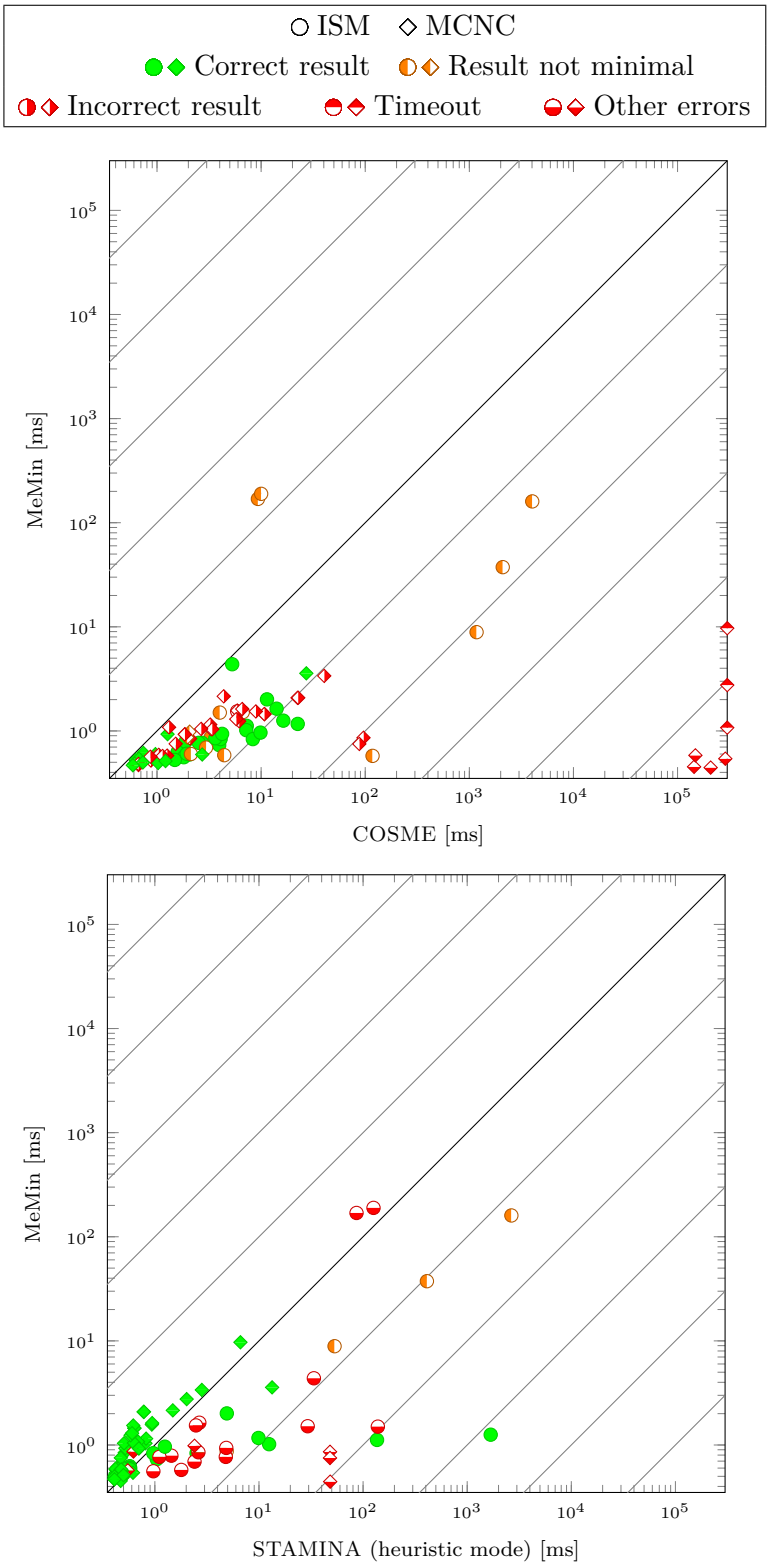

Figure 6.2: Benchmark results — heuristic approaches, © 2015 IEEE

MeMin was able to solve all 34 benchmarks in under 0.2 s, and 30 of them even in less than 10 ms. BICA was on all benchmarks at least 16 times slower; it could solve only one benchmark in less than 10 ms. 16 benchmarks took more than 100 ms, seven more than 1 s, and one could not finish within a timeout of five minutes.

STAMINA (exact mode) was not able to solve 13 benchmarks on our machine. This mostly corresponds to what was also reported in previous publications, except for ifsm1, for which [PO99] reported a time about twice as high as for *ifsm2*, and *fo.20* and *th.30*, for which [AS13] reported around 36 s and 84 s, respectively, though it is not clear whether they used the exact or the heuristic mode of STAMINA. Furthermore, for five benchmarks, the results of STAMINA (exact mode) were not minimal.

The execution time of the heuristic mode of STAMINA was in only one case significantly different from the exact mode. 15 benchmarks lead to error messages on our machine. [HM96] reported results for the heuristic mode of STAMINA for these cases. For the *fo.** and *th.** benchmarks, the resulting machines were in many cases significantly larger than the minimal machines (e.g., 24 states instead of 8 for *th.55*, or 14 instead of 7 for *fo.70*).

COSME was in two cases (*fo.60* and *fo.70*) significantly faster than MeMin, however, the minimized machines had three and two additional states, respectively. In seven cases, COSME was more than ten times slower than MeMin, and in 12 cases, the resulting machines did not have the minimal number of states.

MCNC Benchmarks

Benchmarks from the MCNC suite [Yan91] are widely used in logic synthesis. Both MeMin and STAMINA were able to solve almost all benchmarks in less than 10ms. However, STAMINA was unable to solve three cases and reported non-minimal results in four cases.

BICA, on the other hand, needed more than 100 ms in 17 cases, and more than 1 s in eight cases. On one benchmark, BICA did not terminate within a timeout of five minutes. On two benchmarks, it ran out of memory.

COSME reported invalid results on a large number of these benchmarks. We assume that this might be due to a bug in handling partially specified inputs, which are used by many MCNC benchmarks. We noticed that the resulting machines sometimes had multiple transitions for the same inputs.

6.6.2 Evaluation of MeMin

In this section, we analyze our approach in detail. Tables 6.1 and 6.2 show the results for several experiments.

The column "SAT only" shows the execution time for a naive implementation that does not precompute a partial solution and uses 1 as the lower bound. The column "SAT+LB" gives the time for an implementation that uses the size of the partial solution as the lower bound, but does not add the constraints from Section 6.4.3 to the SAT problem. The column "SAT+LB+PS" shows the time for the implementation that also adds these constraints (i.e., the same implementation that was also used for the scatter plots). For this implementation, the column "SAT share" displays the time the SAT solver needed in relation to the total execution time. The column $|Q|$ shows the number of states before minimization, $|Q_m|$ the number of states after minimization, and $|P|$ the size of the partial solution.

All benchmarks for which the minimized machines have more than nine states could not be solved by the naive implementation within a timeout of five minutes. On the other hand, the solution that uses the lower bound was able to solve all but one benchmark in under five minutes (and all but five in under one second). We observed that the SAT solver typically needs much more time to determine that there is no solution for $|Q_m| - 1$ than to find a solution of size $|Q_m|$. For many benchmarks, the size of the partial solution already corresponds to the size of the minimal solution; thus the second implementation often does not need to consider the $|Q_m| - 1$ case.

The additional constraints used in the third implementation were in particular helpful for larger machines (e.g., *s298* and *scf*), as well as those machines, where $|P| < |Q_m|$. The most noticeable speed-up was on *fo.60* and *fo.70* (more than 600 times faster), and on *ifsm1*, for which the first and second implementation did not terminate within five minutes, while the final implementation was able to solve the benchmark in 1.6 ms.

6.6.3 Other Tools

A number of other state minimization tools have been described in the literature. Unfortunately, for the approaches described in [KVBSV94, HM96, AD01, HXB04, KS13], we were unable to obtain working implementations.

For the ISM tool [KVBSV94], which implements the implicit enumeration approach based on BDDs, there is no public release available [Vil15]. However, the authors of BICA compared their tool with ISM on the same set of

Table 6.1: Evaluation of MeMin (ISM benchmarks), © 2015 IEEE

					Solution times [ms]		
	Benchmark	$\|Q\|$	$\|Q_m\|$	$\|P\|$	SAT only	SAT+LB	SAT+LB+PS	SAT share
ISM benchmarks	alex1	42	6	6	5.9	2.9	1.1	35%
	intel_edge.dummy	28	4	3	1.7	1.6	.8	36%
	isend	40	4	4	2.1	1.6	1.0	35%
	pe-rcv-ifc.fc	46	2	2	1.4	1.3	.8	19%
	pe-rcv-ifc.fc.m	27	2	2	1.3	1.2	.7	17%
	pe-send-ifc.fc	70	2	2	1.9	1.7	1.1	23%
	pe-send-ifc.fc.m	26	2	2	1.5	1.3	.8	16%
	vbe4a	58	3	3	2.2	1.8	1.2	32%
	vmebus.master.m	32	2	2	2.7	2.5	2.0	14%
	fo.16	17	3	2	1.1	1.1	.6	30%
	fo.20	21	3	3	1.1	1.0	.5	20%
	fo.30	31	3	2	1.2	1.2	.7	38%
	fo.40	41	4	4	3.2	2.9	1.5	66%
	fo.50	51	6	5	16.0	15.1	4.3	87%
	fo.60	61	7	6	116423.5	117156.3	169.2	99%
	fo.70	71	7	4	116371.3	114812.6	189.4	99%
	th.20	21	4	3	1.4	1.3	.7	42%
	th.25	26	4	3	1.3	1.2	.7	40%
	th.30	31	5	5	2.2	1.7	.6	31%
	th.35	36	7	7	8.1	2.0	.8	42%
	th.40	41	8	8	40.1	2.2	.9	45%
	th.55	55	8	8	94.1	8.5	1.5	63%
	ifsm0	38	3	3	2.2	1.9	.9	9%
	ifsm1	74	14	13	timeout	timeout	1.6	54%
	ifsm2	48	9	9	1616.7	7.2	1.5	35%
	rubin1200	1200	3	3	39.6	37.8	37.4	13%
	rubin18	18	3	3	1.0	1.0	.5	17%
	rubin2250	2250	3	3	163.9	159.3	160.1	8%
	rubin600	600	3	3	10.1	9.3	8.8	20%
	e271	19	2	2	1.1	1.1	.5	15%
	e285	19	2	2	1.1	1.0	.6	15%
	e304	19	2	2	1.3	1.0	.6	17%
	e423	19	2	2	1.1	1.0	.5	14%
	e680	19	2	2	1.1	1.0	.5	15%

