Chips Need DIP: Time-Proportional Per-Instruction Cycle Stacks at Dispatch

Silvio Campelo de Santana
silvio.h.c.de.santana@ntnu.no
Norwegian University of Science and Technology
Trondheim, Norway

Joseph Rogers
joseph.c.p.rogers@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

Despite continuous accelerator improvements, high single-thread performance remains crucial due to Amdahl's Law. With technology scaling slowing, developers must produce code that is as performant as possible, which often involves instruction-level performance analysis. Such analysis can be visualized as Per-Instruction Cycle Stacks (PICS), which, when created at the commit stage of the processor, report each static instruction's contribution to overall execution time and capture the performance events it was subject to. Sadly, PICS at commit ($PICS_C$) often cannot explain performance because they solely capture how effectively instructions *egress* from the processor core's out-of-order execution window. In contrast, developers must typically also gain insight into how efficiently instructions *ingress* into the execution window to fully understand application performance.

$PICS_C$ must hence be complemented by PICS at dispatch ($PICS_D$) because execution window ingress issues result in specific instructions exhibiting high dispatch latencies. We therefore propose dispatch-time profiling (DIP), which combines time-proportional at dispatch attribution policies with statistical sampling to accurately report each static instruction's contribution to dispatch time as well as list the reason(s) for delayed ingress. DIP is simple to implement and incurs low overhead, i.e., storage and execution time and overheads of 49 bytes and ~1%, respectively, while delivering high accuracy (average profile error of 5.2%). This is a significant improvement over the 26.9% error of state-of-the-art dispatch-tagging as implemented in AMD IBS, Arm SPE, and IBM RIS. We demonstrate that needing $PICS_D$ and $PICS_C$ is the common case by showing that 18 out of our 22 SPEC2017 benchmarks simultaneously exhibit both ingress and egress issues. Additionally, we leverage DIP's $PICS_D$ to optimize the fotonik3d benchmark, improving its performance by 8.5%.

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

Keywords: Performance analysis, dispatch-time profiling, per-instruction cycle stacks

ACM Reference Format:
Silvio Campelo de Santana, Joseph Rogers, Lieven Eeckhout, and Magnus Jahre. 2026. Chips Need DIP: Time-Proportional Per-Instruction Cycle Stacks at Dispatch. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779212.3790139

1 Introduction

The imminent end of Moore's law, the end of Dennard scaling, mounting concerns about the sustainability of computing, and the increasing heterogeneity and complexity of computing systems make it increasingly important and challenging to ensure that applications are as efficient as possible. Interestingly, the increasing focus on accelerators results in CPU performance becoming relatively more important, because the relative performance impact of non-acceleratable code grows with accelerator popularity. This follows from Amdahl's law: Overall execution time is increasingly dominated by non-acceleratable code as accelerators become more prevalent and efficient [6]. Developers and architects thus need performance analysis approaches that identify the root cause(s) of application inefficiency so that they can be addressed through software and hardware optimizations. The impact of software optimization can be massive, e.g., Leiserson et al. [40] report a four orders of magnitude speed-up between a naive and a highly optimized kernel.

Addressing challenging performance issues requires identifying the hottest instructions, i.e., the (typically small) subset of static instructions that the application spends the most



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASPLOS '26, Pittsburgh, PA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790139

Table 1. DIP versus state-of-the-art profilers.

	Egress ($PICS_C$)	*Ingress ($PICS_D$)*
Not time-proportional	PEBS [34]	DT (IBS [19], SPE [5], RIS [33])
Time-proportional	TEA [27]	**DIP (This work)**

time on. For example, Netflix used instruction-level analysis to understand a challenging performance issue that, once addressed, resulted in a 3.5× performance improvement while also reducing average and tail latency [24]. Due to the sheer volume of instructions to analyze, instruction-level performance analysis relies on statistical sampling, i.e., infrequently inspecting the processor core to retrieve performance sampling units.[1] The sampling units are then taken as representative of the time that has passed since the last sampling unit was collected and used to create a statistical performance profile. A performance profile is time-proportional if the weight assigned to each static instruction is proportional to its impact on overall (execution) time [26]. Time-proportional profiles thus only suffer from statistical (random) error, whereas non-time-proportional profiles additionally incur systematic error (bias) [25].

Per-Instruction Cycle Stacks (PICS) enable analyzing an application's hottest instructions by breaking down their overall time contribution across the performance events or architectural states encountered during various dynamic executions — thereby explaining why the instruction took time and in turn enabling developers to derive optimization strategies that address key performance issues. The state-of-the-art approaches that create PICS come in two forms. AMD IBS [19], Arm SPE [5], and IBM RIS [33] support Dispatch-Tagging (DT); dispatch is the stage where the dynamic micro-operations (μops)[2] leave the in-order front-end of the core and enter the out-of-order execution window of its back-end. In contrast, TEA [27] and TIP [26] sample at commit and are time-proportional in terms of execution time. The reasons are that (i) the execution time impact of a μop is exposed at commit, and (ii) TEA and TIP attribute each sampling unit to the μop(s) that the core is exposing the execution time impact of in the cycle the sampling unit is collected. Intel PEBS [34] also samples at commit, but is not time-proportional due to bias in its attribution policies [26, 28]. TEA tracks performance events to break down each static instruction's impact on overall execution time across the (combinations of) event(s) its μops were subjected to, providing PICS at commit, or $PICS_C$ for short.

[1] In statistical terminology, a sampling unit is an individual measurement. Sampling units are combined to create a statistical sample, i.e., an application's performance profile in this work.

[2] An instruction is mapped to one or more μops during decode. The term (static) instruction hence refers to the instruction in the application binary, while the term μop refers to the dynamic execution of one of its μops. Cores track the address of each μop's instruction, e.g., to handle exceptions.

Perhaps surprisingly, we find that $PICS_C$ by themselves rarely explain application performance because they report solely how effective the core is at removing μops from the out-of-order execution window (egress), e.g., due to a long-latency load blocking commit and causing the Re-Order Buffer (ROB) to fill, eventually halting execution. Equally important is the core's ability to insert instructions into the execution window (ingress), e.g., due to a misspeculation-induced pipeline flush disrupting the core's ability to supply μops to its execution units. To address ingress issues, developers must identify the static instructions that incur high dispatch latency, i.e., they need time-proportional PICS at dispatch ($PICS_D$). Fundamentally, the out-of-order execution window decouples μop ingress from egress, which often hides ingress issues when looking at egress, and vice versa. $PICS_D$ and $PICS_C$ are hence complementary because they, in combination, identify the specific instructions that cause ingress and egress issues.

