TEA: Time-Proportional Event Analysis

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

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

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

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.

CCS CONCEPTS

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

KEYWORDS

Performance analysis, time proportionality, performance events

ISCA '23, June 17–21, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589058

ACM Reference Format:
Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2023. TEA: Time-Proportional Event Analysis. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589058

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 [16]. Industry has responded by exploiting specialization and integrating heterogeneous compute engines including GPU and domain-specific accelerators alongside conventional CPU cores [28]. 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 [6]. 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 [31] and AMD μProf [2], 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?

The first question (Q1) is typically addressed with a performance profiler. The state-of-the-art performance profiler TIP [22] is *time-proportional*, in contrast to other performance profilers [3, 4, 18, 19, 21, 30, 38], 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

[1] While attributing time to functions can be sufficient to address simple performance issues, addressing challenging performance issues requires instruction-level analysis [22]. Note that instruction-level analysis can also always be aggregated for presentation at coarser granularity whereas the opposite is not true.

execution time is the fraction of time it prevents the core from committing instructions [22]. 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 [22] 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 [19], Arm SPE [4], and IBM RIS [29] 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 [3, 20, 30, 54] 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 [22] 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 [32]. 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 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 [22]. 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 [58], 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 [19], Arm SPE [4], and IBM RIS [29][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 [46] benchmarks *lbm* and *nab*. For both benchmarks, the PICS provided by TEA explains the 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.

[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.

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 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.

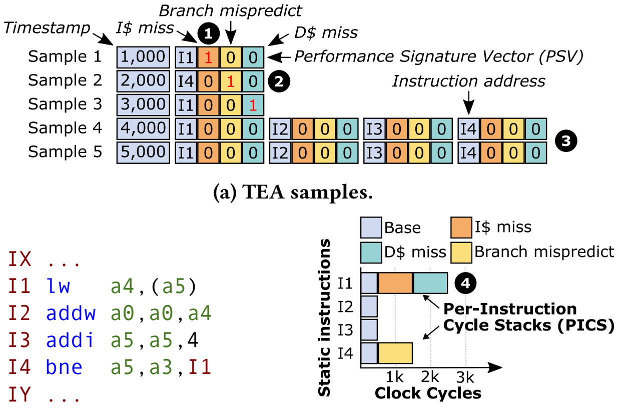
In summary, we make the following major contributions:

- 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 [58] 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 [22] 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 [22], 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).



(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.

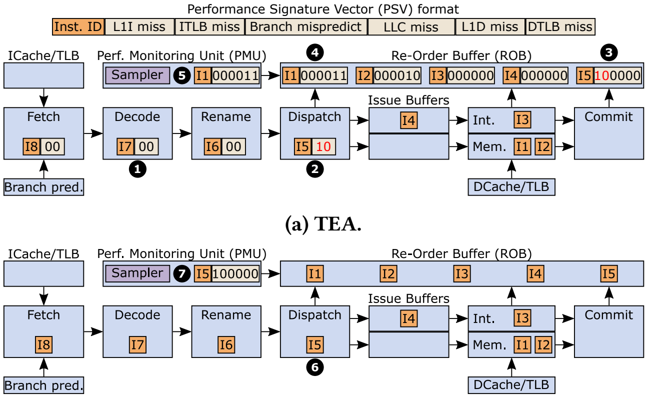
Figure 1: Example explaining how TEA creates PICS. *TEA explains how performance events cause performance loss.*

- **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 1 illustrates how TEA works in practice when an application executes the short loop in Figure 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 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` [38].) In Sample 1, the ROB has drained due to an instruction cache miss when fetching I1 and TIP [22] 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 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



(a) TEA.

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

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

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 1c). Each sample is mapped to static instructions using the address(es) of the instruction(s) and then categorized according to the 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 metadata (e.g., the instruction address) which the processor updates and forwards as the instruction flows through the pipeline. Figure 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 [19], Arm SPE [4], and IBM RIS [29] fall short because they tag instructions at dispatch or fetch and are hence not time-proportional. Figure 2b illustrates the operation of dispatch-tagging with the same core state as we used to explain TEA in Figure 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. (Recall that TEA sampled I1 in Figure 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 [58] in this section, the

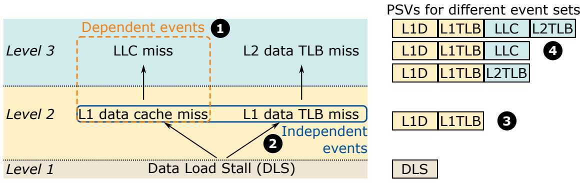


Figure 3: Performance event hierarchy for the Stalled (ST) commit state.

