Doctoral theses at NTNU, 2024:52

Björn Gottschall

Time-Proportional Performance Analysis for Out-of-Order Processors

Doctoral thesis

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of Philosophiae Doctor
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of Science and Technology

Björn Gottschall

Time-Proportional Performance Analysis for Out-of-Order Processors

Thesis for the Degree of Philosophiae Doctor

Trondheim, March 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Information Technology and Electrical Engineering
Department of Computer Science

ISBN 978-82-326-7606-4 (printed ver.)
ISBN 978-82-326-7605-7 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

Doctoral theses at NTNU, 2024:52

Printed by NTNU Grafisk senter

Abstract

The quest for an increase in processor performance has become difficult due to the inherent power limitations of today's chips. As processors become more complex with deeper and wider pipelines, out-of-order execution, and the integration of heterogeneous accelerators, software developers face increasing challenges in utilizing these resources efficiently. Therefore, understanding the performance characteristics of our workloads running on these complex architectures has never been more important to enable optimization that increases efficiency and performance. In this thesis, we present three contributions that collectively answer the two fundamental questions of performance analysis for out-of-order processors by explaining what an application spends time on and why.

Our first contribution is *TIP: Time-Proportional Instruction Profiling*, which establishes the time-proportional principle of performance analysis and identifies the four states of commitment that a time-proportional performance analyzer must be able to differentiate. Time-proportional instruction profiling is able to attribute execution time to instruction accurately, unlike contemporary performance profilers, which are not time-proportional.

Our second contribution is *TEA: Time-Proportional Event Analysis*. TEA combines time-proportional instruction profiling with accurate performance event attribution, thereby explaining why certain instructions are performance-critical.

The evaluation of the accuracy of performance profilers was made possible through *TraceDoctor*, which is our third key contribution. TraceDoctor is a high-performance tracing framework that enabled the creation of a golden performance reference and proved its flexibility by enabling the evaluation of accuracy and overhead in sampled simulations.

To demonstrate the potential of time-proportional performance analysis, we used it to optimize the industry-standard SPEC CPU2017 benchmarks Imagick, lbm, and nab, and achieved a speedup of 1.93, 1.28, and 2.45 times, respectively. Contemporary performance profilers, such as Intel PEBS, AMD IBS, Arm SPE, and IBM RIS are not time-proportional and hence do not clearly identify these optimization opportunities.

Structure of Thesis

This thesis is a collection of papers organized into two parts.

Part I provides an overview of the research contributions and introduces the most relevant background.

Part II includes the published papers. For improved readability, the format of the papers has been altered from their original published forms, but their content remains unchanged. The three papers included in this thesis are listed below.

Paper A. **TIP: Time-Proportional Instruction Profiling**
Björn Gottschall, Lieven Eeckhout, Magnus Jahre
54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2021
Best Paper Runner-Up Award, HiPEAC Paper Award

Paper B. **TEA: Time-Proportional Event Analysis**
Björn Gottschall, Lieven Eeckhout, Magnus Jahre
50th Annual International Symposium on Computer Architecture (ISCA), 2023
Nominated for Best Paper Award

Paper C. **Balancing Accuracy and Evaluation Overhead in Simulation Point Selection**
Björn Gottschall, Silvio Campelo de Santana, Magnus Jahre
2023 IEEE International Symposium on Workload Characterization (IISWC), 2023
Nominated for Best Paper Award

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Professor Magnus Jahre, for his invaluable guidance and support throughout the years of my doctoral studies. Thank you for always keeping your door open for me to seek help and advice. I also feel very fortunate to have had the opportunity to work and collaborate with Professor Lieven Eeckhout, who consistently encouraged me to seek out the missing pieces in my research.

A big round of applause goes to the NTNU Computer Architecture Lab and all the amazing people with our lunch sessions, presentations, and discussions. There is always someone listening to you, bouncing off ideas, or helping out in tricky situations. If it were not for you, my workplace would not have been as amazing as it was.

Over the years I have been fortunate to make some wonderful friends, whom I am incredibly thankful for. They have supported me in countless ways, and we have enjoyed many activities together. Whether it has been payday beers, hiking, skiing, going on cabin trips, or brewing beer, I have always had a great time with you.

And finally, I am deeply grateful to my wife, children, and family for their endless patience, love, and support during this intense period of my life. I am coming home now.

Contents

Part I

Research Overview

Chapter 1

Introduction

In today's world, computers, cloud services, and mobile and embedded devices are ubiquitous. Increasing computing power and efficiency have become the fundamental driving forces of computer architecture, catering to the ever-growing demand for higher performance and less energy consumption. Thanks to Moore's Law, technology has advanced tremendously over the last few decades, which has helped scale integrated circuits' performance [1]. However, shrinking transistors becomes increasingly difficult, requiring to manufacture larger chips to fit more transistors. Current EUV lithography equipment, used for the most advanced silicon technologies, limits the maximum chip area, which is already exhausted with today's high-performance graphic accelerators, meaning they cannot grow any bigger. A solution that the chip industry is adopting is the use of chiplets [2], which allows for increased transistor counts by integrating multiple chips on a single interposer.

Increasing the clock frequency was another solution for many years to improve the performance of integrated circuits, which has mostly stopped due to the breakdown of Dennard scaling [3]. In the past, scaling voltages, currents, and clock frequencies were mostly free as transistors became smaller. However, the shrinkage of transistors has resulted in leakage currents dominating the energy consumed by chips, leading to power densities that strain cooling systems [4]. As clock frequencies and silicon size have increased, chips have become power-limited. This started the era of dark silicon, where it is impossible to power and use all chip resources due to power constraints and cooling limitations [5]. This means that during times of high energy usage, certain areas of the chip may need to be power or clock-gated, making some resources temporarily unavailable.

With new and improved semiconductor technologies, multi-core design, and heterogeneous architectures, the number of transistors on a chip is still increasing. Power constraints do not allow the use of all of these resources simultaneously, which means wasting power is, in effect, wasting performance. Performance analysis helps understand application behavior, optimize workloads to achieve maximum hardware and software performance, and utilize the chip resources as efficiently as possible. As processors become increasingly complex, the task of *performance analysis* — to understand an application's performance characteristics — becomes increasingly challenging and more important.

1.1 Research Scope

This thesis focuses on performance analysis, which must answer two fundamental questions when profiling an application on a specific architecture.

Question 1: What takes time? Q1

Software consists of smaller components such as functions, basic blocks, and instructions, and their execution takes a certain amount of time. A performance profiler aims to answer the question of how long the different parts of an application take to execute.

Question 2: Why does it take time? Q2

Application performance can be affected by performance events to which instructions may be subjected. Performance analysis must identify the instructions and events that impacted performance to explain their execution times.

Answering the first question requires identifying the *performance-critical instructions* within an application, which are the instructions that impact the execution time the most. Answering the second question allows connecting the performance-critical instructions to the events that are the cause of their execution time and thereby explain why they are performance-critical.

By answering both questions, a developer can optimize and change the application to improve performance, waste less resources, and save energy. A key challenge in performance analysis is achieving high accuracy since a performance profiler that cannot accurately point out performance problems in an application is not helpful in the endeavor of optimization. Inaccurate performance profiles mislead developers to focus on unimportant parts of an application, missing optimization potential.

In this thesis, we will reveal and explain why today's performance profilers are inaccurate and propose time-proportional performance analysis as the accurate solution to this problem. Time-proportional performance analysis answers both Q1 and Q2 with a very low error and overhead, revealing the performance-critical instructions in an application and explaining why they exhibit such a long execution time. We focus on single-threaded applications in this thesis because this is a necessary first step, as performance analysis cannot be accurate for multi-threaded applications without being accurate at the level of a single thread.

1.2 Time-Proportional Performance Analysis

Our core contribution of this thesis is time-proportional performance analysis, which combines accurate time and event profiling, revealing the performance-critical instructions of an application and explaining their architectural bottlenecks. This allows for quickly identifying problems and more successful software optimizations that squeeze most performance out of today's hardware. Time-proportional performance analysis adheres to the time-proportional principle (D1), which states that the probability of sampling an instruction must be proportional to its contribution to overall runtime. We use statistical sampling because it can be used on any application without prior knowledge or changes to the application, and it exhibits low overheads that could impact the application's performance. We found that all existing profilers such as Intel PEBS, AMD IBS, Arm SPE, IBM RIS, Lynsyn, Xilinx TCF, Linux perf, OProfile, GNU gprof, and PPerf are inaccurate because they are not time-proportional, which is explained in more detail in Section 2.3 and Section 2.4.

Time-proportional performance analysis focuses on the commit stage of the processor core, which is the point at which instructions retire after successful execution. Thus, it is the place where instruction latencies in a core are exposed. For example, a stalling core means that an instruction has reached the commit stage but has not finished execution yet and thus cannot retire. It is then this instruction that is responsible for the time the core cannot progress further. A key observation that time-proportional performance

Definition 1: Time-Proportionality D1

To accurately distribute the execution time of the instructions of an application using statistical sampling, the probability of sampling an instruction must be proportional to its contribution to overall runtime.

Commit State 1: Computing C1

In the computing commit state, one or more instructions are committing, and the core is advancing through the application. When profiling during this state, a profiler must account time to all currently committing instructions equally.

Commit State 2: Stalled C2

In the stalled state, exactly one instruction is stalling the commit stage and preventing the core from making progress, for example, a long latency load. A profiler profiling during this state must account time to the stalling instruction.

Commit State 3: Flushed C3

In the flushed state, no instructions are currently executing because a previously retired instruction has flushed the cores pipeline, for example, in the case of a mispredicted branch. When profiling during this state, a profiler must account time to the instruction that caused the flush.

Commit State 4: Drained C4

In the drained state, the core ran out of instructions to execute, for example, an instruction cache miss in the frontend. When profiling during this state, a profiler must account time to the instruction that comes next to execute.

analysis made is that a processor core can only be in one of four commit states: Computing (C1), Stalled (C2), Flushed (C3), or Drained (C4).

Time-proportional performance analysis can differentiate all four commit states and retrieve the instructions responsible for runtime. When the processor is in the computing commit state, it will retrieve all instructions that commit in the cycle the sample is taken. If the processor is in the stalled commit state, it will retrieve the oldest stalling instruction as the sample, since it is the instruction that prevents the core from making progress. In the flushed commit state, a previously committed instruction has flushed the pipeline, and it is in this cycle this last committed instruction that is to be blamed and, therefore, sampled. In case the core runs out of instructions, for example, when the frontend cannot supply enough instructions to the backend because of an instruction cache miss, commit is in the drained state, and the first instruction that enters the pipeline will be sampled. Time-proportional performance analysis retrieves additionally the performance events that the instructions were subjected to. This makes it possible to attribute execution time to instructions accurately and explain why they exhibit a longer latency during their execution by exposing the architectural bottlenecks.

We evaluate the accuracy of our time-proportional performance analyzer and contemporary performance analysis approaches by comparing them to a golden reference that accounts for all dynamic instruction, clock cycles, and performance events.

With time-proportional performance analysis, we profiled industry-standard benchmarks from SPEC CPU2017 [6] and PARSEC [7] executed end-to-end using reference input sets in full-stack Linux environments. While contemporary performance profilers missed many optimization potentials, with time-proportional performance analysis, we were able to identify the performance problems and optimize the following benchmarks.

- **SPEC CPU2017 Imagick.** Time-proportional profiling identified flushing instructions responsible for nearly half of the runtime. These flushing instructions turned out to be necessary in this benchmark, and once we removed them, the benchmark sped up by 1.93 times.

- **SPEC CPU2017 lbm.** Time-proportional profiling identified one load instruction to be the performance-critical instruction due to numerous non-hidden cache misses. We employed software prefetching to load the data ahead of time and sped up this benchmark by 1.28 times.

- **SPEC CPU2017 nab.** Time-proportional profiling identified a floating point instruction as performance-critical that follows a flushing instruction. We recompiled the benchmark with a relaxed IEEE 754 floating-point standard, which increased the floating-point pipeline utilization by avoiding the flushing instruction. This sped up the benchmark by 2.45 times.

1.3 Thesis Contributions

The contributions of this thesis are framed within three papers.

In Paper A we present *TIP: Time-Proportional Instruction Profiling*, which introduces the principle of time proportionality (D1) and the four commit states (C1, C2, C3, and C4). TIP explains on which instructions the application spends time on and with that answers Q1. TIP is an instruction-level performance profiler that adds hardware support inside the processor core for time-proportionally retrieving the addresses of the instructions that the out-of-order processor is exposing the latency of in the cycle the sample is taken.

In Paper B we present *TEA: Time-Proportional Event Analysis*, which extends the time-proportional instruction profiling with accurate performance event attribution. TEA answers both Q1 and Q2 by identifying which instructions take time in an application using TIP's attribution policies, and attributes the time accurately to performance events that the instructions were subjected to. Therefore, TEA adds hardware support to track performance events for every in-flight instruction within the processor pipeline and time-proportionally retrieve the instruction addresses and performance events when sampling.

To quantify the accuracy of TIP and TEA, we needed to create a golden reference that captures all clock cycles, dynamic instruction, and performance events. To enable this, I implemented TraceDoctor, a very flexible and high-performance tracing interface for cycle-accurate simulations, which is the first to allow end-to-end tracing of industry-standard benchmarks in full-stack environments. TraceDoctor is the first contribution from Paper C, and to demonstrate that TraceDoctor is not limited to performance profiling and event analysis, we used it to analyze the accuracy and overhead sampled simulations using SimPoint [8]. Sampled simulation is an often-used methodology in computer architecture research to shorten the simulation overhead by focusing on representative parts of an application. In the second contribution of Paper C, we analyzed for the first time how accurate sampled simulations are when executing large-scale workloads to completion on high-detail architecture models.

We found that sampled simulations can accurately represent benchmark performance when collecting relatively many and relatively small simulation points over the whole benchmark execution. Since the simulation overhead increases with the simulation point size and the number of simulation points, it must be balanced with the overhead of the microarchitectural state that needs to be collected and warmed up for each simulation point. By selecting sensible parameters, it is possible to simulate state-of-the-art benchmarks with high accuracy and a low simulation overhead, while common practice often achieves the opposite.

Chapter 2

Background

Performance profiling has been around for a long time, and many tools exist that promise to help a developer understand the performance characteristics of software. Performance profiling tools, as shown in Figure 2.1, often rely on sampling, instrumentation, or tracing. Each of these techniques provides different depths of information about the application and its execution (information capabilities), with the tradeoff on performance and profiling overhead.

A problem with contemporary performance profilers is that the tools that aim to collect the same type of performance-related data often show vastly different results when profiling the same application on the same machine. The reason is that each tool exhibits a different systematic bias that affects its profiling results and is explained in more detail later. These biases can even lead to some tools showing big variations across several of the same profiling runs, leaving developers clueless about what can be trusted.

This chapter gives an overview of performance analysis and explains statistical sampling in detail. Finally, the chapter describes how today's performance profilers and other approaches work to understand why they fall short in accurate performance analysis.

2.1 Performance Analysis

The first step in performance analysis is to figure out what takes time in an application, which is surprisingly challenging since it requires high accuracy. The easiest method of timing the execution of an application is to take the difference between the start and end times. However, this is not helpful for optimizing big and complex applications, which requires identifying and

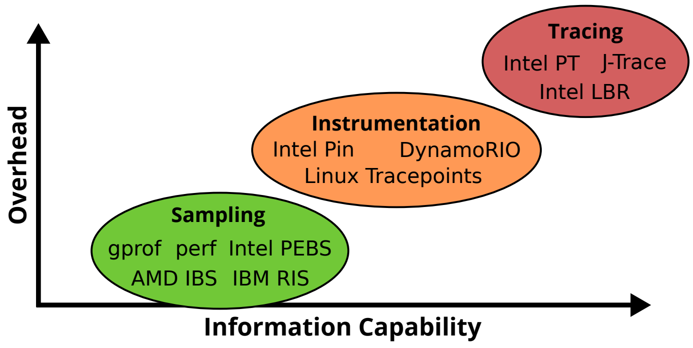


Figure 2.1: The three types of performance profiling with examples are compared based on their overheads and information capabilities. Sampling has the least impact on application execution, storage, and manual effort than tracing and instrumentation.

pinpointing performance bottlenecks towards smaller components such as functions, basic blocks, or instructions. Measuring the performance of these smaller components is not straightforward and can lead to high overheads and profiling errors.

Performance profiling that can give more insights into the execution of an application can be divided into three types, as shown in Figure 2.1, each having advantages and disadvantages. This thesis focuses on sampling as it is — when applied accurately — a good starting point for performance analysis due to its low overhead and wide applicability. A low overhead is very important in performance profiling, as every bit of impact on the system's performance through the technique skews the data one intends to gather.

2.1.1 Sampling

Sampling is the most widespread technique used for performance profiling, which periodically retrieves some information about the executing application. The obtained data can represent the performance characteristics of the whole application execution. However, this is only achieved when the collected samples represent a statistical distribution that is free from systematic biases over the execution of an application and is discussed in more detail in Section 2.2. Sampling can be applied to any application without prior knowledge and does not require any modifications to the application. Due to its low overhead, sampling can achieve low profiling errors and profile full application executions.

One limitation of sampling is scenarios in which the application runs only for a very short time, and the collected samples are not sufficient to accurately represent the application's performance. One example is Functions-as-a-Service (FaaS), where applications typically execute one relatively short task, but are invocated many many times. Short and complex applications can still be analyzed using sampling by scaling up their workloads and runtime to gather sufficient samples. If the executed code is short enough, applying other profiling techniques, such as instrumentation, might also be trivial.

Since sampling relies on statistics, it is crucial to retrieve the profiling data through random sampling because small biases during this process are amplified and will lead to big errors in the performance profile. Unfortunately, as this thesis will show, all contemporary performance profilers that employ sampling, such as Linux perf [9], Intel Processor Event Based Sampling (PEBS) [10], AMD Instruction Based Sampling (IBS) [11], Arm Statistical Profiling Extension (SPE) [12] and IBM Random Instruction Sampling (RIS) [13] suffer from systematic errors.

2.1.2 Instrumentation

Instrumentation inserts instructions into an application and allows for gathering information about its execution. There are two types of instrumentation: static instrumentation is adding additional code into the application's source either manually or from within the compiler, and dynamic instrumentation injects instructions into an already compiled application. Dynamic instrumentation is used by tools such as Intel Pin [14], DynamoRIO [15] or Linux tracepoints [16], whereas static instrumentation is often a manual effort by the developer or is applied from compiler frameworks such as GCC [17], [18] or LLVM [19].

Static instrumentation has the benefit that the developer can target the points of interest inside the application, while dynamic instrumentation can gather data about the execution of an application without the need for the source code. This, however, comes with the disadvantage that the instrumentation code must be specifically targeted within the application to avoid the creation of high overheads.

For example, adding code into very tight loops can create a bias that introduces additional performance problems and mislead any optimization efforts. To avoid this, a performance analyst must have already acquired insights about the performance characteristics of an application before instrumentation can be employed.

2.1.3 Tracing

Tracing is a technique that records very fine-grained details about the execution of an application. This can be, for example, the exact order of instructions executed or functions called and is often coupled with more information like memory accesses or call stacks. Tracing allows for reconstructing an application's execution and, with additional architectural state, provides insights into the exact workings of the application.

Tracers such as Segger J-Trace [20] are popular on embedded platforms, which, through their relatively low performance, create a manageable amount of tracing data that can be recorded out-of-band, in parallel to the application execution and, thus, without performance impact on the target platform. In high-performance processors, tracing can significantly reduce the application performance due to the large amount of generated data. As a result, tracing affects the performance data collected, making it unrepresentative of non-traced executions. Therefore, tracers such as Intel Last Branch Record (LBR) and Intel Processor Trace (PT) [21] must typically be limited to small timeframes. Consequently, tracing is not an adequate approach for performance analysis when it comes to large and complex workloads running on high-performance processors.

2.1.4 Event Analysis

Modern architectures have grown exceedingly complex and use many strategies, such as branch predictors or caches, to gain performance in software execution. As a result of this situation, it has become more challenging to analyze the performance of an application and reason why instructions take longer than others. Performance events play an important role in such architectures in order to explain why certain instructions experience a longer execution latency. Performance event profilers such as Intel VTune [22], AMD µProf [23], and Linux perf [24] are able to sample event counts over the complete execution of an application[1]. Other event profilers such as Intel PEBS [10], AMD IBS [11], ARM SPE [12] or IBM RIS [13] can record performance events that specific instructions are subjected to. These performance event profiles then aim to explain the architectural bottlenecks of an application.

While performance events are useful for explaining performance characteristics in simpler architectures, they are more difficult to make sense of on deeper and wider out-of-order processor pipelines that are designed to hide such events. To better utilize core and memory resources, out-of-order pipelines execute many instructions in parallel and, as the name implies, possibly out-of-order. With the parallel execution of instructions, performance events will overlap, one hiding the latency of another. For example, load instructions often experience hidden and non-hidden cache misses, but hidden cache misses do not directly impact the application's performance.

To account for overlapping effects of performance events, it is often combined with performance profiling, which is called performance analysis in this thesis, and allows — when done time proportionally — for an accurate performance profile that exposes the instructions that take time together with the performance events that caused them to take time.

2.2 Statistical Sampling

To profile an application's execution, data must be retrieved to represent its performance characteristics. To minimize the impact on the performance of an application, statistical sampling is used since it can keep the overhead in check. Statistical sampling intermittently retrieves performance data from the target application. In most cases, it retrieves the currently active or executing instruction. It can also retrieve other kinds of data like, for example, performance event information that helps to reason about architectural bottlenecks.

From the application's perspective, instructions are executed in order, one after another. Figure 2.2 shows such an example of an instruction stream in which each instruction exposes a certain latency. When applying statistical

[1]Event counts are attributed to the instructions of an application through statistical sampling, that creates an event distribution over the executed instruction. Instructions that experience more events are more likely to be sampled and represented in this distribution.

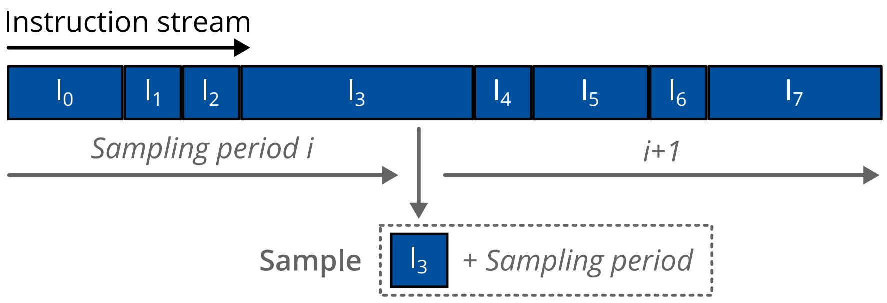


Figure 2.2: Statistical sampling periodically retrieves the currently executing instruction, which is taken as the representative of the sampling period. The collection of samples is used to create a runtime distribution over the executing application.

sampling, an instruction is retrieved after each sampling period and taken as representative of it. By sampling executions for longer, a collection of samples is recorded, which is then used to create a distribution over the full execution. Periodic time-based sampling is typically used in performance profiling, which uses a small timeframe or, as used in Linux perf, a sampling frequency to collect samples periodically. Thus, instructions that execute more often or experience higher execution latencies will collect more samples and are attributed with more sampling periods. This runtime distribution then represents the performance profile showing which instruction took longer in the application's execution and which did not.

When a different metric is used as the sampling period, it is possible to use statistical sampling to record other distributions over an execution. For example, a distribution of cache misses of an application is gathered when using a certain number of cache misses as the sampling period. In this case, instructions that are causing more cache misses are more likely to be sampled. This thesis will solely focus on time-based statistical sampling as any other sampling technique cannot be time-proportional [25]. For example, event sampling is suffering from overlapping effects in high-performance processors that cause the latency of some events to be hidden and, thus, their occurrence to have no impact on time.

The sampling period for statistical sampling can itself introduce a bias in the collected distribution when the period aligns with reoccurring patterns in the application execution. One way to avoid this bias is to sample with a fully random sampling period. However, when using a fixed timeframe as the sampling period, it is very unlikely that the sampling aligns with any execution patterns since the execution time inherently varies through memory

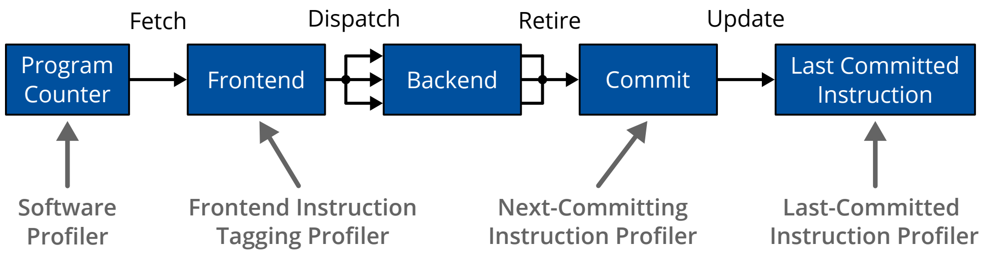

Fetch
Dispatch
Retire
Update
Program Counter
Frontend
Backend
Commit
Last Committed Instruction
Software Profiler
Frontend Instruction Tagging Profiler
Next-Committing Instruction Profiler
Last-Committed Instruction Profiler

Figure 2.3: Performance profilers sample instructions from different stages of the processor pipeline, but are unable to accurately attribute execution time to instructions.

operations and external factors such as interrupts.

Time-proportional statistical sampling is the key to performance profiling because an accurate performance profile that is supposed to be representative of the overall execution time, must be collected in a time-proportional manner. To create such an accurate performance profile, every instruction must be represented proportional to its contribution to overall application execution time (D1). Statistical sampling only infrequently retrieves the currently executing instruction, and the probability of sampling this instruction must be time-proportional. Equation 2.1 computes the sampling probability p of instruction I_x by relating the time the instruction contributes to the overall runtime t_{total} of the application:

$$p(I_x) = \frac{t(I_x)}{t_{total}} \tag{2.1}$$

Statistical sampling is limited to applications and workloads that accumulate sufficient samples to represent the overall distribution of runtime. Applications with a very short execution time must scale up their workloads or be sampled more frequently using shorter sampling periods. Fortunately, in performance profiling, the interest is focused on instructions that execute more often or take longer to execute and are, thus, more likely to be sampled.

2.3 Software Performance Profilers

Software performance profilers are well known and have the benefit that they work almost on all systems independent from the hardware. There are many different software performance profilers, and while their inner workings differ, they all have in common that they periodically sample the program counter of the running application. To achieve this, they interrupt the currently running application using a timer and then record its program counter.

GNU Gprof [17], LLVM PGO [19], and Google performance tools [26] work very similarly by setting a signal handler within the application and setting up a periodic timer interrupt that issues the signal to the application. Within the signal handler, both then retrieve the return address from the stack, which is the program counter of the application at which execution resumes. While also using a periodic timer signal, PPerf [27] uses the Linux ptrace API to control the process, allowing it to profile multi-threaded applications and record the program counters from the threads control blocks. OProfile [28] and Linux perf [24] are performance profilers embedded in the Linux kernel and can profit from a low overhead by avoiding unnecessary context switches. When the regular timer interrupt occurs, they can record the program counters directly from the process control blocks inside the kernel. Unlike the userspace software performance profilers, kernel-based performance profilers are also able to profile kernel code and multi-core systems running multi-threaded applications. While Linux perf is also used in the later described hardware-assisted performance profilers (Section 2.4), it uses software-based performance profiling by default.

Software performance profilers exhibit a bias called skid [29], that makes them very inaccurate when it comes to performance profiling. The time-proportional principle (D1) states that the probability of sampling an instruction must be proportional to the instruction's contribution to overall runtime. Figure 2.3 illustrates why this is not the case for a software performance profiler. All software performance profilers rely on interrupts to record a sample. An interrupt diverts the frontend of a core to the address of the interrupt handler, while the instructions queued in the backend, which can be hundreds, still finish execution. The program counter that is then recorded is the address of the instruction that execution is resuming after the sample is recorded. The problem that software performance profilers cannot overcome is that the instruction they sample is possibly hundreds of instructions away from when the interrupt was issued, and the instruction they sample is not even close to executing. This means software performance profilers are unable to distinguish any of the commit states as shown in Table 2.1 and are, therefore, not time-proportional.

Table 2.1: Overview of the commit states covered by different profilers. Unlike TIP and TEA, other profilers systematically misattribute time by being unable to detect all commit states.

Commit State	Software	Frontend Instruction Tagging	Next Committed Instruction	Last Committed Instruction	TIP & TEA
Computing (C1)	✗	✗	✗	✗	✓
Stalled (C2)	✗	✗	✓	✗	✓
Flushed (C3)	✗	✗	✗	✓	✓
Drained (C4)	✗	✓	✓	✗	✓

2.4 Hardware-Assisted Performance Profilers

Performance profiling can benefit from hardware support inside the processor core to record performance data during the execution of instructions without interference. Many profilers have the ability to capture additional data related to an instruction and store it in designated registers. This information can then be accessed through the profiling software by interrupting the execution, or it can be saved to a memory buffer [21].

Hardware-assisted performance profiling can achieve highly accurate results by respecting the time proportionality principle. Unfortunately, current hardware-assisted profilers produce inaccurate performance profiles due to systematic biases that cause them to sample an instruction other than the one that the processor is exposing the latency of.

- Frontend Instruction Tagging Profilers — Section 2.4.1
  Profilers such as AMD IBS, Arm SPE, and IBM RIS periodically mark instructions in the frontend and sample them when they commit. By doing so, they disregard the computing, stalled, and flushed commit states, and misattribute time to the wrong instructions.

- Next-Committing Instruction Profilers — Section 2.4.2
  Profilers such as Intel PEBS periodically retrieve the next committing instruction when sampling, ignoring the computing and flushed commit states, and misattributing time from instructions, such as mispredicted branches.

- Last-Committed Instruction Profilers — Section 2.4.3
  Profilers such as Xilinx TCF and Lynsyn periodically sample the last committed instruction, ignoring the computing, stalled, and drained

commit states, which causes them to misattribute time from instructions, such as long latency loads.

2.4.1 Frontend Instruction Tagging Profilers

Frontend instruction tagging profilers periodically tag an instruction in the frontend and follow it through its execution in the backend until it commits. Once it commits, the recorded information is stored in designated registers, ready to be retrieved from the profiling software.

IBM RIS [13] is tagging instructions after they are fetched, Arm SPE [12] is tagging instructions at dispatch, and AMD IBS [11] can tag instructions either at fetch or dispatch. They can record additional information about the instruction, such as which latency is exhibited during execution, what performance events it is subjected to, or even things like the data address for load instructions. These profilers have all in common: they provide one set of designated registers that hold the sampling data for a tagged instruction, and with that, they are the only contemporary profilers that provide instruction sampling combined with accurate event attribution.

While the event attribution of RIS, SPE, and IBS is accurate in the sense that the sampled instruction has, in fact, experienced these events, the selected instructions are not representative of their contribution to overall runtime. Similar to software-based performance profilers, frontend instruction tagging profilers decide which instruction to sample in the frontend and thus before it has executed. Therefore, instructions accumulate samples and with that time irrespective of their actual execution latency. Unlike time-proportional profiling, frontend instruction tagging profilers are only able to identify the draining commit state (C4) out of the four commit states (see Table 2.1).

2.4.2 Next-Committing Instruction Profilers

Next-committing instruction profilers periodically sample the instruction that is next to finish execution and commit. This instruction is saved to designated sampling registers that are then retrieved by the profiling software. One of the limitations of these profilers is that they can only gather limited additional information, such as performance events, about the sampled instruction. This is because, in most cases, by the time the decision is made on which instruction to sample, it is already in execution or has fully executed.

Intel PEBS [10] is such a next-committing instruction profiler that is also able to retrieve specific events with the sampled instructions. When profiling events using Intel PEBS it is limited to only one event at a time, and its sampling period must be configured to a certain number of occurrences of this event. By not using time for the sampling period, the resulting performance profile will not represent a time distribution over the instructions but rather an event distribution. For example, Intel PEBS can precisely capture cache missing instructions by setting it up to sample an instruction every nth cache miss, and the resulting profile will show which instructions are missing more or less in the cache. However, this profile will not have any direct correlation to execution time or performance, as cache misses can be hidden through overlapping execution in the out-of-order pipeline.

As Figure 2.3 illustrates, next-committing instruction profilers retrieve the instructions from the commit stage and, therefore, should be much better at identifying the correct commit states. However, they also experience a bias imposed by their fixed sampling policy of always choosing the next committing instruction and are not able to detect two of the four commit states (see Table 2.1). The first is the computing commit state (C1), in which the processor can commit more than one instruction in a cycle. This state requires that all committing instructions are retrieved to be time-proportional. A next committing instruction profiler only retrieves one instruction and is, thus, not able to cover the computing commit state. The second missed commit state is the flushed commit state, in which no instructions are available in the pipeline to be sampled. This profiler will then choose the instruction that is next to commit and, with that, ignores the flushed commit state. For example, after a mispredicted branch retires, the pipeline will be flushed, and when a next-committing instruction profiler takes a sample during that time, it will sample the next instruction to come instead of the mispredicted branch that was responsible. Ignoring these two commit states violates the time-proportional principle (D1) since the probability of sampling an instruction is no longer proportional to the instruction's contribution to overall runtime.

2.4.3 Last-Committed Instruction Profilers

Last-committed instruction profilers periodically sample the instruction that was last committed. Unlike the other hardware-assisted performance profilers, these make use of external hardware to collect and analyze the data out-of-band. Since these profilers have only access to the program counter of the last committed instruction, they are not able to record any other information, such as performance events.

Xilinx Target Communication Framework (TCF) [30], ULINKplus [31], and Lynsyn [32] are last-committed instruction profilers that make use of the exact same technique to access a special program counter debug register on Arm platforms via JTAG. This program counter debug register is updated every time an instruction is committed; thus, it always holds the address to the last committed instruction. In contrast to other profiling techniques that rely on interrupts for data retrieval, these registers are read over an external debug interface, which is non-intrusive and does not interfere with the target platform's performance.