Although state-of-the-art DT profilers AMD IBS [19], Arm SPE [5], and IBM RIS [33] tag μops at dispatch, they cannot generate PICS that are time-proportional at dispatch ($PICS_D$). This is because they (i) do not account for individual dispatch slots, and (ii) invalidate sampling units that hit misspeculated μops. We hence propose dispatch-time profiling (DIP) which, unlike DT, is time-proportional at dispatch, i.e., it attributes dispatch time to the μop(s) that the core is exposing the dispatch latency of in the cycle the sampling unit is taken. DIP achieves this by observing that each dispatch slot can be in four fundamental states: *Base*, *Stall*, *Front-End*, and *Misspeculation*. *Base* covers the unavoidable dispatch time of any correct-path μop, *Stall* accounts for waiting on back-end resources, *Front-End* accounts for front-end events disrupting μop supply, and *Misspeculation* is the dispatch time cost of mispredicted branches, exceptions, and serializing μops. DIP records the architectural state of the core at the time of sampling, e.g., stalled on a full ROB or misspeculating. This enables breaking down $PICS_D$ across the situations that caused dispatch latency, thereby explaining the architectural cause(s) of dispatch time. DIP's hardware is simple, requiring only 49 bytes of additional storage, and sampling keeps performance overhead low (~1%). DIP's capabilities are compared to other profilers in Table 1.

We implement DT and DIP within the BOOM [67] processor core and profile 22 benchmarks from SPEC CPU2017 [53]. We compare DT and DIP to a time-proportional at dispatch golden reference that accounts for all μops, dispatch slots, and architectural causes. This comparison is done using *profile error* [26], a metric for quantifying the amount of correctly attributed cycles for statistical sampling profilers (see Section 4 for details). We find that DT is generally inaccurate with an average profile error of 26.9% (up to 55.1%). In contrast, DIP is accurate and yields an average profile error of 5.2% (up to 20.9%). We observe that DT's policy of invalidating sampling units that hit misspeculated μops can be

improved through trivial extensions to modern cores, yielding 'DT with Misspeculation' (DT-M). DT-M leverages the Last Branch Record (LBR) available in, for example, Intel Raptor Lake [34] or AMD Zen 4 [2] to attribute the dispatch time cost of mispredictions to the responsible taken branch instruction; DT-M also accounts for misspeculation due to not taken mispredicted branches, exceptions, and serialization (see Section 3.3). Although DT-M reduces profile error compared to DT (average of 16.8% and up to 34.6%), it does not account for dispatch parallelism and is therefore not time-proportional at dispatch; this explains why DT-M's profile error is (much) higher than DIP's profile error.

We use DIP's time-proportional at dispatch $PICS_D$ and TEA's time-proportional at commit $PICS_C$ [27] to study the prevalence of ingress and egress issues across the SPEC CPU2017 benchmarks. Interestingly, we find that the vast majority, i.e., 18 out of 22 benchmarks, exhibit both ingress and egress issues simultaneously, demonstrating that it is critical to consider both $PICS_D$ and $PICS_C$ during software and hardware optimization. To further demonstrate the complementarity of $PICS_D$ and $PICS_C$, we use them to analyze the SPEC CPU2017 benchmark fotonik3d. $PICS_D$ identifies an ingress issue caused by a load instruction that experiences frequent memory ordering exceptions, while $PICS_C$ shows an egress issue due to two other loads that frequently miss in the Last Level Cache (LLC). We observe that the ingress issue can be removed by avoiding a register spill, which, once implemented, improves the performance of fotonik3d by 8.5%.

In summary, we make the following key contributions:

- We propose dispatch-time profiling (DIP), which is time-proportional at dispatch, and provides accurate $PICS_D$ (5.2% profile error), at negligible storage (49 bytes) and performance (~1%) overheads; a significant gain over the 29.6% profile error of state-of-the-art DT (i.e., IBS, SPE, and RIS).
- We are the first to demonstrate PICS at both dispatch ($PICS_D$) and commit ($PICS_C$) are needed for comprehensive instruction-level performance profiling.
- Our analysis of SPEC CPU2017 shows that 18 of 22 benchmarks exhibit simultaneous ingress and egress issues. Needing $PICS_D$ *and* $PICS_C$ is hence common.
- We use $PICS_D$ and $PICS_C$ to identify and address a memory ordering exception in fotonik3d, improving performance by 8.5%.

2 Complementarity of $PICS_D$ and $PICS_C$

Ingress and egress performance issues. We now explain why performance issues manifest at both execution window ingress and egress. This motivates why both $PICS_D$ and $PICS_C$ are needed to identify favorable optimization strategies. Figure 1 considers a simplified view of an out-of-order processor pipeline. The front-end fetches instructions from

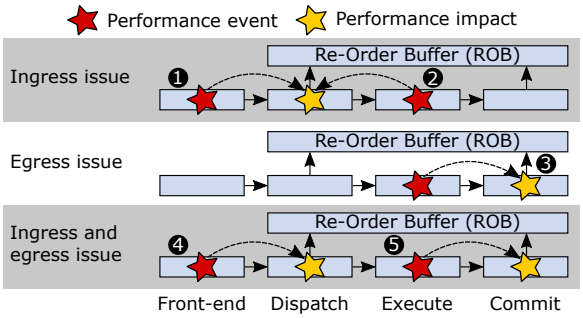


Figure 1. Types of performance issues. *Understanding how µops ingress into and egress out of the out-of-order execution window is key to optimizing performance.*

memory, which are decoded and mapped to µop(s). Then, µops are dispatched to the Re-Order Buffer (ROB) and to the issue buffers or queues of their respective execution unit in program order, thereby entering the out-of-order execution window and the core's back-end. As µops complete, the ROB ensures that µops commit in program order to support precise exceptions.

Figure 1 shows that execution window ingress issues occur when performance events disrupt the supply of µops into the execution window. One example is an instruction cache miss, see ❶, which stalls the front-end and empties the dispatch stage, thereby halting the ingress of µops into the execution window. Another example is a mispredicted branch which first causes the core to (potentially) fetch, decode, dispatch, and execute misspeculated µops. The branch mispredict is eventually detected when the branch µop executes, and the misspeculated µops are then flushed from the pipeline, see ❷. Misspeculation thus disrupts execution window ingress by first supplying µops that should not have been there, before emptying dispatch while waiting for the front-end to supply correct-path µops. Performance issues also occur because µops cannot egress from the execution window. This typically occurs when the execution latency of a µop is sufficiently high for it to reach the head of the ROB before it has fully executed, e.g., a load µop missing in the LLC, causing commit to stall and blocking egress ❸.