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.

Figure 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 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.

Table 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	✓	✓	✓	✓

TEA's performance events. Table 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 1 also shows that the instruction-driven approaches AMD IBS [1, 19], Arm SPE [4, 5], and IBM RIS [29] capture many of the same events as TEA which indicates that some events are important regardless of the specific architecture.

TEA microarchitecture. Figure 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, 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

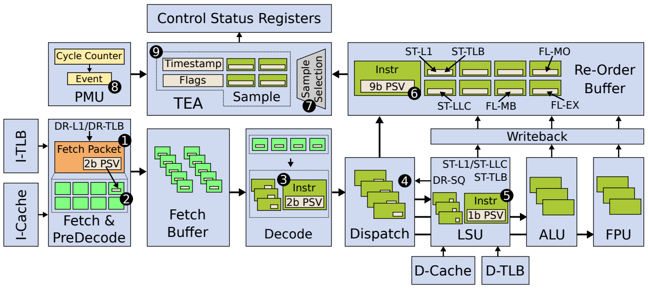


Figure 4: TEA microarchitecture.

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 [22] 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` [38]. 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 [22], 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 [10] and estimated its power consumption with Cadence Joules [11]. 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 [17] 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% [22]. 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 [33], a cycle-accurate FPGA-accelerated full-system simulator, to evaluate the different performance

Table 2: Baseline architecture configuration.

Part	Configuration
Core	OoO BOOM [58]: 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

profiling strategies. The simulated model is the BOOM 4-way superscalar out-of-order core [58], see Table 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 FPGAs in NTNU's Idun cluster [45]. 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 [23] 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 [46] 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 [1], Arm SPE [4], and IBM RIS [29]. 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 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 [30] which has been shown to be significantly more accurate than tagging instructions at fetch or dispatch [22]. **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 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

[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.

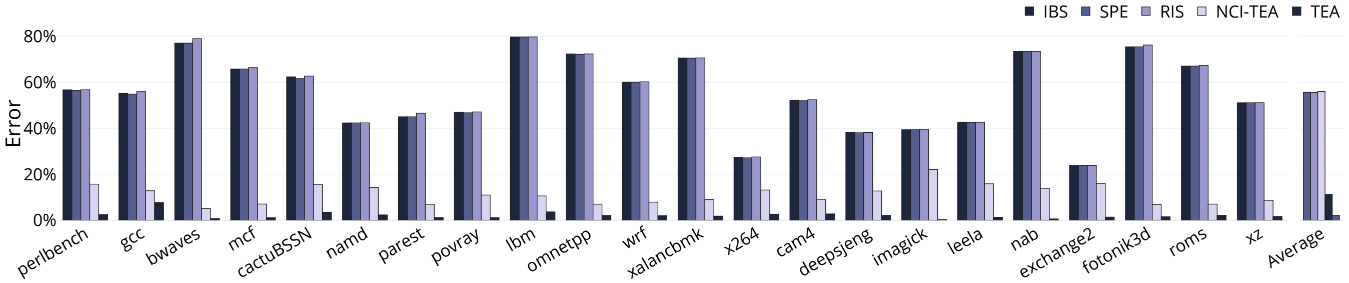
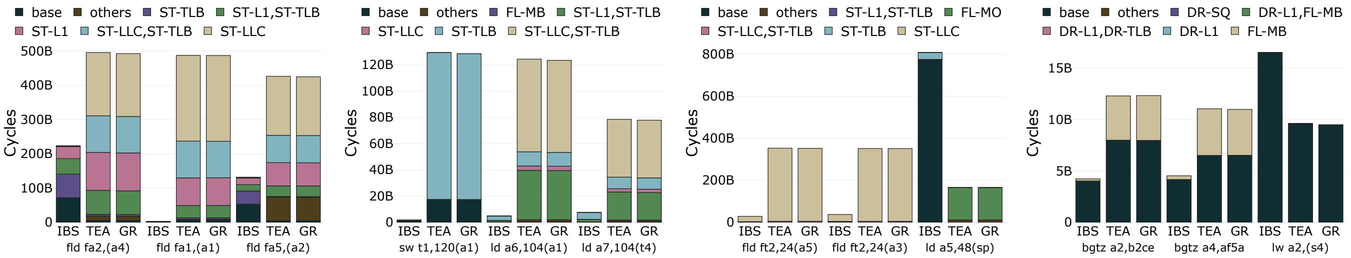


Figure 5: Quantifying the error for the PICS obtained through IBS, SPE, RIS, NCI-TEA, and TEA. *TEA achieves the highest accuracy within 2.1% (and at most 7.7%) compared to the golden reference.*



(a) bwaves **(b) omnetpp** **(c) fotonik3d** **(d) exchange2**

Figure 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.*

RIS tag instructions at dispatch or fetch which leads to non-time-proportional performance profiles. (This confirms the observation from prior work [22].) 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.