While non-intrusive statistical sampling can reduce the error of performance profiling compared to non-profiled executions, last-committed instruction profilers have a significant systematic bias by overly sampling instructions that do not contribute much to runtime. The reason is that the last-committed instruction register is not updated whenever an instruction stalls (e.g., a cache missing load). The probability of accounting time to the last-committed instruction increases, even though it is not responsible for the stall. That violates the time-proportional principle (D1) by overly representing instructions that execute before long latency instructions. By sampling the last committed instruction, these profilers can correctly identify only the flushed commit state (C3) out of the four commit states (see Table 2.1).

2.5 Other Approaches to Performance Analysis

Performance event profiling, as provided by tools like Linux perf [24], Intel VTune [22] or AMD µProf [23], is a common technique used in performance analysis, that samples selected performance events over the execution of an application. Although performance events can indicate a bottleneck in an application, they fall short in several fundamental ways. Firstly, a developer must choose the events that are likely to be problematic. Secondly, performance events do not accurately reflect performance loss, as modern out-of-order architectures are designed to hide latencies. BayesPerf [33] tries to minimize event sampling error when sampling bigger event sets that require multiplexing, but unlike the approaches presented in this thesis, cannot infer actual performance data from event counts.

Approaches exist that try to combine performance profiling with event analysis. Intel's top-down model [34] first profiles an application to find out where most of the time is spent and then iteratively drills down to which events might be the cause for that. Apart from the time it takes to profile an application many times iteratively, the top-down model suffers from the systematic bias that is described for next-committing instruction profilers in Section 2.4 and is also not able to accurately attribute execution time to the performance events it identifies. DCPI [35] does time and event profiling on the older Alpha architecture that allows, unlike other software-based profiling approaches, to actually interrupt the currently executing instruction and, thus, can more accurately measure the runtime of instructions. While it is also able to retrieve event counters with every sample, it cannot attribute time to the events that matter for performance. On modern, more complex architectures, DCPI would perform similarly to regular software-based profilers and thus suffer from high errors.

Interval analysis [36] can be used to create Cycles-per-Instruction (CPI) stacks from the dispatch stage, allowing to reason about performance bottlenecks an application experiences on an out-of-order processor. Unlike the Per-Instruction Cycle Stacks (PICS) that TEA produces with negligible overhead, interval analysis incurs higher overheads as the CPI stack granularity becomes finer. In practice, it is unattainable to use interval analysis to sample every single cycle as required for PICS. Therefore, interval analysis is unable to provide instruction-level performance analysis that can pinpoint performance problems within an application.

Xu et al. [29] argue for regular software performance profiling and propose a mathematical model to correct the error. In this thesis, we have shown that software performance profilers are highly inaccurate by blaming instructions for execution time that have not been executed. To achieve low profiling errors, hardware support for performance profiling is critical, and hence, developers should make use of this feature when it is available.

Chapter 3

Conclusion and Future Work

This thesis presented time-proportional performance analysis for out-of-order processors that allows for highly accurate performance and event profiling of applications. This chapter will summarize the contributions and provide an outlook on future work that is foreseen in the area of performance profiling with a special focus on time-proportionality.

3.1 Conclusion

Time-proportional performance analysis makes use of the time-proportional principle, which attributes runtime to instructions in proportion to their impact on overall runtime. In addition to that, time-proportional performance event analysis records the events an instruction is subjected to during execution and exposes them to the profiler. With these instruments in place, time-proportional performance analysis identifies the instructions that are performance-critical in an application and explains the reasons why they are performance-critical. This creates actionable performance profiles that enable developers to optimize and resolve software and hardware bottlenecks. Time-proportional performance analysis made detecting and resolving performance problems possible, enabling us to speed up the SPEC CPU2017 benchmarks Imagick, lbm and nab by 1.93, 1.28 and 2.45 times respectively. Other contemporary performance profilers are not time-proportional and missed these optimization opportunities.

TraceDoctor is the crucial enabler of all contributions in this thesis as it is the first tracing interface that allows for end-to-end tracing of industry-standard benchmarks and with that made it possible to create a golden reference for performance analysis. Its flexible and highly parallel design eased the analysis and comparison of performance profilers on an unprecedented scale.

We demonstrated TraceDoctor's versatility outside of evaluating performance profilers with a first-time analysis of the overhead and accuracy of sampled simulations on large-scale workloads. We confirmed that sampled simulation can be accurate and reduce the simulation overhead significantly when using relatively many and relatively small simulation intervals captured over the execution of the whole benchmark.

3.2 Future Work

The domain of performance profiling and analysis holds numerous research topics for the future. Software and hardware continuously grow more complex, and performance analysis must keep up with this trend. Profiling tools must continue to support the engineer and developer in optimizing software and hardware.

3.2.1 Multi-Threaded Workloads

The thesis did not give much attention to multi-threaded workloads and multi-core processors, which requires further exploration. Performance profilers and profiling circuitry are multi-core agnostics, meaning they treat and profile code executed on each core individually and are, therefore, already capable of profiling multi-threaded applications. However, the performance characteristics of multi-threaded applications, mainly caused by resource contentions and synchronization behavior, are very different from serial code. Time-proportional performance analysis exposes these performance bottlenecks simply through the additional time incurred to instructions but cannot pinpoint the exact causes of thread contention, thus, is unable to expose the thread that prevents the progress of others. Recording additional events and information from within the operating system scheduler might improve the interpretability of these cases. Presumably, this requires hardware-software co-design of the profiling circuitry and software to analyze the multi-threaded workloads.

Solving these issues would help developers optimize the throughput and efficiency of multi-threaded applications with time-proportional analysis that can expose the performance bottlenecks for each thread down to instruction granularity.

3.2.2 GPU Performance Profiling

GPUs have evolved from graphic processors to general-purpose computing accelerators. Although unsuitable for all workloads, they can achieve much higher performance and throughput than general-purpose processors. Their highly parallel architecture operates after the Single Instruction Multiple Data (SIMD) principle and executes many (of the same) instructions on big datasets inside the Streaming Multiprocessors (SM).

The GPU execution model is very different from multi-core processors. GPU threads are scheduled in warps or wavefronts, of which numerous execute on each streaming multiprocessor in parallel. They are orchestrated by a hardware thread scheduler that tries to minimize stall time caused by resource contentions and dependencies. Similar to CPUs, GPUs have memory hierarchies that use caches to speed up access to the global and local memory, and data locality is, therefore, equally important to lower the stall time of threads.

Time-proportional profiling can play an important role in GPU performance analysis to transparently explain the exact reasons for performance bottlenecks in this complex parallel architecture and pinpoint the threads and instructions of the application that stalled the most.

References

[1] R. Schaller, “Moore’s Law: Past, Present and Future,” *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[2] G. H. Loh and R. Swaminathan, “The Next Era for Chiplet Innovation,” in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[3] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[4] F. J. Pollack, “New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (Keynote Address)(Abstract Only),” in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 32, Haifa, Israel: IEEE Computer Society, 1999, p. 2.

[5] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark Silicon and the End of Multicore Scaling,” *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 365–376, Jun. 2011. Available: `https://doi.org/10.1145/2024723.2000108`.

[6] SPEC, *SPEC CPU 2017*, `https://www.spec.org/cpu2017/`, 2019.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT, Association for Computing Machinery, 2008, pp. 72–81.

[8] G. Hamerly, E. Perelman, J. Lau, and B. Calder, “Simpoint 3.0: Faster and More Flexible Program Analysis,” in *Journal of Instruction Level Parallelism (JILP)*, vol. 7, 2005, pp. 1–28.

[9] Linux, *Perf Wiki*, 2023. Available: `https://perf.wiki.kernel.org/index.php/Main%5C_Page`.

[10] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`, 2021.

[11] P. J. Drongowski, "Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors," AMD, Tech. Rep., 2007.

[12] Arm, *ARM Architecture Reference Manual Supplement Statistical Profiling Extension, for ARMv8-A*, `https://static.docs.arm.com/ddi0586/a/DDI0586A_Statistical_Profiling_Extension.pdf`, 2017.

[13] IBM, *POWER9 Performance Monitor Unit User's Guide*, `https://ibm.ent.box.com/s/8kh0orsr8sg32zb6zmq1d7zz6hud3f8j`, 2018.

[14] C.-K. Luk, R. Cohn, R. Muth, *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 190–200. Available: `https://doi.org/10.1145/1065010.1065034`.

[15] D. L. Bruening and S. Amarasinghe, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," AAI0807735, Ph.D. dissertation, USA, 2004.

[16] M. Desnoyers, *Linux Tracepoints*, `https://www.kernel.org/doc/Documentation/trace/tracepoints.txt`, 2023.

[17] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: a Call Graph Execution Profiler," in *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN, Association for Computing Machinery, 1982, pp. 120–126.

[18] F. S. Foundation, *GCC online documentation*, `https://gcc.gnu.org/onlinedocs/`, 2023.

[19] C. Lattner and V. 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*, ser. CGO, IEEE Computer Society, 2004, p. 75.

[20] SEGGER, *SEGGER J-Trace Streaming Trace Probes*, 2023. Available: `https://www.segger.com/products/debug-probes/j-trace/`.

[21] Intel, *Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023. Available: https://intel.com/content/www/us/en/developer/articles/technical/intel-sdm.

[22] Intel, *VTune Profiler User Guide*, 2021. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/vtune-profiler-user-guide.pdf.

[23] AMD, *μProf*, https://developer.amd.com/amd-uprof/, 2021.

[24] Linux, *perf*, https://perf.wiki.kernel.org/index.php/Main_Page, 2020.

[25] B. Gottschall, L. Eeckhout, and M. Jahre, "TIP: Time-Proportional Instruction Profiling," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, Association for Computing Machinery, 2021, pp. 15–27.

[26] Google, *gperftools*, https://github.com/gperftools/gperftools, 2020.

[27] NTNU, *PPerf*, https://github.com/EECS-NTNU/pperf, 2020.

[28] J. Levon, *OProfile*, https://oprofile.sourceforge.io/news/, 2021.

[29] H. Xu, Q. Wang, S. Song, L. K. John, and X. Liu, "Can We Trust Profiling Results? Understanding and Fixing the Inaccuracy in Modern Profilers," in *Proceedings of the International Conference on Supercomputing*, ser. ICS, Association for Computing Machinery, 2019, pp. 284–295.

[30] AMD, *Xilinx TCF*, https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/TCF-Profiling, 2023.

[31] Arm, *ULINKplus*, http://www2.keil.com/mdk5/ulink/ulinkplus/, 2020.

[32] A. Djupdal, B. Gottschall, F. Ghasemi, and M. Jahre, "Lynsyn and LynsynLite: The STHEM Power Measurement Units," in *Towards Ubiquitous Low-Power Image Processing Platforms*, M. Jahre, D. Göhringer, and P. Millet, Eds., Springer International Publishing, 2021, pp. 93–114.

[33] S. S. Banerjee, S. Jha, Z. Kalbarczyk, and R. K. Iyer, "BayesPerf: Minimizing Performance Monitoring Errors Using Bayesian Statistics," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, Association for Computing Machinery, 2021, pp. 832–844.

[34] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer Society, 2014, pp. 35–44.

[35] J. M. Anderson, L. M. Berc, J. Dean, *et al.*, "Continuous Profiling: Where Have All the Cycles Gone?" *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 357–390, 1997.

[36] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A Performance Counter Architecture for Computing Accurate CPI Components," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, Association for Computing Machinery, 2006, pp. 175–184.

Acronyms

TIP Time-Proportional Instruction Profiling

TEA Time-Proportional Event Analysis

PICS Per-Instruction Cycle Stacks

AMD RIS AMD Random Instruction Sampling

IBM IBS IBM Instruction Based Sampling

Arm SPE Arm Statistical Profiling Extension

Intel PEBS Intel Processor Event Based Sampling

Xilinx TCF Xilinx Target Communication Framework

Intel LBR Intel Last Branch Record

Intel PT Intel Processor Trace

Part II

Publications

Paper A

TIP: Time-Proportional Instruction Profiling

Authors:
Björn Gottschall, Lieven Eeckhout, Magnus Jahre

Published at conference:
54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

Nominations/Awards:
Best Paper Runner-Up Award, HiPEAC Paper Award

Paper A

Gottschall *et al.* (2021)

TIP: Time-Proportional Instruction Profiling

Björn Gottschall[1], Lieven Eeckhout[2], Magnus Jahre[1]

[1]Norwegian University of Science and Technology, Norway
[2]Ghent University, Belgium

Abstract

A fundamental part of developing software is to understand what the application spends time on. This is typically determined using a performance profiler which essentially captures how execution time is distributed across the instructions of a program. At the same time, the highly parallel execution model of modern high-performance processors means that it is difficult to reliably attribute time to instructions — resulting in performance analysis being unnecessarily challenging.

In this work, we first propose the Oracle profiler which is a golden reference for performance profilers. Oracle is golden because (i) it accounts every clock cycle and every dynamic instruction, and (ii) it is time-proportional, i.e., it attributes a clock cycle to the instruction(s) that the processor exposes the latency of. We use Oracle to, for the first time, quantify the error of software-level profiling, the dispatch-tagging heuristic used in AMD IBS and Arm SPE, the Last-Committing Instruction (LCI) heuristic used in external monitors, and the Next-Committing Instruction (NCI) heuristic used in Intel PEBS, resulting in average instruction-level profile errors of 61.8%, 53.1%, 55.4%, and 9.3%, respectively. The reason for these errors is that all existing profilers have cases in which they systematically attribute execution time to instructions that are not the root cause of performance loss. To overcome this issue, we propose *Time-Proportional Instruction Profiling (TIP)* which combines Oracle's time attribution policies with statistical sampling to enable practical implementation. We implement TIP within the Berkeley

Out-of-Order Machine (BOOM) and find that TIP is highly accurate. More specifically, TIP's instruction-level profile error is only 1.6% on average (maximally 5.0%) versus 9.3% on average (maximally 21.0%) for state-of-the-art NCI. TIP's improved accuracy matters in practice, as we exemplify by using TIP to identify a performance problem in the SPEC CPU2017 benchmark Imagick that, once addressed, improves performance by 1.93×.

1 Introduction

The imminent end of Moore's law implies that software inefficiencies can no longer be hidden through technology scaling. Analyzing performance-critical workloads in detail is extremely challenging though given the high (and continuously increasing) complexity of both software and hardware in modern-day computer systems. Software developers thus critically need practical and accurate tools to automatically attribute execution time to source code constructs such as instructions, basic blocks, and functions [1].

A performance profile (statistically) attributes execution time to application-level symbols. Depending on the use case, developers can select symbols at different granularities, including functions, basic blocks, and individual instructions. Gathering profiles without hardware support is inherently inaccurate (see Figure A.1). Software-level profilers (e.g., Linux `perf` [2][1]) interrupt the application and retrieve the address of the instruction that execution will resume from after the interrupt has been handled. Hence, the current in-flight instructions will drain before the interrupt handler is executed which means that the sampled instruction can be tens or even hundreds of instructions away from the instruction(s) that the processor was committing at the time the sample was taken. This phenomenon is known as skid [3] and can be addressed by adding hardware support for instruction sampling (e.g., Intel PEBS [4], AMD IBS [5], or Arm SPE [6]).

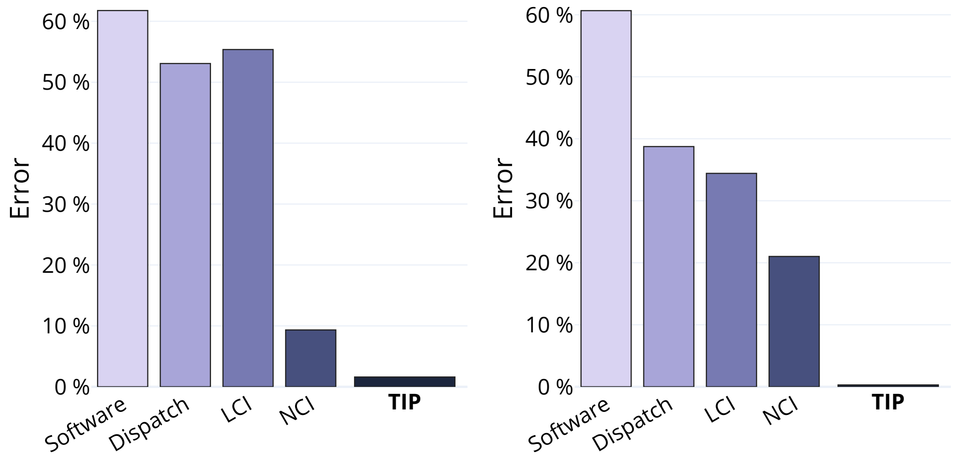
Hardware-supported profiling enables sampling in-flight instructions without interrupting the application and hence eliminates skid by (practically) removing the latency from sampling decision to sample collection. While all hardware profilers rely on sampling, i.e., collecting an instruction address at regular time intervals, their instruction selection policies differ. Intel's

[1]Software-level profiling is the default for `perf`, but it can be configured to use PEBS or IBS for instruction sampling when available.

Processor Event-Based Sampling (PEBS) [4] returns the address of the next instruction that commits after the sample is taken, i.e., a *Next Committing Instruction (NCI)* heuristic. Profiling approaches [7], [8] that use debug interfaces, such as Arm CoreSight [9], systematically sample the *Last Committed Instruction (LCI)*. Finally, AMD's Instruction-Based Sampling (IBS) [5] and Arm's Statistical Profiling Extension (SPE) [6] tag an instruction at *Dispatch* and then retrieve the sample when the instruction commits (which unlike the commit-focused approaches enable gathering data about how this instruction flows through the processor back-end [10]). Unfortunately, it is entirely unclear if these heuristics result in accurate performance profiles because we lack a golden reference — an unsolved problem that has plagued researchers and practitioners [3], [11], [12].

Oracle Profiler. We hence propose the Oracle profiler as a golden reference for performance profiling. The fundamental principle when deriving the Oracle profiler is that a profiler must perform *time-proportional attribution*, i.e., that every clock cycle is attributed to the instruction(s) that the processor exposes the latency of. The Oracle profiler hence focuses on the processor's commit stage because this is where the latency cost of each instruction is resolved and becomes visible to software. More specifically, the best-case instruction latency in a processor that can commit w instructions per cycle is $1/w$ cycles — meaning that the processor has been able to hide all of the instruction latency except for $1/w$ cycles. If the processor is unable to fully hide an instruction's execution latency, the instruction will stall at the head of the reorder buffer (ROB) and thereby block forward progress; i.e., the time commit blocks is the instruction's contribution to the application's execution time.

The Oracle profiler enables us to establish the accuracy of state-of-the-art hardware performance profiling approaches. (Section 4 describes our experimental setup and error metric.) Figure A.1a shows that Software profiling, the Dispatch-tagging strategy used by AMD's IBS [5], and the LCI-strategy of external profilers [7]–[9] all yield inaccurate instruction-level profiles with average errors of 61.8%, 53.1%, and 55.4%, respectively. The NCI-strategy used in Intel PEBS [4] is more accurate, but still leaves room for improvement (9.3% average error). These errors occur because *existing profilers are not time-proportional.* More specifically, they (i) do not account for ILP — i.e., incorrectly attributing the latency of co-committed instructions to only one of the instructions — and (ii) all suffer from systematic misattribution — i.e., attributing the latency of processor stalls to a different instruction



(a) Average error. **(b)** Imagick.

Figure A.1: Instruction-level profile error of state-of-the-art profilers compared to our Time-Proportional Instruction Profiler (TIP). *Existing profilers are inaccurate due to lack of ILP support and systematic latency misattribution.*

than the one that caused the stall. For example, NCI systematically blames the instruction after a pipeline flush for stalls due to misspeculation which results in a 21.0% error on the flush-intensive Imagick benchmark (see Figure A.1b). While Oracle is time-proportional, it cannot be implemented in real systems because accounting every instruction and every clock cycle generates an impractical amount of data (179 GB/s in our setup).

Time-Proportional Instruction Profiler (TIP). TIP bridges the gap between the state-of-the-art profilers and Oracle by combining the time attribution policies of Oracle with statistical sampling, thereby reducing the amount of profiling data by several orders of magnitude compared to Oracle (i.e., 192 KB/s versus 179 GB/s at the commonly used 4 KHz sampling frequency [2]) at the cost of introducing statistical error. Interestingly, Figure A.1 shows that statistical error is negligible in practice. More specifically, the average instruction-level profile error of TIP is merely 1.6% — hence TIP reduces average error by 5.8×, 34.6×, 33.2×, and 38.6× compared to NCI, LCI, Dispatch, and Software profiling, respectively. We implemented TIP in the Berkeley Out-of-Order Machine (BOOM) [13] within the FireSim [14] simulation infrastructure.[2]

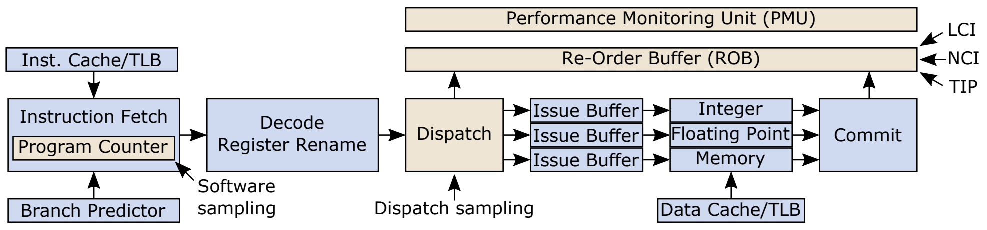
While low profile error is attractive, the real benefit of accurate performance profiling comes from helping developers write more efficient applications. To

[2]Our tools are available at `https://github.com/EECS-NTNU`.

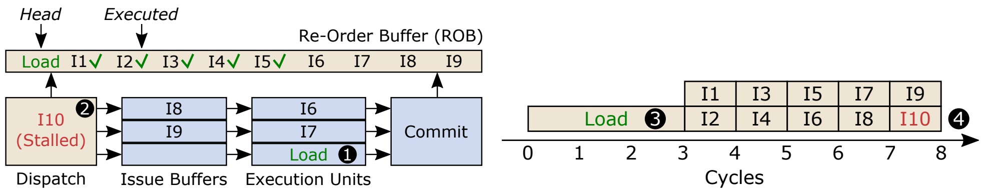
illustrate that TIP's accuracy matters in practice, we use TIP and NCI to analyze the SPEC CPU2017 benchmark Imagick. We find that while both TIP and NCI are accurate at the function-level (0.3% and 0.6% average error, respectively), the function-level profile does not clearly identify the performance problem; this is a common challenge with function-level profiles as developers use functions to organize functionality rather than performance. At the instruction-level, TIP correctly attributes time to Control Status Register (CSR) instructions that cause pipeline flushes whereas NCI misattributes execution time to the next-committing instruction (see Section 6 for details). Interestingly, Imagick does not need to execute the CSR instructions, and replacing them with `nop` instructions yields a 1.93× speed-up compared to the original, mostly due to the second-order effect that removing flushes improves the processor's ability to hide latencies.

Key Contributions:

- We propose a golden reference — the Oracle profiler — which enables quantifying performance profiler accuracy. To ensure that Oracle is robust, we implement it within a 4-wide BOOM core [13], and use the FPGA-accelerated FireSim [14] to simulate SPEC CPU2017 [15] and PARSEC [16] benchmarks to completion in a full-system setup.
- We explain how time-proportional performance profiles can be constructed, and show that existing profilers fall short because they are not time-proportional, i.e., they do not account for ILP and systematically misattribute latencies. More specifically, software-level profiling [2], the dispatch-tagging heuristic used in AMD IBS [5] and Arm SPE [6], the LCI-heuristic used in external monitors [7]–[9], and the NCI-heuristic used in Intel PEBS [4], yield average errors of 61.8%, 53.1%, 55.4%, and 9.3%, respectively.
- We propose the Time-Proportional Instruction Profiler (TIP) which combines Oracle's time attribution policies with statistical sampling to retain high accuracy (1.6% average error) while enabling real-system implementation. TIP is significantly more accurate than existing profilers, i.e., it reduces instruction-level profile error by 5.8×, 34.6×, 33.2×, and 38.6× compared to NCI, LCI, Dispatch, and Software profiling, respectively.
- We use TIP and NCI to analyze the SPEC CPU2017 benchmark Imagick. TIP pinpoints a performance problem that, once addressed, improves performance by 1.93× whereas NCI's profile is inconclusive.



(a) Out-of-order architecture.



(b) Dispatch sampling example. **(c)** Commit timeline.

Figure A.2: LCI, NCI and TIP sample instructions at commit whereas Dispatch (Software) samples at dispatch (fetch). *Dispatch and Software are biased because (i) different instructions spend more time in some pipeline stages than others, and (ii) the time an instruction spends at the head of the ROB directly impacts execution time.*

2 Time-Proportional Profiling

Practical performance profilers rely on statistical sampling to create a profile, i.e., they randomly retrieve the address(es) of (a) currently executing instruction(s). Since sampling is random in time, the probability of sampling an instruction — and time hence being attributed to it — should be proportional to the instruction's impact on overall execution time, and we refer to this property as *time-proportional attribution*. Consider for example a processor that executes a single instruction at a time: an instruction that takes two clock cycles to execute should be attributed twice as much time as a single-cycle instruction.

Understanding why sampling at the commit stage enables time-proportional attribution requires going into some detail on how an out-of-order processor operates (see Figure A.2a). Out-of-order processors consist of an in-order front-end that fetches and decodes instructions, predicts branches, performs register renaming, and finally dispatches instructions to the reorder buffer (ROB) and to the issue queues of the appropriate execution unit [17]. Then, instructions are executed as soon as their inputs are available (possibly out-of-order). Instructions are typically committed in program order to support precise exceptions, and the ROB is used to track instruction order. *Sampling at commit hence enables time-proportional attribution because this is where, not only an instruction's execution becomes visible to software, but also its latency impact on overall execution time becomes visible.*

Sampling at commit is a necessary but not sufficient condition for achieving time-proportional attribution because the profiler must also attribute time to the instruction that the processor spends time on (e.g., the time spent resolving a mispredicted branch must be attributed to the branch and not some other instruction). We find that none of the existing profilers we consider in this work do time-proportional attribution as Dispatch and Software do not sample at commit whereas NCI and LCI misattribute time. We will first exemplify why not sampling at commit is inaccurate in Section 2.1 before we explain why our Oracle profiler does time-proportional attribution, and why NCI and LCI do not, in Section 2.2.

2.1 Dispatch and Software Profiling

Dispatch sampling (as used in AMD IBS [5], Arm SPE [6], and ProfileMe [10]) selects the instruction to be profiled at the dispatch stage and then tracks it through the processor back-end. While this provides interesting insight regarding how an individual instruction progresses through the pipeline, it is not time-proportional. Figure A.2b shows the state of a processor that is currently stalling on a load instruction (see ❶). Since the processor has a number of independent instructions to process, it is able to execute these instructions while the load is pending. However, this leads to the ROB filling up with instructions which in turn stalls dispatch (see ❷). This results in instruction *I10* getting stuck at dispatch due to the back-pressure created by the load instruction. *I10* will hence attract samples under the dispatch sampling policy as it spends more time in the dispatch stage than other instructions. Figure A.2c shows the situation in Figure A.2b from the perspective of the commit stage. If we sample at commit, the load instruction will attract samples as it spends more time at the head of the ROB than the other instructions (see ❸). Sampling at commit hence enables time-proportional attribution, i.e., the load instruction is sampled more frequently because the processor spends more time executing it. In fact, the processor only exposes a half-clock-cycle latency for *I10* because its execution latency was almost completely hidden (see ❹).

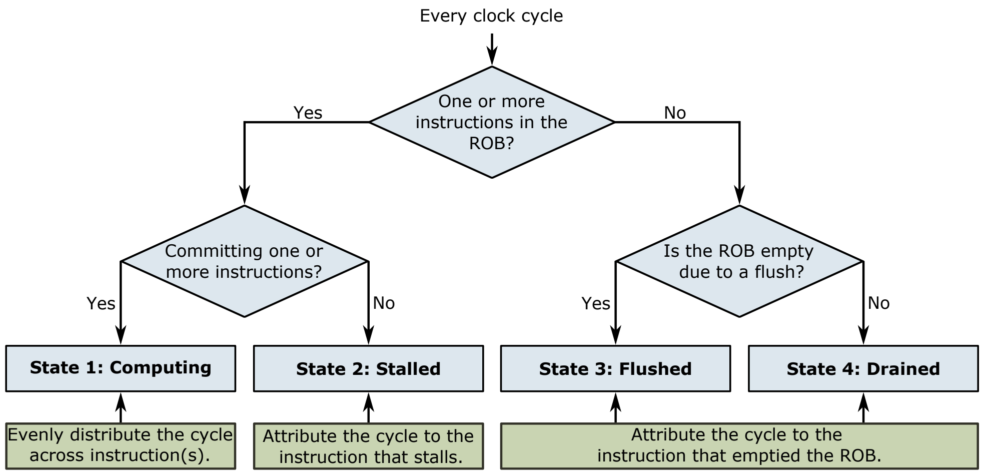

Every clock cycle
One or more instructions in the ROB?
Yes
No
Committing one or more instructions?
Yes
No
Is the ROB empty due to a flush?
Yes
No
State 1: Computing
State 2: Stalled
State 3: Flushed
State 4: Drained
Evenly distribute the cycle across instruction(s).
Attribute the cycle to the instruction that stalls.
Attribute the cycle to the instruction that emptied the ROB.

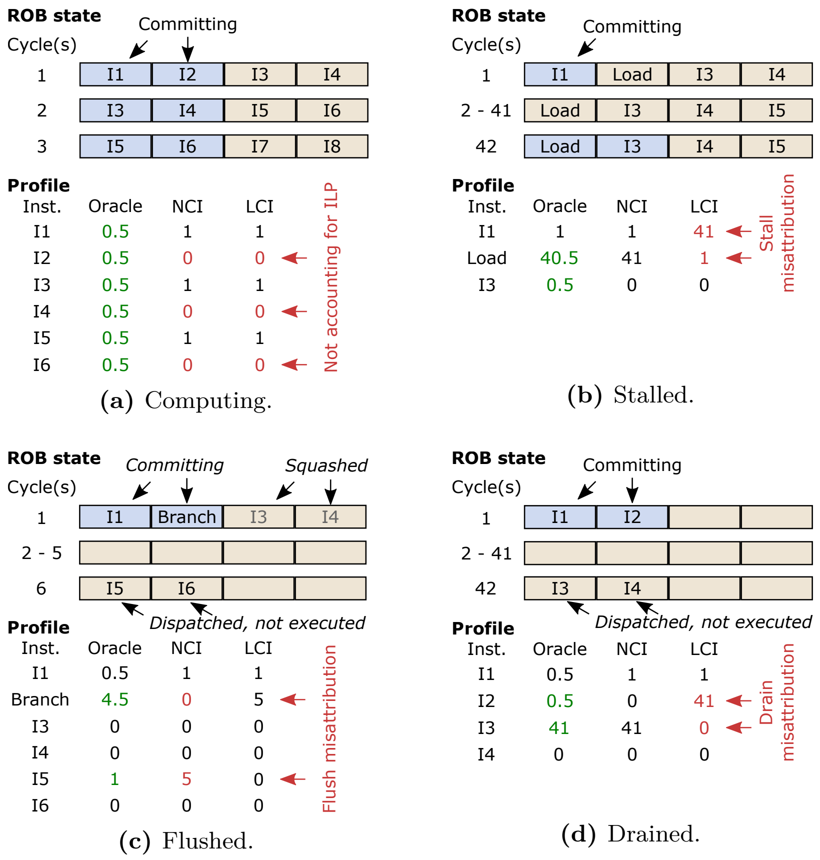
Figure A.3: Oracle profiler clock cycle attribution overview.

Software profiling is also not time-proportional due to a phenomenon prior work referred to as skid [3], [5]. As with Dispatch, long-latency instructions lead to commit stalls that attract samples, but, unlike Dispatch, Software attributes time to instructions that are fetched around the time the sample is taken. The reason is that Software relies on interrupts. Upon an interrupt, the processor stores the application's current Program Counter (PC) and transfers control to the interrupt handler which then attributes the sample to the instruction address in the PC. Software hence tends to attribute latency to instructions that are even further away from the stalled instruction in the instruction stream than Dispatch.

2.2 Oracle Profiling

In this section, we present Oracle which is time-proportional by design, i.e., it attributes each clock cycle during program execution to the instruction(s) which the processor exposed the latency of in this cycle. While NCI and LCI both sample at commit, they employ different instruction selection policies. More specifically, NCI (as supported by Intel PEBS [4]) samples the next-committing instruction, whereas LCI (as supported by external monitors [7], [8], [18]–[20]) samples the last-committed instruction, and we will now explain why neither policy is time-proportional.

Oracle overview. Oracle leverages the fundamental insight that the commit



(a) Computing. (b) Stalled. (c) Flushed. (d) Drained.

Figure A.4: Example illustrating the Oracle, NCI, and LCI profilers on a 2-wide out-of-order processor. *NCI and LCI fall short because they do not account for ILP at the commit stage and misattribute pipeline stall, flush and/or drain latencies.*

stage is in one of four possible states in each clock cycle. Hence, every clock cycle, the Oracle first checks if the ROB contains instructions (i.e., it is not empty). If the ROB contains (an) instruction(s), the Oracle profiler checks if the processor is committing (an) instruction(s) in this cycle. If so, the processor is in the *Computing* state (State 1 in Figure A.3), and the Oracle attributes $1/n$ clock cycles to each of the n committing instructions. If the processor is not committing instructions and there are instructions in the ROB, it is in the *Stalled* state (State 2 in Figure A.3). In this case, there is an instruction at the head of the ROB but it cannot be committed as it has not yet fully executed. The Oracle hence attributes the cycle to the instruction at the head of the ROB as it is blocking commit.

If the ROB is empty, Oracle attributes the clock cycle to the instruction that cleared the ROB. If the ROB is empty due to misspeculation, the processor is in the *Flushed* state (State 3 in Figure A.3). More specifically, the processor is in the flushed state if it committed all non-speculative in-flight instructions before the ROB could be refilled. In this case, the Oracle attributes the cycle to the instruction that caused the flush (e.g., a mispredicted branch). The ROB can also be empty because the front-end is not supplying instructions, typically due to an instruction cache or instruction Translation Lookaside Buffer (TLB) miss. In this case, the processor is in the *Drained* state (State 4 in Figure A.3), and the Oracle attributes the cycle to the first instruction that enters the ROB after the stall as this instruction delayed the front-end.