The common case, however, is that an application suffers from both ingress ❹ and egress ❺ issues. The out-of-order execution window, i.e., the window of µops that are available in the ROB and are waiting for operands, executing, or have completed execution and are waiting for older µop(s) to execute or commit, effectively decouples ingress from egress. Looking at commit will hence typically point to instructions that are hot due to egress issues while hiding ingress issues. This is because commit is the stage where µops egress from the execution window. Conversely, looking at dispatch will typically highlight instructions that are hot due to ingress issues while hiding egress issues. This

happens because dispatch is where μops ingress into the execution window. Hence, considering only dispatch ($PICS_D$) or commit ($PICS_C$) provides a one-sided view of performance. Instead, developers should obtain both $PICS_D$ and $PICS_C$ to fully understand how ingress and egress issues affect application performance. Ingress and egress issues can be aligned, i.e., provide two complementary perspectives on the same underlying performance issue, or unaligned, i.e., they occur in different code regions and therefore provide separate avenues for performance improvement. We analyze the prevalence of ingress versus egress issues experimentally in Section 6.

Analysis example. We illustrate how $PICS_D$ and $PICS_C$ collectively provide insight into ingress and egress issues with an example in Figure 2. Figure 2a shows the performance-critical loop of our PSum compute kernel which computes a partial sum of a large array (i.e., `data`).[3] The array elements to include in the partial sum are given by the `use` array, i.e., element i is included in the partial sum when `use[i]` equals one. Figure 2b shows an excerpt of PSum's assembly code, while Figures 2c and 2d report the $PICS_D$ and the $PICS_C$ of its three hottest instructions, respectively, after executing PSum on BOOM [67] within FireSim [37]. (Section 4 describes our experimental setup in detail.) We use DIP (see Section 3.2) to obtain $PICS_D$, whereas $PICS_C$ are provided by TEA [27].

The hottest instruction in the $PICS_D$ (Figure 2c) is `beqz|88`, a difficult-to-predict branch that determines if an element should be added to the partial sum; we see this because the $PICS_D$ is dominated by the 'Misspeculation' component. (We identify instructions by their type and the two least significant hexadecimal digits of their address.) In contrast, the hottest instruction in the $PICS_C$ (Figure 2d) is a load instruction that fetches `data` elements from memory (i.e., `lw|8a`). This instruction is hot because it often misses in the LLC, as shown by the stalled-by-LLC (ST-LLC) event dominating the $PICS_C$. ST-L1 refers to stalled-by-L1-data events PSum hence suffers from both an ingress and an egress issue. A developer using state-of-the-art TEA would only see `lw|8a` and hence be unaware of the opportunity to optimize `beqz|88`.

Optimization example. By analyzing the $PICS_D$ and $PICS_C$, the developer understands that PSum can be optimized by (i) avoiding the difficult-to-predict branch, or (ii) improving the locality of the `data` array. Figure 2e shows the OPSum kernel, which is the result of applying the first strategy, i.e., avoiding the branch at the cost of using more memory by precomputing the addresses of the elements included in the partial sum. This optimization exploits that logical false in C equals 0, and the address computation loop hence keeps overwriting the current `indices` value until it contains a valid address. Ultimately, OPSum provides exactly the same functionality as PSum while improving performance by 27.3%. This is supported by its $PICS_D$ (Figure 2g) and $PICS_C$ (Figure 2h) which show that the number of clock cycles attributed to the hottest instructions is significantly lower than for PSum (Figures 2c and 2d). Moreover, the long-latency load `lw|b6` that accesses the `data` array is now the hottest instruction at both dispatch (Figure 2g) and commit (Figure 2h), but its execution time impact is lower, because removing the ingress issue allows more `lw|b6` μops to enter the execution window at the same time — hiding memory latencies via memory-level parallelism. Notice that PSum's ingress issue (the misspeculation) is not present in the $PICS_C$ (generated by TEA) in Figure 2d. This demonstrates that DIP uncovers performance issues that state-of-the-art approaches such as TEA [27] miss.

[3] We obtained PSum by combining the hard-to-predict branch and data cache miss microbenchmarks provided by the Vertical Research Group [55].

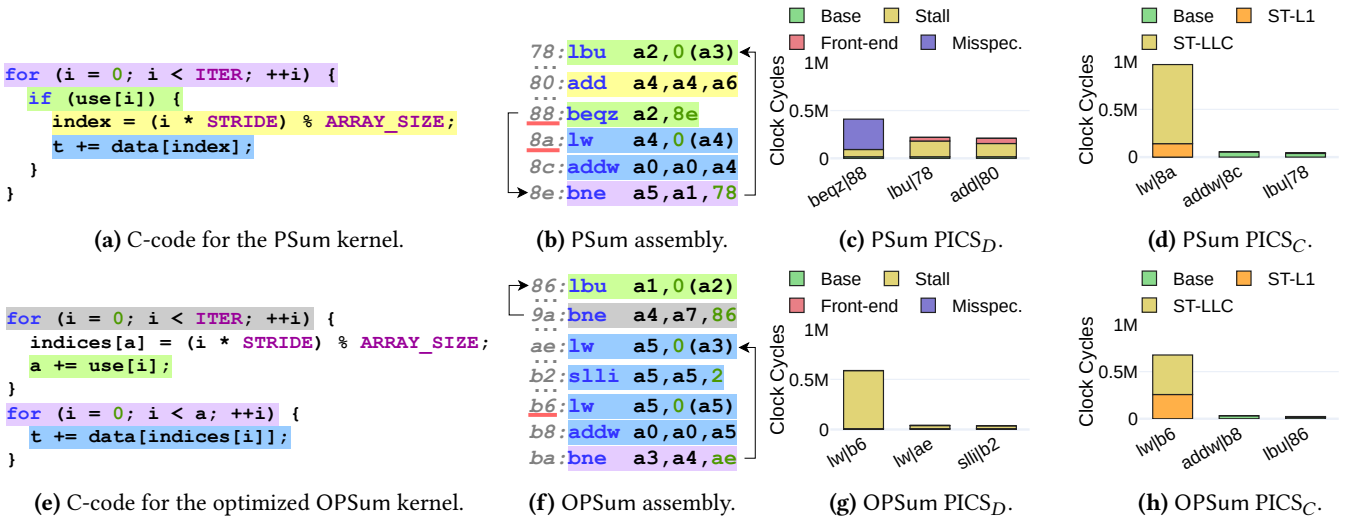
3 Per-Instruction Cycle Stacks at Dispatch

We now turn our attention to designing dispatch-time profiling (DIP), thereby providing the hardware support needed to create accurate $PICS_D$ on out-of-order cores. DIP is accurate because it, by design, creates time-proportional at dispatch performance profiles. We achieve this by generalizing the time-proportionality principle [26], originally defined in the context of execution time. More specifically, it states that if a profiler attributes each sampling unit to the instruction(s) of which the core is exposing the execution time in the cycle the sampling unit is taken, the instruction's weight in the profile will be proportional to its contribution to overall execution time. The time-proportionality principle however applies to any notion of time. Thus, creating a time-proportional at dispatch performance profile simply amounts to attributing a sampling unit to the μop(s) of which the core is (are) exposing the dispatch latency in the cycle the sampling unit is taken.

