TIP: Time-Proportional Instruction Profiling

Björn Gottschall
bjorn.gottschall@ntnu.no
Norwegian University of Science and Technology
Trondheim, Norway

Lieven Eeckhout
lieven.eeckhout@ugent.be
Ghent University
Ghent, Belgium

Magnus Jahre
magnus.jahre@ntnu.no
Norwegian University of Science and Technology
Trondheim, Norway

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×.

CCS CONCEPTS

• **General and reference → Performance**; **Measurement**; • **Hardware → Integrated circuits**; • **Computer systems organization → Architectures**.

MICRO '21, October 18–22, 2021, Virtual Event, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480058

ACM Reference Format:
Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2021. TIP: Time-Proportional Instruction Profiling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480058

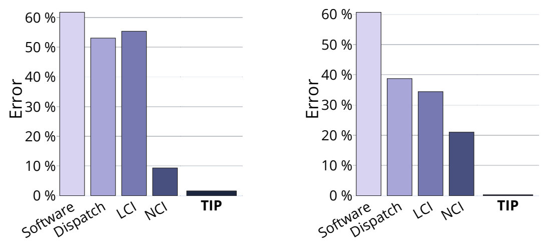
1 INTRODUCTION

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 [22].

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 1). Software-level profilers (e.g., Linux `perf` [34][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 [52] and can be addressed by adding hardware support for instruction sampling (e.g., Intel PEBS [26], AMD IBS [15], or Arm SPE [2]).

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 Processor Event-Based Sampling (PEBS) [26] returns the address of the next instruction that commits after the sample is taken, i.e., a *Next Committing Instruction (NCI)* heuristic. Profiling approaches [14, 47] that use debug interfaces, such as Arm CoreSight [3], systematically sample the *Last Committed Instruction (LCI)*. Finally, AMD's Instruction-Based Sampling (IBS) [15] and Arm's Statistical Profiling Extension (SPE) [2] 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

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



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

Figure 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.*

through the processor back-end [13]). 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 [5, 37, 52].

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 1a shows that Software profiling, the Dispatch-tagging strategy used by AMD's IBS [15], and the LCI-strategy of external profilers [3, 14, 47] 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 [26] 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 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 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).

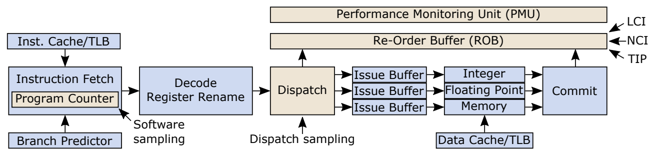
Time-Proportional Instruction Profiler (TIP). TIP bridges the gap between the state-of-the-art profilers and Oracle by combining the time attribution policies of Oracle with statistical sampling, thereby reducing the amount of profiling data by several orders of magnitude compared to Oracle (i.e., 192 KB/s versus 179 GB/s at the commonly used 4 KHz sampling frequency [34]) at the cost of introducing statistical error. Interestingly, Figure 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) [57] within the FireSim [28] 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 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.

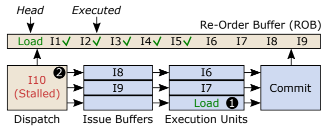
Key Contributions:

- 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 [57], and use the FPGA-accelerated FireSim [28] to simulate SPEC CPU2017 [44] and PARSEC [6] 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 [34], the dispatch-tagging heuristic used in AMD IBS [15] and Arm SPE [2], the LCI-heuristic used in external monitors [3, 14, 47], and the NCI-heuristic used in Intel PEBS [26], 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.

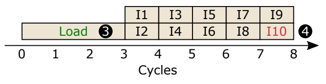
[2]Our tools are available at https://github.com/EECS-NTNU.



(a) Out-of-order architecture.



(b) Dispatch sampling example.



(c) Commit timeline.

Figure 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 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 [19]. 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 [15], Arm SPE [2], and ProfileMe [13]) 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 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 2c shows the situation in Figure 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).