Comparing Oracle against NCI and LCI. We now explain Oracle in more detail for the four fundamental states, and compare against NCI and LCI to explain in which cases they do or do not misattribute clock cycles.

State 1: Computing. In the computing state, Oracle accounts $1/n$ cycles to each committed instruction where n is the number of instructions committed in that cycle (i.e., n is a number between 1 and the processor's commit width). Figure A.4a illustrates this behavior by showing the four oldest ROB-entries of a processor with 2-wide commit. In cycle 1, instructions *I1* and *I2* are committing and Oracle hence accounts 0.5 cycles to both. In contrast, NCI and LCI select a single instruction to attribute the clock cycle to. This is undesirable as it overly attributes cycles to some instructions while missing others — possibly to the extent that certain instructions are executed but not represented in the profile. Oracle, on the other hand, accounts for every clock cycle and every dynamic instruction.

Not acknowledging ILP within the commit stage renders the NCI and LCI profiles difficult to interpret. The key reason is that many applications execute similar instruction sequences over and over. Since NCI and LCI select instructions to sample with a fixed policy, they will be biased towards selecting certain instructions at the expense of others. It is hence difficult for developers to ascertain if a latency difference between instructions in straight-line code segments is due to a performance issue (e.g., some instructions stalling more than others) or attribution bias.

State 2: Stalled. Figure A.4b illustrates how Oracle, NCI, and LCI handle pipeline stalls that occur when instructions reach the head of the ROB before they have been executed. In this example, *I1* is committed in cycle 1 before commit stalls for 40 cycles on the load instruction from cycle 2 to 41; a

40-cycle latency is consistent with a partially hidden Last-Level Cache (LLC) hit in our setup. Oracle attributes the 40 cycles where the processor is stalled to the oldest instruction in the ROB since this is the instruction that the processor is stalling on, before attributing 0.5 cycles to the load and 0.5 cycles to *I3* when they both commit in cycle 42. NCI agrees with Oracle with the exception of missing *I3* in cycle 42 because it does not handle ILP. LCI, on the other hand, completely misattributes the load stall as *I1* is the last-committed instruction from cycle 1 to cycle 41, i.e., LCI attributes 41 cycles to *I1* and only a single cycle to the load (when it commits in cycle 42).

State 3: Flushed. Pipeline flushes occur when the processor has speculatively fetched and (possibly) executed instructions that should not be committed. Figure A.4c illustrates how Oracle handles this case for a mispredicted branch. Some cycles before the example starts, the branch instruction was executed, and the processor discovered that the branch was mispredicted. The processor hence squashed all speculative instructions (e.g., *I3* and *I4*). In cycle 1, *I1* and the branch are committed, and Oracle attributes 0.5 cycles to both instructions. In parallel, the front-end fetches instructions along the correct path which ultimately leads to instructions being dispatched in cycle 6; branch mispredicts lead to the ROB being empty for 3.5 cycles on average in our setup. Oracle hence attributes the 4 cycles the ROB is empty to the branch instruction and 1 cycle to *I5* (since the processor is stalling on it in cycle 6). LCI correctly attributes the stall cycles to the mispredicted branch whereas NCI does not. More specifically, NCI attributes the empty ROB cycles to *I5* as it will be the next instruction to commit. Moreover, it attributes zero cycles to the branch instruction since it is committed in parallel with *I1*. It will undoubtedly be challenging for a developer to understand that an instruction that appears to not take any time is in fact responsible for the ROB being empty.

While the above attribution policy is sufficient to handle other misspeculation cases such as load-store ordering (i.e., a younger load was executed before an older store to the same address), flushes due to exceptions need to be handled differently. More specifically, an exception fires when the excepting instruction reaches the head of the ROB which in turn results in the pipeline being flushed and control transferred to the OS exception handler. When the exception has been handled (e.g., the missing page has been installed in the page table), the excepting instruction is re-executed. Hence, Oracle attributes the cycles where the ROB is empty due to an exception to the instruction that caused the exception. Once the instructions of the exception handler are dispatched, the Oracle attributes cycles to these instructions (i.e., the Oracle does not differentiate between application and system code).

State 4: Drained. The ROB drains when the processor runs out of instructions to execute, for instance due to an instruction cache miss. This situation differs from pipeline flushes in that all instructions to be drained from the ROB are on the correct path and hence will be executed and committed. Figure A.4d exemplifies this situation. In cycle 1, *I1* and *I2* are committed. This leaves the ROB empty until cycle 42. The culprit is that the processor missed in the instruction cache when fetching *I3*, and that the latency of retrieving the cache block and resuming execution was only partially hidden by executing previously fetched instructions. Oracle hence attributes 0.5 cycles to *I1* and *I2* since they both commit in cycle 1. It also attributes 41 cycles to *I3*; 40 cycles is due to the drain and one cycle is attributed because *I3* is stalled at the head of the ROB in cycle 42. Similar to the stalled case, NCI is mostly correct since *I3* is the next instruction to commit when the instruction cache miss is resolved. In contrast, LCI misattributes the empty ROB cycles to *I2*.

Putting-it-all-together. We have so far discussed the four fundamental states of the commit stage (mostly) independently, but instructions often accumulate cycles across multiple states. For example, *I5* moves from the Flushed state to the Stalled state within the example in Figure A.4c, and the processor will be in the Computing state when *I5* eventually commits. The same applies to *I3* from Drained to Stalled (Figure A.4d). This observation is critical to understand how Oracle handles more complex situations, and we now describe how the four states are sufficient for serialized instructions (e.g., fences and atomic instructions) and page misses.

Serialized instructions require that (i) all prior instructions have fully executed before they are dispatched, and (ii) that no other instructions are dispatched until they have committed. While the ROB drains, Oracle will account time to the preceding instructions according to the time they spend at the head of the ROB. When the last preceding instruction commits, the serialized instruction is dispatched and hence immediately becomes the oldest in-flight instruction. Oracle hence accounts time to this instruction as Stalled while it executes and as Computing the cycle it commits. Once it has committed, the subsequent instruction is dispatched and Oracle will account it as Stalled while it executes.

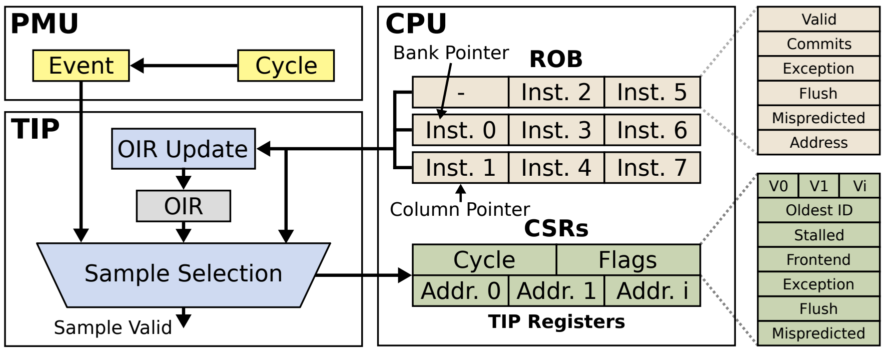


Figure A.5: Structural overview of our Time-Proportional Instruction Profiler (TIP). *TIP is triggered by the PMU, collects a sample, and finally exposes the sample to software.*

Another example is a page miss on a load instruction. In this case, the load accesses the data TLB and L1 data cache in parallel. This results in a TLB miss which invokes the hardware page table walker. Eventually, the page table walker concludes that the requested page is not in memory which causes the exception bit to be set in the load's ROB-entry. If the load reaches the head of the ROB before the page table walk completes, the Oracle starts accounting time as stalled. When the page table walk completes, the load is marked as executed and the exception is triggered once it reaches the head of the ROB. The cycles from the exception to dispatching the first instruction in the OS exception handler are attributed to the load. Once the OS has handled the exception by installing the missing page in memory, the load is re-executed. The load will then incur more stall cycles as it waits at the ROB head for its page mapping to be installed in the TLB and its data to be fetched from memory.

3 TIP: Time-Proportional and Practical Profiling

We now build upon the cycle-level attribution insights of Oracle to design our practical and accurate Time-Proportional Instruction Profiler (TIP).

3.1 Implementing TIP

Figure A.5 shows that TIP is located between the Performance Monitoring Unit (PMU) and the ROB. We now describe in detail how TIP captures samples, as well as how profiling software such as Linux `perf` [2] retrieves TIP's samples at runtime and, once the application terminates, post-processes the samples to create a performance profile.

Sample collection. As TIP is tightly coupled to the processor's ROB, we first quickly explain its main operation. In the BOOM core [13], the ROB consists of b banks, and up to one instruction per bank can be committed in each clock cycle (i.e., b is the commit width). Instructions are allocated to banks in the order of the bank identifiers. The instruction in bank i is hence always older than the instruction in bank $i+1$ within a column, but the b oldest ROB-entries may be distributed across two columns (see Figure A.5). Identifying the head of the ROB hence requires a pointer to the head bank and another pointer to the head column. The core can commit b instructions each cycle since the b oldest instructions will always be allocated in different banks. Similarly, b ROB-entries can be allocated concurrently at dispatch as long as b entries are available between the tail pointers and the current head pointers. When there are no invalid entries between the tail and head pointers, the ROB is full and dispatch stalls until one or more instructions commit. While the exact ROB realization may differ between architectures, it must fundamentally allow b-wide reads (which TIP exploits).

Figure A.5 shows that TIP consists of an *Offending Instruction Register (OIR)* and two functional units (*OIR Update* and *Sample Selection*), and Figure A.6 fleshes out the details of the *Sample Selection* unit. (The color-coding maps the components of Figure A.6 to units in Figure A.5.) When the ROB is not empty, TIP simply copies the addresses[3] of the head ROB-entries into its address registers (see ❶). To enable identifying the oldest ROB-entry, TIP stores the ROB bank pointer in the *Oldest ID* register (see ❷). The address valid bits are selected from the commit and valid signals (see ❸) in the Computing state and Stall state, respectively (see Figure A.3). During post-processing, these states are identified by inspecting TIP's *Stalled* flag which is 1 when no instructions are committed (see ❹). If the *Stalled* bit is 0, the core is in the Computing state, and the sample should be attributed

[3]Architectures commonly divide instructions into one or more µOps. In such implementations, TIP exploits that processors track the µOp-to-instruction mapping to handle interrupts and exceptions.

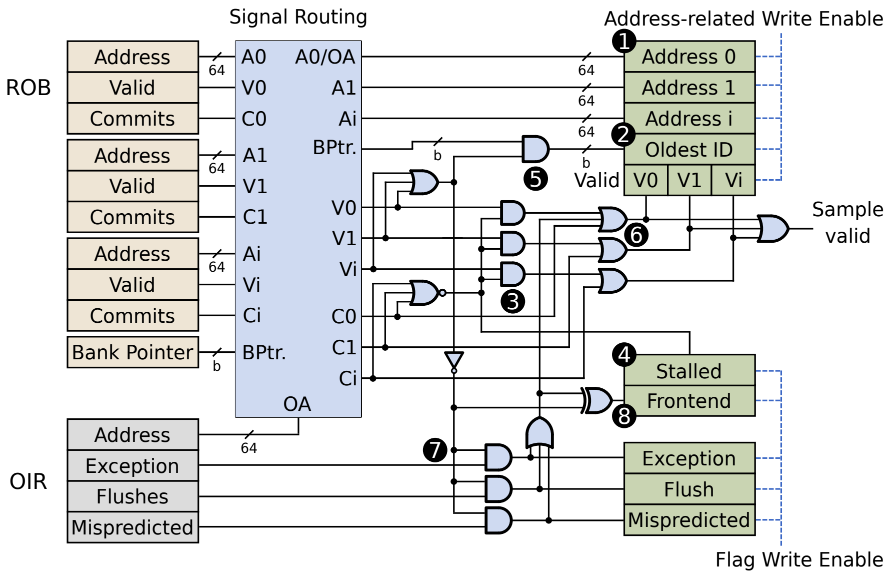


Figure A.6: TIP sample selection logic. *TIP classifies samples based on the the core state, ROB-flags, and OIR-flags.*

to all valid address CSRs. Conversely, the sample should be attributed to the address identified by the *Oldest ID* flag if the *Stalled* flag is 1. TIP only needs to record that the core stalled on this particular instruction since the stall type can be identified by inspecting the instruction type in the binary during post-processing.

If the processor is neither committing nor stalling, the ROB is empty due to a flush or a drain. TIP's *OIR Update* unit hence continuously tracks the last-committed and last-excepting instruction (see Figure A.5). More specifically, TIP updates the OIR with the address and relevant ROB-flags of the youngest committing ROB-entry every cycle; the relevant flags record if the instruction is a mispredicted branch or triggered a pipeline flush. If the processor is not committing instructions, TIP checks if the core is about to trigger an exception. If it is, TIP writes the address of the excepting instruction and an exception flag into the OIR. Returning to Figure A.6, we see that when all head ROB-entries are invalid, TIP (i) places the OIR address in the *Address 0* CSR, (ii) sets the oldest ID to 0 (see ❺), (iii) sets *V0* to 1 and remaining valid bits to 0 (see ❻), and (iv) sets the *Exception*, *Flush*, or *Mispredicted* TIP-flags based on the OIR-flags (see ❼). If one of these flags is set, the core is in the Flushed state.

If the ROB is not empty due to a flush, it must have drained (see Figure A.3). TIP hence immediately sets the *Front-end* flag as (i) the ROB is empty, and (ii) none of the *Exception*, *Flush*, or *Mispredicted* flags are set (see ❽). TIP then deasserts the write enable signal of the flags to prevent further updates, but keeps the write enable signal of the address-related CSRs and flags asserted. When the first instruction (eventually) dispatches, its ROB-entry becomes valid and TIP copies this address into the address CSR corresponding to the ROB-bank the entry is dispatched to (and sets the *Oldest ID* and valid bits accordingly). TIP then deasserts the address-related write enable signal to prevent further updates.

Creating an application profile. We have designed TIP to interface cleanly with Linux `perf` [2]. When using hardware support for profiling, `perf` configures the PMU to collect samples at a certain frequency (4 KHz is the default), and the profiler issues an interrupt when a new sample is ready. This interrupt invokes `perf`'s interrupt handler which simply copies the profiler's CSRs into a memory buffer; the profile is written to non-volatile storage when the buffer is full. At the end of application execution, `perf` has written the raw samples to a file which then needs to be post-processed. To build the profile, we use a data structure in which a zero-initialized counter is assigned to each unique instruction address in the profile. For each sample, we then add $1/n$ of the value in the cycles register to each instruction's counter when the sample contains n instructions. We also track the total number of cycles to enable normalizing the profile.

To help developers understand why some instructions take longer than others, TIP combines the information provided by its status flags with analysis of the application binary. We label cycles where the application is committing (an) instruction(s) as execution cycles and cycles where the ROB has drained as front-end cycles. If the processor is stalled, TIP uses the application binary to determine the instruction type and thereby understand if the oldest instruction is an ALU-instruction, a load, or a store. Moreover, we differentiate between flushes due to branch mispredicts and miscellaneous flushes based on TIP's status flags. (We group the miscellaneous flushes as they only account for 1.4% of application execution time on average.) While this categorization serves our purpose for this work, TIP can easily support more fine-grained categories if necessary.

3.2 TIP Overhead Analysis

Hardware overhead. TIP is extremely lean as it mostly relies on functionality that is already available either in the ROB or the PMU. The storage overhead of TIP is the OIR register (64-bit address and a 3-bit flag) and the CSRs (i.e., cycle, flags, and b address CSRs); we merge all TIP flags into a single CSR. All CSRs are 64-bit since RISC-V's CSR instructions operate on the full architectural bit width, resulting in an overall storage overhead of 57 B for our 4-wide BOOM core (9 B for the OIR and 48 B for the six CSRs). The logic complexity for collecting the samples is also negligible; the main overhead is two multiplexors, one to select the oldest ROB-entry in *OIR Update* and one to choose between the OIR and the address in ROB-bank 0 in *Sample Selection* (see Figure A.5). TIP's logic is not on the critical path of our BOOM core. If necessary, the logic can be pipelined.

Sampling overhead. As aforementioned, we assume that TIP interrupts the core when a new sample is ready. Another possible approach would be for TIP to write samples to a buffer in memory and then interrupt the core once the buffer is full. This requires more hardware support (i.e., inserting memory requests and managing the memory buffer), but reduces the number of interrupts. However, the interrupts become longer (as more data needs to be copied), so the total time spent copying samples is similar.

For each sample, `perf` reads the OS kernel structures to determine key metadata including core, process, and thread identifiers which account for 40 B per sample in total. For our 4-wide BOOM core, the non-ILP-aware profilers (e.g., NCI) capture a single instruction address and the cycle counter (an additional 16 B) whereas TIP captures four instruction addresses, the cycle counter, and the flags CSR (an additional 48 B). At `perf`'s default 4 KHz sampling frequency, TIP hence generates data at 352 KB/s whereas the data rate of the non-ILP-aware profilers is 224 KB/s. To quantify the performance overhead of TIP, we compare PEBS' default sample size (i.e., 56 B per sample) to a configuration with TIP-sized samples on an Intel Core i7-4770. We mimic TIP by including additional general-purpose registers from the PEBS record to reach TIP's 88 B sample size. We find that the increased data rate of TIP adds negligible overhead. More specifically, it increases application runtime by 1.1% compared to a configuration with profiling disabled; the performance overhead with PEBS' default sample size is 1.0%.

Table A.1: Simulated Configuration.

Part	**Configuration**
Core	OoO BOOM: RV64IMAFDCSUX @ 3.2 GHz
Front-end	8-wide fetch, 32-entry fetch buffer, 4-wide decode, 28 KB TAGE branch predictor, 40-entry fetch target queue, max 20 outstanding branches
Execute	128-entry ROB, 128 int/fp physical registers, 24-entry dual-issue MEM queue, 40-entry 4-issue INT queue, 32-entry dual-issue FP queue
LSU	32-entry load/store queue
L1	32 KB 8-way I-cache, 32 KB 8-way D-cache w/ 8 MSHRs, next-line prefetcher from L2
L2/LLC	512 KB 8-way L2 w/ 12 MSHRs, 4 MB 8-way LLC w/ 8 MSHRs
TLB	Page Table Walker, 32-entry fully-assoc L1 D-TLB, 32-entry fully-assoc L1 I-TLB, 512-entry direct-mapped L2 TLB
Memory	16 GB DDR3 FR-FCFS quad-rank, 25.6 GB/s maximum bandwidth, 14-14-14 (CAS-RCD-RP) latencies @ 1 GHz, 8 queue depth, 32 max reads/writes
OS	Buildroot, Linux 5.7.0

Multi-threading. Although we have so far described TIP in the context of single-threaded applications, this is not a fundamental limitation. More specifically, `perf` adds the core, process, and thread identifiers to each sample; the core identifier maps to a logical core under Simultaneous Multithreading (SMT). Apart from this, TIP will attribute time to (an) instruction(s) as in the single-threaded case. For example, if a physical core is committing instruction *I1* on logical core *C1* and instruction *I2* on logical core *C2* in the same cycle, TIP attributes half of the time to *I1* and half to *I2*. Each physical core needs its own TIP unit.

4 Experimental Setup

Simulator. We use the FireSim cycle-accurate FPGA-accelerated full-system simulator [14] to evaluate the different performance profiling strategies. The simulated model uses the BOOM 4-way superscalar out-of-order core [13], see Table A.1 for its configuration, which runs a common buildroot 5.7.0 Linux kernel. The BOOM core is synthesized to and run on the FPGAs in the Amazon's EC2 F1 nodes [21]. We account for the frequency difference between the FPGA-realization of the BOOM core and the FPGA's memory system using FireSim's token mechanism. We enable the hardware profilers when the system boots and profile until the system shuts down after the benchmark has terminated. However, we only include the samples that hit

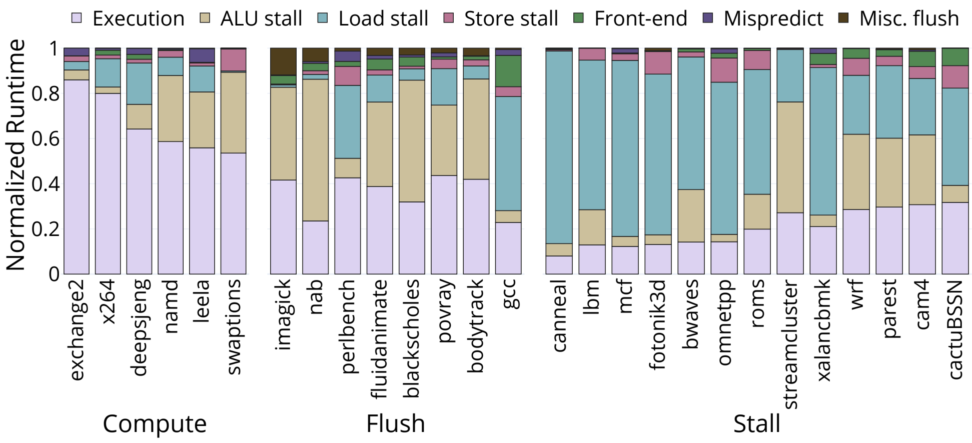


Figure A.7: Normalized cycle stacks collected at commit.

application code in our profiles as (i) the time our benchmarks spend in OS code (e.g., syscalls) is limited (1.1% on average), and (ii) we do not want to include boot and shutdown time in the profiles.

We modified FireSim to trace out the instruction address and the valid, commit, exception, flush, and mispredicted flags of the head ROB-entry in each ROB bank every cycle; the trace includes the ROB's head and tail pointers which we need to model Dispatch. We feed this trace to a highly parallel framework on the CPU-side to enable on-the-fly processing with only minimal simulation slowdown. The profilers are hence modeled on the CPUs that operate in lock-step with the FPGA by processing the traces. This allows us to simulate and evaluate multiple profiler configurations out-of-band in a single simulation run; we run up to 19 profiler configurations on 8 CPUs per FPGA simulation run. For this paper, the total time spent on Amazon EC2 amounts to 5,459 FPGA hours and 30,778 CPU hours. We evaluate multiple profilers with a single simulation run because (i) it enables fairly comparing profilers as they sample in the exact same cycle, and (ii) it reduces the evaluation time (and cost) on Amazon EC2.

Benchmarks. We run 27 SPEC CPU2017 [15] and PARSEC 3.0 [16] benchmarks that are compatible with our setup. (We use `x264` from PARSEC). We simulate the benchmarks to completion using the reference inputs for CPU2017 and the native inputs for PARSEC; we run single-threaded versions of PARSEC. We compile all benchmarks using GCC 10.1 with the `-O3 -g` compilation flags and static linking.

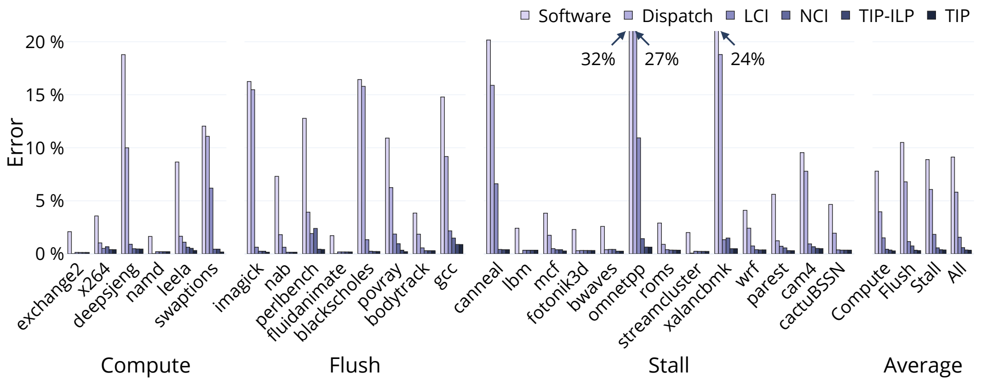


Figure A.8: Function-level errors for the different profilers. *TIP, TIP-ILP, NCI, and LCI are accurate at the function level, while Software and Dispatch are largely inaccurate.*

The benchmarks' execution characteristics are shown in Figure A.7 which reports normalized cycle stacks captured at commit [22], i.e., we attribute every cycle to a specific type, and we then represent the cycle types as a stacked bar with the execute component shown at the bottom, followed by the other cycle types on top; we introduced the categories in Section 3.1. We use the cycle stacks to classify our benchmarks: (i) a benchmark is classified as *Compute-Intensive* if it spends more than 50% of its execution time committing instructions; (ii) if not, and if the benchmark spends more than 3% of its time on pipeline flushing, the benchmark is classified as *Flush-Intensive*; and (iii) the rest of the benchmarks are classified as *Stall-Intensive* as they spend a major fraction of their execution time on processor stalls.

Quantifying profile error. Practical profilers incur inaccuracies compared to the (impractical) Oracle since they rely on statistical sampling and hence record a small percentage of instruction addresses, which are then attributed to symbols in the application binary; the symbols are individual instructions, basic blocks or functions, depending on profile granularity. There are two fundamental sources of error. *Unsystematic errors* occur because sampling is random and the distribution of sampled symbols does not exactly match the distribution obtained with Oracle. Unsystematic errors can be reduced by increasing sampling rate, as we will quantify in the evaluation. *Systematic errors*, on the other hand, occur because the profiling strategy attributes samples to the wrong symbol. We focus on systematic error in the evaluation by quantifying to what extent the different profilers attribute samples to the correct symbol as determined by the Oracle. Because we sample the exact

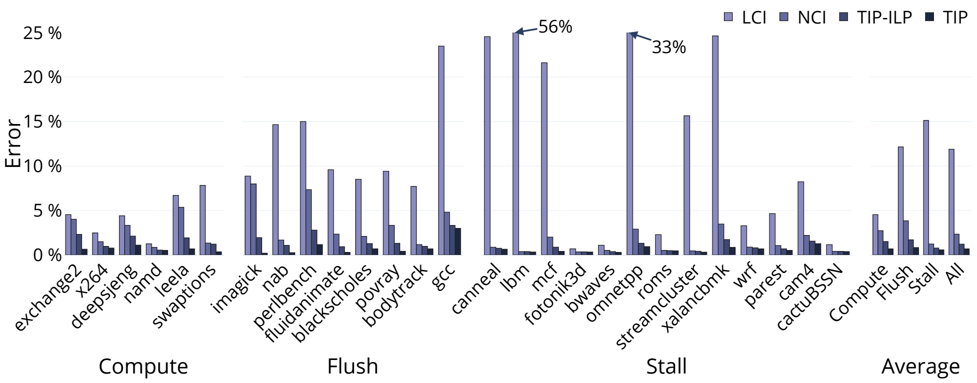


Figure A.9: Basic-block-level errors for the different profilers. (Software and Dispatch are not shown because of their high error.) *TIP, TIP-ILP, and NCI are accurate at the basic block level, whereas LCI (and Software and Dispatch) are not.*

same cycle for all the practical profilers in a single simulation run, we can precisely quantify and compare a profiler's systematic error.

Each sample is taken as a representative for the entire time period since the last sample. By comparing the symbol the sample is attributed to by the practical profiler against the symbol identified by Oracle, we determine whether a sample is correctly or incorrectly attributed. By aggregating the cycles correctly attributed to symbols (i.e., c_{correct}) and relating this to the total number of cycles it takes to execute the application (i.e., c_{total}), we can compute the relative error e (i.e., $e = (c_{\text{total}} - c_{\text{correct}})/c_{\text{total}}$). Error is a lower-is-better metric varying between 100% and 0%, where 100% means that all samples were incorrectly attributed, while 0% means that the practical profiler attributes each sample to the same symbol as Oracle. Profile error can be computed at any granularity, i.e., instruction, basic block, or function level; incorrect attribution at lower granularity can be correct at higher granularity (e.g., misattributing a sample to an instruction within the function that contains the correct instruction). We aggregate errors across benchmarks using the arithmetic mean.

5 Results

We compare the following profilers:

- **Software** generates an interrupt and samples the instruction after the interrupt (e.g., Linux `perf` [2]).
- **Dispatch** tags an instruction at dispatch and samples when it commits (e.g., AMD IBS [5] and Arm SPE [6]).
- **Last Committed Instruction (LCI)** selects the last-committed instruction (e.g., Arm CoreSight [9])
- **Next Committing Instruction (NCI)** selects the next-committing instruction (e.g., Intel PEBS [4]).
- **ILP-Oblivious Time-Proportional Instruction Profiling (TIP 'minus' ILP, or TIP-ILP)** follows TIP (see Section 3), but omits ILP accounting, i.e., when multiple instructions commit in the sampled cycle, the sample is attributed to a single instruction.
- **Time-Proportional Instruction Profiling (TIP)** is the profiler proposed in Section 3.

We compare against Oracle which attributes every cycle to the symbol at the profiling granularity of interest, using the policy described in Section 2.2. As mentioned before, the error differences between the hardware profiling strategies (i.e., all profilers except Software) are due to systematic inaccuracies only as we sample in the exact same cycle. We assume periodic sampling at a typical sampling frequency of 4 KHz, unless mentioned otherwise. We explore the impact of periodic versus random sampling and the impact of sampling frequency in our sensitivity analyses.

5.1 Profile Error

Function-level profiling. Figure A.8 reports error at the function level across all the profilers considered in this work. While TIP is the most accurate profiler (average error 0.3%), TIP-ILP, NCI, and LCI are also accurate with average errors of 0.4%, 0.6%, and 1.6%, respectively. (Note there are some outliers though for LCI up to 10.9%.) Software and Dispatch are much less accurate (9.1% and 5.8% average error, and up to 31.7% and 27.4%, respectively) because tagging instructions at fetch and dispatch creates significant bias. More specifically, samples are attracted to the instructions that are being fetched or dispatched while the processor is experiencing long-latency stalls. The overall conclusion is that *all profilers, except Software and Dispatch, are accurate at function-level granularity.* Since Software and

Dispatch are inherently inaccurate, we will exclude them for the smaller profiling granularities to more clearly show the differences between the more accurate profilers. However, we will report their average errors in the text for completeness.

Basic-block-level profiling. Correctly attributing samples to functions does not necessarily mean that a performance analyst will be able to identify the most performance-critical basic blocks. We hence need to dive deeper and evaluate our profilers at the basic block level. Figure A.9 shows profile errors at the basic block level for all profiling strategies, except Software and Dispatch which are largely inaccurate (average error of 29.9% and 22.4%, respectively). TIP and TIP-ILP are most accurate with average errors of 0.7% and 1.2%, respectively. NCI is also reasonably accurate with an average error of 2.3%, whereas LCI is inaccurate at this level with an average error of 11.9% and up to 56.1%. The reason is that LCI incorrectly attributes stalls on long-latency instructions (e.g., LLC load misses) to the instruction that last committed before the stall. For example, load stalls and functional unit stalls dominate lbm's runtime (66.2% and 15.6%, respectively). The performance-critical loop nest in lbm also contains significant control flow which leads LCI to attribute samples to the wrong basic block, which results in an overall error of 56.1%. The overall conclusion is that *TIP, TIP-ILP, and NCI are accurate at the basic block level, whereas Software, Dispatch, and LCI are not.*

It is also interesting to note that the error is higher at the basic block level compared to the function level; and this is true for all profilers. The most striking example is lbm: the LCI's function-level error is merely 0.3% and then increases to 56.1% at the basic block level. The reason is that a single function accounts for 99.7% of lbm's total runtime, which means that an incorrect attribution at the basic block level most likely still leads to a correct attribution at the function level. This reinforces our claim that fine-granularity profiles are critical as knowing that 99.7% of runtime is spent in a (non-trivial) function is too high-level to clearly identify optimization opportunities.

Instruction-level profiling. Performance analysts need profiling information that is even more detailed than the basic block (and function) level, i.e., performance stranglers need to be identified at the instruction level so that the performance analysts can understand and hopefully mitigate these bottlenecks. Figure A.10 reports instruction-level profile error for TIP, TIP-ILP, and NCI. Software, Dispatch, and LCI are not included here as they

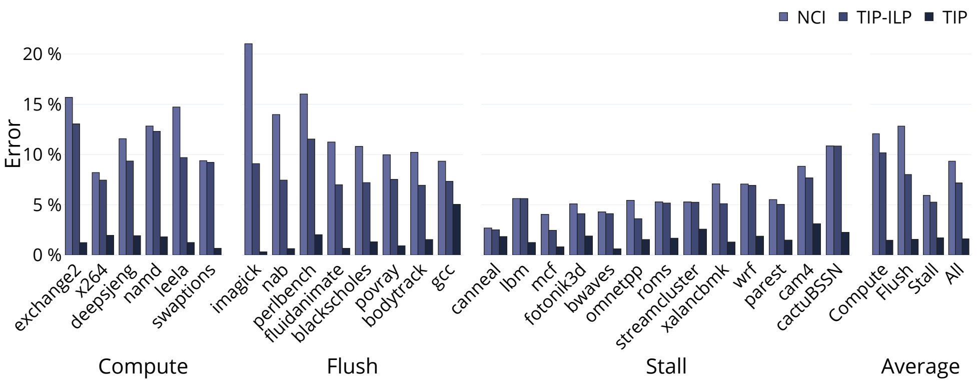
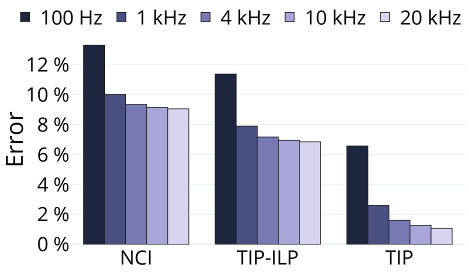


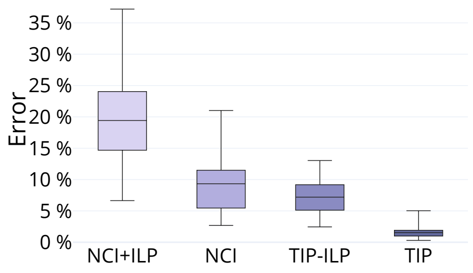
Figure A.10: Instruction-level errors for the different profilers. (Software, Dispatch, and LCI are omitted because of their large errors.) *TIP is the only accurate profiler at the instruction level.*

are largely inaccurate (i.e., average error of 61.8%, 53.1%, and 55.4%, respectively). The key conclusion is that *TIP is the only accurate profiler at the instruction level.* Indeed, the average profile error for TIP equals 1.6%, while the errors for TIP-ILP and NCI are significantly higher, namely 7.2% and 9.3%, respectively. Hence, TIP reduces average error by 5.8×, 34.6×, 33.2×, and 38.6× compared to NCI, LCI, Dispatch, and Software, respectively. We observe the highest error under TIP for gcc (5.0%), and find that the error can be reduced significantly by increasing the sampling frequency, as we will discuss later.