3.1 Golden Reference $PICS_D$

The first step towards achieving time-proportional $PICS_D$ is to define a golden reference. Specifically, we devise a set of attribution policies that identify the μop(s) that the core is exposing the latency of that collectively cover all possible situations that can occur at dispatch. In this way, we obtain golden reference $PICS_D$. These account for all μops, all dispatch slots, all clock cycles, and all architectural causes that we can then use to quantify the error of the $PICS_D$ obtained with sampling. This approach is in line with previous work [26, 27]. Since the golden reference relies on specific microarchitectural details, it is specific to a microarchitecture. We therefore design it for the BOOM core [67] we use for our evaluation.

Fundamental dispatch states. A dispatch slot can be in one of only four fundamental states as shown in Figure 3: It can be (i) dispatching a correct-path μop into the execution window which we refer to as *Base*; (ii) on the correct-path and *Stall* because the back-end cannot accept its μop due



(a) C-code for the PSum kernel. **(b)** PSum assembly. **(c)** PSum $PICS_D$. **(d)** PSum $PICS_C$.

(e) C-code for the optimized OPSum kernel. **(f)** OPSum assembly. **(g)** OPSum $PICS_D$. **(h)** OPSum $PICS_C$.

Figure 2. Analysis and optimization example. *The* PSum *$PICS_D$ identify a difficult-to-predict branch (ingress issue), whereas $PICS_C$ only show a long-latency load (egress issue). We eliminate the branch in* OPSum *to improve execution time by 27.3%.*

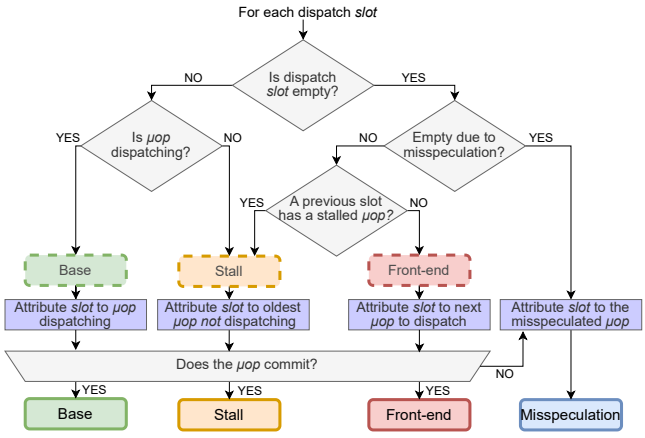


Figure 3. Time-proportional at dispatch attribution policy.

to a full buffer or queue; (iii) on the correct-path but empty because a *Front-End* event (e.g., instruction cache miss) disrupted μop supply; or (iv) dispatching a misspeculated μop or empty due to *Misspeculation*. A μop is considered speculative until it commits. For instance, it could be immediately following a late-resolving mispredicted branch in program order. Figure 3 hence shows that slots attributed to *Base*, *Stall*, and *Front-End* are tentative until the μop commits. The architectural reason for dispatch latency follows directly from the attribution policy. If the slot is stalled, we need to record the reason(s) for the stall (e.g., the ROB is full). If the slot is empty due to a front-end event, the cause is the front-end event, and if the slot contains a misspeculated μop or is empty due to misspeculation, the architectural cause of dispatch latency is the type of misspeculation (e.g., an exception).

We now discuss the attribution policies for the fundamental dispatch states. Since the golden reference accounts for all dispatch slots across all cycles, the dispatch-time contribution of a single dispatch slot is always $1/w$ for a core that can dispatch w μops per cycle. The accounting policies thus need to identify the dispatch slots that should be attributed to a specific μop across all dispatch states, as these can then be counted and converted to cycles (time) by dividing by w.

State 1: Base. This state covers the situation in which a correct-path μop dispatches in a cycle. Each μop dispatches once, and all correct-path μops thus have exactly one dispatch slot in this state. Notice however that a static instruction can be executed many times (e.g., loops), and the *Base* slots may thus add up. We call this state 'Base' because (i) it is unavoidable and (ii) the core dispatches both correct-path and misspeculated μops; referring to this state as dispatching would hence not be fully precise.

State 2: Stall. A *Stall* occurs when a correct-path μop cannot dispatch in this cycle, typically due to some (combination of) full buffer(s) or queue(s). In the BOOM, the specific reasons for dispatch stalls are a full ROB, a full issue buffer, a full load queue, a full store queue, a μop requiring serialization, and register rename stalling due to running out of physical registers. (The BOOM renames registers in parallel with decoding the μop, and failed renaming thus results in the μop stalling at dispatch.) Since μops are dispatched in program order, a stall on a μop μ_i, where the subscript i signifies program order, will also cause μops μ_{i+1}, μ_{i+2}, ... to stall. The full buffer(s) or queue(s) that cause μ_i to stall are the architectural reason for dispatch stalls, and the core is thus exposing the dispatch latency of μ_i in this cycle. The golden reference thus attributes the dispatch slots of μ_i, the slots used by younger μops (e.g., μ_{i+1}), and any empty slots to μ_i.

Earlier μops (e.g., μ_{i-1}) may dispatch in the same cycle as μ_i is stalled. This typically happens in the first cycle when dispatch stalls, i.e., the last dispatchable μop(s) leave(s) dispatch while the stalling μop stalls itself and any later μop(s). The golden reference correctly handles this situation because it attributes time to μops at the level of individual dispatch slots. Consider a core that can dispatch four μops per cycle, and a cycle in which it contains μops μ_{i-1}, μ_i, μ_{i+1} and μ_{i+2}; μ_{i-1} is dispatching while μ_i stalls on a full load queue. The golden reference thus attributes one *Base* slot to μ_{i-1} and three *Stall* slots to μ_i, i.e., the slots occupied by μ_i, μ_{i+1} and μ_{i+2} in this cycle, amounting to 1/4 and 3/4 cycles, respectively.