Table 6.2: Evaluation of MeMin (MCNC benchmarks), © 2015 IEEE

					Solution times [ms]		
	Benchmark	$\|Q\|$	$\|Q_m\|$	$\|P\|$	SAT only	SAT+LB	SAT+LB+PS	SAT share
MCNC benchmarks	bbara	10	7	7	7.8	1.4	.5	9%
	bbsse	16	13	13	timeout	8.1	.9	14%
	bbtas	6	6	6	2.2	1.2	.5	8%
	beecount	7	4	4	1.3	1.2	.5	11%
	cse	16	16	16	timeout	4.2	.8	13%
	dk14	7	7	7	10.1	1.6	.6	9%
	dk15	4	4	4	1.3	1.1	.5	8%
	dk16	27	27	27	timeout	32.7	.9	16%
	dk17	8	8	8	30.9	1.6	.5	9%
	dk27	7	7	7	5.8	1.3	.4	9%
	dk512	15	15	15	timeout	3.4	.5	12%
	donfile	24	1	1	1.0	1.0	.5	8%
	ex1	20	18	18	timeout	9.3	1.0	12%
	ex2	19	5	4	3.8	3.5	.9	48%
	ex3	10	4	2	1.6	1.6	.8	47%
	ex4	14	14	14	timeout	3.0	.5	13%
	ex5	9	3	2	1.1	1.0	.5	22%
	ex6	8	8	8	39.3	1.6	.5	11%
	ex7	10	3	3	1.1	1.0	.5	16%
	keyb	19	19	19	timeout	6.9	1.1	13%
	kirkman	16	16	16	timeout	5.9	2.1	9%
	lion9	9	4	4	1.2	1.0	.4	11%
	lion	4	4	4	1.2	1.0	.4	8%
	mark1	15	12	12	timeout	9.5	1.0	35%
	mc	4	4	4	1.2	1.0	.4	8%
	modulo12	12	1	1	1.0	.9	.4	8%
	opus	10	9	9	265.3	1.7	.5	10%
	planet1	48	48	48	timeout	78.7	1.4	26%
	planet	48	48	48	timeout	78.3	1.4	26%
	pma	24	24	24	timeout	8.1	.8	17%
	s1488	48	48	48	timeout	137.7	2.0	22%
	s1494	48	48	48	timeout	134.0	2.0	23%
	s1a	20	1	1	3.3	3.2	2.7	8%
	s1	20	20	20	timeout	8.3	1.0	15%
	s208	18	18	18	timeout	7.6	1.0	13%
	s27	6	5	5	1.7	1.2	.5	9%
	s298	218	135	135	timeout	5273.0	9.7	36%
	s386	13	13	13	timeout	3.0	.7	13%
	s420	18	18	18	timeout	8.7	1.0	13%
	s510	47	47	47	timeout	89.6	1.2	30%
	s820	25	24	24	timeout	16.6	1.5	15%
	s832	25	24	24	timeout	18.9	1.6	15%
	s8	5	1	1	.9	.9	.4	7%
	sand	32	32	32	timeout	32.9	1.5	18%
	scf	121	97	97	timeout	1492.3	3.3	45%
	shiftreg	8	8	8	37.6	1.7	.4	9%
	sse	16	13	13	timeout	8.2	.9	14%
	styr	30	30	30	timeout	29.4	1.2	17%
	tav	4	4	4	1.2	1.1	.5	7%
	tbk	32	16	16	timeout	35.5	3.5	6%
	tma	20	18	18	timeout	4.7	.7	20%
	train11	11	4	4	1.2	1.1	.5	12%
	train4	4	4	4	1.1	1.0	.4	8%

benchmarks that we used in Section 6.6.1. ISM was slightly faster than BICA on only one benchmark (*ifsm1*). It was more than 100 times slower than BICA on five benchmarks, and was unable to solve seven benchmarks that BICA was able to solve.

Slim [HM96] is not available because it is Fujitsu's proprietary [Hig15]. Also, the source code of VOID [AD01] is not available anymore [Ahm15]. While we were able to obtain a copy of CHESMIN [GF07] from the authors, unfortunately, on our machine, the version we obtained yields segmentation faults on most benchmarks.

6.6.4 Experimental Setup

All experiments were run on an Intel Core i5-4590 (3.3GHz) with 4 GB of RAM. We disabled dynamic frequency scaling, and copied all executables and benchmark files to a RAM disk, to minimize timing variations due to hard drive accesses. The execution times were measured using the perf tool [per].

We used BICA in version 5.0.3. We specified a hash table of size 100,000 (parameter "-h 100,000"). With the default size of 10,000 some benchmarks failed with the error message "Too many collisions. Specify a large hash table." We used the version of STAMINA that is included in SIS 1.3.6 [Cho05]. For the exact mode, we specified the parameter "-s 0", and for the heuristic mode "-s 3" (which combines the "tight upper bound" and "isomorphic" heuristics). We used a version of COSME (as of Aug. 2010) that was provided to us by one of the authors. We specified the same parameters as in the evaluation in [AS13] ("--comparemode --shownum --xincha").

6.7 Conclusions and Future Work

With respect to the set of benchmarks used to evaluate previous approaches, one can consider the problem of minimizing incompletely specified Mealy machines to be solved: Our method can solve all of these benchmarks in less than 0.2 seconds, and all but four in less than 10 ms.

To evaluate the limits of our approach, one problem that future work will need to address is to identify a new set of challenging, realistic benchmarks.

6.A Appendix: Complete Benchmark Results

Table 6.3: ISM-benchmark results (exact approaches)

		MeMin (M)			Bica (B)				Stamina (SE)			
Benchmark	$\|Q\|$	$\|Q\|$	Time	stdev	$\|Q\|$	Time	stdev	B/M	$\|Q\|$	Time	stdev	SE/M
alex1	42	6	.0011	2.95%	6	.2870	0.27%	260.90	6	.1291	0.16%	117.36
intel_edge.dummy	28	4	.0008	2.21%	4	.0396	0.29%	49.50	~~5~~[1]	~~.0033~~	~~0.30%~~	~~4.12~~
isend	40	4	.0010	1.59%	4	.1594	0.07%	159.40	4	.0103	0.11%	10.30
pe-rcv-ifc.fc	46	2	.0008	2.43%	2	.1469	0.16%	183.62	2	.0021	0.51%	2.62
pe-rcv-ifc.fc.m	27	2	.0007	2.91%	2	.0837	0.23%	119.57	2	.0009	0.82%	1.28
pe-send-ifc.fc	70	2	.0011	1.52%	2	.4558	0.47%	414.36	2	.0089	0.09%	8.09
pe-send-ifc.fc.m	26	2	.0008	2.34%	2	.1729	0.27%	216.12	2	.0009	1.37%	1.12
vbe4a	58	3	.0012	1.29%	3	.3702	0.08%	308.50	3	1.6005	0.02%	1333.75
vmebus.master.m	32	2	.0020	2.37%	2	13.3897	0.09%	6694.85	2	.0048	0.33%	2.40
fo.16	17	3	.0006	4.05%	3	.0188	0.60%	31.33	3	2.9122	0.08%	4853.66
fo.20	21	3	.0005	3.28%	3	.0225	0.65%	45.00	—[4]	~~2.0142~~	~~0.01%~~	~~4028.40~~
fo.30	31	3	.0007	2.47%	3	.0304	0.28%	43.42	—[3]	~~3.8240~~	~~0.01%~~	~~5462.85~~
fo.40	41	4	.0015	1.71%	4	.1389	0.21%	92.60	—[3]	~~11.5049~~	~~0.01%~~	~~7669.93~~
fo.50	51	6	.0043	1.01%	6	.0999	0.07%	23.23	—[3]	~~7.2752~~	~~0.09%~~	~~1691.90~~
fo.60	61	7	.1692	0.13%	7	149.9053	0.70%	885.96	—[3]	~~9.8675~~	~~0.06%~~	~~58.31~~
fo.70	71	7	.1894	0.18%	—[2]	—	—	—	—[3]	~~9.1619~~	~~0.09%~~	~~48.37~~
th.20	21	4	.0007	2.45%	4	.0227	0.79%	32.42	~~6~~[1]	~~4.3983~~	~~0.05%~~	~~6283.28~~
th.25	26	4	.0007	3.26%	4	.0264	0.23%	37.71	—[4]	~~2.0331~~	~~0.02%~~	~~2904.42~~
th.30	31	5	.0006	3.31%	5	.0343	0.12%	57.16	—[4]	~~12.2109~~	~~0.03%~~	~~20351.50~~
th.35	36	7	.0008	2.01%	7	.0410	0.10%	51.25	—[3]	~~9.5360~~	~~0.01%~~	~~11920.00~~
th.40	41	8	.0009	2.03%	8	.0481	0.07%	53.44	—[3]	~~4.5429~~	~~0.06%~~	~~5047.66~~
th.55	55	8	.0015	1.22%	8	25.1838	0.17%	16789.20	—[3]	~~7.2665~~	~~0.00%~~	~~4844.33~~
ifsm0	38	3	.0009	1.81%	3	.0520	0.07%	57.77	3	.0012	0.72%	1.33
ifsm1	74	14	.0016	1.38%	14	.1488	0.05%	93.00	—[4]	~~1.4877~~	~~0.03%~~	~~929.81~~
ifsm2	48	9	.0015	1.10%	9	3.0301	0.21%	2020.06	9	5.2103	0.05%	3473.53
rubin1200	1200	3	.0374	0.15%	3	1.9400	0.03%	51.87	~~1200~~[1]	~~.4046~~	~~1.10%~~	~~10.81~~
rubin18	18	3	.0005	3.53%	3	.0083	0.50%	16.60	3	.0051	0.35%	10.20
rubin2250	2250	3	.1601	0.11%	3	11.9879	0.04%	74.87	~~2250~~[1]	~~2.6435~~	~~0.71%~~	~~16.51~~
rubin600	600	3	.0088	0.27%	3	.2803	0.07%	31.85	~~600~~[1]	~~.0542~~	~~0.79%~~	~~6.15~~
e271	19	2	.0005	3.01%	2	.0580	0.08%	116.00	2	.0005	1.65%	1.00
e285	19	2	.0006	2.69%	2	.0240	0.16%	40.00	2	.0005	1.72%	.83
e304	19	2	.0006	2.93%	2	.0231	0.14%	38.50	2	.0005	1.84%	.83
e423	19	2	.0005	2.80%	2	.0331	0.10%	66.20	—[3]	~~3.5725~~	~~0.03%~~	~~7145.00~~
e680	19	2	.0005	2.93%	2	.0271	0.18%	54.20	2	.0005	1.58%	1.00

[1]Invalid result [2]Timeout [3]Segmentation fault [4]*** glibc detected *** ./stamina: double free or corruption

Table 6.4: ISM-benchmark results (heuristic approaches)