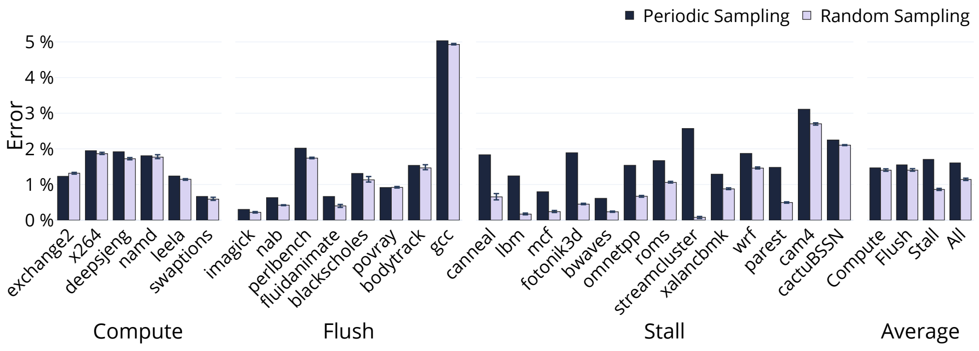
There are two reasons why TIP is the most accurate profiler. First, we observe a significant decrease in profile error when comparing NCI versus TIP-ILP for the flush-intensive benchmarks (see Figure A.10). The reason is TIP-ILP (and TIP) correctly attributes a sample that hits a branch misprediction or pipeline flush to the instruction that is responsible for refilling the pipeline, namely the mispredicted branch or the flush instruction, which is the instruction that was last committed. NCI on the other hand incorrectly attributes the sample to the instruction that will be committed next. Second, we observe the largest decrease in profile error between TIP-ILP and TIP for the compute-intensive benchmarks (see Figure A.10). The compute-intensive benchmarks commit multiple instructions per cycle, and hence attributing an equal share of the sample to all the committing instructions is the correct approach. TIP-ILP and NCI on the other hand attribute the sample to a single instruction which leads to a biased performance profile.



(a) Sampling frequency.



(c) Parallelism-aware NCI+ILP



(b) Periodic versus random sampling.

Figure A.11: Sensitivity analyses. *(a) TIP's accuracy continues to measurably improve beyond 4 KHz unlike the other profilers. (b) Periodic sampling is only slightly more inaccurate than random sampling while being simpler to implement in hardware. (c) Making NCI commit-parallelism-aware increases profile error, in contrast to TIP.*

5.2 Sensitivity Analyses

We now perform various sensitivity analyses with respect to sampling rate, sampling method, and commit-ILP accounting. We focus on instruction-level profiling and consider the most accurate profilers only, namely TIP, TIP-ILP, and NCI.

Sampling rate. As mentioned before, our default sampling rate is set to 4 KHz. We now focus on unsystematic error by evaluating how profiling error varies with sampling frequency from 100 Hz to 20 KHz, see Figure A.11a. As expected, profiling error decreases with increasing sampling frequency; and this is true for all profilers. Moreover, the reduction in error is more significant for the lower frequencies as these have more unsystematic error.

The most interesting observation is that TIP's accuracy continues to measurably improve as the sampling frequency is increased beyond 4 KHz, while it saturates for the other profilers. The most notable example is gcc for which the error decreases from 5.0% at 4 KHz (see Figure A.10) to 2.6% at 20 KHz. Profiling continues to decrease with frequency under TIP because it, unlike TIP-ILP and NCI, attributes high-ILP commit cycles to multiple instructions.

Sampling method. The sampling method used so far assumes periodic sampling, i.e., we take a sample every 250 µs (sampling frequency of 4 KHz). Periodic sampling may lead to an unrepresentative profile if the sampling frequency aligns unfavorably with the application's time-varying execution behavior (cf. Shannon-Nyquist sampling theorem). Random sampling may alleviate this by selecting a random sample within each 250 µs sampling interval. Figure A.11b quantifies profile error for periodic versus random sampling. We find that the impact is small for most benchmarks, except for a handful stall-intensive benchmarks such as streamcluster, lbm, and fotonik; these benchmarks exhibit repetitive time-varying execution behavior that is susceptible to sampling bias. On average, the error decreases from 1.6% under periodic sampling to 1.1% under random sampling. Because random sampling is more complicated to implement in hardware, we opt for periodic sampling in this work.

Commit-parallelism-aware NCI. TIP is more accurate than NCI because it correctly accounts for pipeline flushes and commit parallelism. Our results show that the biggest contribution comes from correctly attributing commit parallelism, i.e., compare the decrease in average instruction-level profile error from 9.3% (NCI) to 7.2% (TIP-ILP) due to correctly attributing pipeline flushing, versus the decrease in profile error from 7.2% (TIP-ILP) to 1.6% (TIP) due to attributing commit parallelism. The question can be raised whether accounting for commit parallelism in NCI would yield a level of accuracy that is similar to TIP, and we hence make NCI commit-parallelism-aware by simply attributing $1/n$ of the sample to the n next-committing instructions.

Figure A.11c presents box plots of the instruction-level error for commit-parallelism-aware NCI, called NCI+ILP, versus TIP, TIP-ILP, and NCI. Surprisingly, the average profile error increases with NCI+ILP, from 9.3% (NCI) to 19.3% (NCI+ILP). The primary reason is that NCI+ILP incorrectly attributes a sample to the n next-committing instructions after a long-latency stall (e.g., LLC miss), instead of attributing the entire sample to the long-latency instruction as done by TIP. The key insight is that commit-parallelism attribution is only beneficial when sample attribution is done in a correct and principled way in the first place, as is the case for TIP.

Validation. We use FireSim for our evaluation because the profilers considered in this work are platform-specific, hence it is impossible to compare the different profilers without reimplementing on a common platform. To evaluate our experimental setup, we conduct a validation experiment for the most accurate profiler in prior work, namely NCI. Lacking an Oracle profiler on real hardware platforms, we have to compare the *relative* difference among existing profilers to gauge their accuracy. In particular, we compare Linux `perf` [2] against PEBS [4] on an Intel i7-4770 system, versus our implementations of the Software profiler and NCI in FireSim, respectively. Obviously, one cannot expect a perfect match because we are comparing across instruction-set architectures (x86-64 versus RISC-V) and thus benchmark binaries. Yet, we still verify that the relative difference (computed using our error metric) between the respective profilers indeed falls within the same ballpark across our set of benchmarks, both at the instruction level and function level. At the instruction level, the difference between PEBS and `perf` on Intel amounts to 69% on average versus 57% on FireSim when comparing NCI versus Software. At the function level, the difference equals 4% versus 7%, respectively.

6 Profiling Case Study

We now perform a case study on the SPEC CPU2017 benchmark Imagick to illustrate how TIP pinpoints the root cause of performance issues. Figure A.12 shows the function- and instruction-level profiles of NCI, TIP, and Oracle for the `ceil` function in Imagick; `ceil` is a math library function and the third hottest function in Imagick. (We report the fraction of total runtime in the function-level profile, and the fraction of time within the function in the instruction-level profile.) The function-level profile does not clearly identify any performance problem (see ❶), suggesting to the developer that no further optimization is possible; a basic-block-level profile suffers from the same limitation. The instruction-level NCI profile attributes most of the execution time to the `feq.d` and the `ret` instructions (see ❷ and ❸, respectively), likely leading to the conclusion that the floating point unit(s) are overloaded and that the return address predictor is ineffective. Hence,

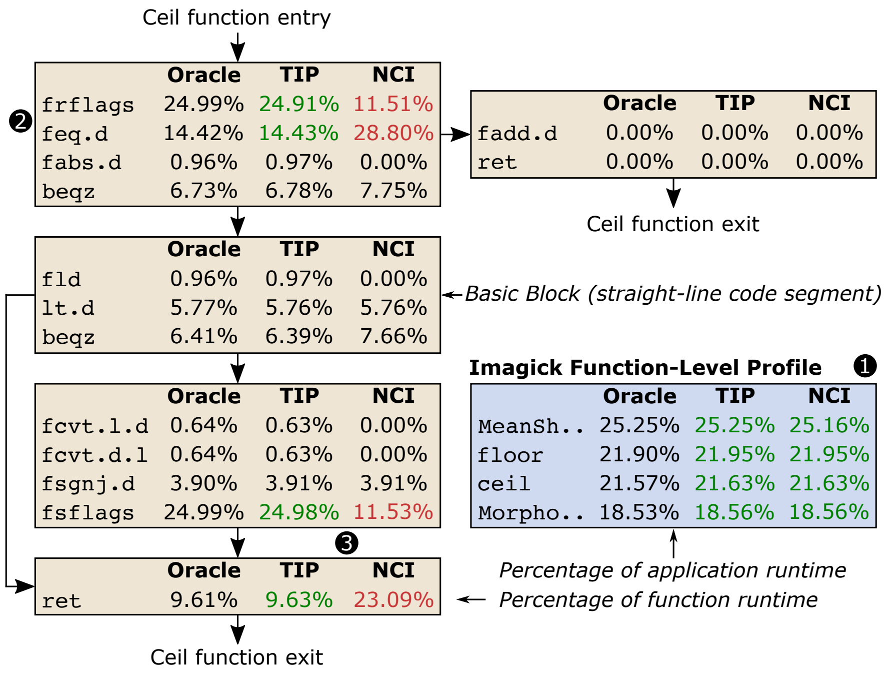

Ceil function entry
❷
Oracle TIP NCI
frflags 24.99% 24.91% 11.51%
feq.d 14.42% 14.43% 28.80%
fabs.d 0.96% 0.97% 0.00%
beqz 6.73% 6.78% 7.75%
Oracle TIP NCI
fadd.d 0.00% 0.00% 0.00%
ret 0.00% 0.00% 0.00%
Ceil function exit
Oracle TIP NCI
fld 0.96% 0.97% 0.00%
lt.d 5.77% 5.76% 5.76%
beqz 6.41% 6.39% 7.66%
Basic Block (straight-line code segment)
Imagick Function-Level Profile ❶
Oracle TIP NCI
fcvt.l.d 0.64% 0.63% 0.00%
fcvt.d.l 0.64% 0.63% 0.00%
fsgnj.d 3.90% 3.91% 3.91%
fsflags 24.99% 24.98% 11.53%
Oracle TIP NCI
MeanSh.. 25.25% 25.25% 25.16%
floor 21.90% 21.95% 21.95%
ceil 21.57% 21.63% 21.63%
Morpho.. 18.53% 18.56% 18.56%
❸
Oracle TIP NCI
ret 9.61% 9.63% 23.09%
Percentage of application runtime
Percentage of function runtime
Ceil function exit

Figure A.12: Function and instruction-level profiles for Imagick for TIP and NCI compared to Oracle.

the developer will probably conclude that further software-level optimization is difficult.

TIP, on the other hand, correctly reports that most of the time in `ceil` is spent on the `frflags` and `fsflags` instructions, and the purpose of these instructions is to mask any changes to the floating-point status register that may occur within the function from the calling code. These instructions are hence necessary if the calling code relies on `ceil` being side-effect free. Interestingly, Imagick never reads the floating-point status register which means that the masking performed within `ceil` is unnecessary. Moreover, the `floor` function suffers from exactly the same problem. We hence optimized Imagick's code by replacing `frflags` and `fsflags` in `ceil` and `floor` with `nop` instructions to remove the unnecessary status register operations.

Figure A.13 presents a cycle stack that compares the original Imagick benchmark (marked "Orig.") to our optimized version (marked "Opt.") across the four hottest functions in the original version. As expected, the original benchmark spends significant time in the "Misc. flush" category because the BOOM core flushes the pipeline after floating-point status register updates to

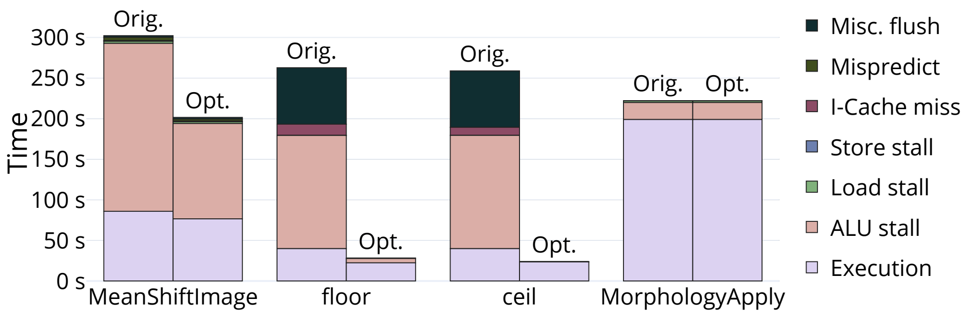


Figure A.13: Time breakdown for the four most runtime-intensive functions in Imagick comparing the original to our optimized version. *The 1.93× speed-up is primarily due improved processor utilization.*

guarantee that instruction dependencies are respected (the BOOM core does not rename status registers) whereas our optimized version does not flush at all. Overall, our optimized version improves performance by 1.93× compared to the original version and hence clearly illustrates that TIP identifies optimization opportunities that matter.

Interestingly, the speedup is (much) higher than expected based on the fraction of time spent executing the `frflags` and `fsflags` instructions (see Figure A.12). More specifically, the instructions collectively account for about 50% of the execution time of two functions that each account for around 22% of overall execution time, yielding an expected speedup of 1.28×. The reason is that the frequent pipeline flushing induced by the floating-point status register accesses has a detrimental effect on the processor's ability to hide latencies. For instance, both `ceil` and `floor` spend significant time on ALU stalls and front-end stalls — since the processor does not have sufficient instructions available to hide functional unit latencies and instruction cache misses. Moreover, our optimization improves IPC from 1.2 to 2.3 which leads to the processor spending less time executing instructions. The effects of improved IPC and reduced stalling carry over to the `MeanShiftImage` function from which `ceil` and `floor` is called, reducing its execution time by roughly one third.

7 Related Work

Hardware-supported profiling. The most related work is the hardware-based instruction profilers employed in current processors: Intel PEBS [4], AMD IBS [5], and Arm SPE [6]; IBS and SPE are inspired by ProfileMe [10]. In addition, external profilers [7], [8], [18]–[20] use debug interfaces such as Arm CoreSight [9] to sample dynamic instructions. TIP is more accurate than these schemes (see Section 5).

Software-level profiling. Software-level profilers [2], [23], [24] are significantly less accurate than TIP and hence sacrifice profile accuracy at the benefit of not requiring hardware support. While TIP helps developers understand how time is distributed across instructions, other performance aspects are also interesting. Vertical profiling [25], [26] combines hardware performance counters with software instrumentation to profile an application across deep software stacks, while call-context profiling [27] efficiently identifies the common orders functions are called in. Causal profiling [11], [28]–[30] is able to identify the criticality of program segments in parallel codes by artificially slowing down segments and measuring their impact. Researchers have also devised approaches for profiling highly optimized code [31], assessing input sensitivity [32], [33], profiling deployed applications [34], and function-level energy attribution [35].

Performance Monitoring Units (PMUs). A large body of work has investigated PMU design [36], and PMUs have a variety of uses (e.g., runtime optimization [37], performance analysis in managed languages [38]–[40], profile-guided compilation [41], [42], and profile-guided meta-programming [43]). Eyerman et al. [44] propose a PMU architecture that enables constructing CPI stacks. In contrast to TIP, CPI stacks capture coarse-grain performance information (e.g., across the entire application) whereas TIP precisely attributes time to individual instructions. The top-down model [45] is also coarse-grain and cannot attribute time to instructions. Researchers have also investigated relating PMU events to application activities [3], [12] and how to make sense of PMU output [46]–[49]; these issues are orthogonal to the problem TIP addresses (i.e., attributing time to instructions).

Instrumentation, simulation, and modeling. Static instrumentation modifies the binary to gather (extensive) application execution data at the cost of performance overhead [1], [50]–[53]. Dynamic instrumentation (e.g., PIN [54] and Valgrind [55]) does not modify the binary which leads to higher performance overheads than static instrumentation. Statistical performance profilers (e.g., TIP and Intel PEBS) do not add instructions and hence have (much) lower overhead than instrumentation-based approaches.

Simulation and modeling can also be used to understand key performance issues. The most related approach to ours is FirePerf [56] which uses FireSim [14] to non-intrusively gather extensive performance statistics. Unlike TIP, which is straightforwardly implementable in an out-of-order core, FirePerf cannot be employed outside of the simulator as it generates a similar amount of data to Oracle. Our approach is also related to interval analysis [57], [58], but interval analysis targets dispatch while we target commit. GDP [59] applies interval modeling at commit, but focuses on slowdown prediction and hence only considers memory loads.

8 Conclusion

We have presented our Oracle profiler, the first golden reference for performance profiling, and used it to show that existing profilers fall short because they are not time-proportional (i.e., they lack ILP support and systematically misattribute instruction latencies). We hence propose the Time-Proportional Instruction Profiler (TIP) which combines the attribution policies of Oracle with statistical sampling to enable practical implementation. TIP is highly accurate (average instruction-level error of 1.6%), and this accuracy enabled us to identify a performance issue in the SPEC CPU2017 benchmark Imagick that, once addressed, yields a 1.93× speed-up.

Acknowledgments

We thank David Metz for his help with FireSim and BOOM, and the anonymous reviewers for their valuable feedback. Lieven Eeckhout is supported through the European Research Council (ERC) Advanced Grant no. 741097. Magnus Jahre is supported by the Research Council of Norway (Grant no. 286596).

References

[1] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, “Performance Debugging in the Large Via Mining Millions of Stack Traces,” in *Proceedings of the International Conference on Software Engineering*, ser. ICSE, IEEE Press, 2012, pp. 145–155.

[2] Linux, *perf*, `https://perf.wiki.kernel.org/index.php/Main_Page`, 2020.

[3] H. Xu, Q. Wang, S. Song, L. K. John, and X. Liu, “Can We Trust Profiling Results? Understanding and Fixing the Inaccuracy in Modern Profilers,” in *Proceedings of the International Conference on Supercomputing*, ser. ICS, Association for Computing Machinery, 2019, pp. 284–295.

[4] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`, 2021.

[5] P. J. Drongowski, “Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors,” AMD, Tech. Rep., 2007.

[6] Arm, *ARM Architecture Reference Manual Supplement Statistical Profiling Extension, for ARMv8-A*, `https://static.docs.arm.com/ddi0586/a/DDI0586A_Statistical_Profiling_Extension.pdf`, 2017.

[7] A. Djupdal, B. Gottschall, F. Ghasemi, and M. Jahre, “Lynsyn and LynsynLite: The STHEM Power Measurement Units,” in *Towards Ubiquitous Low-Power Image Processing Platforms*, M. Jahre, D. Göhringer, and P. Millet, Eds., Springer International Publishing, 2021, pp. 93–114.

[8] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan, “Aveksha: a Hardware-Software Approach for Non-Intrusive Tracing and Profiling of Wireless Embedded Systems,” in *Proceedings of the Conference on Embedded Networked Sensor Systems*, ser. SenSys, Association for Computing Machinery, 2011, pp. 288–301.

[9] Arm, *CoreSight Architecture Specification v3.0*, `https://developer.arm.com/documentation/ihi0029/`, 2017.

[10] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, IEEE Computer Society, 1997, pp. 292–302.

[11] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the Accuracy of Java Profilers," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2010, pp. 187–197.

[12] S. S. Banerjee, S. Jha, Z. Kalbarczyk, and R. K. Iyer, "BayesPerf: Minimizing Performance Monitoring Errors Using Bayesian Statistics," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, Association for Computing Machinery, 2021, pp. 832–844.

[13] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, *SonicBOOM: the 3rd Generation Berkeley Out-of-Order Machine*, Fourth Workshop on Computer Architecture Research with RISC-V, 2020.

[14] S. Karandikar, H. Mao, D. Kim, *et al.*, "Firesim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *Proceedings of the International Symposium on Computer Architecture*, ser. ISCA, IEEE Press, 2018, pp. 29–42.

[15] SPEC, *SPEC CPU 2017*, `https://www.spec.org/cpu2017/`, 2019.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT, Association for Computing Machinery, 2008, pp. 72–81.

[17] A. González, F. Latorre, and G. Magklis, *Processor Microarchitecture: An Implementation Perspective* (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers, 2010.

[18] A. Sadek, A. Muddukrishna, L. Kalms, *et al.*, "Supporting Utilities for Heterogeneous Embedded Image Processing Platforms (STHEM): An Overview," in *Applied Reconfigurable Computing (ARC)*, 2018.

[19] IAR, *I-jet*, `https://www.iar.com/iar-embedded-workbench/add-ons-and-integrations/in-circuit-debugging-probes/`, 2020.

[20] Arm, *ULINKplus*, `http://www2.keil.com/mdk5/ulink/ulinkplus/`, 2020.

[21] Amazon, *Amazon EC2 F1 Instances*, `https://aws.amazon.com/ec2/instance-types/f1/`, 2021.

[22] S. Eyerman, W. Heirman, K. Du Bois, and I. Hur, "Multi-Stage CPI Stacks," *IEEE Computer Architecture Letters (CAL)*, vol. 17, no. 1, pp. 55–58, 2018.

[23] NTNU, *PPerf*, `https://github.com/EECS-NTNU/pperf`, 2020.

[24] Google, *gperftools*, `https://github.com/gperftools/gperftools`, 2020.

[25] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, "Vertical Profiling: Understanding the Behavior of Object-Priented Applications," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA, Association for Computing Machinery, 2004, pp. 251–269.

[26] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer, "Automating Vertical Profiling," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA, Association for Computing Machinery, 2005, pp. 281–296.

[27] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi, "Accurate, Efficient, and Adaptive Calling Context Profiling," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2006, pp. 263–271.

[28] C. Curtsinger and E. D. Berger, "Coz: Finding Code That Counts with Causal Profiling," in *Proceedings of the Symposium on Operating Systems Principles*, ser. SOSP, Association for Computing Machinery, 2015, pp. 184–197.

[29] A. Yoga and S. Nagarakatte, "Parallelism-Centric What-If and Differential Analyses," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2019, pp. 485–501.

[30] B. Pourghassemi, A. Amiri Sani, and A. Chandramowlishwaran, "What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Association for Computing Machinery, 2019, pp. 87–88.

[31] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan, "Binary Analysis for Measurement and Attribution of Program Performance," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2009, pp. 441–452.

[32] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-Sensitive Profiling," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2012, pp. 89–98.

[33] D. Zaparanuks and M. Hauswirth, "Algorithmic Profiling," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2012, pp. 67–76.

[34] C. H. Kim, J. Rhee, H. Zhang, *et al.*, "IntroPerf: Transparent Context-Sensitive Multi-Layer Performance Inference Using System Stack Traces," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS, Association for Computing Machinery, 2014, pp. 235–247.

[35] L. Mukhanov, D. S. Nikolopoulos, and B. R. d. Supinski, "ALEA: Fine-Grain Energy Profiling with Basic Block Sampling," in *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 87–98.

[36] G. Kornaros and D. Pnevmatikatos, "A Survey and Taxonomy of on-Chip Monitoring of Multicore Systems-on-Chip," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 2, pp. 1–38, 2013.

[37] D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. De Bosschere, "Using HPM-Sampling to Drive Dynamic Compilation," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA, Association for Computing Machinery, 2007, pp. 553–568.

[38] P. F. Sweeney, M. Hauswirth, B. Cahoon, *et al.*, "Using Hardware Performance Monitors to Understand the Behavior of Java Applications," in *Proceedings of the Conference on Virtual Machine Research and Technology Symposium*, ser. VM, USENIX Association, 2004, p. 5.

[39] J. Whaley, "A Portable Sampling-Based Profiler for Java Virtual Machines," in *Proceedings of the Conference on Java Grande*, ser. JAVA, Association for Computing Machinery, 2000, pp. 78–87.

[40] Y. Zheng, L. Bulej, and W. Binder, "Accurate Profiling in the Presence of Dynamic Compilation," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA, Association for Computing Machinery, 2015, pp. 433–450.

[41] T. M. Conte, K. N. Menezes, and M. A. Hirsch, "Accurate and Practical Profile-Driven Compilation Using the Profile Buffer," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, IEEE Computer Society, 1996, pp. 36–45.

[42] T. M. Conte, B. A. Patel, and J. S. Cox, "Using Branch Handling Hardware to Support Profile-Driven Optimization," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, Association for Computing Machinery, 1994, pp. 12–21.

[43] W. J. Bowman, S. Miller, V. St-Amour, and R. K. Dybvig, "Profile-Guided Meta-Programming," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2015, pp. 403–412.

[44] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A Performance Counter Architecture for Computing Accurate CPI Components," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, Association for Computing Machinery, 2006, pp. 175–184.

[45] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer Society, 2014, pp. 35–44.

[46] V. M. Weaver and S. A. McKee, "Can Hardware Performance Counters Be Trusted?" In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, IEEE Computer Society, 2008, pp. 141–150.

[47] V. M. Weaver, D. Terpstra, and S. Moore, "Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer Society, 2013, pp. 215–224.

[48] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of Performance Counter Measurements," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer Society, 2009, pp. 23–32.

[49] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan, "Time Interpolation: So Many Metrics, So Few Registers," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, IEEE Computer Society, 2007, pp. 286–300.

[50] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying Page Load Performance with WProf," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI, USENIX Association, 2013, pp. 473–486.

[51] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: a Call Graph Execution Profiler," in *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN, Association for Computing Machinery, 1982, pp. 120–126.

[52] C. Lattner and V. 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*, ser. CGO, IEEE Computer Society, 2004, p. 75.

[53] T. B. Schardl, T. Denniston, D. Doucet, B. C. Kuszmaul, I.-T. A. Lee, and C. E. Leiserson, "The CSI Framework for Compiler-Inserted Program Instrumentation," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 1, no. 2, 2017.

[54] C.-K. Luk, R. Cohn, R. Muth, *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2005, pp. 190–200.

[55] N. Nethercote and J. Seward, “Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2007, pp. 89–100.

[56] S. Karandikar, A. Ou, A. Amid, *et al.*, “FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, Association for Computing Machinery, 2020, pp. 715–731.

[57] T. S. Karkhanis and J. E. Smith, “A First-Order Superscalar Processor Model,” in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[58] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, “A Mechanistic Performance Model for Superscalar Out-of-Order Processors,” *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 2, pp. 1–37, 2009.

[59] M. Jahre and L. Eeckhout, “GDP: Using Dataflow Properties to Accurately Estimate Interference-Free Performance At Runtime,” in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 296–309.

Paper B

TEA: Time-Proportional Event Analysis

Authors:
Björn Gottschall, Lieven Eeckhout, Magnus Jahre

Published at conference:
50th Annual International Symposium on Computer Architecture (ISCA)

Nominations/Awards:
Best Paper Nomination

Paper B

Gottschall *et al.* (2023)

TEA: Time-Proportional Event Analysis

Björn Gottschall[1], Lieven Eeckhout[2], Magnus Jahre[1]

[1]Norwegian University of Science and Technology, Norway
[2]Ghent University, Belgium

Abstract

As computer architectures become increasingly complex and heterogeneous, it becomes progressively more difficult to write applications that make good use of hardware resources. Performance analysis tools are hence critically important as they are the only way through which developers can gain insight into the reasons why their application performs as it does. State-of-the-art performance analysis tools capture a plethora of performance events and are practically non-intrusive, but performance optimization is still extremely challenging. We believe that the fundamental reason is that current state-of-the-art tools in general cannot explain *why* executing the application's performance-critical instructions take time.

We hence propose Time-Proportional Event Analysis (TEA) which explains why the architecture spends time executing the application's performance-critical instructions by creating time-proportional Per-Instruction Cycle Stacks (PICS). PICS unify performance profiling and performance event analysis, and thereby (i) report the contribution of each static instruction to overall execution time, and (ii) break down per-instruction execution time across the (combinations of) performance events that a static instruction was subjected to across its dynamic executions. Creating time-proportional PICS requires tracking performance events across all in-flight instructions, but TEA only increases per-core power consumption by ~3.2 mW (~0.1%) because we carefully select events to balance insight and overhead. TEA leverages statistical sampling to keep performance overhead at 1.1% on average while incurring an average error of 2.1% compared to a non-sampling golden reference; a significant improvement upon the 55.6%, 55.5%, and 56.0% average error for AMD IBS, Arm SPE, and IBM RIS. We demonstrate that TEA's accuracy matters by using TEA to identify performance issues in the SPEC CPU2017 benchmarks lbm and nab that, once addressed, yield speedups of 1.28× and 2.45×, respectively.

1 Introduction

The end of Dennard scaling and the imminent end of Moore's law means that we can no longer expect general-purpose CPU architectures to deliver performance scaling [1]. Industry has responded by exploiting specialization and integrating heterogeneous compute engines including GPU and domain-specific accelerators alongside conventional CPU cores [2]. Counter-intuitively perhaps, sequential CPU code becomes relatively more performance-critical in heterogeneous systems due to Amdahl's Law, i.e., acceleratable code regions take much less time to execute while non-acceleratable code still takes the same amount of time [3]. Performance tuning of sequential CPU code to better exploit the underlying hardware is hence becoming increasingly critical. Unfortunately, this is a time-consuming and tedious endeavor because of how state-of-the-art CPU architectures optimize performance through various forms and degrees of instruction-level parallelism, speculation, caching, prefetching, and latency hiding.

Performance tuning is practically impossible without advanced performance analysis tools, such as Intel VTune [4] and AMD µProf [5], whose purpose it is to help developers answer two fundamental questions:

Q1 Which instructions does the application spend most time executing? Or in other words, which are the *performance-critical instructions*?[1]

Q2 Why are instructions performance-critical? What are the microarchitectural events (cache misses, branch mispredictions, etc.) that render these instructions performance-critical?

[1]While attributing time to functions can be sufficient to address simple performance issues, addressing challenging performance issues requires instruction-level analysis [6]. Note that instruction-level analysis can also always be aggregated for presentation at coarser granularity whereas the opposite is not true.

The first question (Q1) is typically addressed with a performance profiler. The state-of-the-art performance profiler TIP [6] is *time-proportional*, in contrast to other performance profilers [7]–[13], which means that the importance of an instruction in its final performance profile is proportional to the instruction's relative contribution to overall execution time. Time proportionality is achieved by analyzing an instruction's impact on performance at commit time because that is where an instruction's latency is exposed. More specifically, an instruction's key contribution to execution time is the fraction of time it prevents the core from committing instructions [6]. Time-proportional performance profiling is practical because it relies on *statistical sampling*, i.e., the profiler infrequently interrupts the CPU to retrieve the address(es) of the instruction(s) that the CPU is exposing the latency of in the cycle the sample is taken.

While performance profiling is a necessary first step, it is not sufficient because it does not answer the second question (Q2). More specifically, performance profilers such as TIP [6] do not explain *why* the architecture spends time on performance-critical instructions because they do not break down the time contribution of an instruction's execution across microarchitectural performance events. State-of-the-art approaches that attempt to address Q2 fall short because they account for performance events in a non-time-proportional manner, hence providing a skewed view on performance. Existing performance analysis approaches can be classified as instruction-driven versus event-driven. Instruction-driven approaches such as AMD IBS [9], Arm SPE [7], and IBM RIS [14] tag instructions at either the fetch or dispatch stage in the pipeline and then record the performance events that a tagged individual instruction is subjected to. Tagging instructions at the fetch or dispatch stages biases the instruction profile towards instructions that spend a lot of time in the fetch and dispatch stages, and not necessarily at the commit stage — hence lacking time-proportionality. Event-driven approaches [8], [10], [15], [16] on the other hand rely on counting performance events (e.g., cache misses, branch mispredicts, etc.). Event counts are then either attributed to instructions or used to generate coarse-grained performance information, such as application-level cycles per instruction stacks. Event-driven approaches also provide a skewed view on performance because the performance event counts they provide do not necessarily correlate with the impact these events have on performance because of latency hiding effects (as we will quantify in Section 5).

Our key insight is that both Q1 and Q2 can be answered by creating time-proportional *Per-Instruction Cycle Stacks (PICS)* in which the time the architecture spends executing each instruction is broken down into the (combinations of) performance events it encountered during program execution.[2] Since our PICS are time-proportional by design, they have the desirable properties that (i) the height of the cycle stack is proportional to a static instruction's impact on overall execution time — addressing Q1 — and (ii) the size of each component in the cycle stack is proportional to the impact on overall performance that this (combination of) performance event(s) incurs — addressing Q2. While time-proportional TIP [6] captures each static instruction's impact on overall execution time (thereby answering Q1), it cannot answer Q2 and create PICS because this requires breaking down each static instruction's performance impact across the events the instruction was subjected to during its dynamic executions.

A key challenge for creating PICS is that contemporary processors record many performance events, e.g., the Performance Monitoring Unit (PMU) of the recent Intel Alder Lake can report 297 distinct performance events [17]. Building time-proportional PICS however requires tracking events across all in-flight instructions — and limiting the number of tracked events is hence key to keeping overheads in check. We address this issue by returning to first principles, i.e., PICS must break down the execution time impact of an instruction according to the architectural behavior that caused the instruction's latency. We must hence focus on the commit stage and exploit that it can be in three non-compute states: (i) Commit *stalled* because an instruction reached the head of the Re-Order Buffer (ROB) before it had fully executed; (ii) it *drained* because of a front-end stall; or (iii) it *flushed* due to, for instance, a mispredicted branch. The task at hand is hence to map these states back to the performance events that caused them. Fortunately, performance events form hierarchies, and we exploit these to select events that make PICS easy to interpret while keeping overheads low. Surprisingly perhaps, we find that capturing only nine events is sufficient to ensure that 99% of the stall cycles incurred by instructions that are not subjected to any event is less than 5.8 clock cycles.