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%).

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 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 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 6d to Figure 5).

This analysis also illustrates TEA's ability to detect combined events. For example, the combination of cache and TLB misses,

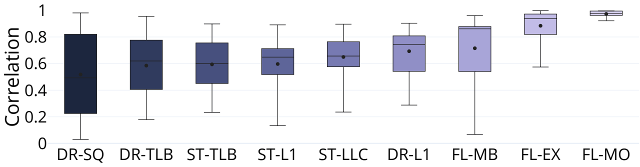


Figure 7: Quantifying the correlation between event count and its impact on performance. *Some event counts correlate strongly with their impact on performance while others do not.*

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 6a and 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 6a and 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 7 reports box plots for the Pearson correlation coefficient for all PSV events across all benchmarks.[5] 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.

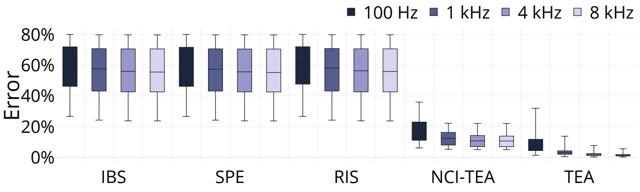
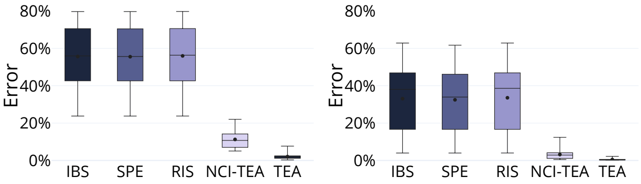


Figure 8: Error versus sampling frequency.



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

Figure 9: Errors at instruction and function granularity.

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 8 reports the accuracy of the various techniques as a function of sampling frequency. Accuracy is rather insensitive to sampling frequency above 4 KHz, which is why we chose it as our baseline sampling frequency to balance accuracy and run-time overhead.

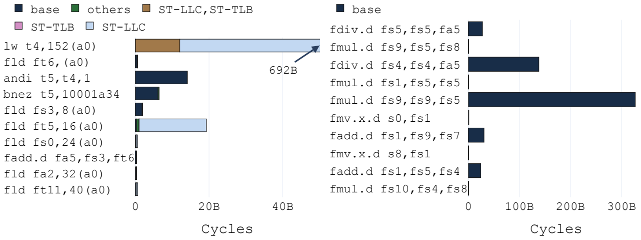
Analysis granularity. Figure 9 evaluates the accuracy of the various techniques when cycle stacks are computed at the instruction and function granularities; basic block and application granularities exhibit the same trends. TEA is uniformly the most accurate technique. While the error decreases at function 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.

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 [22],

[5] Event-driven approaches such as Intel PEBS [30] and DCPI [3] 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 captured in the same cycle, yielding independent profiles.



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

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

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 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 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 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.

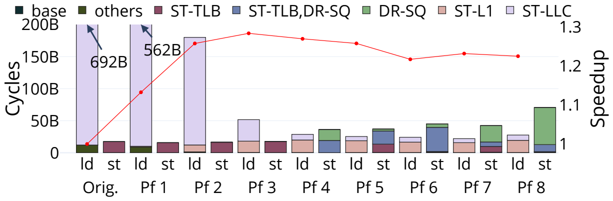
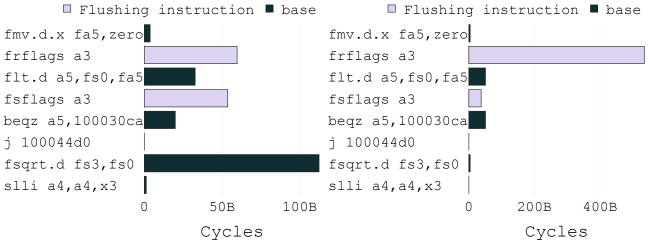


Figure 11: PICS and speedup for the most performance-critical load instruction and store instruction of lbm across a range of prefetch distances.



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

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

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 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 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 11).