Software profiling is also not time-proportional due to a phenomenon prior work referred to as skid [15, 52]. 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 [26]) samples the next-committing instruction, whereas LCI (as supported by external monitors [4, 14, 25, 42, 47]) 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 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 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 3). In this case, there is an instruction at the head of the ROB but it cannot

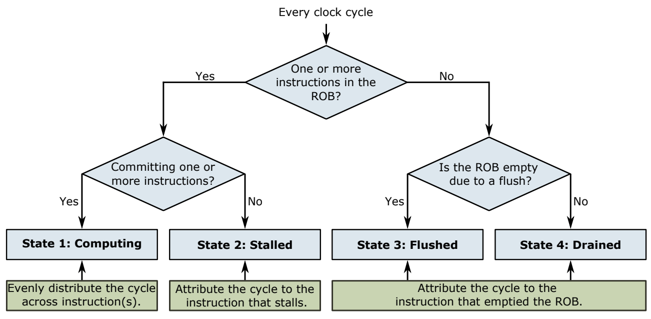

Every clock cycle
One or more instructions in the ROB?
Yes
No
Committing one or more instructions?
Is the ROB empty due to a flush?
Yes
No
Yes
No
State 1: Computing
State 2: Stalled
State 3: Flushed
State 4: Drained
Evenly distribute the cycle across instruction(s).
Attribute the cycle to the instruction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

Figure 3: Oracle profiler clock cycle attribution overview.

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 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 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 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.

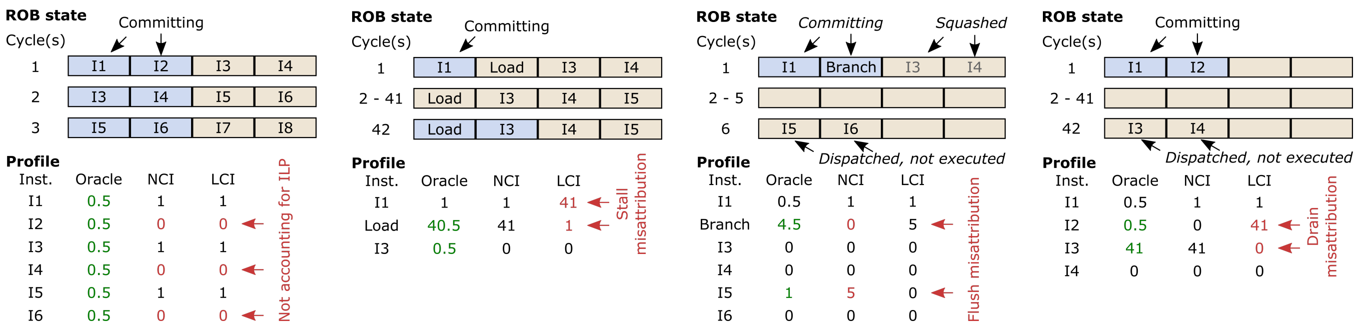
Not acknowledging ILP within the commit stage renders the NCI and LCI profiles difficult to interpret. The key reason is that many applications execute similar instruction sequences over and over. Since NCI and LCI select instructions to sample with a fixed policy, they will be biased towards selecting 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 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 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 4d exemplifies this situation. In cycle 1, *I1* and *I2* are committed. This leaves the ROB empty until



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

Figure 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.*

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 4c, and the processor will be in the Computing state when *I5* eventually commits. The same applies to *I3* from Drained to Stalled (Figure 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.

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

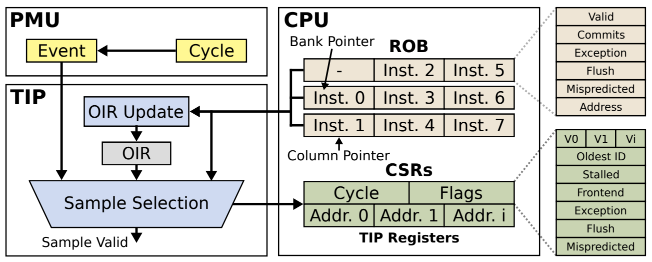


Figure 5: Structural overview of our Time-Proportional Instruction Profiler (TIP). *TIP is triggered by the PMU, collects a sample, and finally exposes the sample to software.*

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 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` [34] 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 [57], 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

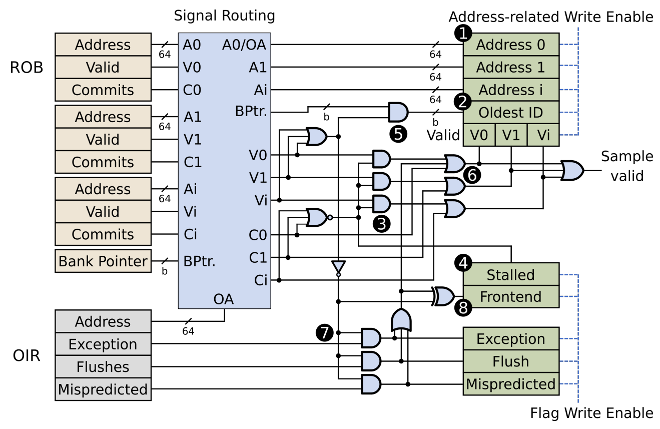


Figure 6: TIP sample selection logic. *TIP classifies samples based on the the core state, ROB-flags, and OIR-flags.*