We hence propose *Time-proportional Event Analysis (TEA)*, which enables creating PICS by adding hardware support for tracking the performance events that each instruction encounters during its execution. More specifically, TEA allocates a *Performance Signature Vector (PSV)* for each dynamically executed instruction which includes one bit for each supported performance

[2] Performance-critical instructions are typically executed in (nested) loops and have many dynamic executions; we collect performance events across multiple sampled dynamic executions per static instruction in the binary.

event. During application execution, TEA uses a cycle counter to periodically collect PSV(s) at a typical 4 KHz sampling frequency. The PMU then retrieves the instruction pointer(s) and PSV(s) of the instruction(s) that the architecture is exposing the latency of at the time of sampling following the time-proportional attribution policies described in prior work [6]. When the sample is ready, the PMU interrupts the core, and the interrupt handler reads the instruction pointer(s) and PSV(s) and stores them in a memory buffer. When the application completes, the PSVs are post-processed to create PICS for each static instruction by aggregating the PSVs captured for each of its dynamic execution samples.

We implement TEA within the Berkeley Out-of-Order Machine (BOOM) core [18], and our implementation tracks nine performance events across all in-flight instructions. TEA incurs only minor overhead, i.e., requires 249 bytes of storage and increases per-core power consumption by only ~3.2 mW (~0.1%). We demonstrate the accuracy of TEA by comparing its PICS to those generated by AMD IBS [9], Arm SPE [7], and IBM RIS [14][3], which are the state-of-the-art instruction-driven performance analysis approaches, and an (unimplementable) golden reference that retrieves the PSVs for all dynamic instructions in all clock cycles. TEA is very accurate with an average error of 2.1% relative to the golden reference which is a significant improvement over the 55.6%, 55.5%, and 56.0% average error of IBS, SPE, and RIS, respectively. Since TEA relies on statistical sampling, the performance overhead of enabling it is only 1.1% on average.

To demonstrate that TEA is useful in practice, we used it to analyze the SPEC CPU2017 [19] benchmarks *lbm* and *nab*. For both benchmarks, the PICS provided by TEA explains the performance problems whereas state-of-the-art approaches do not. The performance problem of *lbm* is due to a non-hidden load instruction, and we address this issue by inserting software prefetch instructions. TEA enabled us to choose a prefetch distance that is large enough to hide most of the load latency while not being too large as this creates contention for store resources in the core and L1 cache, yielding an overall performance improvement of 1.28×. For *nab*, the high accuracy of TEA enabled us to deduce that a floating-point square root instruction was performance-critical because an earlier instruction flushed the pipeline and hence caused it to be issued too late for its execution latency to be

[3]The key benefit of the front-end-tagging strategy used by IBS, SPE, and RIS is that it only requires tracking performance events for a single instruction which yields a single byte of storage overhead (compared to 249 bytes for TEA). Unfortunately, this lower overhead comes at the cost of poor accuracy.

hidden. We addressed this issue by relaxing IEEE 754 compliance with the `-finite-math` and `-fast-math` compiler options which yielded speedups of 1.96× and 2.45×, respectively.

In summary, we make the following major contributions:

- We observe that time-proportional *Per-Instruction Cycle Stacks (PICS)* provide all the necessary information to explain both which instructions are performance-critical (i.e., answering Q1) and *why* these instructions are performance-critical (i.e., answering Q2) — thereby helping developers understand tedious performance problems.
- We propose *Time-proportional Event Analysis (TEA)* which captures the information necessary to create PICS by tracking key performance events for all in-flight instructions with *Performance Signature Vectors (PSVs)*.
- We implement TEA at the RTL-level within the out-of-order BOOM core [18] and demonstrate that it achieves a 2.1% average error relative to the golden reference; a significant improvement upon the 55.6%, 55.5%, and 56.0% error of IBS, SPE, and RIS, respectively. TEA has low overhead (i.e., storage overhead of 249 bytes, power consumption overhead of ~0.1%, and performance overhead of 1.1%).
- We used TEA to analyze the *lbm* and *nab* benchmarks from SPEC CPU2017. TEA identifies two performance problems that are difficult to identify with state-of-the-art approaches, and addressing them yields speedups of 1.28× and 2.45×, respectively.

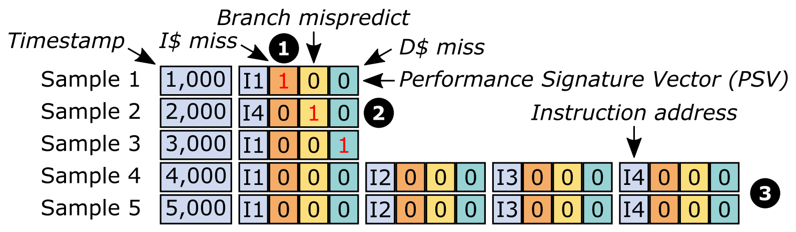
2 Background and Motivation

Time-proportional performance profiling [6] is based on the observation that determining the contribution of each instruction to overall execution time requires determining the instruction(s) that the core is currently exposing the latency of. (We assume a baseline that already supports TIP [6], the state-of-the-art time-proportional performance profiler.) Time-proportional profiling needs to focus on the commit stage of the pipeline because this is where the non-hidden instruction latency is exposed. Focusing on commit is a necessary but not a sufficient condition for time-proportionality because, depending on the state of the CPU, it will expose the latency of different instruction(s). More specifically, the processor will be in one of four commit states in any given cycle:

- **Compute:** The processor is committing one or more instructions. Time-proportionality hence evenly distributes time across the committing instructions (i.e., $1/n$ cycles to each instruction when n instructions commit in parallel).
- **Drained:** The ROB is empty because of a front-end stall, for instance due to an instruction cache miss. Time is hence attributed to the next-committing instruction.
- **Stalled:** An instruction I is stalled at the head of the ROB because it has not yet been fully executed. Time is hence attributed to I which is the next-committing instruction.
- **Flushed:** An instruction I caused the ROB to flush, for instance due to a mispredicted branch, and the ROB is empty. Time is hence attributed to I but unlike in the stall and drain states it has already committed, i.e., it is the last-committed instruction.

Explaining why it takes time to execute a particular instruction hence requires mapping the non-compute commit states Drained, Stalled, and Flushed to the performance events that caused them to occur. (Execution latency is fully hidden in the Compute state, and there is hence no additional execution time to explain in this state.)

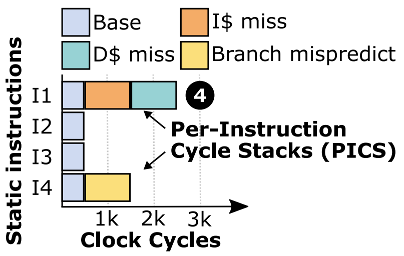
TEA example. Figure B.1 illustrates how TEA works in practice when an application executes the short loop in Figure B.1b on an out-of-order processor that supports three performance events (i.e., instruction cache miss, data cache miss, and branch mispredict). TEA relies on statistical sampling and for the purpose of this example we assume that it samples once every 1,000 cycles; the samples that TEA collects are shown in Figure B.1a. (In our evaluation, TEA samples at 4 kHz, i.e., once every 800,000 cycles at 3.2 GHz, which is the default for Linux `perf` [11].) In Sample 1, the ROB has drained due to an instruction cache miss when fetching I1 and TIP [6] hence samples I1. TEA additionally tracks the performance events that each dynamic instruction was subjected to by attaching a *Performance Signature Vector (PSV)* to each in-flight instruction. The PSV consists of one bit for each supported performance event and hence consists of three bits in this example. Since I1 was subjected to an instruction cache miss, its instruction cache miss event bit is set in its PSV, see ❶. A TEA sample hence consists



(a) TEA samples.

```
IX ...
I1 lw   a4,(a5)
I2 addw a0,a0,a4
I3 addi a5,a5,4
I4 bne  a5,a3,I1
IY ...
```

(b) Example code.



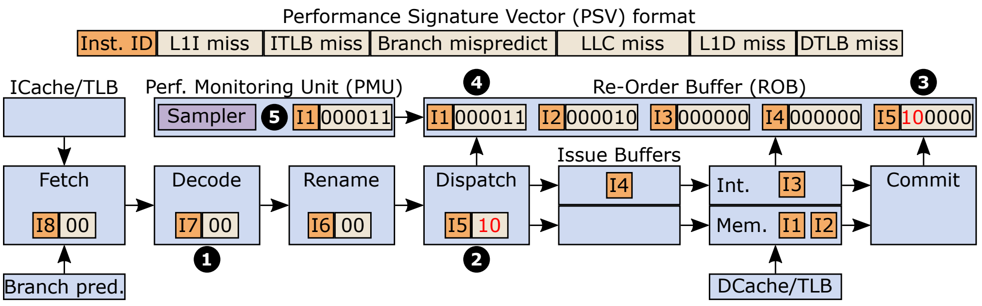
(c) Time-proportional PICS.

Figure B.1: Example explaining how TEA creates PICS. *TEA explains how performance events cause performance loss.*

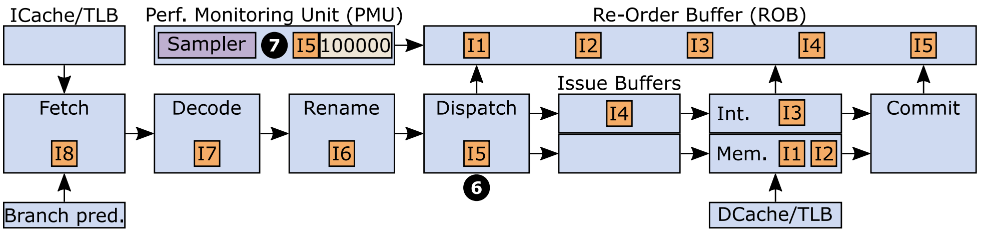
of a PSV for all sampled instructions in addition to the information returned by TIP (i.e., instruction address(es) and timestamp).

In Sample 2, the ROB has again emptied, but now the reason is that branch instruction I4 was mispredicted. I4 hence committed while all younger instructions were squashed, resulting in the processor being in the Flushed state. The sample is hence attributed to I4, and TEA provides a PSV where the branch mispredict bit is set, see ❷. In Sample 3, I1 is again the cause of performance loss, but this time it is stalled on a cache miss. The processor is therefore in the Stalled state, and TEA explains why because the data cache miss event bit is set in the PSV. In Samples 4 and 5, the working set of I1 has been loaded into the L1 cache and the branch predictor has learned how to predict I4 correctly. The 4-wide core is thus able to commit I1, I2, I3, and I4 in parallel and is in the Compute state. All PSV entries are 0 because none of the instructions were subjected to any performance event, see ❸.

The application terminates without additional samples being collected, and TEA then uses the captured samples to create PICS for I1, I2, I3, and I4 (see Figure B.1c). Each sample is mapped to static instructions using the address(es) of the instruction(s) and then categorized according to the



(a) TEA.



(b) Dispatch-tagging (e.g., IBS).

Figure B.2: Example comparing TEA to dispatch-tagging. *TEA is time-proportional whereas dispatch-tagging is not.*

PSV value — which identifies the (combination of) performance event(s) that caused the processor to expose the latency of this instruction in this sample. From Samples 1 and 3, TEA attributes 1,000 cycles to I1 due to the instruction cache miss event and data cache miss event, respectively, see ❹. Similarly, TEA attributes 1,000 cycles to I4 for the mispredicted branch in Sample 2. The remaining cycles are distributed evenly across I1, I2, I3, and I4 since they commit in parallel in Samples 4 and 5. This category is labeled 'Base' since none of the instructions were subjected to performance events.

If an instruction is subjected to multiple events, multiple bits are set in the PSV, and we refer to events that impact the same instruction as combined events. Combined events are often serviced sequentially, e.g., an instruction cache miss must resolve for a load to be executed and subjected to a data cache miss. The stall cycles caused by this load are hence caused by both events and it is challenging to tease apart the stall impact of each event. TEA hence reports combined events as separate categories. Out of all dynamic instruction executions that encounter at least one event, 30.0% are subjected to combined events (see Section 5). Combined events are hence not too common, but can help to explain challenging issues.

Capturing PSVs. Creating PICS requires recording the performance events each instruction is subjected to during its execution, i.e., creating a PSV for each instruction packet. An instruction packet is the instruction (or µop) itself and its associated meta-data (e.g., the instruction address) which the processor updates and forwards as the instruction flows through the pipeline. Figure B.2a illustrates how TEA captures PSVs by showing the execution state of an out-of-order core and PSVs for each instruction; this architecture supports six performance events and hence has a six-bit PSV format. (We will explain how we implement TEA in detail in Section 3.) In the front-end, the PSVs need to capture and pass along the events that can occur in this and previous pipeline stages which are instruction cache and TLB misses in this example, see ❶. At dispatch, TEA initializes the six-bit PSV associated with each ROB entry by setting the front-end bits of the PSV to their respective values and all remaining PSV-bits to zero. At ❷, instruction I5, which was subjected to an instruction cache miss, hence has its two most significant bits set to 10 as these are the front-end PSV-bits (see ❸). In the cycle we focus on, I1 is the oldest instruction and stalled due to an L1 cache miss and a TLB miss and its two least significant PSV-bits are hence both 1, see ❹. Similarly, I2 is also a data cache miss while the PSVs for I3 and I4 are all zeros because they so far have not been subjected to any performance events. Since TEA is time-proportional, its hardware sampler selects I1 and its PSV before interrupting the processor such that the software sampling function can retrieve the sample and store it in a memory buffer.

Instruction-driven performance analysis. AMD IBS [9], Arm SPE [7], and IBM RIS [14] fall short because they tag instructions at dispatch or fetch and are hence not time-proportional. Figure B.2b illustrates the operation of dispatch-tagging with the same core state as we used to explain TEA in Figure B.2a. Dispatch-tagging marks the instruction that is dispatched in the cycle the sample is taken, i.e., I5 (see ❻). Fetch-tagging works in the same way except that it tags at fetch rather than at dispatch and would hence tag I8 in this example. The key benefit of tagging instructions in the front-end is that the scheme only needs to track events for the tagged instruction, i.e., it needs one PSV to record the events that the tagged instruction is subjected to, see ❼.

Tagging at dispatch or fetch does however incur significant error because it is not time-proportional. More specifically, sampling I5 or I8 is not time-proportional because I1 is stalled at the head of the ROB at the time the sample is taken, i.e., the processor is exposing the latency of I1 in this cycle.

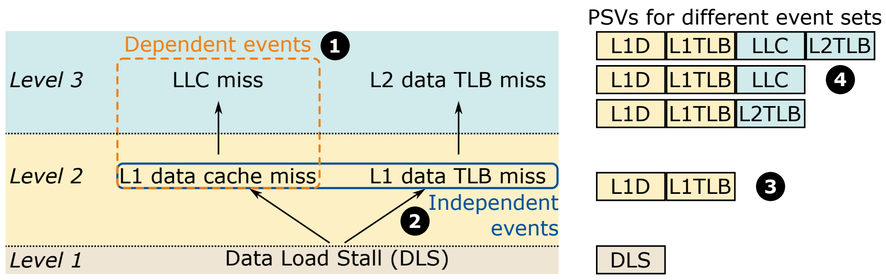


Figure B.3: Performance event hierarchy for the Stalled (ST) commit state.

(Recall that TEA sampled I1 in Figure B.2a.) This situation is common because performance-critical instructions tend to stall at commit which in turn stalls the front-end — resulting in the PSVs of the instructions that are dispatched or fetched during stalls being overrepresented in the PICS. Tagging at dispatch or fetch also captures events that may not impact performance. For example, I1 is stalled on a combined data cache and TLB miss event, but dispatch-tagging captures I5's instruction cache miss (which is hidden under I1's events).

3 Time-Proportional Event Analysis

We now explain the details of our TEA implementation. While we focus on the open-source BOOM core [18] in this section, the approach will be similar for other microarchitectures, i.e., some implementation details will be different but the flow of information will remain the same.

Performance event hierarchies. PICS help developers understand why instructions are performance-critical, and TEA provides this information by mapping the non-compute commit states to the most important performance events that cause them. However, TEA has to track performance events for all in-flight instructions, and we hence need to carefully select a small set of performance events that collectively capture key architectural bottlenecks to keep overheads in check. Fortunately, performance events can be grouped according to the non-compute commit state they can cause. Performance events hence form hierarchies that we can exploit to trade off overhead against interpretability, i.e., the ability of the selected set of performance counters to explain commit stalls.

Table B.1: The performance events of TEA, IBS, SPE, and RIS.

Event	**Description**	**TEA**	**IBS**	**SPE**	**RIS**
DR-L1	L1 instruction cache miss	✓	✓	✗	✓
DR-TLB	L1 instruction TLB miss	✓	✓	✗	✓
DR-SQ	Store instruction stalled at dispatch	✓	✗	✗	✓
FL-MB	Mispredicted branch	✓	✓	✓	✓
FL-EX	Instruction caused exception	✓	✗	✓	✗
FL-MO	Memory ordering violation	✓	✗	✗	✗
ST-L1	L1 data cache miss	✓	✓	✓	✓
ST-TLB	L1 data TLB miss	✓	✓	✓	✓
ST-LLC	LLC miss caused by a load instruction	✓	✓	✓	✓

Figure B.3 explains how event hierarchies enable reasoning about event selection by focusing on the Stalled (ST) commit state. Performance events can be dependent or independent. Dependent performance events can only occur if a prior performance event has occurred, e.g., a load can only miss in the LLC if it has already missed in the L1 cache ❶. Independent performance events in contrast occur independently of each other, e.g., a load can hit in the L1 cache independently of it hitting or missing in the L1 TLB ❷. We can hence exploit the event hierarchy to balance how easy it is for a developer to interpret PICS — which favors capturing more events and thereby explaining increasingly complex architectural behaviors — against overheads — which increases with event count because TEA must track events for all in-flight instructions.

We refer to the events captured by a PSV as an *event set*. For the events in Figure B.3, we can create one single-bit PSV which only captures that a load stall occurred and hence has low overhead but offers limited insight. We can improve interpretability by moving to a 2-bit PSV. In this case, the most favorable option is to include the L1 data cache and TLB miss events as they cover all possible Level 2 events in the event hierarchy, see ❸. We can improve interpretability by adding the dependent events of the L1 data and TLB misses as exemplified by the 3-bit and 4-bit PSVs ❹. In this case, we still need to report the root event of each dependency chain to avoid losing interpretability. For example, if we capture LLC misses and not L1 misses, we can no longer identify LLC hits.

TEA's performance events. Table B.1 lists the nine performance events that TEA captures in our BOOM implementation. We name the performance events on the form *X*-*Y* where *X* is the commit state and *Y* is the event, e.g., an L1 data cache miss is labeled ST-L1 since it explains the Stalled commit state. To explain the Drained state, TEA captures that an instruction missed in the L1 instruction cache (DR-L1), missed in the L1 instruction TLB (DR-TLB), and that the ROB drains due to a full store queue (DR-SQ). The DR-SQ event captures the case where the ROB drains because a store cannot dispatch because the Load/Store Queue (LSQ) is full of completed but not yet retired stores; this improves interpretability when the application is sensitive to store bandwidth. For the Flushed state, TEA captures that an instruction is a mispredicted branch (FL-MB), caused an exception (FL-EX), and caused a memory ordering violation (FL-MO). A memory ordering violation occurs when a load executes before an older store to the same address and hence has read stale data. It is addressed by re-executing the load and squashing all younger in-flight instructions (which is time-consuming). To explain the Stalled state, TEA captures L1 data cache misses (ST-L1), L1 data TLB misses (ST-TLB), and LLC misses caused by load instructions (ST-LLC). Capturing LLC misses improves interpretability for memory-sensitive applications.

TEA exploits event hierarchies to balance interpretability and overhead. Retaining interpretability means that TEA should assign events to instructions that caused long stalls, i.e., stalls that cannot be explained by instruction execution latencies and dependencies, because these determine the expected stall time in the absence of miss events. We evaluate TEA from this perspective by capturing the stalls caused by any dynamic instruction. Our golden reference provides this data because it captures all dynamic instructions and all clock cycles (see Section 4 for details regarding our experimental setup). We further extract the instructions that stall commit and TEA does not assign an event to. Overall, 99% of these instructions cause stalls that are shorter than 5.8 clock cycles, and TEA hence captures the events that can majorly impact performance.

Table B.1 also shows that the instruction-driven approaches AMD IBS [9], [20], Arm SPE [7], [21], and IBM RIS [14] capture many of the same events as TEA which indicates that some events are important regardless of the specific architecture.

TEA microarchitecture. Figure B.4 illustrates how we implement TEA in the BOOM core. The DR-L1 and DR-TLB events occur in Fetch which requires allocating a 2-bit PSV in the fetch packet, see ❶. Because the first instruction of the fetch packet always incurs the DR-L1 and DR-TLB events,

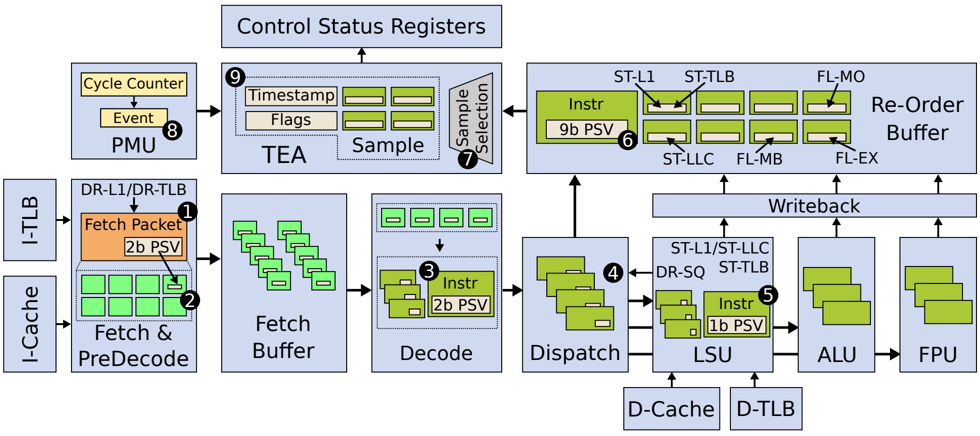


Figure B.4: TEA microarchitecture.

TEA only requires a single PSV ❷. When the fetch packet is expanded into individual instructions and added to the Fetch Buffer, the PSV of the first instruction is copied and the PSVs of all other instructions are initialized to zero. In Decode, the instructions from the fetch buffer are decoded into μops and the PSV of each μop is passed along ❸. Dispatch inserts μops into the ROB and the issue queues of the functional units. Dispatch detects DR-SQ when a store is the oldest μop and cannot dispatch due to a full LSQ ❹. To avoid complicating the LSU-to-ROB interface, we allocate storage for an ST-TLB event in each LSU entry because it is detected before the cache responds ❺. ST-L1 and ST-LLC events in contrast become available upon a cache response and can hence be communicated immediately (through Writeback). The complete 9-bit PSV of each μop is stored in the μop's ROB-entry ❻. The FL-MB, FL-EX, and FL-MO events are already detected by the ROB because they require flushing the pipeline, and the ROB can hence record them in the PSV.

TEA is connected to the head of the ROB with the time-proportional sample selection logic inherited from TIP [6] ❼. Once a cycle counter event is emitted by the PMU (see ❽), the Sample Selection unit identifies the commit state (i.e., Computing, Stalled, Flushed, or Drained) and selects the appropriate instruction(s) given the state. TEA delays returning the sample in the Stalled and Drained state until the next μop commits to ensure that the μop's PSV is updated. A sample contains a timestamp, flags (i.e., commit state and valid bits) as well as the instruction address(es) and PSV(s) ❾. TEA is hence indifferent to tracking μops or dynamic instructions since it in both cases maps them to static instructions when sampling. Finally, the sample is written to TEA's Control and Status Registers (CSRs) and an interrupt is issued.

Sample collection and PICS generation. The interrupt causes the sampling software to retrieve TEA's sample as well as inspect other CSRs to determine the logical core identifier and process and thread identifiers before writing all of this information to a buffer in memory (which is flushed to a file when necessary); this is the typical operation of Linux `perf` [11]. The logical core identifier maps to a hardware thread under Simultaneous Multi-Threading (SMT) and a physical core otherwise; we require one TEA unit per physical core. While we focus on single-threaded applications in this work, TEA is hence equally applicable to multi-threaded applications since we capture sufficient information to create PICS for each thread. The ability of profiling tools to map samples to processes also enables creating PICS for any piece of software (e.g., operating system code and just-in-time compilers).

All collected samples are hence available in a file when the application terminates. We have created a tool that takes this sample file as input and then aggregates cycles across the PSV signatures of each static instruction, thereby creating PICS for each static instruction in which each category corresponds to a specific (combination of) performance event(s). A developer can then use this tool to analyze application performance by visualizing PICS at various granularities (e.g., static instructions and functions).

Overheads. We assume a baseline that implements TIP [6], and TIP incurs a storage overhead of 57 B compared to an unmodified BOOM core. TEA additionally needs to track PSVs across all in-flight instructions and hence requires adding two bits per entry in the 48-entry fetch buffer to store the DR-L1 and DR-TLB events (12 B) and a 9-bit PSV field to each ROB entry (216 B for our 192-entry ROB). TEA also needs three 2-bit registers in fetch to track DR-L1 and DR-TLB for all fetch packets and 2 bits for each entry in decode and dispatch to track these events through the rest of the front-end. TEA needs a one-bit register in dispatch to track the DR-SQ event and one bit in each LSU entry to track ST-TLB until the load completes. TEA also needs a register for the PSV of the last-committed instruction to correctly handle the Flushed state (2 B). The overall storage overhead of TEA is hence 249 B per core (and 306 B per core for TEA and TIP).

Since IBS, SPE, and RIS tag instructions in the front-end, they know which instruction to capture PSVs from and hence only require storing 6, 5, and 7 bits, respectively, i.e., one byte. They do however capture other information such as branch targets, memory addresses, and various latencies when implemented in commercial cores. The minimum storage requirements of IBS, SPE, and RIS are hence negligible, but this benefit is due to tagging instructions in the front-end which is also the root cause of their large errors.

To better understand the power overhead of TEA (and TIP), we synthesized the ROB and fetch buffer modules of the BOOM core in a commercially available 28 nm technology with and without TEA using Cadence Genus [22] and estimated its power consumption with Cadence Joules [23]. We focus on the ROB and fetch buffer because they account for 91.7% of TEA's storage overhead. (Recall that the events TEA captures are already identified in the microarchitecture.) Overall, TEA increases the power consumption of these units by 4.6%. In absolute terms, supporting TEA in these units increases power consumption by 3.2 mW which is negligible. For example, RAPL [24] reports a core power consumption of 32.7 W on a recent laptop with an Intel i7-1260P chip running `stress-ng` on all 8 physical cores which yield 4.7 W per core. Implementing TEA on this system would hence increase per-core power consumption by ~0.1%. If this power overhead is a concern, the PSVs can be clock or power-gated and enabled ahead of time such that the PSVs for all in-flight instructions are updated when sampling.

TEA's performance overhead is the same as TIP because we can pack the PSVs into the CSR that TIP uses to communicate sample metadata to software. A CSR must be 64 bit wide to match the width of the other registers in the architecture, but TIP only uses 10 bits for metadata. Communicating four PSVs requires 36 bits which result in TEA using 46 out of 64 CSR bits. TEA hence retains the 88 B sample size from TIP which results in a performance overhead of 1.1% [6]. TEA's logic is not on any critical path of the BOOM core, and TEA hence does not impact cycle time.

4 Experimental Setup

Simulator. We use FireSim [25], a cycle-accurate FPGA-accelerated full-system simulator, to evaluate the different performance profiling strategies. The simulated model is the BOOM 4-way superscalar out-of-order core [18], see Table B.2 for its configuration, which runs a common buildroot 5.7.0 Linux kernel. The BOOM core is synthesized to and runs on Xilinx U250

Table B.2: Baseline architecture configuration.

Part	**Configuration**
Core	OoO BOOM [18]: RV64IMAFDCSUX @ 3.2 GHz
Front-end	8-wide fetch, 48-entry fetch buffer, 4-wide decode, 28 KB TAGE branch predictor, 60-entry fetch target queue, max. 30 outstanding branches
Execute	192-entry ROB, 192 integer/floating-point physical registers, 48-entry dual-issue memory queue, 80-entry 4-issue integer queue, 48-entry dual-issue floating-point queue
LSU	64-entry load/store queue
L1	32 KB 8-way I-cache, 32 KB 8-way D-cache w/ 16 MSHRs, 64 SDQ/RPQ entries, next-line prefetcher
LLC	2 MiB 16-way dual-bank w/ 12 MSHRs
TLB	Page Table Walker, 32-entry fully-assoc L1 D-TLB, 32-entry fully-assoc L1 I-TLB, 1024-entry direct-mapped L2 TLB
Memory	16 GB DDR3 FR-FCFS quad-rank, 16 GB/s maximum bandwidth, 14-14-14 (CAS-RCD-RP) latencies @ 1 GHz, 8 queue depth, 32 max reads/writes
OS	Buildroot, Linux 5.7.0

FPGAs in NTNU's Idun cluster [26]. We account for the frequency difference between the FPGA-realization of the BOOM core and the FPGA's memory system using FireSim's token mechanism. We use TraceDoctor [27] to capture cycle-by-cycle traces that contain the instruction address and the valid, commit, exception, and flush flags as well as the PSV of the head ROB-entry in each ROB bank; the trace includes the ROB's head and tail pointers which we need to model dispatch-tagging. We configure a highly parallel framework of TraceDoctor workers on the host to enable on-the-fly processing while minimizing simulation slowdown. The performance analysis approaches are hence modeled on the host CPUs that operate in parallel with the FPGA by processing the traces. This allows us to simulate and evaluate multiple configurations out-of-band in a single simulation run; we run up to 15 configurations on 12 CPUs per FPGA simulation run. We evaluate multiple configurations with a single run because (i) it enables fairly comparing analysis approaches as they sample in the exact same cycle, and (ii) it reduces evaluation time.

Benchmarks. We run a broad set of SPEC CPU2017 [19] benchmarks that are compatible with our setup. We simulate the benchmarks to completion using the reference inputs. We compile all benchmarks using GCC 10.1 with the `-O3 -g` compilation flags and static linking. We enable the performance analyzers when the system boots up until the system shuts down after the benchmark has terminated. We only retain the samples that hit user-level code because (i) the time our benchmarks spend in OS code (e.g., syscalls) is limited (1.7% of total time), and (ii) we do not want to include system boot and shutdown time in the profiles.

Golden reference. The baseline we compare against computes PICS for every instruction, i.e., we know for each instruction how it contributes to the total execution time and where it spends its time — we consider this to be our golden reference. This is clearly impractical to implement in a real system because it would require communicating the PSVs to software for every dynamically executed instruction which would incur too high performance overhead. More specifically, the golden reference requires communicating and parsing 2.7 petabytes of performance data in total at a rate of 116 GB/s. This golden reference is nevertheless extremely useful because it represents the ideal performance profile to compare against.

Error metric. Quantifying the accuracy of the cycle stacks obtained by TEA (or any other technique) requires an error metric that quantifies the error across all components in the cycle stack. Moreover, we want to be able to compute the error metric at the level of granularity at which the cycle stack is computed. We consider instruction and function granularities in this work. We refer to a component in the cycle stack as $C_{i,j}$, $1 \leq j \leq N$ with N being the number of components in the stack and i being a unit of granularity, i.e., an instruction, a basic block, a function or the entire application. The corresponding component in the cycle stack as obtained through the golden reference is referred to as $C^R_{i,j}$. The correctly attributed cycle count per component hence equals $\min(C_{i,j,}, C^R_{i,j})$. Summing up these correctly attributed cycle counts across all components and all units G at the granularity of interest yields the total number of correctly attributed cycles, i.e., $T_{correct} = \sum_{i=1}^{G}\sum_{j=1}^{N}\min(C_{i,j,}, C^R_{i,j})$. The error is defined as the relative difference between the total cycle count T_{total} and the correctly attributed cycle count, i.e., $E = (T_{total} - T_{correct})/T_{total}$. Not all techniques that we evaluate generate the same set of components. In particular, IBS, SPE, and RIS do not provide the same components as TEA. For fair comparison against the golden reference, we hence compare each scheme against a golden reference with the same set of components as the scheme supports.

5 Results

The state-of-the-art approaches for creating Per-Instruction Cycle Stacks (PICS) are represented by **IBS**, **SPE**, and **RIS** which are our best-effort implementations[4] of AMD IBS [20], Arm SPE [7], and IBM RIS [14]. IBS and SPE tags instructions at dispatch whereas RIS tags instructions while forming instruction groups in the fetch stage. IBS, SPE, and RIS all record the performance events that tagged instructions are subjected to while they travel through the pipeline but support different event sets (see Table B.1). We also compare against two variants of TEA. **NCI-TEA** combines the events supported by TEA with the Next-Committing Instruction (NCI) sampling policy used by Intel PEBS [8] which has been shown to be significantly more accurate than tagging instructions at fetch or dispatch [6]. **TEA** is our approach as described in Section 3 which uses time-proportional PSV sampling. We sample instructions at a frequency of 4 KHz for all techniques, unless mentioned otherwise. We also evaluated a version of TEA that tags instructions at dispatch which yields similar accuracy to IBS, SPE, and RIS, but we could not include this configuration due to page restrictions.

5.1 Average Accuracy

We first focus on the accuracy of TEA for generating PICS, and Figure B.5 reports error per benchmark. A couple of interesting observations can be made. First, IBS, SPE, and RIS are significantly less accurate than NCI-TEA and TEA. The reason is that IBS, SPE, and RIS tag instructions at dispatch or fetch which leads to non-time-proportional performance profiles. (This confirms the observation from prior work [6].) Fundamentally, tagging an instruction in the front-end skews the profile to instructions that spend a significant amount of time at dispatch or fetch — which are not necessarily the instructions the application spends time on at commit. RIS performs slightly worse than IBS and SPE because it captures more events and the cycle stacks thus have more components. By consequence, accurately capturing all components in the stack is more challenging. The marginal difference between IBS and SPE is also due to capturing different event sets.