		Cosme (C)				Stamina (SH)			
Benchmark	$\|Q\|$	$\|Q\|$	Time	stdev	C/M	$\|Q\|$	Time	stdev	SH/M
alex1	42	6	.0072	1.67%	6.54	6	.1361	0.09%	123.72
intel_edge.dummy	28	5[5]	.0032	0.25%	4.00	5[5]	.0048	0.25%	6.00
isend	40	4	.0072	1.51%	7.20	4	.0125	0.19%	12.50
pe-rcv-ifc.fc	46	2	.0083	0.33%	10.37	2	.0025	0.46%	3.12
pe-rcv-ifc.fc.m	27	2	.0039	0.67%	5.57	2	.0010	0.57%	1.42
pe-send-ifc.fc	70	2	.0224	0.42%	20.36	2	.0098	2.34%	8.90
pe-send-ifc.fc.m	26	2	.0040	0.15%	5.00	2	.0009	0.80%	1.12
vbe4a	58	3	.0163	0.36%	13.58	3	1.6815	0.02%	1401.25
vmebus.master.m	32	2	.0113	0.65%	5.65	2	.0049	0.17%	2.45
fo.16	17	3	.0015	0.88%	2.50	3	.0005	1.96%	.83
fo.20	21	3	.0018	0.26%	3.60	—[6]	~~.0009~~	~~1.64%~~	~~1.80~~
fo.30	31	3	.0027	0.36%	3.85	—[6]	~~.0048~~	~~0.56%~~	~~6.85~~
fo.40	41	6[5]	.0040	0.32%	2.66	—[6]	~~.1386~~	~~0.05%~~	~~92.40~~
fo.50	51	6	.0052	1.09%	1.20	—[6]	~~.0336~~	~~0.32%~~	~~7.81~~
fo.60	61	10[5]	.0093	2.15%	.05	—[6]	~~.0864~~	~~1.23%~~	~~.51~~
fo.70	71	9[5]	.0100	0.29%	.05	—[6]	~~.1259~~	~~0.11%~~	~~.66~~
th.20	21	4	.0018	0.34%	2.57	—[6]	~~.0014~~	~~0.73%~~	~~2.00~~
th.25	26	4	.0025	0.71%	3.57	—[6]	~~.0011~~	~~1.68%~~	~~1.57~~
th.30	31	6[5]	.0029	0.21%	4.83	—[6]	~~.0024~~	~~0.40%~~	~~4.00~~
th.35	36	7	.0036	0.39%	4.50	—[6]	~~.0026~~	~~0.57%~~	~~3.25~~
th.40	41	8	.0042	0.53%	4.66	—[6]	~~.0048~~	~~0.87%~~	~~5.33~~
th.55	55	10[5]	.0066	0.13%	4.40	—[6]	~~.0293~~	~~0.45%~~	~~19.53~~
ifsm0	38	3	.0098	0.27%	10.88	3	.0012	0.81%	1.33
ifsm1	74	14	.0140	0.12%	8.75	—[6]	~~.0026~~	~~0.31%~~	~~1.62~~
ifsm2	48	~~3~~[1]	~~.0058~~	~~0.41%~~	~~3.86~~	—[6]	~~.0024~~	~~0.37%~~	~~1.60~~
rubin1200	1200	15[5]	2.0928	0.05%	55.95	1200[5]	.4101	1.43%	10.96
rubin18	18	3	.0014	0.26%	2.80	3	.0004	2.01%	.80
rubin2250	2250	15[5]	4.0188	0.06%	25.10	2250[5]	2.6564	0.63%	16.59
rubin600	600	15[5]	1.1732	0.67%	133.31	600[5]	.0532	1.26%	6.04
e271	19	2	.0020	0.41%	4.00	2	.0005	1.57%	1.00
e285	19	2	.0019	0.27%	3.16	2	.0005	1.64%	.83
e304	19	19[5]	.0021	0.35%	3.50	2	.0005	1.91%	.83
e423	19	19[5]	.1175	0.04%	235.00	—[6]	~~.0017~~	~~0.58%~~	~~3.40~~
e680	19	19[5]	.0044	0.29%	8.80	2	.0005	1.69%	1.00

[1]Invalid result [5]Result not minimal [6]Other error

Table 6.5: MCNC-benchmark results (exact approaches)

		MeMin (M)			Bica (B)				Stamina (SE)			
Benchmark	$\|Q\|$	$\|Q\|$	Time	stdev	$\|Q\|$	Time	stdev	B/M	$\|Q\|$	Time	stdev	SE/M
bbara	10	7	.0005	2.83%	7	.0252	0.14%	50.40	7	.0005	1.59%	1.00
bbsse	16	13	.0009	1.73%	13	4.2474	0.03%	4719.33	13	.0006	1.30%	.66
bbtas	6	6	.0005	3.74%	6	.0015	0.87%	3.00	6	.0004	2.05%	.80
beecount	7	4	.0005	3.41%	4	.4787	0.11%	957.40	4	.0005	2.91%	1.00
cse	16	16	.0008	2.03%	16	.0075	0.35%	9.37	16	.0005	1.71%	.62
dk14	7	7	.0006	2.82%	7	.0017	1.09%	2.83	7	.0004	2.06%	.66
dk15	4	4	.0005	3.22%	4	.0016	0.80%	3.20	4	.0004	2.65%	.80
dk16	27	27	.0009	2.09%	27	.0021	0.97%	2.33	27	.0005	1.24%	.55
dk17	8	8	.0005	3.34%	8	.0016	1.23%	3.20	8	.0004	4.12%	.80
dk27	7	7	.0004	3.44%	7	.0015	0.86%	3.75	7	.0004	1.83%	1.00
dk512	15	15	.0005	2.81%	15	.0016	0.97%	3.20	15	.0004	1.65%	.80
donfile	24	1	.0005	3.92%	1	.0058	0.27%	11.60	1	.0006	2.64%	1.20
ex1	20	18	.0010	1.82%	18	6.4799	0.03%	6479.90	18	.0008	1.63%	.80
ex2	19	5	.0009	2.00%	5	.1767	0.03%	196.33	~~14~~[1]	~~.6162~~	~~0.15%~~	~~684.66~~
ex3	10	4	.0008	2.24%	4	.0864	0.10%	108.00	~~5~~[1]	~~.0028~~	~~0.50%~~	~~3.50~~
ex4	14	14	.0005	2.57%	14	.0030	0.54%	6.00	14	.0004	1.90%	.80
ex5	9	3	.0005	3.09%	3	.0213	0.30%	42.60	~~4~~[1]	~~.0008~~	~~0.92%~~	~~1.60~~
ex6	8	8	.0005	3.28%	8	.0025	1.57%	5.00	8	.0004	2.13%	.80
ex7	10	3	.0005	3.45%	3	.0138	0.35%	27.60	~~4~~[1]	~~.0013~~	~~0.42%~~	~~2.60~~
keyb	19	19	.0011	1.64%	19	.0796	0.35%	72.36	19	.0008	2.97%	.72
kirkman	16	16	.0021	0.67%	—[7]	~~.0007~~	~~1.24~~	~~.33~~	16	.0014	0.45%	.66
lion9	9	4	.0004	3.81%	4	.0156	0.61%	39.00	4	.0004	1.26%	1.00
lion	4	4	.0004	3.91%	4	.0015	1.10%	3.75	4	.0004	1.77%	1.00
mark1	15	12	.0010	1.72%	~~0~~[1]	~~.0008~~	~~1.38%~~	~~.80~~	12	.0006	1.44%	.60
mc	4	4	.0004	4.40%	4	.0015	0.52%	3.75	4	.0004	2.21%	1.00
modulo12	12	1	.0004	4.27%	1	.0053	0.45%	13.25	—[3]	~~.0489~~	~~0.40%~~	~~122.25~~
opus	10	9	.0005	3.07%	~~0~~[1]	~~.0008~~	~~1.27%~~	~~1.60~~	9	.0004	1.60%	.80
planet1	48	48	.0014	1.49%	48	.3190	0.04%	227.85	48	.0006	1.40%	.42
planet	48	48	.0014	1.30%	48	.3190	0.01%	227.85	48	.0006	0.91%	.42
pma	24	24	.0008	2.20%	24	.0287	0.13%	35.87	—[3]	~~.0482~~	~~0.16%~~	~~60.25~~
s1488	48	48	.0020	0.76%	48	.9242	0.08%	462.10	48	.0007	1.05%	.35
s1494	48	48	.0020	1.03%	48	.9369	2.64%	468.45	48	.0007	1.22%	.35
s1a	20	1	.0027	0.81%	1	.0558	0.06%	20.66	1	.0019	0.38%	.70
s1	20	20	.0010	1.50%	20	.1329	0.07%	132.90	20	.0005	2.48%	.50
s208	18	18	.0010	1.41%	18	.0462	0.05%	46.20	18	.0006	2.27%	.60
s27	6	5	.0005	3.32%	5	.0305	0.07%	61.00	5	.0004	2.12%	.80
s298	218	135	.0097	0.42%	135	34.2797	0.08%	3533.98	135	.0066	0.59%	.68
s386	13	13	.0007	2.24%	13	.0264	0.09%	37.71	13	.0004	1.85%	.57
s420	18	18	.0010	1.56%	18	.0642	0.04%	64.20	18	.0006	1.54%	.60
s510	47	47	.0012	1.47%	—[2]	—	—	—	47	.0005	1.66%	.41
s820	25	24	.0015	2.02%	—[8]	~~74.5850~~	~~0.70~~	~~49723.33~~	24	.0009	2.43%	.60
s832	25	24	.0016	1.37%	—[8]	~~75.4407~~	~~1.01~~	~~47150.43~~	24	.0009	2.04%	.56
s8	5	1	.0004	3.90%	1	.0054	0.19%	13.50	1	.0004	1.64%	1.00
sand	32	32	.0015	1.18%	32	19.4550	0.02%	12970.00	32	.0006	1.61%	.40
scf	121	97	.0033	0.72%	—[7]	~~.0010~~	~~0.91~~	~~.30~~	97	.0027	0.34%	.81
shiftreg	8	8	.0004	3.90%	8	.0015	0.89%	3.75	8	.0004	2.15%	1.00
sse	16	13	.0009	2.08%	13	4.2638	0.14%	4737.55	13	.0006	1.77%	.66
styr	30	30	.0012	1.39%	30	.2308	0.02%	192.33	30	.0006	1.78%	.50
tav	4	4	.0005	3.03%	4	.0020	1.21%	4.00	4	.0004	1.68%	.80
tbk	32	16	.0035	0.62%	16	.3170	0.06%	90.57	16	.0132	0.29%	3.77
tma	20	18	.0007	2.51%	18	.0639	0.12%	91.28	—[3]	~~.0486~~	~~0.22%~~	~~69.42~~
train11	11	4	.0005	3.91%	4	.0163	0.26%	32.60	4	.0004	2.60%	.80
train4	4	4	.0004	3.55%	4	.0016	1.38%	4.00	4	.0004	3.66%	1.00