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 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 5 shows that TIP consists of an *Offending Instruction Register (OIR)* and two functional units (*OIR Update* and *Sample Selection*), and Figure 6 fleshes out the details of the *Sample Selection* unit. (The color-coding maps the components of Figure 6 to units in Figure 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 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 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 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 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 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` [34]. 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

[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.

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 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%.

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 [28] to evaluate the different performance profiling strategies. The simulated model uses the BOOM 4-way superscalar out-of-order core [57], see Table 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 [1]. 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 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.

Table 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

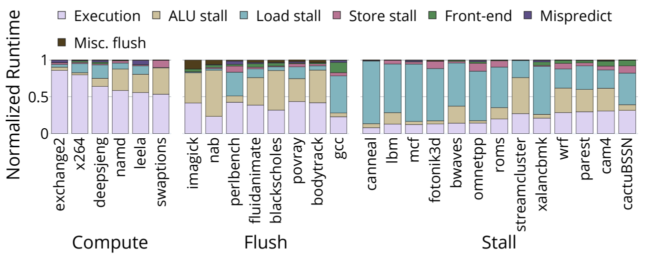


Figure 7: Normalized cycle stacks collected at commit.

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 [44] and PARSEC 3.0 [6] 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.

The benchmarks' execution characteristics are shown in Figure 7 which reports normalized cycle stacks captured at commit [18], 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 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` [34]).
- **Dispatch** tags an instruction at dispatch and samples when it commits (e.g., AMD IBS [15] and Arm SPE [2]).
- **Last Committed Instruction (LCI)** selects the last-committed instruction (e.g., Arm CoreSight [3])
- **Next Committing Instruction (NCI)** selects the next-committing instruction (e.g., Intel PEBS [26]).
- **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 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 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

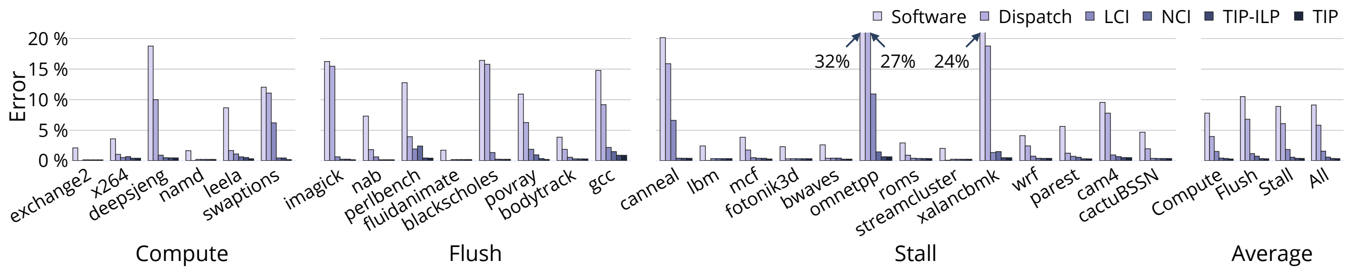


Figure 8: Function-level errors for the different profilers. *TIP, TIP-ILP, NCI, and LCI are accurate at the function level, while Software and Dispatch are largely inaccurate.*