[4]While we took great care to implement and configure IBS, SPE, and RIS as faithfully as possible, we are ultimately limited by the information that has been disclosed publicly. The fundamental issue with these approaches is however that they are not time-proportional.

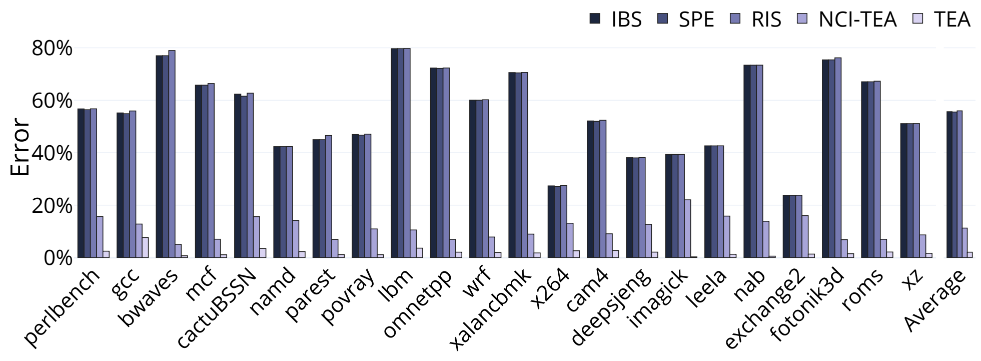
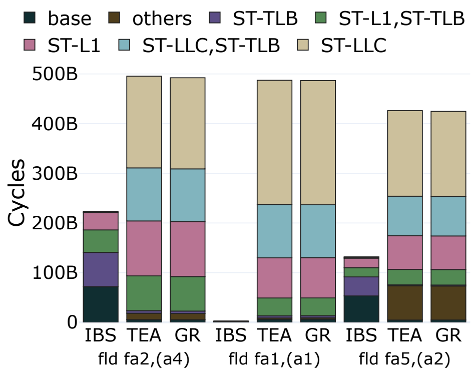


Figure B.5: Quantifying the error for the PICS obtained through IBS, SPE, RIS, NCI-TEA, and TEA. *TEA achieves the highest accuracy within 2.1% (and at most 7.7%) compared to the golden reference.*

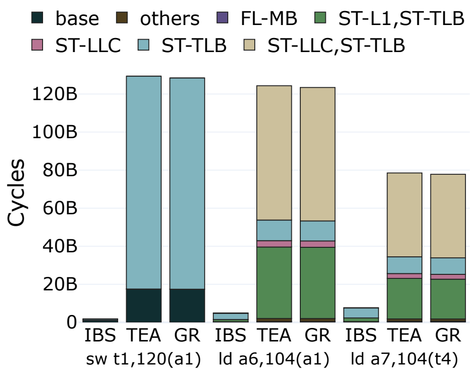
Second, sampling instructions at commit substantially improves accuracy as is evident from comparing NCI-TEA versus IBS, SPE, and RIS. NCI-TEA samples the instructions as they contribute to execution time, i.e., an instruction that stalls commit has a higher likelihood of being sampled, and, as a result, the cycle stack is more representative of the contribution of this instruction to the program's overall execution time.

Third, sampling at commit is not a sufficient condition for obtaining accurate cycle stacks. We need to attribute the sample to the correct instruction *and* we need to attribute the sample to the correct signature. Attributing the sample to the next-committing instruction (NCI) is inaccurate in case of a pipeline flush due to a mispredicted branch or an exception. The instruction which is to blame is not the next-committing instruction but the instruction that was last committed, namely the mispredicted branch or the excepting instruction. TEA solves this issue by keeping track of the PSV of the last-committing instruction as previously described in Section 3.

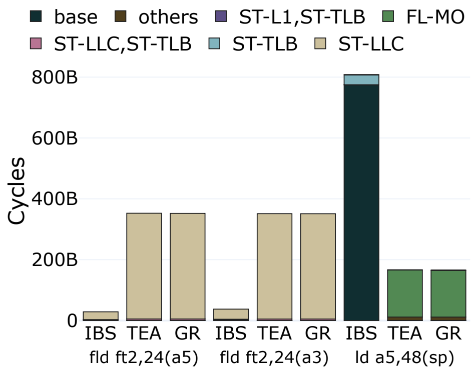
Overall, TEA achieves an average error of 2.1% (and at most 7.7%). This is significantly more accurate compared to the other techniques: NCI-TEA (11.3% average error and up to 22.0%), RIS (56.0% average error and up to 79.7%), IBS (55.6% average error and up to 79.7%), and SPE (55.5% average error and up to 79.7%).



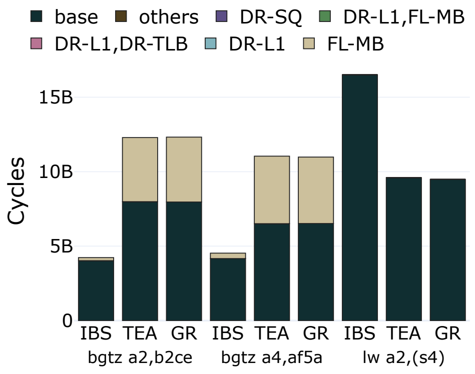
(a) bwaves



(b) omnetpp



(c) fotonik3d



(d) exchange2

Figure B.6: PICS for the top-3 instructions as provided by IBS, TEA, and the golden reference (GR). *The PICS provided by TEA are accurate compared to the golden reference, in contrast to IBS.*

5.2 Per-Instruction Accuracy

The previous section quantified the average accuracy of the PICS across all instructions within a benchmark. We now zoom in on the accuracy for individual instructions. Figure B.6 reports the PICS of the top-3 (most execution time) instructions for four benchmarks for IBS, TEA, and the golden reference; we take IBS as representative of SPE and RIS since their accuracy is very similar (see Figure B.5). We select *bwaves*, *omnetpp*, and *fotonik3d* because they collectively illustrate how TEA reports solitary versus combined events, and *exchange2* because it is the benchmark for which IBS yields the lowest error. The overall conclusion is that the PICS reported by IBS are inaccurate for two reasons: (i) the height of the cycle stacks is inaccurate because IBS is not time-proportional, and (ii) the relative importance of the components within the cycle stacks is inaccurate because of signature misattribution. This also applies to *exchange2* which is the benchmark for which IBS incurs the lowest error (i.e., comparing Figure B.6d to Figure B.5).

This analysis also illustrates TEA's ability to detect combined events. For example, the combination of cache and TLB misses, i.e., (ST-L1, ST-TLB) and (ST-LLC, ST-TLB), accounts for a significant fraction of the PICS of the top-3 instructions for *bwaves* and *omnetpp* (see Figures B.6a and B.6b). Out of all dynamic instruction executions that are subjected to at least one event, 30.0% encounter combined events. Combined events are hence not too common, but they can help explain specific performance problems. Optimizing *bwaves* would for example require improving both cache and TLB utilization, whereas optimizing *fotonik3d* can focus solely on improving cache utilization (see Figures B.6a and B.6c).

5.3 Why Event-Driven Analysis Falls Short

As aforementioned in the introduction, event-driven performance analysis attempts to answer question (Q2) of why instructions are performance-critical by counting performance events (e.g., cache misses, TLB misses, branch mispredicts, etc.). This is a widely used approach for software tuning. Unfortunately, it is extremely tedious and time-consuming and appears to be more of an art than a science, i.e., performance tuning requires intimate familiarity with the code and the underlying hardware. The fundamental reason is that event counts do not necessarily correlate with the impact these events have on overall performance. Having developed a method to compute accurate PICS, we can now quantify the adequacy of performance event counting.

We do this by computing the correlation between event counts and the corresponding components in the cycle stack. We compute the Pearson correlation coefficient r which varies between -1 and +1. In our context, r close to one implies an almost perfect positive correlation; on the other hand, r close to zero means no correlation. Figure B.7 reports box plots for the Pearson correlation coefficient for all PSV events across all benchmarks.[5]

[5]Event-driven approaches such as Intel PEBS [8] and DCPI [10] cannot detect combined events because they must fundamentally sample based on the event they are counting; many events only apply to certain instruction types (e.g., only loads and stores can miss in the cache). When counting multiple events in parallel, the events will not be

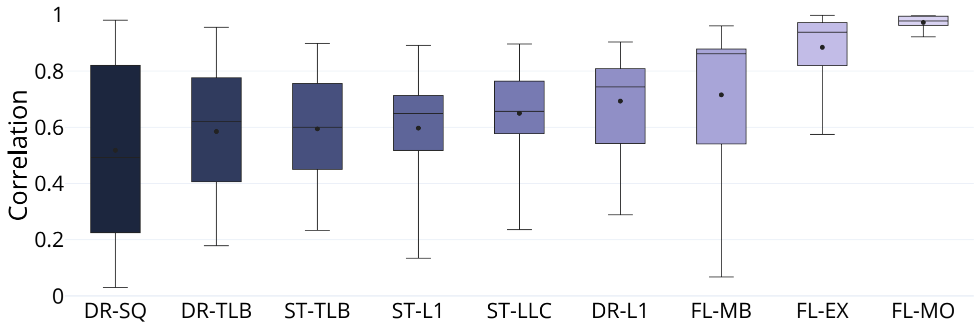


Figure B.7: Quantifying the correlation between event count and its impact on performance. *Some event counts correlate strongly with their impact on performance while others do not.*

Some performance events strongly correlate with performance, as is the case for branch mispredictions (FL-MB), exceptions (FL-EX), and memory ordering violations (FL-MO). The reason is that these events lead to a pipeline flush, which in most cases cannot be hidden. TLB misses (DR-TLB and ST-TLB) and cache misses (ST-L1, ST-LLC, and DR-L1) on the other hand show moderate correlation with performance, with LLC misses (SL-LLC) showing higher correlation than L1 data cache misses (ST-L1). The reason is that cache misses can be partially hidden, and this is true more so for L1 data cache misses than for LLC misses. The least correlation and the largest spread are observed for store queue stalls (DR-SQ), i.e., in some cases, a full store queue is completely hidden while in other cases a full store queue stalls the processor.

While the above analysis is intuitively understood, i.e., architects are well aware of latency hiding effects, this work is the first to quantify the (lack of) correlation between event counts and their impact on performance. This is also the fundamental reason why performance tuning using event counts is so tedious and time-consuming. TEA solves this problem by providing accurate PICS.

5.4 Sensitivity analysis

Sampling frequency. Figure B.8 reports the accuracy of the various techniques as a function of sampling frequency. Accuracy is rather insensitive to

captured in the same cycle, yielding independent profiles.

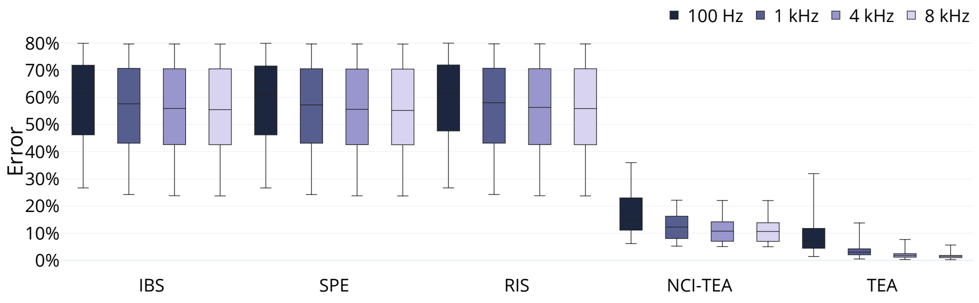
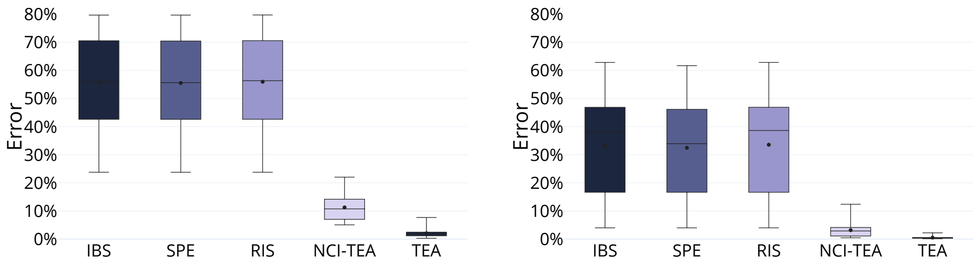


Figure B.8: Error versus sampling frequency.

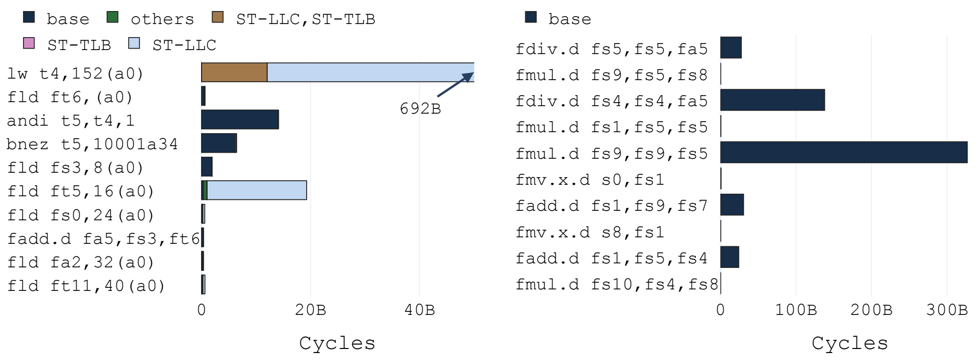


(a) Instruction **(b)** Function

Figure B.9: Errors at instruction and function granularity.

sampling frequency above 4 KHz, which is why we chose it as our baseline sampling frequency to balance accuracy and run-time overhead.

Analysis granularity. Figure B.9 evaluates the accuracy of the various techniques when cycle stacks are computed at the instruction and function granularities; basic block and application granularities exhibit the same trends. TEA is uniformly the most accurate technique. While the error decreases at function granularity for the alternative approaches, it does not decrease as steeply as one may expect. The reason is that cycles are systematically misattributed to the wrong events. As a result, the alternative approaches fall short, even at coarse granularity. This reinforces the need for a more adequate analysis technique such as TEA.



(a) PICS generated by TEA. **(b)** PICS generated by IBS.

Figure B.10: Lbm performance analysis. *TEA identifies the performance-critical load whereas IBS does not.*

6 Case Studies

We now demonstrate that TEA — by identifying the performance-critical instructions (Q1) and explaining why they are performance-critical (Q2) — comprehensively identifies application optimization opportunities that state-of-the-art approaches miss by analyzing and optimizing *lbm* and *nab*. As in Section 5.2, we take IBS as representative of SPE and RIS.

Analyzing lbm. When using current state-of-the-art approaches, the first step is to collect a performance profile. If we use TIP [6], the profile is time-proportional and hence reports the contribution of each static instruction to overall execution time (i.e., answers Q1). TIP however does not explain why a particular instruction is performance-critical and therefore forces developers to guess what the problem could be from the instruction type and TIP's flags. In the case of *lbm*, TIP will identify the performance-critical load instruction and, unsurprisingly perhaps, report that this load stalls commit.

TEA in contrast provides PICS as shown in Figure B.10a which (i) identify the performance-critical `lw` instruction — thereby answering Q1 — and (ii) explains that this `lw` instruction always misses in the LLC while hiding the latency of the following load instructions that also miss in the LLC — hence answering Q2; TEA's PICS are practically identical to the PICS generated by the golden reference. Figure B.10b shows PICS generated by IBS for the region of the code which it identifies as performance-critical. IBS attributes the performance problem to some floating-point arithmetic instructions that

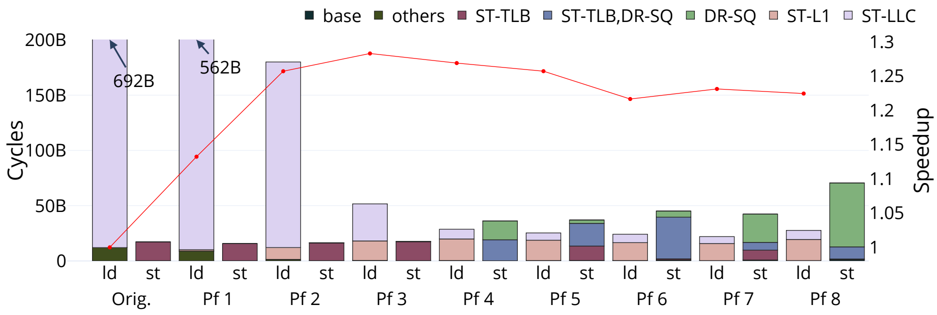
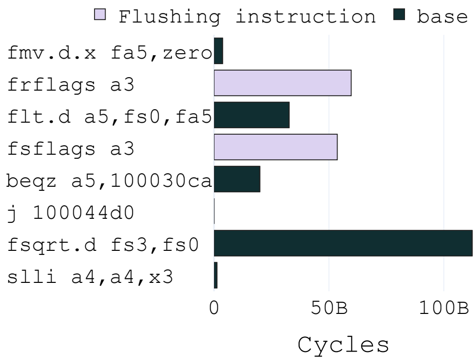


Figure B.11: PICS and speedup for the most performance-critical load instruction and store instruction of lbm across a range of prefetch distances.

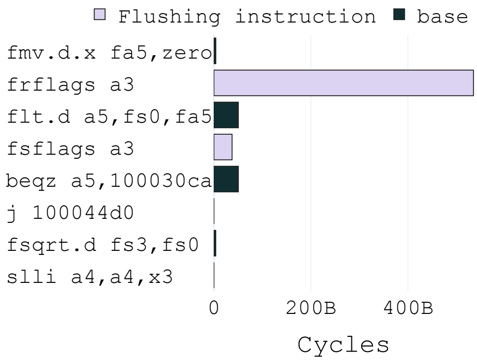
happen to dispatch while the performance-critical `lw` instruction is stalled at the head of the ROB. The event-driven analysis is also unclear because *lbm* has 11 load instructions in the inner loop which all incur between 3.3 and 3.9 billion misses each. The key problem is that event counting does not differentiate between hidden and non-hidden misses.

TEA explains that the key performance problem of *lbm* is that (i) its working set exceeds the size of the LLC, and (ii) the architecture is not able to issue the load instructions sufficiently early to hide their latency. More specifically, the body of the inner loop of *lbm* contains sufficient compute instructions to fill the ROB and hence blocks the processor from issuing the loads of the next iteration while processing a previous iteration. TEA, unlike TIP, IBS, and event counting, provides all of this information in its PICS — and thereby explains that software prefetching is the optimization to apply.

Optimizing lbm. Applying software prefetching is challenging because the developer must insert prefetches sufficiently early to hide memory latencies while at the same time taking care not to bottleneck other core resources (e.g., the LSQ) or pollute the caches. (Since the BOOM core does not support software prefetching, we implemented a custom software prefetch instruction using its ROCC interface.) Figure B.11 shows the TEA-generated PICS for the most performance-critical load and store instructions when issuing software prefetches for the three cache lines *lbm* requires to execute the body of its inner loop n iterations before it is used (we refer to this as a prefetch distance of n). The PICS show that as we increase prefetch distance, the impact of the most performance-critical load instruction on overall execution time goes down until it saturates at prefetch distance 4, i.e., LLC hits (ST-L1) accounts for most of its execution time impact. This increases performance



(a) PICS generated by TEA.



(b) PICS generated by IBS.

Figure B.12: Nab performance analysis. *TEA identifies that the fsqrt.d instruction issues too late to hide its execution latency.*

which in turn increases store bandwidth requirements. The performance impact of the most performance-critical store instruction hence increases, mainly due to categories involving a full store queue (DR-SQ). *Lbm* writes 19 cache lines in each iteration, and prefetching hence moves its bottleneck from load latency to store bandwidth. While latency issues typically affect one static instruction, a bandwidth problem is typically distributed over multiple instructions, e.g., *lbm* has seven store instructions with a runtime over 10 billion clock cycles at distance 4.

Addressing this performance problem requires sweeping prefetch distances to identify the point where the load latency and store bandwidth effects balance out which exemplifies why TEA — by providing a comprehensive view on performance after running the application once — is desirable. The optimal prefetch distance for this architecture is 3 which yields a speedup of 1.28× over the original (see the line in Figure B.11).

Analyzing nab. Figure B.12a shows the PICS as reported by TEA for the code region that contains the performance-critical `fsqrt.d` instruction of *nab*. Again, the PICS reported by TEA are very similar to the golden reference whereas the PICS generated by IBS are not (Figure B.12b). (Flushing instructions such as `fsflags` and `frflags` always flush the pipeline in this architecture and can hence be identified statically.) In this example, none of the instructions are subjected to performance events, and the key advantage of TEA is hence that the developer can trust that (i) the time attributed to `fsqrt.d` is accurate, and (ii) that TEA did not miss any performance events that can majorly impact performance.

`Fsqrt.d` is hence performance-critical because its execution latency was not hidden. The reason is that the `fsflags` and `frflags` instructions that were executed just prior to it always flush the pipeline in this architecture. These instructions are inserted by the compiler to be compliant with the IEEE 754 standard because `flt.d` by default should not trigger an exception upon a comparison involving a NaN value. The RISC-V ISA however does not include a non-excepting version of the `flt.d` instruction, and the `fsflags` and `frflags` instructions are hence required to mask exceptions. While understanding this (involved) behavior is possible when looking at the PICS of these exact instructions, it would be extremely challenging to understand otherwise.

Optimizing nab. Addressing this problem is simple because *nab* does not require any special handling of comparisons involving NaN values. More specifically, enabling the compiler options `-finite-math` or `-fast-math` yields speed-ups of 1.96× and 2.45×, respectively. The reason for the significant speedups is that avoiding pipeline flushes enables the processor to fetch and execute further ahead into the instruction stream, thereby better hiding the execution latencies of independent floating-point instructions.

7 Related Work

The most related approaches to TEA are the instruction-driven performance analysis approaches AMD IBS [9], Arm SPE [7], and IBM RIS [14] which are inaccurate because they are not time-proportional (see Section 5).

A large body of work relies on event-driven performance analysis using Performance Monitoring Counters (PMCs) as provided by Intel [8] and DCPI [10]. Researchers have hence investigated PMU design [28], and PMUs have a variety of uses (e.g., runtime optimization [29], performance analysis in managed languages [30]–[32], profile-guided compilation [33], [34], and profile-guided meta-programming [35]). Xu et al. [36] focus on providing correct offsets in PMC sampling by exploiting counters that are the same when running on real hardware and during binary instrumentation (e.g., retired instructions). BayesPerf [37] encodes known relationships between performance counters in a machine learning model and then infers which performance counter values can be trusted. It is well-known that PMCs can be challenging to make sense of [38]–[40], and approaches have been proposed for reducing the consequences of the fact that only a limited number of events can be monitored concurrently (e.g., [41]). We demonstrated in Section 5 that optimization based on PMCs is challenging because PMC counts often correlate poorly with performance, and adopting TEA will hence also address these issues.

Eyerman et al. [16] propose a PMC architecture that enables constructing Cycles Per Instruction (CPI) stacks. The top-down model [15], which combines PMC output with a performance model to classify the application as mainly retiring instructions or being front-end-bound, back-end-bound, or suffering from bad speculation, can be viewed as a restricted form of a cycle stack because it presents a classification of an application's predominant performance bottleneck whereas a CPI stack breaks down an application's overall CPI across the units of the processors in which time was spent. Unlike TEA, these approaches cannot produce per-instruction cycle stacks — and our case studies demonstrate that instruction-level analysis is critical to understand performance issues.

While TEA explains why instructions are performance-critical, other performance aspects are also interesting. Vertical profiling [42], [43] combines hardware performance counters with software instrumentation to profile an application across deep software stacks, while call-context profiling [44] efficiently identifies the common orders functions are called in. Causal profiling [45]–[48] is able to identify the criticality of program segments in parallel codes by artificially slowing down segments and measuring their impact. Researchers have also devised approaches for profiling highly optimized code [49], assessing input sensitivity [50], [51], and profiling deployed applications [52].

Static instrumentation modifies the binary to gather (extensive) application execution data at the cost of performance overhead [53]–[57]. Dynamic instrumentation (e.g., Pin [58] and Valgrind [59]) does not modify the binary which leads to higher performance overheads than static instrumentation. Statistical performance analysis approaches (e.g., TEA, IBS, SPE, and RIS) do not modify the binary and hence have (much) lower overhead than instrumentation-based approaches. Simulation and modeling can also be used to understand key performance issues. The most related approach to ours is FirePerf [60] which uses FireSim [25] to non-intrusively gather extensive performance statistics. FirePerf would hence, unlike TEA, incur a significant performance overhead if used in a non-simulated environment.

8 Conclusion

We have presented Time-Proportional Event Analysis (TEA) which explains execution time by mapping commit stalls to the performance events that caused them — thereby enabling the creation of time-proportional Per-Instruction Cycle Stacks (PICS). To generate PICS, TEA tracks performance events across all in-flight instructions, but, by carefully selecting which events to track, it only increases per-core power consumption by ~0.1%. TEA relies on statistical sampling and hence has a performance overhead of merely 1.1%, yet only incurs an average error of 2.1% compared to a non-sampling golden reference. We demonstrate the utility of TEA by using it to identify performance problems in the SPEC CPU2017 benchmarks *lbm* and *nab* that, once addressed, yield speedups of 1.28× and 2.45×, respectively.

Acknowledgments

We thank the reviewers for their valuable feedback. Lieven Eeckhout is supported in part by the UGent-BOF-GOA grant No. 01G01421, the Research Foundation Flanders (FWO) grant No. G018722N, and the European Research Council (ERC) Advanced Grant agreement No. 741097. Magnus Jahre is supported by the Research Council of Norway (Grant No. 286596).

References

[1] W. J. Dally, Y. Turakhia, and S. Han, "Domain-Specific Hardware Accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020.

[2] M. D. Hill and V. J. Reddi, "Accelerator-Level Parallelism," *Communications of the ACM*, vol. 64, no. 12, pp. 36–38, 2021.

[3] M. Arora, S. Nath, S. Mazumdar, S. B. Baden, and D. M. Tullsen, "Redefining the Role of the CPU in the Era of CPU-GPU Integration," *IEEE Micro*, vol. 32, no. 6, pp. 4–16, 2012.

[4] Intel, *VTune Profiler User Guide*, 2021. Available: `https://www.intel.com/content/dam/develop/external/us/en/documents/vtune-profiler-user-guide.pdf`.

[5] AMD, *μProf*, `https://developer.amd.com/amd-uprof/`, 2021.

[6] B. Gottschall, L. Eeckhout, and M. Jahre, "TIP: Time-Proportional Instruction Profiling," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, Association for Computing Machinery, 2021, pp. 15–27.

[7] Arm, *ARM Architecture Reference Manual Supplement Statistical Profiling Extension, for ARMv8-A*, `https://static.docs.arm.com/ddi0586/a/DDI0586A_Statistical_Profiling_Extension.pdf`, 2017.

[8] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`, 2021.

[9] P. J. Drongowski, "Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors," AMD, Tech. Rep., 2007.

[10] J. M. Anderson, L. M. Berc, J. Dean, *et al.*, "Continuous Profiling: Where Have All the Cycles Gone?" *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 357–390, 1997.

[11] Linux, *perf*, `https://perf.wiki.kernel.org/index.php/Main_Page`, 2020.

[12] Google, *gperftools*, `https://github.com/gperftools/gperftools`, 2020.

[13] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, IEEE Computer Society, 1997, pp. 292–302.

[14] IBM, *POWER9 Performance Monitor Unit User's Guide*, `https://ibm.ent.box.com/s/8kh0orsr8sg32zb6zmq1d7zz6hud3f8j`, 2018.

[15] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer Society, 2014, pp. 35–44.

[16] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A Performance Counter Architecture for Computing Accurate CPI Components," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, Association for Computing Machinery, 2006, pp. 175–184.

[17] Intel, *Performance Monitoring Event Reference*, `https://perfmon-events.intel.com/`, 2022.

[18] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, *SonicBOOM: the 3rd Generation Berkeley Out-of-Order Machine*, Fourth Workshop on Computer Architecture Research with RISC-V, 2020.

[19] SPEC, *SPEC CPU 2017*, `https://www.spec.org/cpu2017/`, 2019.

[20] AMD, *Processor Programming Reference (PPR) for AMD Family 19h Model 21h, Revision B0 Processors*, `https://www.amd.com/en/support/tech-docs/preliminary-processor-programming-reference-ppr-for-amd-family-19h-model-21h`, 2021.

[21] Arm, *Arm Neoverse N2 Core Technical Reference Manual*, `https://developer.arm.com/documentation/102099/0000/Statistical-Profiling-Extension-support/Statistical-Profiling-Extension-events-packet`, 2022.

[22] Cadence, *Genus Synthesis Solution*, `https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html`, 2022.

[23] Cadence, *Joules RTL Power Solution*, `https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/power-analysis/joules-rtl-power-solution.html`, 2022.

[24] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory Power Estimation and Capping," in *Proceedings of the International Symposium on Low Power Electronics and Design*, ser. ISLPED, Association for Computing Machinery, 2010, pp. 189–194.

[25] S. Karandikar, H. Mao, D. Kim, *et al.*, "Firesim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *Proceedings of the International Symposium on Computer Architecture*, ser. ISCA, IEEE Press, 2018, pp. 29–42.

[26] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*, 2019. arXiv: `1912.05848 [cs.DC]`.

[27] B. Gottschall and M. Jahre, *TraceDoctor: Versatile High-Performance Tracing for FireSim*, The First FireSim and Chipyard User and Developer Workshop at ASPLOS, 2023.

[28] G. Kornaros and D. Pnevmatikatos, "A Survey and Taxonomy of on-Chip Monitoring of Multicore Systems-on-Chip," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 2, pp. 1–38, 2013.

[29] D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. De Bosschere, "Using HPM-Sampling to Drive Dynamic Compilation," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ser. OOPSLA, Association for Computing Machinery, 2007, pp. 553–568.

[30] P. F. Sweeney, M. Hauswirth, B. Cahoon, *et al.*, "Using Hardware Performance Monitors to Understand the Behavior of Java Applications," in *Proceedings of the Conference on Virtual Machine Research and Technology Symposium*, ser. VM, USENIX Association, 2004, p. 5.

[31] J. Whaley, "A Portable Sampling-Based Profiler for Java Virtual Machines," in *Proceedings of the Conference on Java Grande*, ser. JAVA, Association for Computing Machinery, 2000, pp. 78–87.

[32] Y. Zheng, L. Bulej, and W. Binder, "Accurate Profiling in the Presence of Dynamic Compilation," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA, Association for Computing Machinery, 2015, pp. 433–450.

[33] T. M. Conte, K. N. Menezes, and M. A. Hirsch, "Accurate and Practical Profile-Driven Compilation Using the Profile Buffer," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, IEEE Computer Society, 1996, pp. 36–45.

[34] T. M. Conte, B. A. Patel, and J. S. Cox, "Using Branch Handling Hardware to Support Profile-Driven Optimization," in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, Association for Computing Machinery, 1994, pp. 12–21.

[35] W. J. Bowman, S. Miller, V. St-Amour, and R. K. Dybvig, "Profile-Guided Meta-Programming," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2015, pp. 403–412.

[36] H. Xu, Q. Wang, S. Song, L. K. John, and X. Liu, “Can We Trust Profiling Results? Understanding and Fixing the Inaccuracy in Modern Profilers,” in *Proceedings of the International Conference on Supercomputing*, ser. ICS, Association for Computing Machinery, 2019, pp. 284–295.

[37] S. S. Banerjee, S. Jha, Z. Kalbarczyk, and R. K. Iyer, “BayesPerf: Minimizing Performance Monitoring Errors Using Bayesian Statistics,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, Association for Computing Machinery, 2021, pp. 832–844.

[38] V. M. Weaver and S. A. McKee, “Can Hardware Performance Counters Be Trusted?” In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, IEEE Computer Society, 2008, pp. 141–150.

[39] V. M. Weaver, D. Terpstra, and S. Moore, “Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer Society, 2013, pp. 215–224.

[40] D. Zaparanuks, M. Jovic, and M. Hauswirth, “Accuracy of Performance Counter Measurements,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE Computer Society, 2009, pp. 23–32.

[41] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan, “Time Interpolation: So Many Metrics, So Few Registers,” in *Proceedings of the International Symposium on Microarchitecture*, ser. MICRO, IEEE Computer Society, 2007, pp. 286–300.

[42] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind, “Vertical Profiling: Understanding the Behavior of Object-Priented Applications,” in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA, Association for Computing Machinery, 2004, pp. 251–269.

[43] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer, “Automating Vertical Profiling,” in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA, Association for Computing Machinery, 2005, pp. 281–296.

[44] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi, “Accurate, Efficient, and Adaptive Calling Context Profiling,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2006, pp. 263–271.

[45] C. Curtsinger and E. D. Berger, “Coz: Finding Code That Counts with Causal Profiling,” in *Proceedings of the Symposium on Operating Systems Principles*, ser. SOSP, Association for Computing Machinery, 2015, pp. 184–197.

[46] A. Yoga and S. Nagarakatte, “Parallelism-Centric What-If and Differential Analyses,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2019, pp. 485–501.

[47] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, “Evaluating the Accuracy of Java Profilers,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2010, pp. 187–197.

[48] B. Pourghassemi, A. Amiri Sani, and A. Chandramowlishwaran, “What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling,” in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Association for Computing Machinery, 2019, pp. 87–88.

[49] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan, “Binary Analysis for Measurement and Attribution of Program Performance,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2009, pp. 441–452.

[50] E. Coppa, C. Demetrescu, and I. Finocchi, “Input-Sensitive Profiling,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2012, pp. 89–98.

[51] D. Zaparanuks and M. Hauswirth, “Algorithmic Profiling,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2012, pp. 67–76.

[52] C. H. Kim, J. Rhee, H. Zhang, *et al.*, "IntroPerf: Transparent Context-Sensitive Multi-Layer Performance Inference Using System Stack Traces," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS, Association for Computing Machinery, 2014, pp. 235–247.

[53] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying Page Load Performance with WProf," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI, USENIX Association, 2013, pp. 473–486.

[54] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance Debugging in the Large Via Mining Millions of Stack Traces," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE, IEEE Press, 2012, pp. 145–155.