[1]Invalid result [2]Timeout [3]Segmentation fault [7]*Inconsistent machine detected. [8]Out of memory

Table 6.6: MCNC-benchmark results (heuristic approaches)

		Cosme (C)				Stamina (SH)			
Benchmark	$\|Q\|$	$\|Q\|$	Time	stdev	C/M	$\|Q\|$	Time	stdev	SH/M
bbara	10	7	.0009	0.78%	1.80	7	.0005	1.81%	1.00
bbsse	16	~~4~~[1]	~~.0018~~	~~0.28%~~	~~2.00~~	13	.0007	1.06%	.77
bbtas	6	6	.0006	0.88%	1.20	6	.0004	2.04%	.80
beecount	7	~~3~~[1]	~~.0008~~	~~0.94%~~	~~1.60~~	4	.0005	1.79%	1.00
cse	16	~~5~~[1]	~~.0020~~	~~0.38%~~	~~2.50~~	16	.0005	0.97%	.62
dk14	7	7	.0007	0.98%	1.16	7	.0004	2.22%	.66
dk15	4	4	.0006	1.11%	1.20	4	.0004	2.43%	.80
dk16	27	27	.0012	0.58%	1.33	27	.0005	1.56%	.55
dk17	8	8	.0006	0.60%	1.20	8	.0004	1.71%	.80
dk27	7	7	.0006	1.29%	1.50	7	.0004	1.32%	1.00
dk512	15	15	.0027	0.39%	5.40	15	.0004	1.77%	.80
donfile	24	—[8]	~~287.4346~~	~~0.44~~	~~574869.20~~	1	.0006	3.21%	1.20
ex1	20	~~6~~[1]	~~.0033~~	~~0.41%~~	~~3.30~~	18	.0008	1.82%	.80
ex2	19	~~9~~[5]	.0020	0.61%	2.22	—[6]	~~.0024~~	~~1.72%~~	~~2.66~~
ex3	10	~~9~~[5]	.0019	0.37%	2.37	—[6]	~~.0006~~	~~2.82%~~	~~.75~~
ex4	14	~~5~~[1]	~~.0012~~	~~0.95%~~	~~2.40~~	14	.0004	1.66%	.80
ex5	9	~~8~~[5]	.0009	1.13%	1.80	—[6]	~~.0005~~	~~1.65%~~	~~1.00~~
ex6	8	~~2~~[1]	~~.0010~~	~~0.58%~~	~~2.00~~	8	.0004	2.09%	.80
ex7	10	~~9~~[5]	.0010	0.48%	2.00	—[6]	~~.0005~~	~~1.64%~~	~~1.00~~
keyb	19	~~3~~[1]	~~.0032~~	~~0.38%~~	~~2.90~~	19	.0008	1.12%	.72
kirkman	16	~~8~~[1]	~~.0043~~	~~0.34%~~	~~2.04~~	16	.0014	0.25%	.66
lion9	9	4	.0010	0.72%	2.50	4	.0004	2.50%	1.00
lion	4	~~2~~[1]	~~.0006~~	~~0.72%~~	~~1.50~~	4	.0004	2.46%	1.00
mark1	15	~~9~~[1]	~~.0013~~	~~0.60%~~	~~1.30~~	12	.0006	1.36%	.60
mc	4	~~2~~[1]	~~.0006~~	~~1.19%~~	~~1.50~~	4	.0004	3.48%	1.00
modulo12	12	—[8]	~~207.9701~~	~~1.10~~	~~519925.25~~	—[3]	~~.0483~~	~~0.40%~~	~~120.75~~
opus	10	~~4~~[1]	~~.0011~~	~~0.59%~~	~~2.20~~	9	.0004	1.48%	.80
planet1	48	~~12~~[1]	~~.0107~~	~~0.15%~~	~~7.64~~	48	.0006	1.97%	.42
planet	48	~~12~~[1]	~~.0107~~	~~0.25%~~	~~7.64~~	48	.0006	1.33%	.42
pma	24	~~11~~[1]	~~.0962~~	~~0.24%~~	~~120.25~~	—[3]	~~.0481~~	~~0.11%~~	~~60.12~~
s1488	48	~~15~~[1]	~~.0223~~	~~0.20%~~	~~11.15~~	48	.0007	1.17%	.35
s1494	48	~~14~~[1]	~~.0227~~	~~0.12%~~	~~11.35~~	48	.0007	1.79%	.35
s1a	20	—[2]	—	—	—	1	.0020	0.43%	.74
s1	20	~~3~~[1]	~~.0033~~	~~0.19%~~	~~3.30~~	20	.0005	1.34%	.50
s208	18	—[2]	—	—	—	18	.0006	3.40%	.60
s27	6	~~2~~[1]	~~.0008~~	~~1.26%~~	~~1.60~~	5	.0004	1.40%	.80
s298	218	—[2]	—	—	—	135	.0066	0.34%	.68
s386	13	~~3~~[1]	~~.0015~~	~~0.25%~~	~~2.14~~	13	.0004	1.51%	.57
s420	18	~~2~~[1]	~~.0026~~	~~0.36%~~	~~2.60~~	18	.0006	1.58%	.60
s510	47	~~9~~[1]	~~.0061~~	~~0.57%~~	~~5.08~~	47	.0005	1.34%	.41
s820	25	~~5~~[1]	~~.0059~~	~~0.22%~~	~~3.93~~	24	.0009	1.36%	.60
s832	25	~~5~~[1]	~~.0065~~	~~2.24%~~	~~4.06~~	24	.0009	0.69%	.56
s8	5	—[8]	~~144.1141~~	~~1.00~~	~~360285.25~~	1	.0004	3.78%	1.00
sand	32	~~3~~[1]	~~.0089~~	~~0.29%~~	~~5.93~~	32	.0006	1.67%	.40
scf	121	~~40~~[1]	~~.0403~~	~~0.08%~~	~~12.21~~	97	.0028	0.40%	.84
shiftreg	8	8	.0007	1.80%	1.75	8	.0004	1.39%	1.00
sse	16	~~4~~[1]	~~.0018~~	~~0.28%~~	~~2.00~~	13	.0007	1.41%	.77
styr	30	~~11~~[1]	~~.0057~~	~~0.23%~~	~~4.75~~	30	.0006	1.00%	.50
tav	4	—[8]	~~148.3782~~	~~0.23~~	~~296756.40~~	4	.0004	4.23%	.80
tbk	32	16	.0271	0.05%	7.74	16	.0133	0.16%	3.80
tma	20	~~12~~[1]	~~.0877~~	~~0.17%~~	~~125.28~~	—[3]	~~.0481~~	~~0.39%~~	~~68.71~~
train11	11	4	.0012	0.57%	2.40	4	.0004	1.85%	.80
train4	4	4	.0005	0.41%	1.25	4	.0004	2.05%	1.00

[1]Invalid result [2]Timeout [3]Segmentation fault [5]Result not minimal [6]Other Error [8]Out of memory

7 Summary, Conclusions, and Future Work

In this thesis, we have developed techniques to automatically generate models of microarchitectural components. Such models are, for example, important for building performance prediction tools, compilers, cycle-accurate simulators, and for showing the presence or absence of microarchitectural security issues. Previous approaches for obtaining microarchitectural models typically required a significant amount of manual work, and the models were often not very accurate and precise.

7.1 Summary and Conclusions

7.1.1 Models of Recent Microarchitectures

In the first part of the thesis, we focused on recent x86 microarchitectures.

nanoBench

We have developed a general tool for evaluating small microbenchmarks using hardware performance counters. Unlike previous tools, our tool can execute microbenchmarks directly in kernel space. This makes it possible to benchmark privileged instructions, and it allows for more accurate measurements by disabling interrupts and preemptions. The reading of the performance counters is implemented with minimal overhead, avoiding function calls and branches. Our tool is precise enough to measure individual memory accesses.

On top of *nanoBench*, we then developed techniques to automatically generate microbenchmarks for characterizing the performance of individual instructions and for inferring properties of the cache architecture.

Instruction Characterizations

We have presented an approach to automatically characterize the latency, throughput, and port usage of more than 13,000 instruction variants. For the latency, we have introduced a more precise definition that, in contrast to previous definitions, considers dependencies between different pairs of input and output operands. We have applied our approach to 16 different Intel and AMD microarchitectures. Our experimental evaluation demonstrates that the obtained instruction characterizations are both more accurate and more precise than those obtained by prior work.

Characterizing Cache Architectures

We have developed tools to analyze undocumented properties of caches. In particular, we have implemented two tools that can fully automatically infer a large class of deterministic cache replacement policies. Furthermore, for caches with nondeterministic policies, we have proposed tools that can help to identify their policies manually. We have applied our techniques to 13 different Intel microarchitectures, and we have found several previously undocumented replacement policies. Furthermore, we have discovered that flushing the cache does not necessarily reset the state of the replacement policy; this could be exploited to leak information in covert-channel or side-channel attacks.

7.1.2 General Models

In the second part of the thesis, we looked at more general techniques for obtaining models of hardware components. In particular, we have proposed the problem of *gray-box learning*, in which learning algorithms can exploit known information about the system to be learned. As a first step toward solving this problem, we have developed an efficient learning algorithm for the serial composition of two Mealy machines, in which the left machine is known and the right machine is unknown.

While this was only a first step—applications to real-world systems will require support for more complex composition topologies—we were able to adapt a central idea of our learning algorithm to the problem of minimizing incompletely specified Mealy machines. We have implemented an exact minimization tool that outperforms previous exact approaches by several orders of magnitude on a number of hard benchmarks and is even competitive with state-of-the-art heuristic techniques.

7.2 Future Work

There are multiple directions for future work.

One direction is to develop microbenchmarks to build models of other performance-relevant components of microarchitectures, such as instruction decoders, branch predictors, or translation lookaside buffers. Like the microbenchmarks described in Chapters 3 and 4, these microbenchmarks could be evaluated using *nanoBench*.

Another direction is to adapt our techniques to non-x86 architectures, such as ARM, MIPS, or RISC-V.

Furthermore, we would like to build tools that use the models we generated in the first part of the thesis to predict or explain the performance of software running on the corresponding microarchitectures. We would also like to integrate our models into existing tools, such as simulators like gem5 [BBB+11], or compilers like GCC [GCC] and LLVM [LA04].

In addition to that, we would like to implement and analyze microarchitectural covert-channel and side-channel attacks, in particular regarding the issues discovered in Section 3.7.4 and Section 4.4.4.

Finally, we would like to close the gap between theory and practice by extending our gray-box learning approach to more complex composition topologies so that it can be applied to real-world systems.