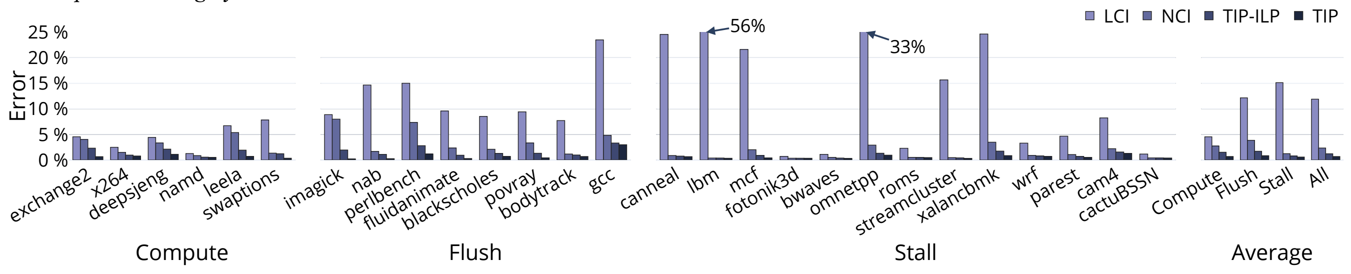


Figure 9: Basic-block-level errors for the different profilers. (Software and Dispatch are not shown because of their high error.) *TIP, TIP-ILP, and NCI are accurate at the basic block level, whereas LCI (and Software and Dispatch) are not.*

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 10 reports instruction-level profile error for TIP, TIP-ILP, and NCI. Software, Dispatch, and LCI are not included here as they 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 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 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.

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 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

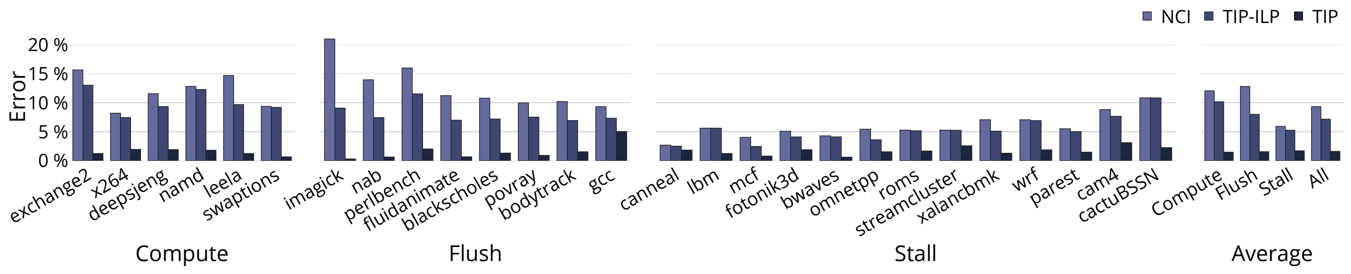
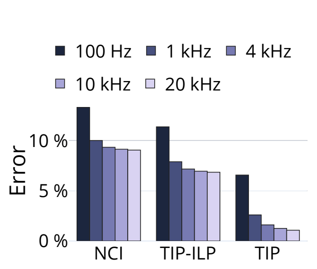
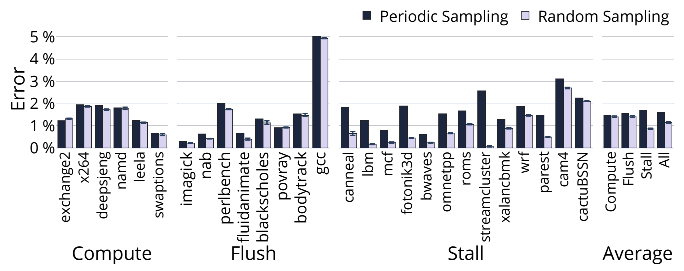


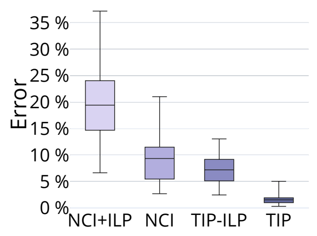
Figure 10: Instruction-level errors for the different profilers. (Software, Dispatch, and LCI are omitted because of their large errors.) *TIP is the only accurate profiler at the instruction level.*



(a) Sampling frequency.



(b) Periodic versus random sampling.



(c) Parallelism-aware NCI+ILP

Figure 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.*

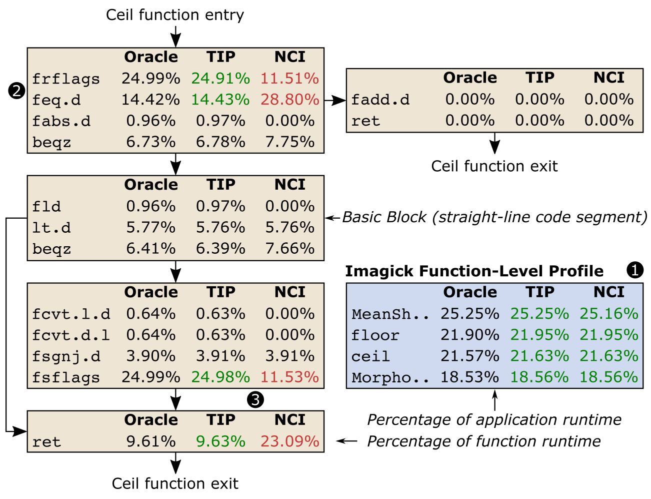
lower frequencies as these have more unsystematic error. The most interesting observation is that TIP's accuracy continues to measurably improve as the sampling frequency is increased beyond 4 KHz, while it saturates for the other profilers. The most notable example is gcc for which the error decreases from 5.0% at 4 KHz (see Figure 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 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 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,