[55] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: a Call Graph Execution Profiler," in *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN, Association for Computing Machinery, 1982, pp. 120–126.

[56] C. Lattner and V. 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*, ser. CGO, IEEE Computer Society, 2004, p. 75.

[57] T. B. Schardl, T. Denniston, D. Doucet, B. C. Kuszmaul, I.-T. A. Lee, and C. E. Leiserson, "The CSI Framework for Compiler-Inserted Program Instrumentation," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 1, no. 2, 2017.

[58] C.-K. Luk, R. Cohn, R. Muth, *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2005, pp. 190–200.

[59] N. Nethercote and J. Seward, "Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, Association for Computing Machinery, 2007, pp. 89–100.

[60] S. Karandikar, A. Ou, A. Amid, *et al.*, “FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, Association for Computing Machinery, 2020, pp. 715–731.

Paper C

Balancing Accuracy and Evaluation Overhead in Simulation Point Selection

Authors:
Björn Gottschall, Silvio Campelo de Santana, Magnus Jahre

Published at conference:
2023 IEEE International Symposium on Workload Characterization (IISWC)

Nominations/Awards:
Best Paper Nomination

Paper C

Gottschall *et al.* (2023)

Balancing Accuracy and Evaluation Overhead in Simulation Point Selection

Björn Gottschall, Silvio Campelo de Santana, Magnus Jahre

Norwegian University of Science and Technology, Norway

Abstract

Simulators are the tool of choice when designing new computer architectures. While software-based simulators are (relatively) easy to adapt to evaluate new architectural mechanisms and can provide extensive non-invasive performance measurements, they are slow, and executing benchmarks to completion with realistic input sets is practically impossible. FPGA-accelerated simulators are orders of magnitude faster but require hardware implementations of all simulated components and provide limited performance measurements.

In this paper, we first propose TraceDoctor to enable extensive non-invasive performance measurement within the FPGA-accelerated FireSim simulator. TraceDoctor enables tracing any architectural signal and keeps overheads in check by enabling architects to implement analysis-specific workers which run on the host computer to filter or compress trace data. We then use TraceDoctor to better understand which part(s) of a benchmark to focus on in software-based simulators, i.e., explaining how architects should configure the SimPoint methodology to yield simulation points that accurately predict whole-benchmark performance while minimizing the number of simulated instructions. Our analysis yields a number of interesting insights. For example, we find that evaluating relatively many, relatively small simulation points accurately predicts whole-benchmark performance while requiring to simulate relatively few instructions. In contrast, the current typical practice of selecting a single large simulation point yields low accuracy yet requires simulating many instructions.

1 Introduction

Developing high-performance hardware is a challenging and complex task, but understanding and alleviating full-stack performance issues is arguably even more challenging. Simulators play a key role in the development process to verify and debug new circuitry because they combine relative ease of development with the ability to provide detailed insight into architectural behavior. Software-based computer architecture simulators such as gem5 [1] and Sniper [2], carefully model the behavior of hardware components, can be (easily) adapted to model novel architectural approaches, and enables architects to non-intrusively collect highly detailed performance information. Unfortunately, they incur long simulation times which in general makes it intractable to simulate applications to completion at high detail with realistic input sets. Register Transfer Level (RTL) simulators, such as Synopsys VCS [3] or Verilator [4], can simulate hardware components at the level of signals, gates, and registers and thereby provide waveforms that exactly represent the switching behavior of the circuits. Register Transfer Level (RTL) simulators, however, incur (much) higher simulation time overhead than computer architecture simulators.

Hardware-accelerated simulators, such as FireSim [5], use Field Programmable Gate Arrays (FPGAs) to accelerate simulation. This reduces simulation time by roughly two orders of magnitude compared to software-based computer architecture simulators and enables simulating SPEC CPU17 [6] benchmarks to completion in full-stack configurations with reference input sets in (many) tens of hours. The root cause of FireSim's high performance is that it requires RTL-level models of key system components such as the processor cores (e.g., Rocket [7] and BOOM [8]). This enables FireSim to instantiate hardware configurations of these components in FPGAs. Simulation is hence massively parallel, i.e., all actions in a given clock cycle occur in parallel as opposed to being modeled with (mostly) sequential instructions in conventional computer architecture simulators. FireSim's high performance hence comes at the cost of making it more challenging to implement novel architecture mechanisms – because fully functional RTL implementations are required — and limited opportunity to collect performance information — because hardware must be added to retrieve it.

Accessing architecture-internal state is however critical to gain insight into architectural behavior, and FireSim hence provides TracerV and AutoCounter to enable out-of-band inspection of architecture-internal state [9]. TracerV focuses on instruction retirement and traces out the addresses of all instructions that commit in each cycle. AutoCounter can be used to add performance counters to signals within the design which is then periodically exported during simulation. While both interfaces are very helpful, they are not without limitations. TracerV traces out all retired instructions — and hence incurs significant storage overhead for all but the shortest timeframes — whereas AutoCounter only supports infrequently retrieving event counts or signal states — and therefore only to a limited extent reports how signals change over time.

We hence propose TraceDoctor which (i) supports tracing any architecture-internal signal every simulator clock cycle, and (ii) keeps storage overhead in check with analysis-specific software workers that run on the host computer and compress or filter trace data. More specifically, TraceDoctor provides a trace vector that connects the signals of interest within the simulated hardware design to TraceDoctor's hardware component; the width of the trace vector is set to accommodate the target signals. The (valid) trace vectors are transferred to the host over the FPGA's Direct Memory Access (DMA) interface and dispatched to software workers within TraceDoctor's simulation driver. TraceDoctor supports executing multiple workers in parallel on the host to (i) minimize the impact of tracing on simulator performance, and (ii) enable performing different analyses on the traced data in parallel. The workers are independent of the hardware design, thereby avoiding time-consuming FPGA synthesis as long as the trace vector is not changed. We demonstrate that TraceDoctor reduces Effective Simulator Frequency (ESF) only by 3.9%, 5.7%, and 3.5% on average when configured with workers that implement the binary, text, and flamegraph output modes of TracerV. TracerV on the other hand reduces ESF by 84.3%, 76.8%, and 38.9% for the binary, text, and flamegraph outputs, respectively, because it stalls simulation while processing sample data on the host.

We then use TraceDoctor to gain insight into a key challenge in software-based simulation, i.e., how to select a representative subset of whole-benchmark execution. SimPoint [10]–[13] is the de facto standard approach, but the runtime overhead of software-based simulators means that SimPoint has not been validated with detailed processor implementations, such as the BOOM core [8], and modern benchmarks, such as SPEC CPU2017 [6], e.g., the most recent SimPoint validation effort [11] used SimpleScalar [14] and SPEC CPU2000. Moreover, the relationship between Cycles Per Instruction (CPI) prediction error and evaluation overhead, i.e., the number of instructions that must be simulated to predict CPI, has not been studied in detail. Finally, current practice tends towards selecting one long simulation point per benchmark that consists of between 100 million and 2 billion instructions (see Section 6.1). In contrast, existing validation studies focus on many short simulation points, e.g., up to 30 simulation points that each consist of 10 million instructions for each benchmark [11].

We hence develop a TraceDoctor worker which retrieves Basic Block Vectors (BBVs) by simulating 21 SPEC CPU2017 benchmarks to completion in a full-system FireSim setup built around a high-performance out-of-order BOOM core [8] configuration. We then sweep a broad range of SimPoint parameters, the most important of which are the maximum number of simulation points and the number of instructions in a simulation point, to generate in total 710,850 simulation points. Our analysis[1] of this extensive data set yields the following key insights:

- We confirm that SimPoint is accurate when configured to collect a sufficient number of simulation points for each benchmark. More specifically, SimPoint achieves the lowest average CPI prediction errors, i.e., 0.9%, 0.7%, 0.5%, 0.4%, and 0.3% for interval sizes of 10, 30, 100, 500, and 1,000 million instructions, respectively, when allowed to select up to 44, 46, 48, 43, and 40 simulation points.
- SimPoint incurs errors when a cluster's centroid — which it selects by solely considering basic block execution frequencies — does not yield average CPI, for instance due to accesses to irregular data structures such as maps and graphs. Such situations are rare in our benchmarks, and the accuracy of SimPoint is hence generally high.
- The best balance between simulation overhead and accuracy is obtained by selecting relatively many simulation points that each consist of relatively few instructions. Architects must however weigh this advantage against operational overheads, e.g., the overheads of generating and storing checkpoints as well as ensuring that history-based structures such as caches and branch predictors are warm.
- If constraints force architects to run few simulation points, these should be large (i.e., 100 million instructions or larger). CPU prediction error however improves significantly when moving from a single simulation point per benchmark to two simulation points per benchmark.

[1]TraceDoctor is available at `https://github.com/EECS-NTNU/firesim` and integration into mainline FireSim is in progress. We have also made the dataset of our SimPoint exploration publicly available [15].

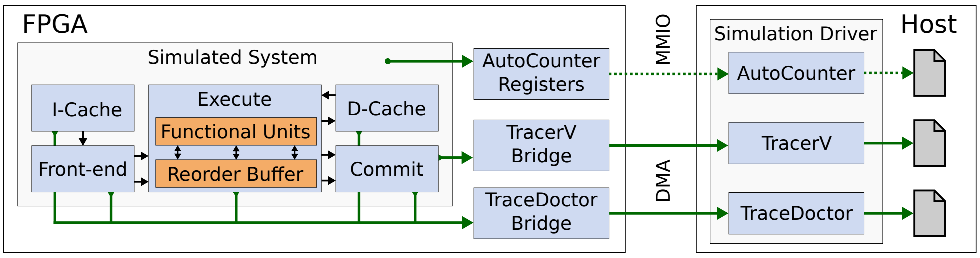


Figure C.1: Overview of how AutoCounter, TracerV, and TraceDoctor integrate with a FireSim simulation.

- Basic block vectors must be captured across a complete benchmark execution with the target data set to be representative of whole-benchmark execution. If architects are forced to profile a subset of benchmark execution, the selected simulation points will only be representative of this subset.

2 Background

We will now briefly introduce TracerV and AutoCounter because this is necessary to understand how TraceDoctor improves upon the state-of-the-art (see Figure C.1).

TracerV traces out the addresses of retiring dynamic instructions per CPU core. To achieve this, TracerV adds signals to the commit stage of the CPU core (e.g., Rocket or BOOM) which exposes the addresses of the committing instructions. The TracerV bridge receives these signals and then inserts the instruction addresses and a timestamp into a DMA queue. Once the queue is full, the TracerV host driver pulls the contents of the DMA queue into a buffer on the host and processes the data. TracerV has a triggering subsystem that enables limiting tracing to a region of interest by specifying start (and end) cycles, addresses, or instructions. The triggers are specified when instantiating the TracerV host driver and can hence be changed without having to synthesize the FPGA design.

The TracerV host driver supports three output modes. The first mode is the binary output which writes raw tracing data to a file in a binary format. The second mode writes trace data as plain text in a Comma-Separated Values (CSV) format, i.e., it retrieves the instruction addresses and the timestamp,

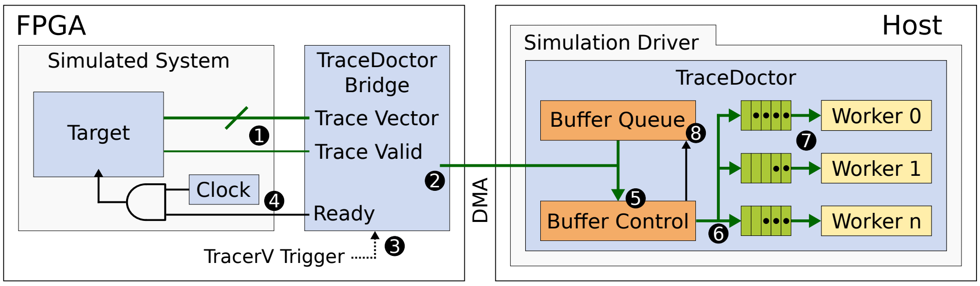


Figure C.2: TraceDoctor overview.

separates them by commas, and writes the resulting string to a file. The third and last mode is FirePerf [9] in which TracerV first creates a symbol table using the debug information in the target binary and then correlates traced instruction addresses with this table. In this way, FirePerf emits only the start and end cycles of each function call into a file and thereby enables generating flamegraphs once simulation completes.

AutoCounter creates identity registers or performance counter registers by annotating signals within the target design. The identity register exposes the value of the selected signal, whereas the performance counter register adds one to the register every time the target signal toggles. These registers are then periodically sampled by the host driver. AutoCounter is hence similar to performance monitoring counters in contemporary processors and enables retrieving statistical information from a simulation but cannot track how signals change at cycle-accurate time scales.

3 TraceDoctor

TraceDoctor overview. Figure C.2 continues where Figure C.1 left off and explains the details of how TraceDoctor integrates with the FireSim ecosystem. The tracing bundle of TraceDoctor consists of a user-configurable trace vector ❶ and a valid signal and can be attached to any signal within the simulated system. We include the valid signal such that the target hardware can choose to not emit a trace vector in cycles that it knows are not interesting (e.g., the processor core is still stalled on the same `load` instruction as it was in the previous cycle). The tracing bundle is connected to the TraceDoctor Bridge which inserts valid trace vectors into the Direct Memory Access (DMA) queue ❷ on the hardware side; invalid trace vectors are discarded. We attach the

TraceDoctor Bridge to the trigger module of TracerV to support starting and stopping tracing in a specific cycle or when a specific instruction or address is observed ❸. If the DMA queue fills up, the TraceDoctor Bridge provides back-pressure by lowering its ready signal ❹ which in turn stalls simulation.

On the host side, the TraceDoctor simulation driver provides a queue of buffers in which each buffer by default stores a single DMA transfer ❺. If an insufficient number of buffers are available in the buffer queue, the DMA queue on the hardware side cannot be drained which in turn (eventually) stalls simulation. The buffer control component ensures that buffers are processed in order by broadcasting each vector to work queues in arrival order ❻; it also initializes the buffer's reference counter to n (where n is the number of workers). By default, each worker executes in a separate thread and consumes buffers from its work queue ❼. The worker decrements the buffer's reference counter when it has been processed, and the buffer can be reused when the counter reaches zero ❽.

Configuring TraceDoctor. Architects should take care to configure TraceDoctor to minimize simulation slowdown. One critical point is to ensure that the tracing vector fits within the bit-width of the DMA engine (which is typically 512 bits). Wider tracing vectors will require multiple DMA tokens for each transfer which will typically result in DMA transfers bottlenecking simulation. Similarly, architects must take care to (i) configure TraceDoctor with a sufficient number of workers, and (ii) ensure that work is well-distributed across workers. Since a buffer cannot be reused until all workers have processed it, host processing will be bottlenecked by the slowest worker, i.e., the slowest worker will execute back-to-back while the latency of all other workers is hidden. An effective strategy for improving the performance of time-consuming analyses is to distribute it across n workers and configure each worker to only process one in n buffers, i.e., for the $n-1$ buffers which the worker will not process, it simply decrements the reference counter before moving on to the next buffer. This strategy is however only applicable when (i) buffers can be analyzed independently, and (ii) the data emitted by the different workers contain sufficient information to be merged when the simulation completes.

Table C.1: Simulator configuration.

Part	Configuration
Core	OoO BOOM [8]: RV64IMAFDCSUX @ 3.2 GHz
Front-end	8-wide fetch, 32-entry fetch buffer, 4-wide decode, 28 KB TAGE branch predictor, 40-entry fetch target queue, max. 20 outstanding branches
Execute	128-entry ROB, 128 integer/floating-point physical registers, 24-entry dual-issue memory queue, 40-entry 4-issue integer queue, 32-entry dual-issue floating-point queue
LSU	32-entry load/store queue
L1	32 KB 8-way I-cache, 32 KB 8-way D-cache w/ 8 MSHRs, next-line prefetcher from L2
L2/LLC	512 KB 8-way L2 w/ 12 MSHRs, 4 MB 8-way LLC w/ 8 MSHRs
TLB	Page Table Walker, 32-entry fully-assoc L1 D-TLB, 32-entry fully-assoc L1 I-TLB, 512-entry direct-mapped L2 TLB
Memory	16 GB DDR3 FR-FCFS quad-rank, 16 GB/s maximum bandwidth, 14-14-14 (CAS-RCD-RP) latencies @ 1 GHz, 8 queue depth, 32 max reads/writes
OS	Buildroot, Linux 5.7.0

4 Experimental Setup

Simulator configuration. Our simulations are performed on either a dual-socket AMD EPYC 7413 host system with 512 GB of main memory and two Xilinx Alveo FPGA accelerators cards (one U250 [16] and one U280 [17]) provided by eX3 [18] or the U250-instances of NTNU's EPIC cluster [19]. We evaluate a single-core system with a 4-wide BOOM core [8] as described in Table C.1. We focus on an FPGA configuration that includes the simulated architecture and the hardware components of TracerV and TraceDoctor, which runs at a clock frequency of 70 MHz. When including AutoCounter in the configuration, the FPGA clock frequency dropped to 50 MHz.

Benchmarks. We run the 21 benchmarks in SPEC CPU2017 [6] that are compatible with our setup. We used GCC 10.1 with `-O3` compiler optimizations enabled and static linkage. Our full system simulator runs all benchmarks to completion using reference input sets. We exclude system boot and power-off time by starting (stopping) tracing before (after) benchmark execution.

Metrics. While our default FPGA configuration runs at a clock frequency of 70 MHz, the simulated architecture will only reach this frequency if it does not interact with the host or non-RTL simulator components (e.g., network access, disk access, or the FASED [20] memory model). We use *Effective Simulator*

Frequency (ESF), i.e., the number of simulated clock cycles per second, as our simulator performance metric. ESF is maximally 70 MHz in our setup, but any interaction with the host or memory system that requires stalling the simulator to maintain cycle accuracy reduces ESF. The ideal tracing interface would avoid reducing ESF beyond what is necessary to preserve simulator accuracy, i.e., the ESF achieved by FireSim without tracing.

When evaluating SimPoint configurations, our evaluation overhead metric is the number of simulated instructions. We report evaluation overhead instead of evaluation time as evaluation time differs between simulators and can differ within the same simulator depending on the level of detail. The number of simulated instructions hence captures the amount of work that the simulator must perform, and the practitioner can then multiply it by the average number of simulated instructions per second for their target simulator configuration to estimate evaluation time.

CPI prediction error e is the absolute relative percentage error of CPI as predicted by SimPoint ($\text{CPI}_{\text{SimPoint}}$) relative to the golden reference CPI which we obtain by simulating each benchmark to completion with reference inputs:

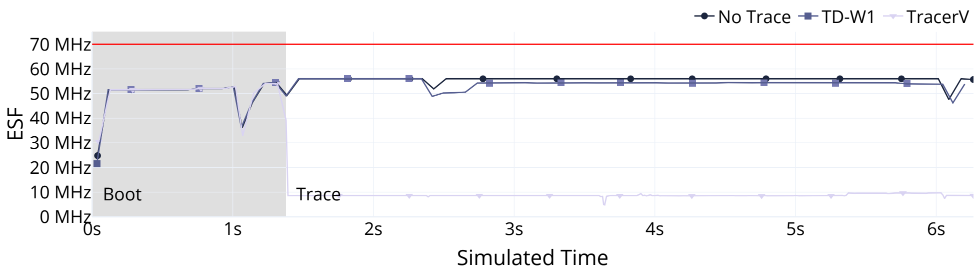
$$e = \frac{|\text{CPI}_{\text{SimPoint}} - \text{CPI}|}{\text{CPI}} \times 100. \tag{C.1}$$

We aggregate errors across benchmarks by computing the average (arithmetic mean) or maximum across per-benchmark errors.

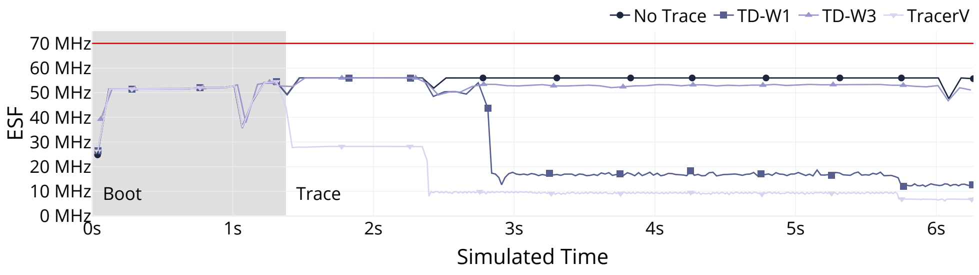
5 TraceDoctor Evaluation

We now compare FireSim's performance with TraceDoctor to its performance with TracerV, AutoCounter, and a configuration in which tracing is disabled (which we refer to as "No Trace"). We label our TraceDoctor configurations TD-Wi where i is the number of workers. TD-W1 is hence a single-worker TraceDoctor configuration whereas the TD-W3 configuration has three workers. To fairly compare TraceDoctor to prior work, we exploit the flexibility of TraceDoctor to configure it to provide the exact same output as TracerV and AutoCounter. Since TracerV and AutoCounter provide different output data, we first compare to TracerV and then to AutoCounter.

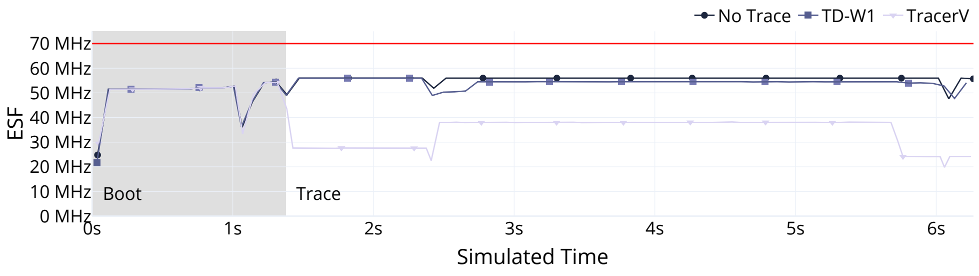
For the experiments in this section, we run *nab* from SPEC CPU2017 [6] with the test input set. This is necessary to keep simulation time sufficiently short to execute to completion with all tracing interfaces. Even this relatively short



(a) Binary



(b) CSV



(c) FirePerf

Figure C.3: Simulator performance with TraceDoctor (TD) and TracerV when generating (a) binary, (b) CSV, and (c) FirePerf traces. *TraceDoctor (i) yields higher performance than TracerV across all modes, and (ii) only minorly slows down simulation time compared to No Trace when configured with a sufficient number of workers.*

timeframe yields traces that contain 7.1 billion dynamic instructions and hence pushes against the storage overhead limits of TracerV. As mentioned before, TracerV writes uncompressed binary or CSV files that quickly become large, i.e., its binary (CSV) trace files for *nab* with the test input set is 1,001 GB (319 GB). While TraceDoctor supports compression of trace outputs by default, adding support for compression is trivial, and we hence disable this feature to ensure a fair comparison. We further ensure that storage latency does not impact simulator performance by writing all tracing output to the null device (i.e., `/dev/null`).

TraceDoctor versus TracerV. Figure C.3 reports ESF for FireSim with TraceDoctor compared to TracerV and no tracing across TracerV's three supported output modes, i.e., binary, CSV, or FirePerf [9] (see Figures C.3a, C.3b and C.3c, respectively). TraceDoctor yields average ESF slowdowns of 3.9%, 5.7%, and 3.5%, whereas TracerV reduces ESF by 84.3%, 76.8%, and 38.9% for the binary, CSV, and FirePerf outputs, respectively. The performance difference is primarily due to TraceDoctor processing trace data on the host in parallel with simulation whereas TracerV stalls simulation during host processing. This however requires that TraceDoctor is configured with a sufficient number of workers to process tracing data at least equally fast as it is being supplied by the FPGA. More specifically, a single worker is sufficient for the binary and FirePerf outputs (i.e., see TD-W1 in Figures C.3a and C.3c) whereas three workers are required in the CSV case (i.e., compare TD-W3 to TD-W1 in Figure C.3b).

TracerV yields lowest performance with binary output, i.e., ESF is consistently below 10 MHz in Figure C.3a. Since TracerV stalls FireSim during the processing of trace data, it directly affects simulator performance. With the binary format, TracerV also (i) generates a massive amount of tracing data (i.e., 1 TB in total), and (ii) writes the data to a file in 8-byte chunks that each generates a syscall — which in combination are the root causes of TracerV's simulator performance degradation. TracerV performs much better in the FirePerf configuration (see Figure C.3c), primarily because it generates significantly less tracing data (i.e., 122 MB).

Both TraceDoctor with a single worker (TD-W1) and TracerV slow down FireSim significantly in the CSV configuration (see Figure C.3b). When tracing is triggered after 1.3 seconds of simulated time, TracerV immediately drops ESF to 28 MHz before further reducing ESF to 10 MHz after 2.4 s. TD-W1 on the other hand initially does not slow down simulation (i.e., an ESF of 55.9 MHz), but ESF then drops to 17.3 MHz after 2.8 s. There are two observations to be made here. First, the phase in which TracerV yields higher ESF is a benchmark phase where fewer instructions are executed. It is hence easier for the host to keep up with the FPGA because fewer instructions need to be written to the CSV file. Second, TraceDoctor's buffer architecture enables it to keep ESF high until the buffer fills up. In this experiment, we configured TraceDoctor to buffer up to 12.9 GB of tracing data before stalling simulation. Since the TD-1W configuration only has a single worker, it is not able to consume trace data at the same rate as the FPGA produces it and hence ultimately stalls simulation. If the benchmark however would have entered a phase in which it commits fewer instructions per cycle, the trace data bandwidth could drop sufficiently for the worker to catch up (and this is why we configure such a large buffer).

The root cause of the low ESF with TD-W1 and TracerV for the CSV output is that it relies on `printf` to format the output which incurs a performance overhead. Figure C.3b shows that the TD-W3 configuration overcomes this limitation by instantiating three workers which each process every third buffer, i.e., worker 0 processes buffers (0, 3, 6, . . .), worker 1 processes buffers (1, 4, 7, . . .) and worker 2 processes buffers (2, 5, 8, . . .). Since each line in the CSV file contains a timestamp, it is trivial to merge the output files of each worker after simulation is complete.

TraceDoctor versus AutoCounter. We now compare TraceDoctor to AutoCounter by implementing a worker that counts the selected signals cycle-accurately and then periodically writes counter values to a file. We compared the ESF of TraceDoctor and AutoCounter across 34 sampling frequencies between 1 kHz and 128 kHz, but we were unable to detect a drop in ESF, i.e., neither TraceDoctor nor AutoCounter has a measurable impact on simulator performance. While TraceDoctor is not intended to replace AutoCounter, this result demonstrates that TraceDoctor can be employed to capture simulator statistics in parallel to other tracing activities. (While enabling AutoCounter forced us to reduce FPGA clock frequency to 50 MHz, we expect that we could regain our default 70 MHz frequency with some optimization.)

6 Validating SimPoint

We now demonstrate the utility of TraceDoctor by using it to validate SimPoint [10]–[13]. More specifically, we use TraceDoctor workers to create instruction-level execution profiles from which we in turn create Basic Block

Table C.2: SimPoint validation space.

SimPoint Parameter	Default	Validation Range
Coverage	Full	Full and first 50b to 1000b instructions (increments of 50b)
Interval size	N/A	10m, 30m, 100m, 500m, and 1b instructions
Dimensionality	15	1 to 50 (increments of 1)
MaxK	30	1 to 50 (increments of 1)
BIC threshold	0.9	0.0 to 1.0 (increments of 0.05)

Vector (BBV) profiles during post-processing. We will first quickly recapitulate the main operation of SimPoint before presenting the results of our experiments.

6.1 A Primer on SimPoint

Table C.2 lists the key parameters of the SimPoint methodology and the ranges we consider in our validation experiments. The first parameter is *benchmark coverage*, i.e., how many instructions are considered when capturing Basic Block Vectors (BBVs). While it is clearly better to capture BBVs across complete benchmark executions, this is not always possible in practice because capturing BBVs can take a long time in some setups.

Basic Block Vector (BBV) collection. SimPoint divides the benchmark into intervals of i dynamic instructions; i is a user-selected parameter, which we refer to as *interval size*. The first i dynamic instructions are hence part of the first instruction interval, the second group of i dynamic instructions forms the second interval, and so on. SimPoint requires a BBV for each instruction interval which is simply the number of times each basic block was executed during the instruction interval the BBV represents. BBVs are typically long since they have one entry for each basic block in the benchmark, e.g., gcc executes 659,156 unique basic blocks in our evaluation. It is hence impractical to use BBVs for clustering directly as each basic block then becomes a dimension in the solution space, and SimPoint thus projects BBVs into a space with fewer dimensions using a random linear projection. The number of dimensions to project onto is 15 by default, and we refer to this parameter as *dimensionality*.

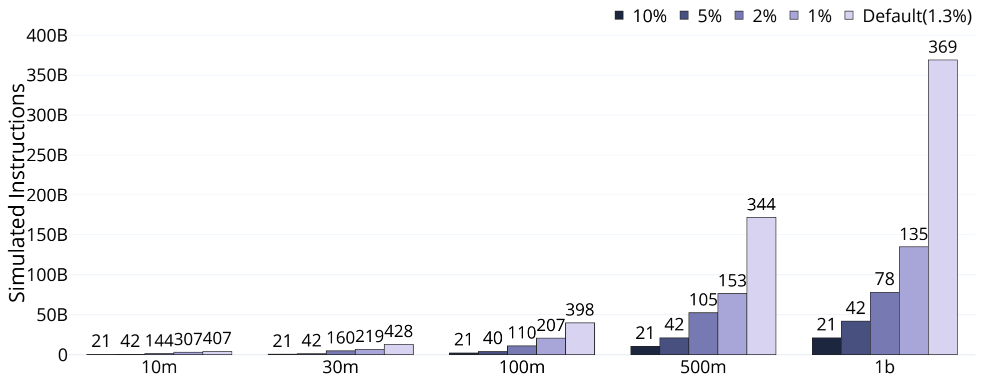


Figure C.4: Aggregate number of simulated instructions for our 21 SPEC2017 benchmarks for various sizes and CPI prediction error constraints compared to the SimPoint default parameters. *SimPoint is generally accurate, but simulation overhead can be substantially reduced by selecting different parameters or tolerating higher error.*

Simulation point selection. The key SimPoint parameters that control simulation point selection are *MaxK* and the *Bayesian Information Criterion (BIC) threshold*. MaxK represents the maximum number of simulation points that can be created, whereas BIC is a statistical criterion that balances the improvement from adding more clusters against how well they explain the data. We use the default search policy of SimPoint, which is to perform a binary search through the range of values of k between 1 and MaxK to find the clustering that maximizes the BIC score. Binary search reduces SimPoint execution time substantially compared to linear search while yielding similar accuracy because BIC scores generally increase with k [11].

For each k, SimPoint runs k-means five times with different random seeds yielding different initial centroids. This is necessary because the initial centroid selection can have a big impact on the final clustering. (The centroid is the element closest to the center of a cluster.) In each iteration, k-means computes the Euclidean distance between each data point and all centroids and moves the data point to the cluster with the closest centroid. It then recomputes the centroids of all clusters because they could have changed due to data point movement. K-means either completes after a fixed number of iterations or when it converges, i.e., the centroids remain the same after iterating over all data points, and we configure SimPoint to run to convergence. SimPoint finally computes the BIC score of each candidate clustering and retains the one with the highest score.

SimPoint has now identified the clustering that yields the best BIC score for each inspected k. It then applies the *BIC threshold*, which expresses the deviation from the maximal BIC score that SimPoint can exploit to reduce the number of simulation points. More specifically, SimPoint normalizes the BIC score of each evaluated clustering to the best observed BIC score and returns the clustering with the fewest number of clusters (i.e., the lowest k) which has a normalized BIC score above the threshold. Consider clusterings k1, k2, k3, k4, and k5 with BIC scores of 100, 200, 300, 400, and 500, respectively, as well as k-values of 1, 2, 3, 4, and 5. The maximum observed BIC score is 500, and the normalized BIC scores are hence 0.2, 0.4, 0.6, 0.8, and 1.0. If the BIC threshold is 0.5, SimPoint will consider k3, k4, and k5 because their normalized BIC score is greater than 0.5 and select k3 because it has the lowest k value. If the threshold is 0.9, SimPoint will select k5 because it is the only clustering with a normalized BIC score above the threshold. Providing a lower BIC threshold hence reduces the number of instructions to simulate because it favors fewer clusters but (potentially) increases prediction error.

Predicting CPI. Our SimPoint configuration represents each cluster by its centroid instruction interval and computes the weight of the cluster as the ratio of the number of intervals in the cluster divided by the number of intervals in the benchmark. From this information, it is straightforward to predict whole-benchmark CPI, i.e., the architect evaluates each simulation point to retrieve its individual CPI and then computes the weighted average of the simulation point CPIs to predicted whole-benchmark CPI.

SimPoint in current practice. The most detailed validation studies of SimPoint focus on relatively large MaxK and short intervals [10], [11]. For example, Hamerly et al. [11] focused on a configuration with MaxK equal to 30 and 10 million instruction intervals. To better understand how SimPoint is used in practice, we surveyed the proceedings of ISCA from 2018 to 2022 and found 25 papers that used SimPoint in their evaluation. Out of these papers, 52% used a MaxK of 1 and 30% used a MaxK larger than one; 16% did not clearly specify their MaxK value. Interestingly, only a single paper used a MaxK of 30, and for the remaining papers, MaxK was typically much less than 10 or not specified. With respect to interval sizes, 16% used interval sizes between 5 million and 30 million instructions, 48% used interval sizes between 100 and 500 million instructions, and 28% used interval sizes greater than or equal to 1 billion instructions; 8% did not clearly report their interval size.