Bibliography

[Abe12] Andreas Abel. Measurement-based inference of the cache hierarchy. Master's thesis, Universität des Saarlandes, December 2012. URL: `http://embedded.cs.uni-saarland.de/literature/AndreasAbelMastersThesis.pdf`. (Cited on pages 82, 83, 84, 86, 90, 111, and 112).

[ABu+19] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887, May 2019. `doi:10.1109/SP.2019.00066`. (Cited on page 76).

[AD01] Imtiaz Ahmad and A. Shoba Das. A heuristic algorithm for the minimization of incompletely specified finite state machines. *Computers & Electrical Engineering*, 27(2):159–172, 2001. `doi:10.1016/S0045-7906(00)00016-1`. (Cited on pages 141, 145, 151, 155, and 158).

[AFBKV15] Fides Aarts, Paul Fiterau-Brostean, Harco Kuppens, and Frits Vaandrager. Learning register automata with fresh value generation. In *Proceedings of the 12th International Colloquium on Theoretical Aspects of Computing (ICTAC 2015)*, pages 165–183. Springer-Verlag, 2015. `doi:10.1007/978-3-319-25150-9_11`. (Cited on pages 17 and 118).

[Ahm01] Imtiaz Ahmad. A distributed algorithm for finding prime compatibles on network of workstations. *Microprocessors and Microsystems*, 25(4):195–202, 2001. `doi:10.1016/S0141-9331(01)00112-0`. (Cited on pages 139 and 145).

[Ahm15] Imtiaz Ahmad. Personal communication, March 2015. (Cited on page 158).

[ALOJ13] Jung Ho Ahn, Sheng Li, Seongil O, and Norman P. Jouppi. McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software, Austin, TX, USA*, pages 74–85, 2013. `doi:10.1109/ISPASS.2013.6557148`. (Cited on pages 40 and 79).

[AMD17] AMD. Software optimization guide for AMD family 17h processors. Publication No. 55723, Revision 3.00, June

2017. URL: https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf. (Cited on pages 43 and 46).

[AMD19] AMD. AMD64 architecture programmer's manual volume 2: System programming. Publication No. 24593, Revision 3.32, October 2019. URL: https://www.amd.com/system/files/TechDocs/24593.pdf. (Cited on page 74).

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, November 1987. doi:10.1016/0890-5401(87)90052-6. (Cited on pages 17, 118, 120, 134, and 145).

[AQH90] M. J. Avedillo, J. M. Quintana, and J. L. Huertas. New approach to the state reduction in incompletely specified sequential machines. In *IEEE International Symposium on Circuits and Systems*, pages 440–443, 1990. doi:10.1109/ISCAS.1990.112075. (Cited on page 146).

[AR12] Andreas Abel and Jan Reineke. Automatic cache modeling by measurements. In *6th Junior Researcher Workshop on Real-Time Computing (in conjunction with RTNS)*, November 2012. URL: http://embedded.cs.uni-saarland.de/publications/CacheModelingJRWRTC.pdf. (Cited on pages 86 and 111).

[AR13] Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Philadelphia, PA, USA, April 9-11, 2013*, pages 65–74, 2013. doi:10.1109/RTAS.2013.6531080. (Cited on pages 82, 83, 90, 96, 111, and 112).

[AR14] Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, March 23-25, 2014*, pages 141–142, 2014. doi:10.1109/ISPASS.2014.6844475. (Cited on pages 19 and 79).

[AR15] Andreas Abel and Jan Reineke. MeMin: SAT-based exact minimization of incompletely specified Mealy machines. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, November*

2-6, 2015, ICCAD '15, pages 94–101. IEEE Press, 2015. doi:10.1109/ICCAD.2015.7372555. (Cited on pages 19 and 139).

[AR16] Andreas Abel and Jan Reineke. Gray-box learning of serial compositions of Mealy machines. In *NASA Formal Methods—Proceedings of the 8th International Symposium, Minneapolis, MN, USA, June 7-9, 2016*, NFM 2016, pages 272–287. Springer-Verlag, June 2016. doi:10.1007/978-3-319-40648-0_21. (Cited on pages 19 and 117).

[AR19] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Providence, RI, USA, April 13-17, 2019*, ASPLOS '19, pages 673–686. ACM, 2019. doi:10.1145/3297858.3304062. (Cited on pages 19 and 39).

[AR20] Andreas Abel and Jan Reineke. nanoBench: A low-overhead tool for running microbenchmarks on x86 systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, USA, August 23–25, 2020*, August 2020. To appear. (Cited on pages 19, 21, and 79).

[AS09] A. Alberto and A. Simao. Minimization of incompletely specified finite state machines based on distinction graphs. In *10th Latin American Test Workshop (LATW '09)*, pages 1–6, March 2009. doi:10.1109/LATW.2009.4813796. (Cited on page 140).

[AS13] Alex D. B. Alberto and Adenilso Simao. Iterative minimization of partial finite state machines. *Central European Journal of Computer Science*, 3(2):91–103, 2013. doi:10.2478/s13537-013-0106-0. (Cited on pages 141, 145, 146, 151, 154, and 158).

[Auf58] D. D. Aufenkamp. Analysis of sequential machines II. *IRE Transactions on Electronic Computers*, EC-7(4):299–306, 1958. doi:10.1109/TEC.1958.5222663. (Cited on page 144).

[AZMM04] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, pages 267–272. ACM, 2004. doi:10.1145/986537.986601. (Cited on page 83).

[BACD97] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347. Association for Computing Machinery, 1997. doi:10.1145/263580.263662. (Cited on page 79).

[BBB+11] 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. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011. doi:10.1145/2024716.2024718. (Cited on pages 40, 79, and 167).

[BBG+12] Ramon Bertran, Alper Buyuktosunoglu, Meeta S. Gupta, Marc Gonzalez, and Pradip Bose. Systematic energy characterization of CMP/SMT processor systems via automated micro-benchmarks. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 199–211, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/MICRO.2012.27. (Cited on page 40).

[BBS12] Domagoj Babic, Matko Botincan, and Dawn Song. Symbolic grey-box learning of input-output relations. Technical Report UCB/EECS-2012-59, University of California, Berkeley, May 2012. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-59.html. (Cited on page 135).

[BC00] Josep M. Blanquer and Robert C. Chalmers. MOB: A memory organization benchmark. Technical report, University of California, Santa Barbara, October 2000. URL: http://www.gnu-darwin.org/www001/ports-1.5a-CURRENT/devel/mob/work/mob-0.1.0/doc/mob.ps. (Cited on pages 86 and 111).

[Ben71] R. G. Bennetts. An improved method of prime C-class derivation in the state reduction of sequential networks. *IEEE Transactions on Computers*, 20(2):229–231, February 1971. doi:10.1109/T-C.1971.223221. (Cited on page 145).

[BF72] Alan W. Biermann and Jerome A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE*

Transactions on Computers, 21(6):592–597, June 1972. doi:10.1109/TC.1972.5009015. (Cited on page 145).

[BHLM13] Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. A fresh approach to learning register automata. In *Proceedings of the 17th International Conference on Developments in Language Theory (DLT)*, pages 118–130, 2013. doi:10.1007/978-3-642-38771-5_12. (Cited on pages 17 and 118).

[Bia18] Andrea Di Biagio. llvm-mca: a static performance analysis tool, 2018. URL: http://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html. (Cited on pages 40 and 77).

[Bil] Olexa Bilaniuk. libpfc. URL: https://github.com/obilaniu/libpfc. (Cited on pages 22, 31, and 35).

[BKMO14] Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Leakage resilience against concurrent cache attacks. In *Principles of Security and Trust—Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France*, pages 140–158, April 2014. doi:10.1007/978-3-642-54792-8_8. (Cited on page 114).

[BMME19] Samira Briongos, Pedro Malagón, José M. Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks, April 2019. arXiv:1904.06278. (Cited on pages 84, 112, and 115).

[BSN+19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 785–800, New York, NY, USA, 2019. ACM. doi:10.1145/3319535.3363194. (Cited on page 76).

[BT09] Vlastimil Babka and Petr Tůma. Investigating cache parameters of x86 family processors. In *Proceedings of the 2009 SPEC benchmark workshop*, pages 77–96. Springer, 2009. doi:10.1007/978-3-540-93799-9_5. (Cited on pages 86 and 111).

[BWL72] R. G. Bennetts, J. L. Washington, and D. W. Lewin. A computer algorithm for state table reduction. *Radio and Electronic Engineer*, 42(11):513–520, November 1972. doi:10.1049/ree.1972.0088. (Cited on page 145).

[C+10] Keith Cooper et al. The platform-aware compilation environment, preliminary design document. Technical report, Department of Computer Science, Rice University, September 2010. URL: http://pace.rice.edu/uploadedFiles/Publications/PACEDesignDocument.pdf. (Cited on page 79).

[CBM+19] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondrej Sykora, Saman Amarasinghe, and Michael Carbin. BHive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, November 2019. URL: http://groups.csail.mit.edu/commit/papers/19/ithemal-measurement.pdf. (Cited on page 36).

[CD01] C. L. Coleman and J. W. Davidson. Automatic memory hierarchy characterization. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 103–110, November 2001. doi:10.1109/ISPASS.2001.990684. (Cited on pages 86 and 111).

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003. doi:10.1007/3-540-36577-X_24. (Cited on page 118).

[Cho05] Philip Chong. SIS 1.3 unofficial distribution, November 2005. URL: https://ptolemy.berkeley.edu/projects/embedded/Alumni/pchong/sis.html. (Cited on page 158).

[CnKR17] Pablo Cañones, Boris Köpf, and Jan Reineke. Security analysis of cache replacement policies. In *Proceedings of the 6th International Conference on Principles of Security and Trust (POST)*, volume 10204, pages 189–209, Berlin, Heidelberg, 2017. Springer-Verlag. doi:10.1007/978-3-662-54455-6_9. (Cited on page 114).

[CnKR19] Pablo Cañones, Boris Köpf, and Jan Reineke. On the incomparability of cache algorithms in terms of timing leakage. *Logical Methods in Computer Science*, Volume 15, Issue 1, March 2019. doi:10.23638/LMCS-15(1:21)2019. (Cited on page 114).

[CRON+14] A. S. Charif-Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue. CQA: A code quality analyzer tool at binary level. In *21st International Conference on High Performance Computing (HiPC)*, pages 1–10, December 2014. URL: http://www.maqao.org/publications/papers/CQA.pdf, doi:10.1109/HiPC.2014.7116904. (Cited on pages 40, 43, and 77).

[CS11] Keith Cooper and Jeffrey Sandoval. Portable techniques to find effective memory hierarchy parameters. Technical report, Rice University, 2011. URL: https://hdl.handle.net/1911/96399. (Cited on pages 86 and 111).