Ceil function entry
❷
❸
Ceil function exit
Ceil function exit
Basic Block (straight-line code segment)
Imagick Function-Level Profile ❶
Percentage of application runtime
Percentage of function runtime

	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%

	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%

	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
ret	9.61%	9.63%	23.09%

	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%

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

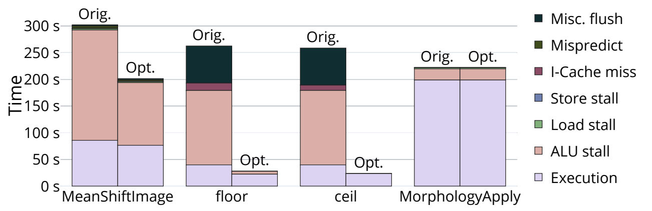


Figure 13: Time breakdown for the four most runtime-intensive functions in Imagick **comparing the original to our optimized version.** *The 1.93× speed-up is primarily due improved processor utilization.*

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` [34] against PEBS [26] 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 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, 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 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 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 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 [26], AMD IBS [15], and Arm SPE [2]; IBS and SPE are inspired by ProfileMe [13]. In addition, external profilers [4, 14, 25, 42, 47] use debug interfaces such as Arm CoreSight [3] to sample dynamic instructions. TIP is more accurate than these schemes (see Section 5).

Software-level profiling. Software-level profilers [20, 34, 40] 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 [23, 24] combines hardware performance counters with software instrumentation to profile an application across deep software stacks, while call-context profiling [59] efficiently identifies the common orders functions are called in. Causal profiling [12, 37, 41, 54] 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 [46], assessing input sensitivity [11, 55], profiling deployed applications [31], and function-level energy attribution [36].

Performance Monitoring Units (PMUs). A large body of work has investigated PMU design [32], and PMUs have a variety of uses (e.g., runtime optimization [8], performance analysis in managed languages [45, 51, 58], profile-guided compilation [9, 10], and profile-guided meta-programming [7]). Eyerman et al. [16] 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 [53] is also coarse-grain and cannot attribute time to instructions. Researchers have also investigated relating PMU events to application activities [5, 52] and how to make sense of PMU output [38, 49, 50, 56]; 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 [21, 22, 33, 43, 48]. Dynamic instrumentation (e.g., PIN [35] and Valgrind [39]) 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 [29] which uses FireSim [28] 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 [17, 30], but interval analysis targets dispatch while we target commit. GDP [27] 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] Amazon. 2021. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

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

[3] Arm. 2017. CoreSight Architecture Specification v3.0. https://developer.arm.com/documentation/ihi0029/.

[4] Arm. 2020. ULINKplus. http://www2.keil.com/mdk5/ulink/ulinkplus/.

[5] Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2021. BayesPerf: Minimizing performance monitoring errors using bayesian statistics. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[7] William J. Bowman, Swaha Miller, Vincent St-Amour, and R. Kent Dybvig. 2015. Profile-guided meta-programming. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 403–412.

[8] Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. 2007. Using HPM-Sampling to Drive Dynamic Compilation. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[9] Thomas M. Conte, Kishore N. Menezes, and Mary Ann Hirsch. 1996. Accurate and practical profile-driven compilation using the profile buffer. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 36–45.

[10] Thomas M. Conte, Burzin A. Patel, and J. Stan Cox. 1994. Using branch handling hardware to support profile-driven optimization. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 12–21.

[11] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive profiling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 89–98.

[12] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code that Counts with Causal Profiling. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*.

[13] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. 1997. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 292–302.