State 3: Front-End. Dispatch can also be empty due to a disruption in μop supply from the front-end while on the correct execution path. This causes empty dispatch slots, and since μops enter dispatch in program order, the next μop to enter is responsible for the lost dispatch time. Architectural causes are determined by recording the performance events a μop was subjected to (e.g., an instruction cache or TLB miss) and keeping track of them until the μop reaches dispatch. Empty dispatch slots can also occur when the first μop to dispatch is not subjected to any performance events. In this case, the golden reference classifies the empty slot(s) as a front-end bubble, which typically occurs in the BOOM when fetch is redirected on a correctly predicted branch. In modern architectures, other front-end bottlenecks can cause a μop supply issues. Macro-op fusion and fission [52], μop cache miss [69], loop buffer miss [17, 35], and pre-decoding and decoding of variable-length instructions [48] are some examples of mechanisms that could cause front-end bubbles. Again, we call this state *Front-End* rather than empty because calling it empty would not be fully precise because dispatch can also be empty while dispatching misspeculated μops.

State 4: Misspeculation. The final dispatch state occurs when a dispatch slot is either (i) on a misspeculated path, or (ii) dispatch is empty because the pipeline was flushed to squash misspeculated μops and the front-end has not yet been able to provide correct-path μops. While misspeculation can be caused by mispredicted conditional branches, exceptions, or serialization in the BOOM, their impact on dispatch time is the same. Specifically, dispatch time exposed by all dispatch slots that were either dispatching misspeculated μops or empty during misspeculation is always attributed to the μop that caused the misspeculation. Since μops must execute to detect misspeculation, they have necessarily already dispatched when this happens. This is straightforward to deal with in the golden reference because it sees all dispatch slots in all cycles, and it attributes all occupied and empty dispatch slots between the last correct-path μop μ_l and the first correct-path μop to dispatch after the pipeline flush μ_f to the μop that caused the flush, i.e., the responsible μop μ_r. For mispredicted branches, the responsible μop is the branch itself, and for exceptions (serialization) μ_r is the excepting (serialized) μop. In the BOOM, serialized μops (e.g., fences and atomic instructions) stall at dispatch until the ROB is empty and flush the pipeline when they commit.

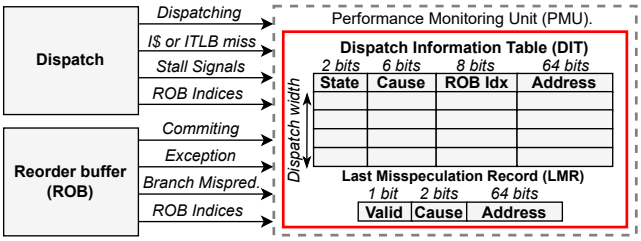


Figure 4. Overview of the DIP hardware architecture.

3.2 DIP: Dispatch-Time Profiling

Having outlined how to create golden reference PICS$_D$, we now describe how dispatch-time profiling (DIP) obtains time-proportional at dispatch PICS$_D$ with low overhead. While the golden reference creates exact PICS$_D$, it does so by accounting for all μops and all dispatch slots across all cycles. Thus, it cannot be implemented in real cores without debilitating overhead. DIP, in contrast, leverages statistical sampling to generate approximate PICS$_D$ while keeping overheads in check. Statistical sampling requires looking specifically at the cycle in which the sampling unit is taken, because considering earlier or later cycles will introduce systematic errors (i.e., bias) [26–28]. DIP thus realizes the hardware support required to implement the attribution policies of the golden reference shown in Figure 3 from the perspective of a performance sampling unit taken in a specific cycle.

DIP architecture. Figure 4 provides an overview of the DIP architecture; we expect that DIP would be integrated within the core's Performance Monitoring Unit (PMU). Key to DIP is the Dispatch Information Table (DIT), which tracks the required dispatch information until all slots have been fully classified according to the time-proportional at dispatch attribution policies in Figure 3. The DIT contains one entry for each dispatch slot, consisting of the *State*, *Cause*, *ROB Index*, and *Address* fields. *State* records the fundamental dispatch state, and *Cause* records the architectural reason for a slot exposing dispatch latency. Cause bits are multi-hot encoded to support all combinations of causes, and interpreted differently depending on the state to reduce storage overhead; Table 2 enumerates all possible causes. *Address* records the address of the static instruction that the slot is attributed to. The static instruction address follows each μop through the pipeline and is hence available at dispatch (and in the ROB).

The *ROB Index* is used to determine if a slot dispatched a correct-path or misspeculated μop as it enables tracking each sampled μop until it is committed or squashed. Upon detecting a pipeline flush affecting the sampled μops, DIP determines which of the slot(s) are affected by misspeculation

Table 2. DIT causes. *The M, S, and F prefixes represent the Misspeculation, Stall, and Front-End states, respectively.*

Name	Description
M-SWE	Software-handled exception.
M-FL	Pipeline flush; misprediction or hardware-handled exception.
S-ISB	Dispatch stall due to full issue buffer.
S-ROB	Dispatch stall due to full reorder buffer.
S-LDQ	Dispatch stall due to full load queue.
S-STQ	Dispatch stall due to full store queue.
S-REN	Dispatch stall due to register rename not ready.
S-SER	Serializing instruction; requires empty pipeline to execute.
F-BUB	Bubble, e.g., fetch redirect on correctly predicted branch.
F-CTM	L1 instruction cache or instruction TLB miss.

and updates their DIT entries accordingly. More specifically, this means changing *State* to *Misspeculation*, setting *Address* to reflect the responsible μop, and updating *Cause*. Additionally, DIP updates the Last Misspeculation Record (LMR) on every flush and invalidates it when the first correct-path μop enters dispatch to detect if a pipeline refill is due to a flush or a front-end issue. Table 2 shows that DIP records if misspeculation is due to a software-handled exception (M-SWE) or a hardware-handled pipeline flush (M-FL). This is sufficient because the exact cause can typically be inferred from the instruction type, e.g., a conditional branch with the M-FL bit set was misspeculated. Some instructions (e.g., loads) can trigger exceptions that must be handled in software (e.g., page faults) as well as exceptions that are handled in hardware (e.g., memory ordering exceptions). DIP records M-SWE and M-FL to distinguish between these situations.

Software interface and runtime overhead. Once all sampled μops have either been committed or flushed, the sampling unit is ready to be retrieved by sampling software. To this end, DIP exposes the DIT as a read-only Control and Status Register (CSR), and getting the data out amounts to interrupting the core to invoke the software sampling routine which then reads the CSR. For example, Linux `perf` [42] typically samples at a 4 kHz frequency. When sampling, `perf` copies the CSR contents to a memory buffer, spilling the buffer to non-volatile storage when the buffer becomes full or the application terminates. The raw sample data is thus available in a file once the application terminates. We have written a tool that post-processes this file to create and visualize PICS_D.