[CV14] Jie Chen and Guru Venkataramani. CC-Hunter: Uncovering covert timing channels on shared processor hardware. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 216–228, Washington, DC, USA, 2014. IEEE Computer Society. doi:10.1109/MICRO.2014.42. (Cited on page 76).

[CWPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. In *Parallel Computing*, volume 27, pages 3–35. Elsevier, 2001. doi:10.1016/S0167-8191(00)00087-9. (Cited on page 79).

[CX18] Keith Cooper and Xiaoran Xu. Efficient characterization of hidden processor memory hierarchies. In *Proceedings of the 18th International Conference on Computational Science (ICCS), Part III*, pages 335–349, June 2018. doi:10.1007/978-3-319-93713-7_27. (Cited on pages 86 and 111).

[DK17] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 406–421, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062388. (Cited on page 114).

[DKMR15] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)*, 18(1), June 2015. doi:10.1145/2756550. (Cited on page 114).

[DKPT17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. PRIME+ABORT: A timer-free high-precision L3 cache attack using Intel TSX. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC '17, pages 51–67, Berkeley, CA, USA, 2017. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=3241189.3241195. (Cited on pages 81 and 114).

[DLM+13] Anthony Danalis, Piotr Luszczek, Gabriel Marin, Jeffrey S. Vetter, and Jack Dongarra. BlackjackBench: Portable hardware characterization with automated results' analysis. *The Computer Journal*, 57(7):1002–1016, 06 2013. doi:10.1093/comjnl/bxt057. (Cited on pages 86 and 111).

[DMM+04] Jack J. Dongarra, Shirley Moore, Philip Mucci, Keith Seymour, and Haihang You. Accurate cache and TLB characterization using hardware counters. In *Computational Science—ICCS 2004*, pages 432–439, 2004. doi:10.1007/978-3-540-24688-6_57. (Cited on pages 86 and 111).

[DSBC69] S. C. De Sarkar, A. K. Basu, and A. K. Choudhury. Simplification of incompletely specified flow tables with the help of prime closed sets. *IEEE Transactions on Computers*, C-18(10):953–956, October 1969. doi:10.1109/T-C.1969.222552. (Cited on page 145).

[DXS19] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. A benchmark suite for evaluating caches' vulnerability to timing attacks. *CoRR*, abs/1911.08619, November 2019. arXiv:1911.08619. (Cited on page 114).

[EGPQ06] Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu. Grey-box checking. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 420–435, 2006. doi:10.1007/11888116_30. (Cited on page 135).

[EMRCS14] J. Echavarria, A. Morales-Reyes, R. Cumplido, and M. A. Salido. FSM merging and reduction for IP cores watermarking using

genetic algorithms. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7. IEEE, December 2014. doi:10.1109/ReConFig.2014.7032525. (Cited on page 140).

[ENBSH11] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Cache pirating: Measuring the curse of the shared cache. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 165–175, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/ICPP.2011.15. (Cited on pages 104 and 115).

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004. doi:10.1007/978-3-540-24605-3_37. (Cited on page 149).

[Fal19] Brandon Falk. Sushi Roll: A CPU research kernel with minimal noise for cycle-by-cycle micro-architectural introspection, August 2019. URL: https://gamozolabs.github.io/metrology/2019/08/19/sushi_roll.html. (Cited on page 37).

[FDTZ04] Basilio B. Fraguela, Ramon Doallo, Juan Tourino, and Emilio L. Zapata. A compiler tool to predict memory hierarchy performance of scientific codes. *Parallel Computing*, 30(2):225–248, 2004. doi:10.1016/j.parco.2003.09.004. (Cited on page 114).

[Fin] FinalWire Ltd. AIDA64. URL: https://www.aida64.com/. (Cited on page 43).

[FJ05] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. doi:10.1109/JPROC.2004.840301. (Cited on page 79).

[Fog] Agner Fog. Test programs for measuring clock cycles and performance monitoring. URL: https://www.agner.org/optimize/. (Cited on pages 32 and 35).

[Fog19] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical University of Denmark, August 2019. URL: http://www.agner.org/optimize/instruction_tables.pdf. (Cited on pages 15, 40, 43, 48, 49, 58, and 69).

[FRMK19] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, Jr., and Dejan Kostić. Make the most out of last level cache in Intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 8:1–8:17, New York, NY, USA, 2019. ACM. doi:10.1145/3302424.3303977. (Cited on page 81).

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, USA, 2011. IEEE Computer Society. doi:10.1109/SP.2011.22. (Cited on page 114).

[GCC] GCC, the GNU compiler collection. URL: https://gcc.gnu.org/. (Cited on pages 40 and 167).

[GDTF+10] Jorge González-Domínguez, Guillermo L. Taboada, Basilio B. Fraguela, María J. Martín, and Juan Touriño. Servet: A benchmark suite for autotuning on multicore clusters. In *IPDPS*, pages 1–9. IEEE, 2010. doi:10.1109/IPDPS.2010.5470358. (Cited on pages 86 and 111).

[GF07] Sezer Gören and F. Joel Ferguson. On state reduction of incompletely specified finite state machines. *Computers & Electrical Engineering*, 33(1):58–69, January 2007. doi:10.1016/j.compeleceng.2006.06.001. (Cited on pages 141, 146, 151, and 158).

[Gin59] Seymour Ginsburg. On the reduction of superfluous states in a sequential machine. *J. ACM*, 6(2):259–282, April 1959. doi:10.1145/320964.320983. (Cited on page 144).

[GJ11] Karthik Ganesan and Lizy K. John. MAximum Multicore POwer (MAMPO): An automatic multithreaded synthetic power virus generation framework for multicore systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 53:1–53:12, New York, NY, USA, 2011. ACM. doi:10.1145/2063384.2063455. (Cited on page 40).

[GJB+10] Karthik Ganesan, Jungho Jo, W. Lloyd Bircher, Dimitris Kaseridis, Zhibin Yu, and Lizy K. John. System-level max power (SYMPO): A systematic approach for escalating system-level power consumption using synthetic benchmarks. In *Proceedings of the 19th International Conference on Parallel Architectures*

and Compilation Techniques, PACT '10, pages 19–28, New York, NY, USA, 2010. ACM. doi:10.1145/1854273.1854282. (Cited on page 40).

[GL65] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IEEE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965. doi:10.1109/PGEC.1965.264140. (Cited on pages 140, 144, and 147).

[GL06] Olga Grinchtein and Martin Leucker. Learning finite-state machines from inexperienced teachers. In *Proceedings of the 8th International Conference on Grammatical Inference: Algorithms and Applications*, pages 344–345. Springer, 2006. doi:10.1007/11872436_30. (Cited on pages 134 and 145).

[GLP06] Olga Grinchtein, Martin Leucker, and Nir Piterman. Inferring network invariants automatically. In *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 483–497. Springer, 2006. doi:10.1007/11814771_40. (Cited on page 134).

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA 2016, pages 300–321. Springer-Verlag, 2016. doi:10.1007/978-3-319-40667-1_15. (Cited on page 114).

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA 2016, pages 279–299, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-319-40667-1_14. (Cited on page 114).

[Goo] Google. EXEgesis. URL: https://github.com/google/EXEgesis. (Cited on page 43).

[GPY02] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer, 2002. doi:10.1007/3-540-46002-0_25. (Cited on page 118).

[GR08] Daniel Grund and Jan Reineke. Estimating the performance of cache replacement policies. In *Proceedings of the Sixth ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 101–112. IEEE, 2008. doi:10.1109/MEMCOD.2008.4547695. (Cited on page 113).

[Gra17] Torbjörn Granlund. Instruction latencies and throughput for AMD and Intel x86 processors, April 2017. URL: https://gmplib.org/~tege/x86-timing.pdf. (Cited on pages 40, 43, 48, 71, and 72).

[GS06] Fei Guo and Yan Solihin. An analytical model for cache replacement policy performance. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, pages 228–239, New York, NY, USA, 2006. ACM. doi:10.1145/1140277.1140304. (Cited on page 113).

[GYCH18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, April 2018. doi:10.1007/s13389-016-0141-6. (Cited on pages 75, 76, and 114).

[GYLH16] Qian Ge, Yuval Yarom, Frank Li, and Gernot Heiser. Your processor leaks information—and there's nothing you can do about it, 2016. arXiv:1612.04474. (Cited on page 110).

[H+10] Stefan Henkler et al. Legacy component integration by the Fujaba real-time tool suite. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 267–270, New York, NY, USA, 2010. ACM. doi:10.1145/1810295.1810349. (Cited on page 135).

[HHEW15] Julian Hammer, Georg Hager, Jan Eitzinger, and Gerhard Wellein. Automatic loop kernel analysis and performance modeling with Kerncraft. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, pages 4:1–4:11, New York, NY, USA, 2015. ACM. doi:10.1145/2832087.2832092. (Cited on pages 40 and 77).

[Hig15] Hiroyuki Higuchi. Personal communication, March 2015. (Cited on page 158).

[HKP15] Mohamed Hassan, Anirudh M. Kaushik, and Hiren D. Patel. Reverse-engineering embedded memory controllers through latency-based analysis. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA*, pages 297–306, 2015. doi:10.1109/RTAS.2015.7108453. (Cited on page 111).

[HL11] Yating Hsu and David Lee. Machine learning for implanted malicious code detection with incompletely specified system implementations. In *IEEE International Conference on Network Protocols*, pages 31–36, Washington, DC, USA, 2011. doi:10.1109/ICNP.2011.6089070. (Cited on page 134).

[HM95] Hiroyuki Higuchi and Yusuke Matsunaga. Implicit prime compatible generation for minimizing incompletely specified finite state machines. In *Proceedings of the Asia and South Pacific Design Automation Conference*, ASP-DAC '95, New York, NY, USA, 1995. ACM. doi:10.1145/224818.224903. (Cited on page 145).

[HM96] Hiroyuki Higuchi and Yusuke Matsunaga. A fast state reduction algorithm for incompletely specified finite state machines. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, pages 463–466, New York, NY, USA, 1996. ACM. doi:10.1145/240518.240606. (Cited on pages 141, 145, 151, 154, 155, and 158).

[Hop71] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971. doi:10.1016/B978-0-12-417750-5.50022-1. (Cited on pages 18 and 140).

[HSJC12] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2012. doi:10.1007/978-3-642-27940-9_17. (Cited on pages 17 and 118).

[HV10] Marijn J. H. Heule and Sicco Verwer. Exact DFA identification using SAT solvers. In *Grammatical Inference: Theoretical Results and Applications*, volume 6339 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2010. doi:10.1007/978-3-642-15488-1_7. (Cited on pages 128 and 131).

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013. doi:10.1109/SP.2013.23. (Cited on pages 81 and 82).

[HXB04] Heng Hu, Hong-Xi Xue, and Ji-Nian Bian. HSM2: A new heuristic state minimization algorithm for finite state machine. *Journal of Computer Science and Technology*, 19(5):729–733, September 2004. doi:10.1007/BF02945600. (Cited on pages 141, 145, 151, and 155).