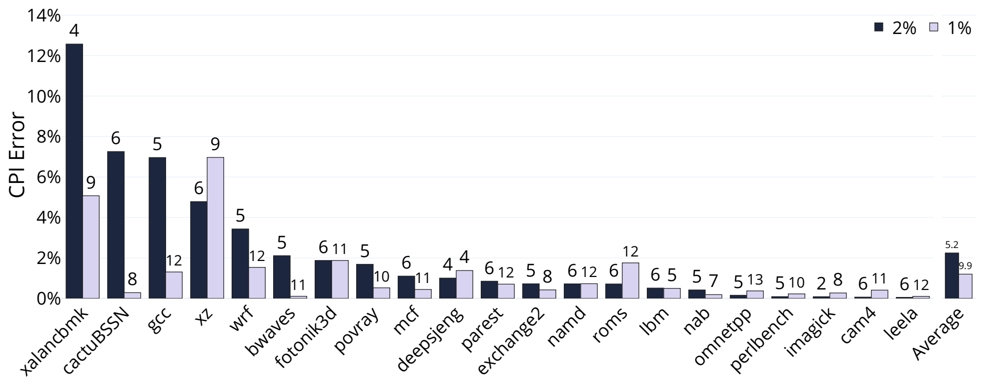


Figure C.5: Per-benchmark CPI prediction error with the 100m interval size and a 2% and 1% average error constraint. *SimPoint predicts CPI accurately for most benchmarks, but outliers occur when BBVs do not predict CPI well.*

6.2 SimPoint Error

The above analysis demonstrates that there is a mismatch between the SimPoint configurations that are used in practice and the configurations that have been validated, and we will now first explore the impact of SimPoint parameters on CPI prediction error before delving into the trade-off between prediction error and evaluation overhead.

Parameters. We focus on the interval size, MaxK, dimensionality, BIC threshold, and benchmark coverage parameters. While Table C.2 shows the extent of the validation space we consider, our validation is not completely orthogonal as this yields intractable evaluation overhead. We perform three parameter sweeps in which we consider interval sizes of 10 million, 30 million, 100 million, 500 million, and 1 billion instructions and vary a subset of parameters. More specifically, we vary MaxK and dimensionality in the first sweep, MaxK and BIC threshold in the second sweep, and MaxK and benchmark coverage in the third sweep.

Evaluation overhead under an error constraint. Figure C.4 plots evaluation overhead versus error for the SimPoint configurations that minimize evaluation overhead under an error bound. More specifically, we select the SimPoint configuration that minimizes simulation overhead within 0.25% of a target average error e (i.e., $e \pm 0.25\%$) across our 21 SPEC CPU2017 benchmarks. We consider target errors e of 10%, 5%, 2%, and 1% as well as interval sizes of 10m, 30m, 100m, 500m, and 1b instructions. We also report

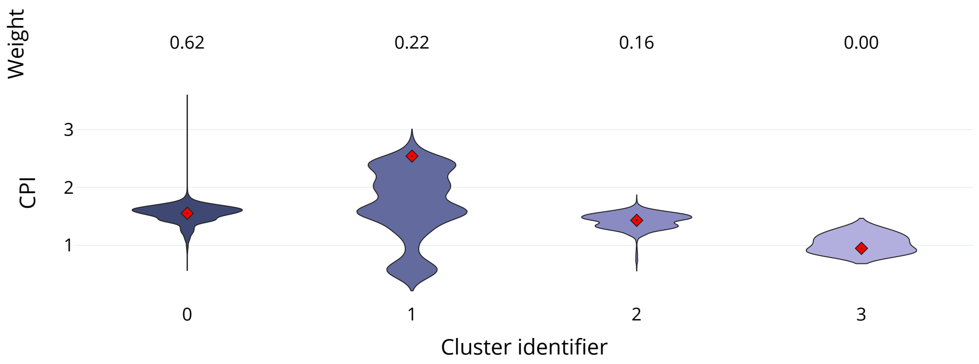


Figure C.6: CPI distribution across clusters for xalancbmk in the 2% error configuration. *CPI prediction error is due to the centroid not being representative of the cluster's average CPI.*

the simulation overhead of the SimPoint default configuration. The numbers above the bars are the number of simulation points required to evaluate all 21 benchmarks.

Our analysis, which unlike prior validation efforts [10], [11] runs modern benchmarks to completion with the reference input sets on a cycle-exact processor core model, confirms that SimPoint is accurate. More specifically, it yields an average error of only 1.3% in its default configuration, i.e., 1.9%, 1.3%, 1.4%, 1.0%, and 1.2% for the 10m, 30m, 100m, 500m, and 1b interval sizes, respectively. The default SimPoint configuration however prioritizes accuracy at the expense of evaluation overhead, and our analysis shows that SimPoint can yield similar accuracy at much lower overhead. For example, the default SimPoint configuration requires evaluating 369 simulation points to achieve the 1.2% average error with the 1b interval size. We find that the same interval size, full instruction coverage, MaxK equal to 8, dimensionality 35, and a BIC threshold of 0.9 yields an average error of 1.2% while evaluating only 135 simulation points — thereby reducing evaluation overhead by 2.7×.

Explaining residual error. Figure C.5 drills down into the 100m interval size with error limits of 2% and 1%. The 2% and 1% SimPoint configurations yield average errors of 2.2% and 1.2% with MaxK of 6 and 14 and dimensionalities of 15 and 8, respectively; both configurations have full coverage and BIC thresholds of 0.9. Figure C.5 shows that errors are low for most benchmarks, but that there are some outliers with higher errors. For example, the average

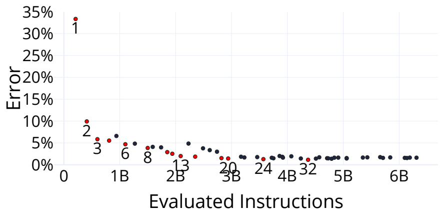
```
 1 bool ValueStore::contains(const FieldValueMap* const other) {
 2  unsigned int otherSize = other->size();
 3  unsigned int tupleSize = fValueTuples->size();
 4  for (unsigned int i=0; i<tupleSize; i++) {
 5   FieldValueMap* valueMap = fValueTuples->elementAt(i);
 6   if (otherSize == valueMap->size()) {
 7    for (unsigned int j=0; j<otherSize; j++) {
 8     ...
 9    }
10  }
11  return false;
12 }
```

Listing C.1: The xalancbmk function responsible for the CPI variation in Cluster 1.

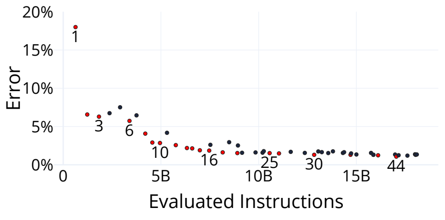
CPI prediction error of xalancbmk in the 2% configuration is 12.6%. The root cause of these outliers is that the CPI of the cluster centroid does not represent the average CPI of the instruction intervals in the cluster.

Figure C.6 explains this observation in more detail by showing the CPI distribution within each of the four clusters selected by SimPoint for the 2% error configuration of xalancbmk. In other words, we retrieved the CPI of each 100m instruction interval within xalancbmk that SimPoint assigns to each cluster and plotted their distribution. The red dots mark the CPI of the selected simulation point and the numbers above each distribution report the weight that SimPoint assigned to this cluster. SimPoint fundamentally assumes that an instruction interval that is close to the center of the cluster in terms of the (projected) BBV vector also yields close-to-average CPI. Figure C.6, and all our previously presented results, demonstrate that this assumption typically holds true, i.e., the red dots (BBV centroids) yield close-to-average CPI for Cluster 0, 2, and 3. For Cluster 1 on the other hand, the red dot is in the upper end of the CPI distribution.

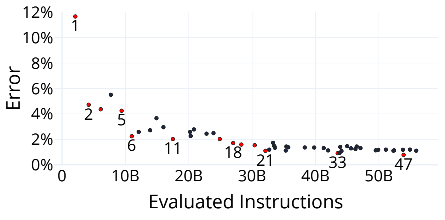
Listing C.1 shows the xalancbmk function which contains the performance-critical basic block in Cluster 1 and hence is the root cause of its CPI variation. Xalancbmk converts XML into HTML and other formats, and this specific function checks if a value map is contained within another. The performance difference is caused by a `load` instruction which is part of the value map dereferencing operation at line 6. The latency of this load varies significantly throughout benchmark execution because locality is a property of the input document, i.e., it determines to what extent xalancbmk accesses cache blocks that are spatially or temporally close to each other. The result is that the CPI of this cluster is high (i.e., up to 2.8) when locality is poor and low (i.e.,



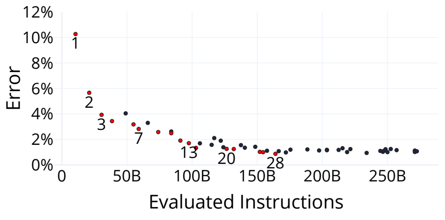
(a) 10 million (avg).



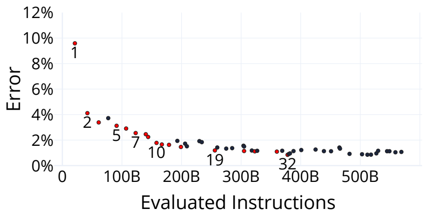
(b) 30 million (avg).



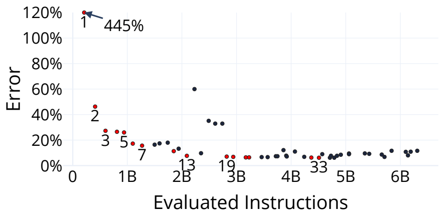
(c) 100 million (avg).



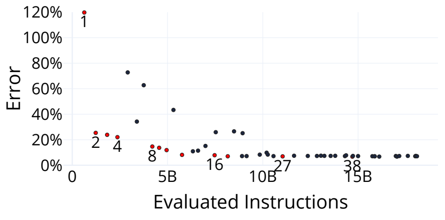
(d) 500 million (avg).



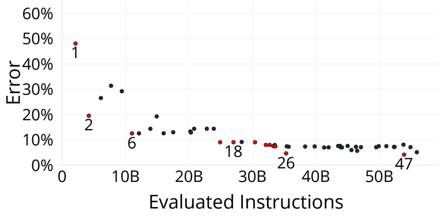
(e) 1 billion (avg).



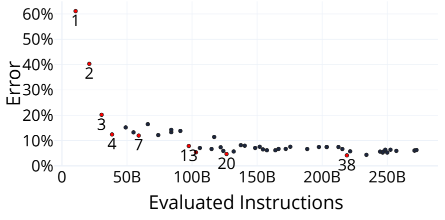
(f) 10 million (max).



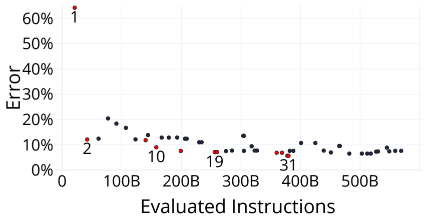
(g) 30 million (max).



(h) 100 million (max).



(i) 500 million (max).



(j) 1 billion (max).

Figure C.7: Average and maximum CPI prediction error versus evaluation overhead for *MaxK* values between 1 and 50. *Selecting a MaxK value close to the error saturation point balances error and evaluation overhead.*

0.5) when locality is good.

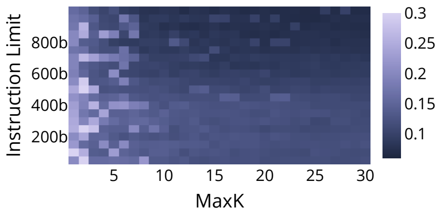
6.3 Selecting Favorable SimPoint Parameters

Having confirmed that SimPoint can yield high accuracy, we now explore the impact of SimPoint parameter selection. Our objective is to devise general guidelines that architects can use to configure SimPoint to yield a favorable balance between accuracy and evaluation overhead.

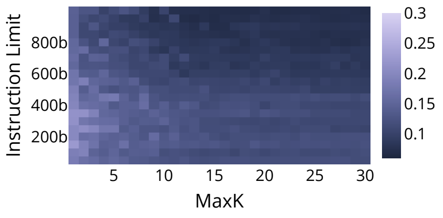
MaxK. The SimPoint parameter that has the most significant impact on error and evaluation overhead is MaxK. The reason is that MaxK bounds the number of simulation points that SimPoints can use for any benchmark and setting it too low results in higher error whereas setting it too high yields higher evaluation overhead because SimPoint returns more clusters. Figure C.7 quantifies this relationship by comparing the average (Figures C.7a to C.7e) and maximum CPI prediction error (Figures C.7f to C.7j) to the number of evaluated instructions for MaxK values between 1 and 50.

Points that are on the Pareto front in Figure C.7 are red, and these points are important because they strike a Pareto-optimal balance between CPI prediction error and evaluation overhead, i.e., reducing error will result in an increase in evaluation overhead and vice versa. Figures C.7a to C.7e show that error improves with increasing MaxK and then saturates. We note that saturation occurs with lower MaxK values for the larger interval sizes. Computer architects should hence select MaxK values around the saturation point as increasing MaxK beyond this increases overhead while only marginally reducing error. To give a specific recommendation, we consider MaxK values that are Pareto optimal with respect to both average and maximal error and select the lowest MaxK value that achieves an average error within one percentage point of the minimal average error, yielding saturation MaxK values of 13, 16, 18, 13, and 10 for the 10m, 30m, 100m, 500m, and 1b interval sizes.

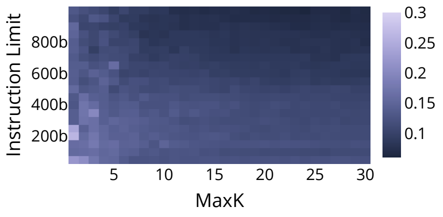
Figures C.7a to C.7e also show that fewer simulation points are needed to achieve a target CPI prediction error with larger simulation points. For example, Figure C.7a shows that MaxK must be set to 6 to achieve an error below 4% with the 10m interval, whereas Figure C.7e shows that setting MaxK to 3 is sufficient with the 1b interval. The reason is that larger simulation points capture a larger part of benchmark execution and therefore fewer are required to accurately predict average CPI. On the flip side, larger



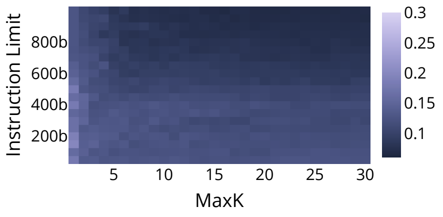
(a) 10 million (avg).



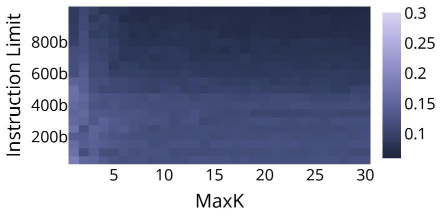
(b) 30 million (avg).



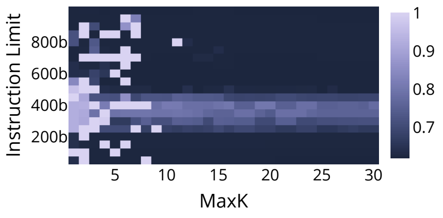
(c) 100 million (avg).



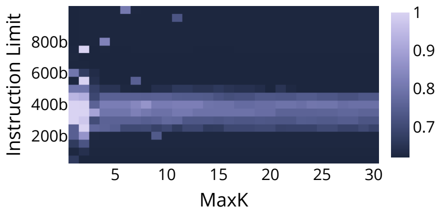
(d) 500 million (avg).



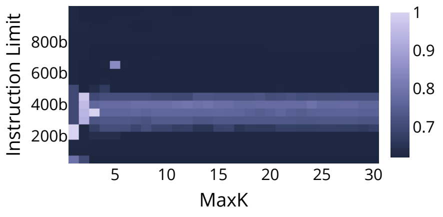
(e) 1 billion (avg).



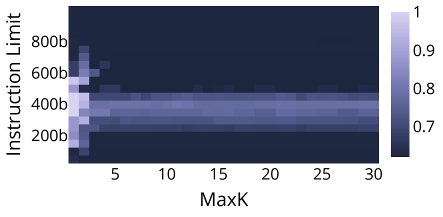
(f) 10 million (max).



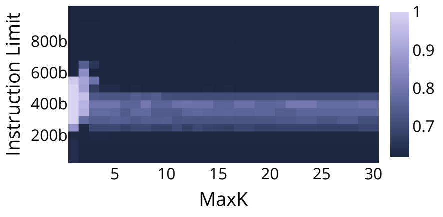
(g) 30 million (max).



(h) 100 million (max).

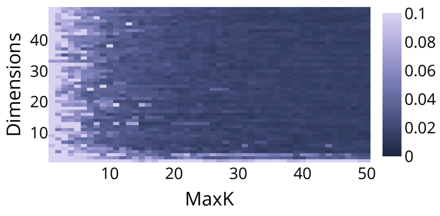


(i) 500 million (max).

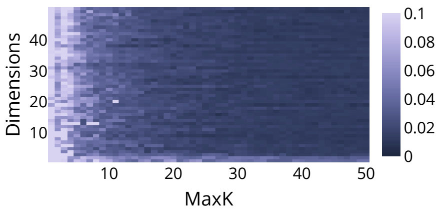


(j) 1 billion (max).

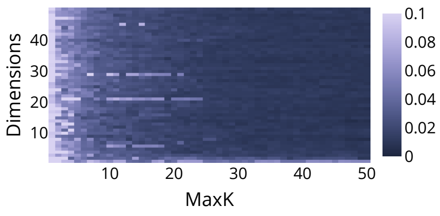
Figure C.8: Saturating average and maximum CPI prediction error for *instruction coverage* versus MaxK. *High errors occur when SimPoint misses key benchmark phases.*



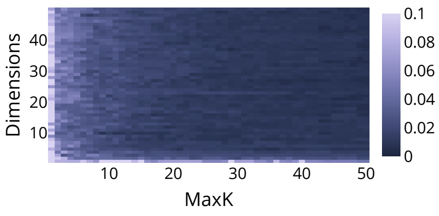
(a) 10 million (avg).



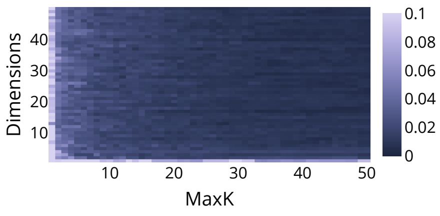
(b) 30 million (avg).



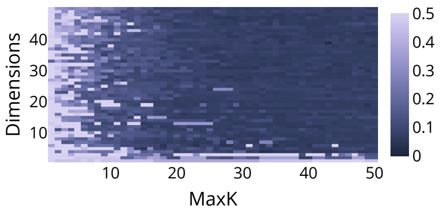
(c) 100 million (avg).



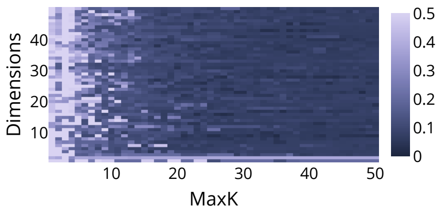
(d) 500 million (avg).



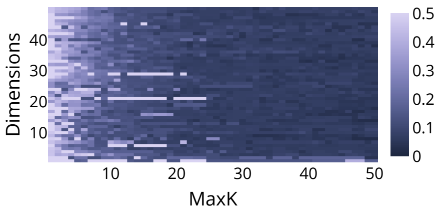
(e) 1 billion (avg).



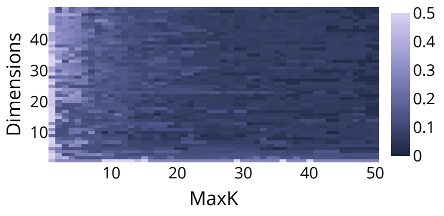
(f) 10 million (max).



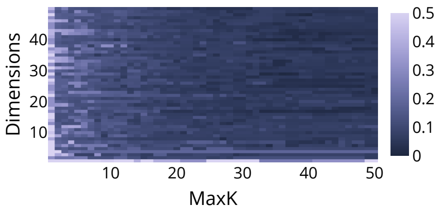
(g) 30 million (max).



(h) 100 million (max).



(i) 500 million (max).



(j) 1 billion (max).

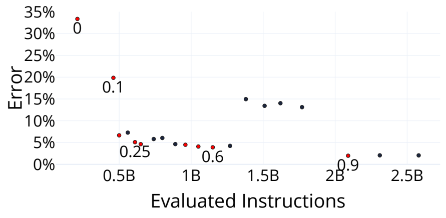
Figure C.9: Saturating average and maximum CPI prediction error for *dimensionality* versus MaxK. *Dimensionality should be 10 or higher, and it is hence unnecessary to deviate from the SimPoint default of 15.*

intervals also yield much higher evaluation overhead as reaching this accuracy requires simulating 1.1 billion instructions with 10m intervals and 61 billion instructions with 1b intervals; 1b intervals hence increase evaluation overhead by 55.5×.

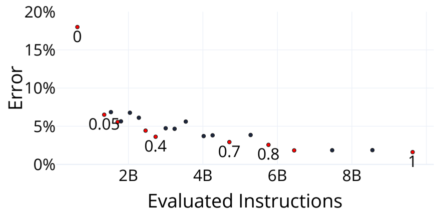
Benchmark coverage. Figure C.8 shows heat maps of average and maximum CPI prediction errors when we vary MaxK and collect BBVs from the first 50 to 1,000 billion dynamic instructions of each benchmark. We let average (maximum) errors saturate at 30% (100%) to improve readability, i.e., the lightest dots in the heat maps represent errors of this magnitude or higher. While the average errors are generally reasonable when MaxK is sufficiently high, the maximum errors are high across the board. The reason is that imagick has an important phase that does not occur in the first 1,000 billion instructions. The line across Figures C.8f to C.8j between 250 billion and 450 billion instructions is due to xz selecting simulation points which are not representative of its overall CPI. Architects should hence be cognizant of the fact that the identified simulation points will only be representative of the part of benchmark execution that they collect BBVs from.

Dimensionality. Figure C.9 reports the average and maximum error when we sweep the validation space spanned by dimensionalities from 1 to 50 and MaxK from 1 to 50. Recall that dimensionality describes the number of dimensions in the space that BBVs are projected onto before running k-means clustering. The key takeaway is that both average and maximum error improves with increasing dimensionality and MaxK, i.e., selecting a higher MaxK cannot compensate for selecting too low dimensionality and vice versa. More specifically, we find that selecting a dimensionality of at least 10 is sufficient; the SimPoint default dimensionality of 15 is hence reasonable. While we do not observe any loss of accuracy with high dimensionality, it marginally increases the runtime of the SimPoint tool.

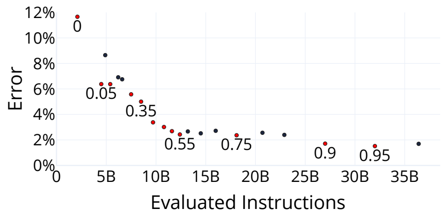
BIC threshold. Figure C.10 plots the average and maximum CPI prediction error versus evaluation overhead for all BIC thresholds between 0 and 1.0 in increments of 0.05. We fix MaxK at the saturation error points for each instruction interval, i.e., 13, 16, 18, 13, and 10 for the 10m, 30m, 100m, 500m, and 1b intervals, respectively, to ensure that we optimize the BIC threshold on top of an efficient configuration. Figure C.10 shows that bringing the BIC threshold down from the SimPoint default of 0.9 can reduce evaluation overhead while maintaining low error, but setting the BIC threshold too low yields high maximum errors. As in our MaxK analysis, we find a balance by considering the BIC thresholds that are Pareto optimal for both average and



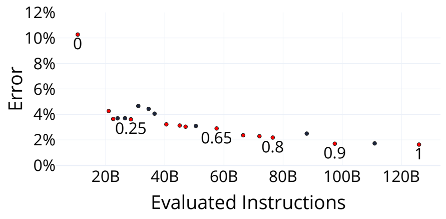
(a) 10 million (avg).



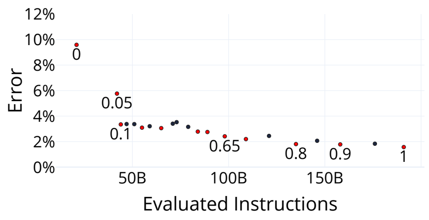
(b) 30 million (avg).



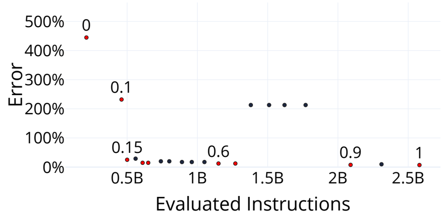
(c) 100 million (avg).



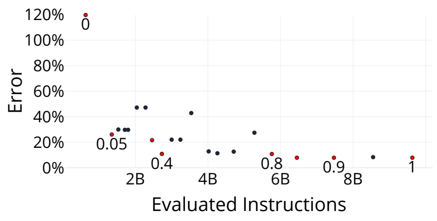
(d) 500 million (avg).



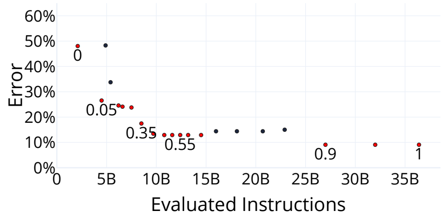
(e) 1 billion (avg).



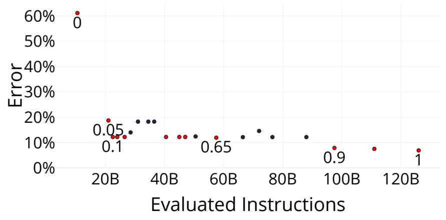
(f) 10 million (max).



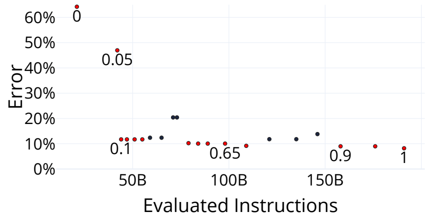
(g) 30 million (max).



(h) 100 million (max).



(i) 500 million (max).



(j) 1 billion (max).

Figure C.10: Average and maximum CPI prediction error versus simulation overhead for *BIC thresholds* between 0 and 1 at saturation MaxKs. *Moderately lowering the BIC threshold reduces evaluation overhead while maintaining low error.*

maximum error and then select the lowest threshold that yields an average error within one percentage point of the minimum average error for each interval size. This yields BIC thresholds of 0.9, 0.8, 0.55, 0.9, and 0.65 for the 10m, 30m, 100m, 500m, and 1b intervals, respectively. The differences are primarily caused by the subset of thresholds that are Pareto optimal. More specifically, the 100m and 1b intervals have configurations close to the error constraint that are Pareto optimal with both errors and therefore yield lower BIC thresholds than the other intervals.

7 Related Work

Benchmark sampling. Sampled simulation methodologies divide the program into representative samples that can be simulated to reduce simulation runtime. The underlying motivation is that software applications consist of regions with repetitive microarchitectural behavior, e.g., similar CPI or number of cache misses. SimPoint [10]–[13], which we focus on in this work, was one of the first approaches to appear, and many later approaches are based on SimPoint. Pinpointing [21], for example, uses dynamic instrumentation to collect BBVs. Other approaches adapt SimPoint to multi-threaded simulation. Perelman et al. [22] proposes an approach that extends SimPoint to analyze phase behavior in parallel programs and selects simulation points based on this phase characterization. BarrierPoint [23] and LoopPoint [24] also adapt SimPoint to multi-threaded benchmarks. BarrierPoint creates the regions for selecting the representative intervals based on barriers, whereas LoopPoint uses loop boundaries to create intervals.

Statistical simulation [25], [26] alternates between a fast low-detail simulation mode and a slow high-detail mode and predicts target metrics (e.g., CPI) from the measurements obtained during detailed simulation. A key benefit of this approach is that the statistical confidence of the prediction can be computed. Time-Based Sampling (TBS) [27] selects the representative portions of the program by sampling based on time instead of instruction count, and TaskPoint [28] proposes a technique for sampled simulation of task-based programs. A small number of task instances are selected as representative and simulated in detail while the remaining are fast-forwarded. LiveSim [29] incrementally simulates the benchmark from memory checkpoints, i.e., checkpoints are randomly selected for simulation, and metrics of interest (with confidence) are reported in real-time.

Tracing mechanisms. The most related tracing mechanism to TraceDoctor is TracerV [9], and we demonstrated that TracerDoctor yields higher performance than TracerV in Section 5. Beyond the FireSim [5] ecosystem, a plethora of hardware-based tracing tools exist [30]–[34]. Fundamentally, the amount of trace data must be adapted to the bandwidth of the tracing interface, and the insights that led to TraceDoctor are hence applicable to these approaches as well. More specifically, it is generally beneficial to (i) provide flexibility in selecting what information to trace, and (ii) empower users to implement problem-specific filtering and compression on the host. Tracing can also be performed in software, e.g., with `perf` [35] and `strace` [36], but software-level tracing typically has a significant performance impact.

8 Conclusion

We have now presented TraceDoctor which is a versatile high-performance tracing interface for FireSim. TraceDoctor minimizes simulator slowdown by executing a configurable number of workers on the host to perform evaluation-specific analyses on trace data in parallel with the FireSim simulation. We used TraceDoctor to independently validate SimPoint as well as study the accuracy versus evaluation overhead trade-off. We confirm that SimPoint is accurate when configured to collect a sufficient number of simulation points and explain that the residual error of SimPoint occurs when the CPI of the cluster centroid is not representative of the average CPI of the instruction intervals in the cluster. Interestingly, we find that relatively many and relatively small simulation points yield low error and low evaluation overhead. This is in contrast to current practice in the field where architects typically select one or a few relatively large simulation points — which yields both relatively high error and relatively high evaluation overhead.

Acknowledgments

We thank the anonymous reviewers for their feedback. We used the Experimental Infrastructure for Exploration of Exascale Computing (eX3) for some of our experiments; eX3 is supported by the Research Council of Norway (Grant No. 270053). Magnus Jahre is supported by the Research Council of Norway (Grant No. 286596).

References

[1] N. Binkert, B. Beckmann, G. Black, *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[2] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12.

[3] Synopsys, *Synopsys VCS*, 2023. Available: `https://www.synopsys.com/verification/simulation/vcs.html`.

[4] W. Snyder, *Verilator*, 2023. Available: `https://verilator.org`.

[5] S. Karandikar, H. Mao, D. Kim, *et al.*, "Firesim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *Proceedings of the International Symposium on Computer Architecture*, ser. ISCA, IEEE Press, 2018, pp. 29–42.

[6] SPEC, *SPEC CPU 2017*, `https://www.spec.org/cpu2017/`, 2019.

[7] K. Asanović, R. Avizienis, J. Bachrach, *et al.*, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[8] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.

[9] S. Karandikar, A. Ou, A. Amid, *et al.*, "FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 715–731.

[10] E. Perelman, G. Hamerly, and B. Calder, "Picking Statistically Valid and Early Simulation Points," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003, pp. 244–255.

[11] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and More Flexible Program Analysis," in *Journal of Instruction Level Parallelism (JILP)*, vol. 7, 2005, pp. 1–28.

[12] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," in *Proceedings International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001, pp. 3–14.

[13] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 45–57, 2002.

[14] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, pp. 59–67, 2002.

[15] B. Gottschall, S. Santana, and M. Jahre, *Balancing Accuracy and Evaluation Overhead in Simulation Point Selection*, Zenodo, 2023. Available: `https://doi.org/10.5281/zenodo.8273178`.

[16] AMD, *Alveo U250 Data Center Accelerator Card*, 2023. Available: `https://www.xilinx.com/products/boards-and-kits/alveo/u250.html`.

[17] AMD, *Alveo U280 Data Center Accelerator Card*, 2023. Available: `https://www.xilinx.com/products/boards-and-kits/alveo/u280.html`.

[18] Simula Research Laboratory, *Experimental Infrastructure for Exploration of Exascale Computing*. Available: `https://www.ex3.simula.no/`.

[19] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*, 2019. arXiv: `1912.05848 [cs.DC]`.

[20] D. Biancolin, S. Karandikar, D. Kim, *et al.*, "FASED: FPGA-Accelerated Simulation and Evaluation of DRAM," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 330–339.

[21] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing Representative Portions of Large Intel ® Itanium ® Programs with Dynamic Instrumentation," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2004, pp. 81–92.

[22] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting Phases in Parallel Applications on Shared Memory Architectures," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2006, 10–pp.

[23] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, “BarrierPoint: Sampled Simulation of Multi-Threaded Applications,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 2–12.

[24] A. Sabu, H. Patil, W. Heirman, and T. E. Carlson, “LoopPoint: Checkpoint-Driven Sampled Simulation for Multi-Threaded Applications,” in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 604–618.

[25] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, “SMARTS: Accelerating Microarchitecture Simulation Via Rigorous Statistical Sampling,” in *ACM SIGARCH Computer Architecture News*, 2003, pp. 84–97.

[26] M. Ekman and P. Stenstrom, “Enhancing Multiprocessor Architecture Simulation Speed Using Matched-Pair Comparison,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005, pp. 89–99.

[27] E. K. Ardestani and J. Renau, “ESESC: A Fast Multicore Simulator Using Time-Based Sampling,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 448–459.

[28] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé, “TaskPoint: Sampled Simulation of Task-Based Programs,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 296–306.

[29] S. Hassani, G. Southern, and J. Renau, “LiveSim: Going Live with Microarchitecture Simulation,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 606–617.

[30] ARM, *ARM CoreSight Architecture*, 2023. Available: `https://developer.arm.com/Architectures/CoreSight%5C%20Architecture`.

[31] Intel, *Intel® 64 and IA-32 Architectures Software Developer Manuals*, 2023. Available: `https://intel.com/content/www/us/en/developer/articles/technical/intel-sdm`.

[32] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, “iWatcher: Efficient Architectural Support for Software Debugging,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2004, p. 224.

[33] SEGGER, *SEGGER J-Trace Streaming Trace Probes*, 2023. Available: `https://www.segger.com/products/debug-probes/j-trace/`.

[34] AMD, *AMD Xilinx Integrated Logic Analyzer*, 2023. Available: `https://www.xilinx.com/products/intellectual-property/ila.html`.

[35] Linux, *Perf Wiki*, 2023. Available: `https://perf.wiki.kernel.org/index.php/Main%5C_Page`.

[36] Linux, *Strace – The Linux Syscall Tracer*, 2023. Available: `https://strace.io/`.

ISBN 978-82-326-7606-4 (printed ver.)
ISBN 978-82-326-7605-7 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

NTNU

Norwegian University of
Science and Technology