The runtime overhead of DIP is hence due to profiling software interrupting the application to retrieve performance sampling units. All sampling-based profilers, e.g., DIP, TEA, and DT, incur overhead from context switching to the software profiler and from transferring the CSR data. When using `perf` to retrieve sampling units, 40 B of metadata is always transferred, corresponding to core, process, and thread identifiers, in addition to the profiler-specific CSR data. TIP [26] reports an overall overhead of 1.1% when retrieving CSR data at a rate of 352 KB/s, slightly higher than the 1.0% overhead of PEBS [34] at 224 KB/s. Overhead is hence dominated by context switching, since CSR data rate increases substantially from PEBS to TIP, while performance overhead changes marginally. DIP for BOOM generates data at a rate of 320 KB/s, and, by interpolating between TIP and PEBS, we obtain a 1.08% (or ~1%) performance overhead for DIP.

DIP hardware overhead. The key hardware overhead of DIP is the storage overhead of the DIT and the LMR. Our BOOM core can dispatch up to 4 μops per cycle; see Section 4 for details. For each dispatch slot, the DIT uses 2 bits to track the dispatch *State* because there are four fundamental states. As *Cause* bits are multi-hot encoded, the DIT needs 6 bits because there are 6 possible causes for the *Stall* state. Our BOOM has a 192-entry ROB, requiring 8 bits for each *ROB Index*, and the *Address* field is 64 bits to store the full (virtual) instruction address. The LMR collectively uses 67 bits for its *Valid*, *Cause*, and *Address* fields. Finally, we pass one bit from fetch until dispatch, incurring a single bit of overhead in fetch and decode (recall that rename happens in parallel with decode in the BOOM). This bit tracks if an I-cache or I-TLB miss has occurred. The total hardware overhead is 389 bits, or less than 49 bytes.

DIP's hardware is off the critical path and therefore does not impact clock frequency. It does however need to track the dispatch and commit stages for a sufficiently long time to collect each sampling unit. Specifically, DIP needs to be enabled early enough for the LMR to be updated in the cycle the sampling unit is collected, and remain active until DIP knows if the μop(s) in the sampling unit are misspeculated or not (i.e., track them until they commit). We conservatively assume that this requires DIP to be active 2,000 cycles every 800,000 clock cycles; a sampling unit is collected every 800,000 clock cycles with a 4 kHz sampling frequency when the clock frequency is 3.2 GHz. We use Cadence Genus [12] to synthesize DIP in a commercially available 28 nm technology, yielding an area overhead of 0.065 mm^2 which is 1.7% of the BOOM's ROB. (We target a 1 GHz clock frequency for the BOOM's ROB to meet timing at our technology node.) We then use Cadence Joules [13] to estimate DIP's static and dynamic power consumption to be 5.6 mW and 1.9 mW, respectively. If DIP is clock-gated when inactive, its overall power consumption is only decimal points higher than its static power consumption, i.e., also 5.6 mW, which is 3.0% of the power consumption of BOOM's ROB and only 0.1% of the 4.7 W per-core power of the Intel i7-1260P [27].

Scalability. DIP scales to higher-performance cores as well, i.e., it handles larger structures (e.g., ROB size and dispatch width) and optimizations (e.g., μop fusion, speculative wakeup, and instruction replay). The size of the ROB only affects the number of bits needed to store the ROB index in the DIT. The size of the DIT itself scales linearly with dispatch width, and

therefore affects the amount of data to read for each sampling unit (remember that DIP's performance overhead is mostly connected to context switch and not data bandwidth). If the core implements μop fusion, DIP's per-instruction policy will attribute cycles to the instruction address associated with the fused μops, i.e., all of the latency is attributed to one of the μops while the latency of the other μop(s) is hidden. Speculative wakeup and instruction replay occur after dispatch and hence do not impact DIP's attribution policies, but can affect performance characteristics. Consider an optimized core that handles memory ordering exceptions by replaying instructions rather than BOOM's approach of flushing the pipeline. The optimized core would then expose the cost of memory ordering exceptions as (potential) dispatch stalls, whereas the BOOM exposes them as misspeculation.

DIP can also accommodate different causes of dispatch latency, requiring changing the number of cause bits in each DIT entry. Note that causes can be combined or divided to trade storage overhead against explainability. Changes to the causes affect the component breakdown of the PICS, but do not change DIP's attribution policy (Figure 3), as the policy is based on the fundamental dispatch states, i.e., *Base*, *Stall*, *Front-end*, and *Misspeculation*, see Section 3.1.

3.3 Dispatch Tagging (DT)

We now explain why existing Dispatch Tagging (DT) profilers, i.e., AMD IBS [19], ARM SPE [5], and IBM RIS [33], are not time-proportional at dispatch; it is already well-known that DT is not time-proportional at commit [26, 27]. Under DT, a μop is tagged at dispatch, and the performance events it experiences are recorded as it continues through the out-of-order back-end and are retrieved when it commits. This approach is not time-proportional at dispatch for two reasons. The first is that DT, unlike DIP, tags a single μop when sampling. This is not time-proportional (e.g., because several μops can dispatch in the same cycle or some μops can dispatch while others are stalled). In these cases, DT attributes the sampling unit to one of these μops, but this is inaccurate because they all expose dispatch latency in the cycle the sampling unit is taken.

The second reason is that IBS, SPE, and RIS invalidate sampling units that tag a misspeculated μop. While this avoids attributing time and performance events to μops that should never have been executed, it is not time-proportional because it does not attribute the sampling unit to the μop responsible for dispatch latency, i.e., the μop that caused the pipeline flush. However, modern CPUs such as Intel Raptor Lake [34] and AMD Zen 4 [2] track activity that we can use to partially address this issue. The main reason for misspeculation is mispredicted branches, and the addresses of the most recent taken branch instructions are tracked by the Last Branch Record (LBR). In the case of software-handled exceptions, instruction addresses must be communicated to system software such that it can (potentially) re-execute after handling the exception. Flushes due to non-taken mispredicted branches, hardware-handled exceptions, and serializing instructions require a trivial hardware extension, i.e., a CSR for the instruction address of the responsible μop.

We hence create 'Dispatch-Tagging with Misspeculation' (DT-M), which modifies DT to attribute sampling units that hit misspeculated μops to the instruction that caused the flush for exceptions and serializing μops and the most recent entry in the LBR otherwise. DT-M is hence representative of the accuracy that could be achieved through trivial extensions to IBS, SPE, and RIS. (We compare DT-M quantitatively to DT and DIP in Section 5.) Unlike DIP, DT-M cannot differentiate between dispatch being empty due to a flush and a front-end stall, because sampling units taken during the pipeline refill following a flush will hit correct-path μops. Moreover, DT-M only records performance events for the tagged μop. During misspeculation however, DT-M attributes the sampling unit to the μop that caused the flush. This means that it cannot report any performance events for the flush-causing μop because it recorded the events of the misspeculated μop.