[IES15] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In *Proceedings of the 2015 Euromicro Conference on Digital System Design*, pages 629–636. IEEE, 2015. doi:10.1109/DSD.2015.56. (Cited on pages 81 and 82).

[IGI+16] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016. doi:10.1007/978-3-662-53140-2_18. (Cited on pages 81 and 82).

[IHS15] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source LearnLib. In *Computer Aided Verification*, pages 487–495. Springer, 2015. doi:10.1007/978-3-319-21690-4_32. (Cited on pages 17, 119, and 133).

[Ins] InstLatx64. x86, x64 instruction latency, memory latency and CPUID dumps. URL: http://instlatx64.atw.hu/. (Cited on pages 40, 43, and 69).

[Inta] Intel Corporation. Intel architecture code analyzer. URL: https://software.intel.com/en-us/articles/intel-architecture-code-analyzer. (Cited on pages 15, 41, 42, and 48).

[Intb] Intel Corporation. Intel VTune Amplifier. URL: https://software.intel.com/vtune. (Cited on pages 22 and 35).

[Intc] Intel Corporation. X86 Encoder Decoder (XED). URL: https://intelxed.github.io/. (Cited on pages 15, 41, and 61).

[Int12] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2012. Order Number: 248966-026. URL: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf. (Cited on pages 14, 40, 42, 47, 69, 70, and 71).

[Int19a] Intel Corporation. 10th generation Intel Core Processor—instruction throughput and latency README, September 2019. URL: https://software.intel.com/en-us/download/10th-generation-intel-core-processor-instruction-throughput-and-latency-docs. (Cited on pages 42 and 64).

[Int19b] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, September 2019. Order Number: 248966-042b. URL: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf. (Cited on pages 14, 27, 40, 42, 44, 47, 48, 53, 54, 62, 71, and 72).

[Int19c] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, October 2019. Order Number: 325462-071US. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. (Cited on pages 33, 43, 61, 70, and 74).

[JB07] Tobias John and Robert Baumgartl. Exact cache characterization by experimental parameter extraction. In *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS)*, pages 65–74, Nancy, France, 2007. URL: https://hal.inria.fr/inria-00168530/file/actes.pdf#page=66. (Cited on pages 86 and 111).

[JEJI08] Ajay Joshi, Lieven Eeckhout, Lizy K. John, and Ciji Isen. Automated microprocessor stressmark generation. In *International Symposium on High-Performance Computer Architecture-Proceedings*, pages 209–219. IEEE Computer Society, February 2008. doi:10.1109/HPCA.2008.4658642. (Cited on page 40).

[JGSW12] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells. Power management of the third generation Intel Core micro architecture formerly codenamed Ivy Bridge. *2012 IEEE Hot Chips 24 Symposium (HCS)*, August 2012. doi:10.1109/hotchips.2012.7476478. (Cited on page 84).

[JTSE10] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM. doi:10.1145/1815961.1815971. (Cited on pages 83, 84, 86, and 93).

[KAGPJ16] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016. doi:10.1145/2897937.2897962. (Cited on pages 81 and 82).

[KDK+14] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, June 2014. doi:10.1109/ISCA.2014.6853210. (Cited on page 114).

[KHF+19] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019. doi:10.1109/sp.2019.00002. (Cited on pages 13 and 22).

[KLA+18] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 974–987. IEEE Press, 2018. doi:10.1109/MICRO.2018.00083. (Cited on page 115).

[KMO12] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV '12, pages 564–580, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31424-7_40. (Cited on page 114).

[KPD+11] Russell Kegley, Jonathan Preston, Brian Dougherty, Jules White, and Anirudda Gokhale. Predictive cache modeling and analysis. Technical report, Lockheed Martin Aeronautics

Corporation, November 2011. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a552968.pdf. (Cited on page 114).

[KPMR12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, USA, 2012. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kim. (Cited on page 115).

[KS91] L. N. Kannan and D. Sarma. Fast heuristic algorithms for finite state machine minimization. In *Proceedings of the European Conference on Design Automation.*, EURO-DAC '91, pages 192–196, Los Alamitos, CA, USA, February 1991. IEEE Computer Society Press. doi:10.1109/EDAC.1991.206388. (Cited on pages 141, 145, and 151).

[KS13] A. S. Klimowicz and V. V. Solov'ev. Minimization of incompletely specified Mealy finite-state machines by merging two internal states. *J. Comput. Syst. Sci. Int.*, 52(3):400–409, May 2013. doi:10.1134/S106423071303009X. (Cited on pages 141, 146, 151, and 155).

[KVBSV94] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *31st Conference on Design Automation*, pages 684–690, June 1994. doi:10.1109/DAC.1994.204189. (Cited on pages 141, 151, and 155).

[KVBSV10] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization.* Springer Publishing Company, 1st edition, 2010. (Cited on page 145).

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society. doi:10.1109/CGO.2004.1281665. (Cited on pages 40, 42, and 167).

[Lam73] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973. doi:10.1145/362375.362389. (Cited on page 75).

[LD97] Stan Liao and Srinivas Devadas. Solving covering problems using LPR-based lower bounds. In *Proceedings of the 34th Annual Design Automation Conference (DAC)*, pages 117–120. ACM, June 1997. doi:10.1145/266021.266046. (Cited on page 145).

[Lee15] David Lee. Personal communication, January 2015. (Cited on page 134).

[LGR+16] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1–05:48, 2016. doi:10.4230/LITES-v003-i001-a005. (Cited on page 79).

[LHH+18] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. Automated instruction stream throughput prediction for Intel and AMD microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 121–131, Los Alamitos, CA, USA, November 2018. IEEE Computer Society. doi:10.1109/PMBS.2018.8641578. (Cited on page 40).

[LHHW19] Jan Laukemann, Julian Hammer, Georg Hager, and Gerhard Wellein. Automatic throughput and critical path analysis of x86 and ARM assembly kernels. *CoRR*, abs/1910.00214, October 2019. arXiv:1910.00214. (Cited on page 40).

[LHS+20] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. ACM, June 2020. To appear. doi:10.1145/3320269.3384746. (Cited on page 114).

[LN12] Martin Leucker and Daniel Neider. Learning minimal deterministic automata from inexperienced teachers. In *ISoLA*, pages 524–538, 2012. doi:10.1007/978-3-642-34026-0_39. (Cited on page 134).

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin,

Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, 2018. arXiv:1801.01207. (Cited on pages 13 and 22).

[LSX09] Gabriel H. Loh, Samantika Subramaniam, and Yuejian Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 53–64, 2009. doi:10.1109/ISPASS.2009.4919638. (Cited on pages 40 and 79).

[LT98] Enyou Li and Clark Thomborson. Data cache parameter measurements. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 376–383, Los Alamitos, CA, USA, October 1998. IEEE. doi:10.1109/ICCD.1998.727077. (Cited on pages 86 and 111).

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015. doi:10.1109/SP.2015.43. (Cited on pages 81, 112, and 114).

[Man04] Stefan Manegold. The calibrator (v0.9e), a cache-memory and TLB calibration tool. http://homepages.cwi.nl/~manegold/Calibrator/, June 2004. (Cited on pages 86 and 111).

[MC17] Xinxin Mei and Xiaowen Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):72–86, 2017. doi:10.1109/TPDS.2016.2549523. (Cited on page 111).

[McC18] John D. McCalpin. Comments on timing short code sections on Intel processors, July 2018. URL: https://sites.utexas.edu/jdm4372/2018/07/23/comments-on-timing-short-code-sections-on-intel-processors/. (Cited on page 33).

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955. doi:10.1002/j.1538-7305.1955.tb03788.x. (Cited on page 144).

[Mee18] Hendrik Meerkamp. *SUACA: A tool for performance analysis of machine programs.* Bachelor thesis, Saarland University, July 2018. URL: http://compilers.cs.uni-saarland.de/publications/theses/meerkamp_bsc.pdf. (Cited on page 77).

[MHSM09] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 261–270, Washington, DC, USA, 2009. IEEE. doi:10.1109/PACT.2009.22. (Cited on page 111).

[MM14] Oded Maler and Irini-Eleftheria Mens. Learning regular languages over large alphabets. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 485–499. Springer, 2014. doi:10.1007/978-3-642-54862-8_41. (Cited on pages 17 and 118).

[MRAC19] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, California, USA, June 2019. PMLR. URL: http://proceedings.mlr.press/v97/mendis19a.html. (Cited on page 36).

[MS96] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, 1996. URL: http://dl.acm.org/citation.cfm?id=1268299.1268322. (Cited on pages 86 and 111).

[MSN+15] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID 2015, pages 48–65, New York, NY, USA, 2015. Springer-Verlag New York, Inc. doi:10.1007/978-3-319-26362-5_3. (Cited on pages 81, 82, and 112).

[MWK17] Heiko Mantel, Alexandra Weber, and Boris Köpf. A systematic study of cache side channels across AES implementations. In *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 213–230. Springer,

July 2017. doi:10.1007/978-3-319-62105-0_14. (Cited on page 114).

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA '06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11605805_1. (Cited on page 114).

[Pao10] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation*, September 2010. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. (Cited on page 32).

[per] perf: Linux profiling with performance counters. URL: https://perf.wiki.kernel.org. (Cited on pages 22, 35, and 158).

[Pfl73] C. P. Pfleeger. State reduction in incompletely specified finite-state machines. *IEEE Transactions on Computers*, C-22(12):1099–1102, December 1973. doi:10.1109/T-C.1973.223655. (Cited on pages 18, 140, and 144).

[PG93] R. Puri and Jun Gu. An efficient algorithm to search for minimal closed covers in sequential machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(6):737–745, June 1993. doi:10.1109/43.229748. (Cited on pages 141, 145, and 151).

[PJ15] Xiaoyue Pan and Bengt Jonsson. A modeling framework for reuse distance-based estimation of cache performance. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 62–71. IEEE, March 2015. doi:10.1109/ISPASS.2015.7095785. (Cited on page 83).

[PO99] J. M. Pena and A. L. Oliveira. A new algorithm for exact reduction of incompletely specified finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(11):1619–1632, November 1999. doi:10.1109/43.806807. (Cited on pages 134, 141, 145, 149, 151, and 154).

[PU59] M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, EC-8(3):356–367, September 1959. doi:10.1109/TEC.1959.5222697. (Cited on pages 140 and 144).

[PWKJ16] Vincent Palomares, David C. Wong, David J. Kuck, and William Jalby. Evaluating out-of-order engine limitations using uop flow simulation. In *Tools for High Performance Computing 2015*, pages 161–181. Springer International Publishing, 2016. doi:10.1007/978-3-319-39589-0_13. (Cited on page 40).