[14] Asbjørn Djupdal, Björn Gottschall, Fatemeh Ghasemi, and Magnus Jahre. 2021. Lynsyn and LynsynLite: The STHEM Power Measurement Units. In *Towards Ubiquitous Low-power Image Processing Platforms*, Magnus Jahre, Diana Göhringer, and Philippe Millet (Eds.). Springer International Publishing, 93–114.

[15] Paul J Drongowski. 2007. *Instruction-based sampling: A new performance analysis technique for AMD family 10h processors.* Technical Report. AMD.

[16] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2006. A performance counter architecture for computing accurate CPI components. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 175–184.

[17] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2009. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)* 27, 2 (2009), 1–37.

[18] Stijn Eyerman, Wim Heirman, Kristof Du Bois, and Ibrahim Hur. 2018. Multi-stage CPI stacks. *IEEE Computer Architecture Letters (CAL)* 17, 1 (2018), 55–58.

[19] Antonio González, Fernando Latorre, and Grigorios Magklis. 2010. *Processor microarchitecture: An implementation perspective.* Morgan & Claypool Publishers.

[20] Google. 2020. gperftools. https://github.com/gperftools/gperftools.

[21] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *ACM SIGPLAN Notices*.

[22] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *International Conference on Software Engineering (ICSE)*.

[23] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. 2005. Automating vertical profiling. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[24] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical profiling: understanding the behavior of object-oriented applications. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

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

[26] Intel. 2021. 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.

[27] Magnus Jahre and Lieven Eeckhout. 2018. GDP: Using dataflow properties to accurately estimate interference-free performance at runtime. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 296–309.

[28] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[29] Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. 2020. 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)*.

[30] T. S. Karkhanis and J. E. Smith. 2004. A first-order superscalar processor model. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*.

[31] Chung Hwan Kim, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Xiangyu Zhang, and Dongyan Xu. 2014. 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 (SIGMETRICS)*.

[32] Georgios Kornaros and Dionisios Pnevmatikatos. 2013. A survey and taxonomy of on-chip monitoring of multicore systems-on-chip. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 18, 2 (2013), 1–38.

[33] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO)*.

[34] Linux. 2020. perf. https://perf.wiki.kernel.org/index.php/Main_Page.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Notices*.

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

[37] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of Java profilers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 187–197.

[38] Todd Mytkowicz, Peter F. Sweeney, Matthias Hauswirth, and Amer Diwan. 2007. Time interpolation: So many metrics, so few registers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 286–300.

[39] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[40] NTNU. 2020. PPerf. https://github.com/EECS-NTNU/pperf.

[41] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. 2019. 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)*.

[42] Ahmad Sadek, Ananya Muddukrishna, Lester Kalms, Asbjørn Djupdal, Ariel Podlubne, Antonio Paolillo, Diana Goehringer, and Magnus Jahre. 2018. Supporting utilities for heterogeneous embedded image processing platforms (STHEM): An overview. In *Applied Reconfigurable Computing (ARC)*.

[43] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. 2017. The CSI Framework for Compiler-Inserted Program Instrumentation. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*.

[44] SPEC. 2019. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[45] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. 2004. Using hardware performance monitors to understand the behavior of java applications. In *Proceedings of the Conference on Virtual Machine Research and Technology Symposium (VM)*.

[46] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. 2009. Binary analysis for measurement and attribution of program performance. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 441–452.

[47] Matthew Tancreti, Mohammad Sajjad Hossain, Saurabh Bagchi, and Vijay Raghunathan. 2011. 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 (SenSys)*.

[48] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Proceedings of the USENIX conference on Networked Systems Design and Implementation (NSDI)*.

[49] Vincent M. Weaver and Sally A. McKee. 2008. Can hardware performance counters be trusted?. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 141–150.

[50] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. 2013. Non-determinism and overcount on modern hardware performance counter implementations. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 215–224.

[51] John Whaley. 2000. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of the ACM Conference on Java Grande (JAVA)*.

[52] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can we trust profiling results? Understanding and fixing the inaccuracy in modern profilers. In *Proceedings of the International Conference on Supercomputing (ICS)*. 284–295.

[53] A. Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44.

[54] Adarsh Yoga and Santosh Nagarakatte. 2019. Parallelism-centric what-if and differential analyses. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 485–501.

[55] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic profiling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 67–76.

[56] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. 2009. Accuracy of performance counter measurements. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 23–32.

[57] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*.

[58] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2015. Accurate profiling in the presence of dynamic compilation. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[59] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. 2006. Accurate, Efficient, and Adaptive Calling Context Profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.