Table 3. Baseline architecture configuration.

Part	Configuration
Core	OoO BOOM [67]: 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

4 Experimental Setup

Simulation. Results are gathered using FireSim [37], a cycle-accurate FPGA-accelerated full-system simulator. The simulated CPU is a BOOM 4-wide out-of-order core [67] running a common buildroot 5.7.0 Linux kernel; Table 3 provides further details. We run FireSim on AMD Alveo U250 FPGAs on an internal cluster. The frequency difference between the FPGA realization of the BOOM core and the FPGA's memory system is accounted for using FireSim's token mechanism.

We use TraceDoctor [25] to acquire cycle-accurate traces from the FPGA's simulated BOOM core. As simulation occurs, trace data is sent from the FPGA to the host CPUs, which run TraceDoctor workers. Each worker runs on its own CPU thread and implements a specific profiler configuration considered in this work. In this way, we (i) ensure that comparisons are fair because all profilers sample in the exact same cycle, and (ii) substantially reduce evaluation time. When simulation completes, we create PICS_D from the raw sampling data as described in Section 3.2. Unless otherwise mentioned, we profile at a sampling frequency of 4 kHz because this is the default in Linux `perf` [42].

Benchmarks. We simulate all but one of the SPEC CPU2017 benchmarks [53]; the blender benchmark cannot run due to a technical infrastructure limitation. We configure FireSim to begin trace collection once the workload starts executing, thereby excluding the operating system's boot-up phase. We execute the first 150 billion cycles for each benchmark to keep simulation turnaround times reasonable. This amounts to an overall instruction count ranging from 16.2 billion (omnetpp) to 116.6 billion (imagick). We only report sampling units that hit benchmark code and (statically linked) libraries to ensure that OS activities do not interfere with the profiles, resulting in removing 0.95% of the sampling units on average.

Golden reference. We create golden reference PICS_D as described in Section 3.1; this is an established practice in performance profiling research [26, 27]. While the reference provides ground truth PICS_D, it cannot be implemented in real hardware, e.g., it would generate profiling data at a rate of 256 GB/s if implemented in the BOOM core. Moreover, the reference profiles would generate 12 TB of raw sampling data per benchmark. To avoid this, TraceDoctor workers on the host process reference data on the fly, resulting in file sizes ranging from 13 MB for lbm to 78 MB for wrf.

Profile error. We use *profile* error [26] to quantify profiler accuracy. We denote the number of (partial) cycles attributed to the PICS_D of a given instruction as C_i, where $1 \leq i \leq G$, and G (granularity) equals the total number of static instructions. The number of cycles attributed to an instruction i in the golden reference PICS_D is C_i^R. We then calculate the total number of correctly attributed cycles across the whole application, $T_{correct}$, using the following summation: $T_{\text{correct}} = \sum_{i=1}^{G} \min(C_i, C_i^R)$. Once the correctly attributed cycles are computed across all PICS_D, profile error is found by taking the relative difference between the total cycle count and the correct cycle count, i.e., $E = (T_{\text{total}} - T_{\text{correct}})/T_{\text{total}}$.

5 Accuracy

This section evaluates the accuracy of the following profilers:

- **Dispatch Tagging (DT).** A profiler representative of AMD IBS [19], Arm SPE [5], and IBM RIS [33].
- **DT-M.** An improvement of DT which accounts for misspeculation (see Section 3.3).
- **DIP.** Our profiler as presented in Section 3.2.

Profile error. Figure 5 reports profile error for DT, DT-M, and DIP across all benchmarks. We order the benchmarks according to the number of uniquely sampled instructions from low (left) to high (right). The benchmarks that are least (most) susceptible to statistical error are hence placed to the left (right). The average profile errors of DT (26.9%) and DT-M (16.8%) are higher than DIP's (5.2%). The reason is that DIP is time-proportional at dispatch, while DT and DT-M are not (see Section 3.3). Although DT-M performs better than DT because it handles misspeculation, it does not account for dispatch parallelism, hence its residual error. Since DIP's attribution policies are time-proportional at dispatch, error is solely due to statistical (random) error, and therefore generally increases from left to right. Without bias, the likelihood of sampling an instruction is proportional to how hot the instruction is. Therefore, the profile of a benchmark with a few (very) hot instructions will converge faster than a benchmark with many less-hot instructions. This explains DIP's worst-case profile error of 20.9% for gcc where its sampling units hit 63,133 unique instructions. The other extreme is occupied by lbm, where DIP incurs an overall profile error of only 2.6% because it hits 654 unique instructions. (We will explore the impact of sampling frequency shortly.)

Qualitative PICS_D Analysis. Figures 6 and 7 present the PICS_D for gcc and deepsjeng, respectively, while internal subfigures (a–d) show the PICS_D of the 5 hottest instructions for DT, DT-M, DIP, and the golden reference. We show gcc because it is the benchmark where DIP has the highest profile error, and deepsjeng because it illustrates how recording performance events versus architectural causes affect the PICS_D. More specifically, the PICS_D of DT and DT-M are broken down across event signatures, i.e., the (combinations of) performance event(s) that the μops belonging to the hottest instructions were subjected to, whereas the PICS_D breakdown of the DIP and the golden reference is based on (combinations of) architectural causes (see Table 2). A few signatures or causes typically dominate a set of PICS_D. Therefore, to improve readability, we report only signatures and causes that account for at least 10% of a shown PICS_D. The remaining causes and signatures are grouped into the 'Other' component.

By comparing DT's PICS_D in Figures 6a and 7a to the golden reference PICS_D in Figures 6d and 7d, respectively, we see that DT's PICS_D are a poor match for the golden reference. For gcc, DT fails to report 3 out of the 5 hottest instructions (i.e., `bne|ac`, `jr|22`, and `beqz|02`), while for deepsjeng, it misses two instructions (`beq|46` and `bnez|0e`). The reason is that DT invalidates sampling units that hit misspeculated μops. Figures 6a and 7a further confirm the key takeaway from Figure 5, i.e., DT struggles with correctly