[QJP+07] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250662.1250709. (Cited on page 86).

[RC13] Guillem Rueda Cebollero. Learning cache replacement policies using register automata. Master's thesis, Uppsala University, Department of Information Technology, December 2013. URL: http://www.diva-portal.org/smash/get/diva2:678847/FULLTEXT01.pdf. (Cited on pages 112 and 116).

[Rei08] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, November 2008. URL: http://embedded.cs.uni-saarland.de/publications/DissertationCachesInWCETAnalysis.pdf. (Cited on pages 79 and 111).

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, November 2007. doi:10.1007/s11241-007-9032-3. (Cited on page 83).

[RHSJ94] June-Kyung Rho, G. D. Hachtel, F. Somenzi, and R. M. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(2):167–177, February 1994. doi:10.1109/43.259940. (Cited on pages 141, 145, and 151).

[RS93] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993. doi:10.1006/inco.1993.1021. (Cited on pages 132 and 134).

[RTHW14] T. Roehl, J. Treibig, G. Hager, and G. Wellein. Overhead analysis of performance counter measurements. In *43rd International Conference on Parallel Processing Workshops (ICCPW)*, pages 176–185, September 2014. doi:10.1109/ICPPW.2014.34. (Cited on page 35).

[Rub75] Frank Rubin. Worst case bounds for maximal compatible subsets. *IEEE Transactions on Computers*, C-24(8):830–831, August 1975. doi:10.1109/T-C.1975.224315. (Cited on page 145).

[Set11] Burr Settles. From theories to queries: Active learning in practice. In *Active Learning and Experimental Design workshop, in conjunction with AISTATS 2010*, volume 16 of *Proceedings of Machine Learning Research*, pages 1–18, Sardinia, Italy, 2011. PMLR. URL: http://proceedings.mlr.press/v16/settles11a.html. (Cited on page 118).

[SG09] Muzammil Shahbaz and Roland Groz. Inferring Mealy machines. In *Proceedings of the 2nd World Congress on Formal Methods*, pages 207–222. Springer, 2009. doi:10.1007/978-3-642-05089-3_14. (Cited on pages 17 and 118).

[SGL95] J.M. Sanchez, A.O. Garnica, and J. Lanchares. A genetic algorithm for reducing the number of states in incompletely specified finite state machines. *Microelectronics journal*, 26(5):463–470, 1995. doi:10.1016/0026-2692(95)98948-Q. (Cited on pages 141, 145, and 151).

[SK13] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM. doi:10.1145/2485922.2485963. (Cited on pages 40 and 79).

[SS95] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995. doi:10.1109/12.467697. (Cited on pages 86 and 111).

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html. (Cited on pages 149 and 150).

[TASS09] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 121–132, New York, NY, USA, 2009. ACM. doi:10.1145/1508244.1508259. (Cited on pages 113 and 114).

[THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the International Conference on Parallel Processing Workshops*, pages 207–216, Washington, DC, USA, 2010. IEEE. doi:10.1109/ICPPW.2010.38. (Cited on page 35).

[TJYD10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010. doi:10.1007/978-3-642-11261-4_11. (Cited on pages 22 and 35).

[TLM18] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *CoRR*, abs/1802.03802, 2018. arXiv:1802.03802. (Cited on page 114).

[TY00] Clark Thomborson and Yuanhua Yu. Measuring data cache and TLB parameters under Linux. In *Proceedings of the Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 383–390, July 2000. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.1427. (Cited on pages 86 and 111).

[VGGK20] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning replacement policies from hardware caches. In *Proceedings of the 41st ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI). Association for Computing Machinery, June 2020. To appear. (Cited on pages 90, 92, 105, 113, and 116).

[Vil15] Tiziano Villa. Personal communication, March 2015. (Cited on page 155).

[Vil19] Pepe Vila. Personal communication, December 2019. (Cited on page 109).

[VKBSV97] T. Villa, T. Kam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Explicit and implicit algorithms for binate covering problems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(7):677–691, July 1997. doi:10.1109/43.644030. (Cited on page 145).

[VKM19] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54, May 2019. doi:10.1109/SP.2019.00042. (Cited on pages 90 and 105).

[vSMK+20] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. https://cacheoutattack.com/, January 2020. (Cited on page 114).

[VSVA05] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Using language inference to verify omega-regular properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 45–60. Springer, 2005. doi:10.1007/978-3-540-31980-1_4. (Cited on page 118).

[W+08] Reinhard Wilhelm et al. The worst-case execution time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008. doi:10.1145/1347375.1347389. (Cited on page 79).

[Wika] Skylake (client) - Microarchitectures - Intel. URL: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client). (Cited on page 44).

[Wikb] Zen - Microarchitectures - AMD. URL: https://en.wikichip.org/wiki/amd/microarchitectures/zen. (Cited on page 46).

[WL06] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 473–482, Washington, DC, USA, December 2006. IEEE Computer Society. doi:10.1109/ACSAC.2006.20. (Cited on page 76).

[WLPA14] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, Volume I: User-level ISA, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html. (Cited on page 37).

[WM08] Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *2008 IEEE International Symposium on Workload Characterization*, pages 141–150. IEEE, September 2008. doi:10.1109/IISWC.2008.4636099. (Cited on page 33).

[Won13] Henry Wong. Intel Ivy Bridge cache replacement policy, January 2013. URL: http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/. (Cited on pages 86 and 112).

[WPSAM10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 235–246, March 2010. doi:10.1109/ISPASS.2010.5452013. (Cited on page 111).

[WTM13] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE, April 2013. doi:10.1109/ISPASS.2013.6557172. (Cited on page 33).

[XS20] Wenjie Xiong and Jakub Szefer. Leaking information through cache LRU states. *CoRR*, abs/1905.08348v2, January 2020. arXiv:1905.08348v2. (Cited on page 115).

[Yan91] Saeyang Yang. Logic synthesis and optimization benchmarks user guide, Version 3.0. Technical report, Mi-

croelectronics Center of North Carolina (MCNC), January 1991. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.591&rank=1. (Cited on page 154).

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, SEC '14, pages 719–732, USA, 2014. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom. (Cited on page 114).

[YGL+15] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. *Cryptology ePrint Archive, Report 2015/905*, 2015. URL: https://eprint.iacr.org/2015/905. (Cited on pages 81 and 82).

[YJS+06] Kamen Yotov, Sandra Jackson, Tyler Steele, Keshav Pingali, and Paul Stodghill. Automatic measurement of instruction cache capacity. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, pages 230–243. Springer, 2006. doi:10.1007/978-3-540-69330-7_16. (Cited on page 111).

[YPS05] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. In *SIGMETRICS*, pages 181–192, New York, NY, USA, 2005. ACM. doi:10.1145/1064212.1064233. (Cited on pages 86 and 111).

[ZBW17] M. Zeiser, J. Betz, and D. Westhoff. Cache covert-channel mitigation in cloud virtualization with XEN's credit scheduler. In *IEEE Global Communications Conference, GLOBECOM 2017, Singapore*, pages 1–7, December 2017. doi:10.1109/GLOCOM.2017.8253984. (Cited on page 110).

[ZGY14] Yi Zhang, Nan Guan, and Wang Yi. Understanding the dynamic caches on Intel processors: Methods and applications. In *Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing*, EUC '14, pages 58–64, USA, August 2014. IEEE Computer Society. doi:10.1109/EUC.2014.18. (Cited on page 112).

Index

µops, 45

active learning, 117, 118
AES instructions, 69
age graph, 92
Angluin's algorithm, 118, 145
assembler, 25

black box, 118
branch instructions, 58

cache, 79–116
  µop cache, 45
  associativity, 81
  block size, 81
  C-Box, 81
  cache set, 81
  congruent addresses, 82
  hash function, 81, 82
  hit, 81
  index, 81
  miss, 81
  organization, 81–82
  prefetching, 33
  slice, 81
  way, 81
cacheInfo, 86
cacheSeq, 88
chain instruction, 52
CLFLUSH instruction, 89, 110
closure, 148
CNF, 128
compatibility
  rows, 124, 129
  states, 143, 148
  words, 122
covert channel, 75, 110, 114
CPUID instruction, 32

decoder, 45
dependency chain, 52
division, 46, 58

encoding, 25
equivalence query, 118
error state, 126, 131
eviction set, 90
execution port, *see* port
explicit operand, 25

functional unit, 46

gray-box learning, 117–138

hardware performance counters, *see* performance counters
hyper-threading, 33, 75

IACA, 42, 63
implicit operand, 25, 62
incompatibility matrix, 147
initialization sequence, 89
instruction variant, 62
interference, 33
ISM benchmarks, 151

L* algorithm, 118, 145
latency
  algorithm, 52–59
  definition, 47
LFENCE instruction, 32
LLVM, 42
loop unrolling, 29

MCNC benchmarks, 151, 154
Mealy machine
  completely specified, 119, 142
  definition, 119, 142
  incompletely specified, 142
  minimization, 139–164
  serial composition, 118, 120
measurements, 62
MeMin, 139–164

memory operand, 25
micro-operation, 45
microbenchmark, 22
move elimination, 45

nanoBench, 21–37
  features, 25–32
  implementation, 32–34
  kernel mode, 29
  noMem mode, 30
  user mode, 29

observation table, 123
  closed, 125
  consistent, 124
  input-complete, 126
  p-closed, 125
  unique, 126
out-of-order execution, 46
output query, 118

partial solution, 130, 149
partition, 124, 128
  closed, 125
performance counters, 23–24
  APERF/MPERF, 24
  core, 24
  fixed-function, 24
  programmable, 24
  uncore, 24, 82
permutation policy, 82, 90
physically-contiguous memory, 34
pipe, 46
pipeline, 44–46
  AMD, 46
  Intel, 44–46
port, 46
  combination, 48
  usage
    algorithm, 49–51
    definition, 48

RDMSR instruction, 24
RDPMC instruction, 24
renamer, 45
reorder buffer, 45
replacement policy, 82
  LRU_3PLRU_4, 96
  $PLRU_8Rand_2$, 102
  $Rand_3PLRU_8$, 102
  adaptive, 86, 92
  FIFO, 82
  LRU, 82
  MRU, 83
  NRU, 83
  permutation policy, 82, 90
  PLRU, 82
  QLRU, 83, 96, 104
  RRIP, 83
reservation station, 46
reset sequence, 92, 110
reset state, 142, 150
right-equivalence, 120, 131

SAT, 128, 147
scheduler, 46
serializing instructions, 32
SHL instruction, 73
side channel, 75, 110, 114
SIMD, 54
status flags, 25, 56–58, 73

throughput
  algorithm, 59–61
  definition, 47

uops.info, 39–77

WBINVD instruction, 89, 109

x86
  assembler, 25
  instruction set, 61
XML file, 63

zero idiom, 45, 54, 72