Analyzing nab. Figure 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 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 [19], Arm SPE [4], and IBM RIS [29] 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 [30] and DCPI [3]. Researchers have hence investigated PMU design [36], and PMUs have a variety of uses (e.g., runtime optimization [9], performance analysis in managed languages [47, 52, 59], profile-guided compilation [12, 13], and profile-guided metaprogramming [8]). Xu et al. [53] 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 [7] 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 [50, 51, 57], 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. [20] propose a PMC architecture that enables constructing Cycles Per Instruction (CPI) stacks. The top-down model [54], 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 [26, 27] combines hardware performance counters with software instrumentation to profile an application across deep software stacks, while call-context profiling [60] efficiently identifies the common orders functions are called in. Causal profiling [15, 40, 43, 55] 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 [48], assessing input sensitivity [14, 56], and profiling deployed applications [35].

Static instrumentation modifies the binary to gather (extensive) application execution data at the cost of performance overhead [24, 25, 37, 44, 49]. Dynamic instrumentation (e.g., Pin [39] and Valgrind [42]) 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 [34] which uses FireSim [33] 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] AMD. 2021. 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.

[2] AMD. 2021. μProf. https://developer.amd.com/amd-uprof/.

[3] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. 1997. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems* 15, 4 (1997), 357–390.

[4] 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.

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

[6] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott B. Baden, and Dean M. Tullsen. 2012. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE Micro* 32, 6 (2012), 4–16.

[7] Subho S. Banerjee, Saurabh Jha, Zbigniew 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)*. Association for Computing Machinery, 832–844.

[8] William J. Bowman, Swaha Miller, Vincent St-Amour, and R. Kent Dybvig. 2015. Profile-Guided Meta-Programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 403–412.

[9] 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 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. Association for Computing Machinery, 553–568.

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

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

[12] 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)*. IEEE Computer Society, 36–45.

[13] 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)*. Association for Computing Machinery, 12–21.

[14] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-Sensitive Profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 89–98.

[15] 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)*. Association for Computing Machinery, 184–197.

[16] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-Specific Hardware Accelerators. *Commun. ACM* 63, 7 (2020), 48–57.

[17] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. Association for Computing Machinery, 189–194.

[18] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. 1997. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 292–302.

[19] Paul J Drongowski. 2007. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors.* Technical Report. AMD.

[20] 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)*. Association for Computing Machinery, 175–184.

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

[22] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2021. TIP: Time-Proportional Instruction Profiling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, 15–27.

[23] Björn Gottschall and Magnus Jahre. 2023. TraceDoctor: Versatile High-Performance Tracing for FireSim. The First FireSim and Chipyard User and Developer Workshop at ASPLOS.

[24] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction (SIGPLAN)*. Association for Computing Machinery, 120–126.

[25] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 145–155.

[26] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. 2005. Automating Vertical Profiling. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Association for Computing Machinery, 281–296.

[27] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical Profiling: Understanding the Behavior of Object-Priented Applications. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Association for Computing Machinery, 251–269.

[28] Mark D. Hill and Vijay Janapa Reddi. 2021. Accelerator-Level Parallelism. *Commun. ACM* 64, 12 (2021), 36–38.

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

[30] 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.

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

[32] Intel. 2022. Performance Monitoring Event Reference. https://perfmon-events.intel.com/.

[33] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. Firesim: FPGA-Accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE Press, 29–42.

[34] 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)*. Association for Computing Machinery, 715–731.

[35] 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)*. Association for Computing Machinery, 235–247.

[36] 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.

[37] 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)*. IEEE Computer Society, 75.

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

[39] 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 *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 190–200.

[40] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the Accuracy of Java Profilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 187–197.

[41] 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)*. IEEE Computer Society, 286–300.

[42] 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)*. Association for Computing Machinery, 89–100.

[43] 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)*. Association for Computing Machinery, 87–88.

[44] 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. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 2, Article 43 (2017), 25 pages.

[45] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. 2019. EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure. arXiv:1912.05848 [cs.DC]

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

[47] 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)*. USENIX Association, 5.

[48] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 441–452.

[49] 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)*. USENIX Association, 473–486.

[50] Vincent M. Weaver and Sally A. McKee. 2008. Can Hardware Performance Counters be Trusted?. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 141–150.

[51] 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)*. IEEE Computer Society, 215–224.

[52] John Whaley. 2000. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of the Conference on Java Grande (JAVA)*. Association for Computing Machinery, 78–87.

[53] 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)*. Association for Computing Machinery, 284–295.

[54] 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)*. IEEE Computer Society, 35–44.

[55] Adarsh Yoga and Santosh Nagarakatte. 2019. Parallelism-Centric What-If and Differential Analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 485–501.

[56] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic Profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 67–76.

[57] 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)*. IEEE Computer Society, 23–32.

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

[59] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2015. Accurate Profiling in the Presence of Dynamic Compilation. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Association for Computing Machinery, 433–450.

[60] 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)*. Association for Computing Machinery, 263–271.