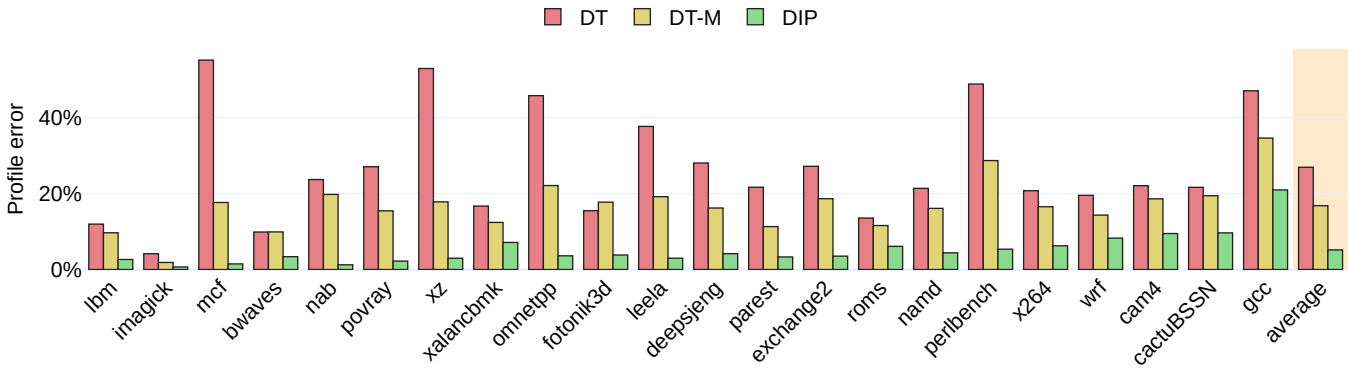


Figure 5. Profile error at a 4 kHz sampling frequency. *DIP has substantially lower average error (5.2%) than DT (26.9%) and DT-M (16.8%) because it is time-proportional at dispatch.*

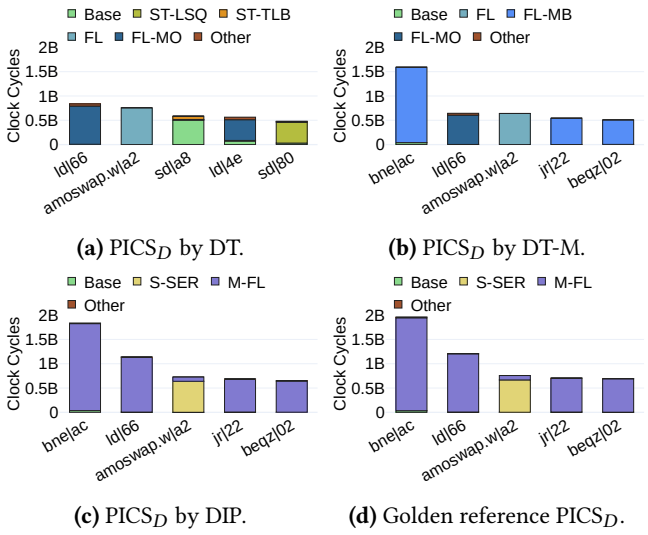


Figure 6. PICS$_D$ for the 5 hottest instructions for gcc. *DIP matches the golden reference even if its profile error is 20.9%.*

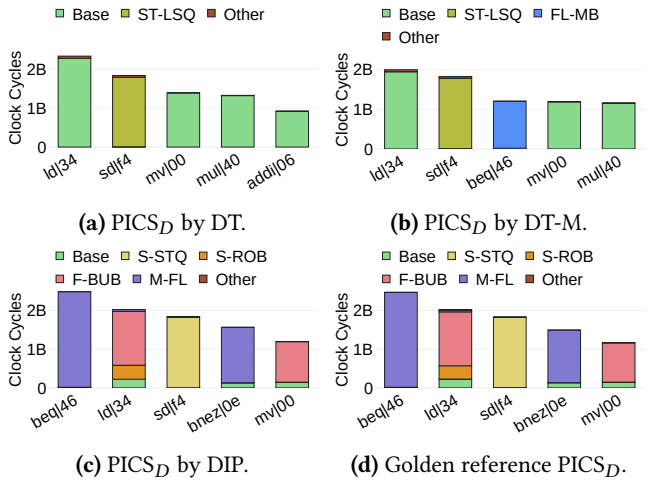


Figure 7. PICS$_D$ for the 5 hottest instructions for deepsjeng. *DT-M falls short as it misclassifies sampling units taken during refills.*

attributing dispatch time to instructions. For example, it underestimates the dispatch latency of `ld|66` for gcc by 31%. DT's PICS$_D$ are hence difficult to interpret because they (i) miss control flow instructions that require direction or target prediction, and (ii) report misleading dispatch time.

DT-M improves upon DT by accounting for misspeculation. Figure 6b shows that DT-M's gcc PICS$_D$ contain the same hot instructions as the golden reference (Figure 6d). However, DT-M lacks time-proportionality. Therefore, the time attributed to individual instructions deviates by up to 47% (`ld|66`). DT-M's deepsjeng PICS$_D$ (Figure 7b) reports different hot instructions than the golden reference (Figure 7d), i.e., `bnez|0e` is missing and `mul|40` is included. Moreover, DT-M mistakenly reports `beq|46` as deepsjeng's third hottest instruction while it is in fact its hottest. The reason is that DT-M mistakenly attributes sampling units taken when the pipeline is refilled after a flush to the next dispatching instruction, instead of attributing the sampling unit to the instruction that caused the flush (i.e., `beq|46` and `bnez|0e`).

Figures 6c and 7c show that DIP's PICS$_D$ for gcc and deepsjeng, respectively, are practically identical to the golden reference PICS$_D$ (Figures 6d and 7d). This is perhaps surprising since DIP incurs its highest profile error for gcc. DIP is however time-proportional at dispatch and therefore only susceptible to statistical (random) error, which goes down as more sampling units hit the same instruction. The hottest instructions are hot because they are sampled many times. Thus, their PICS$_D$ are less affected by statistical error. Notice that this effect does not apply to DT and DT-M because they additionally suffer from systematic error (bias).

Comparing the PICS$_D$ of DT and DT-M to the PICS$_D$ provided by DIP and the golden reference also illustrates why it

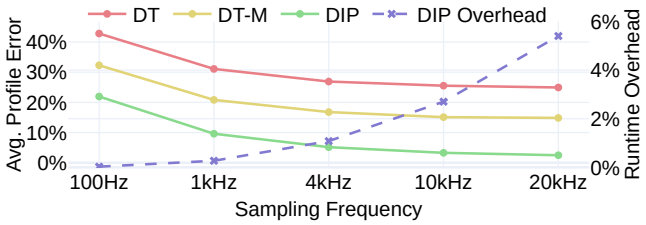


Figure 8. Average profile error and runtime overhead versus sampling frequency. *DT and DT-M consistently yield higher error than DIP at equivalent runtime overhead.*

is less favorable to break down $PICS_D$ according to performance event signatures than architectural causes. DT and DT-M for example cannot explain the high dispatch latency of `ld|34` because they focus on performance events, i.e., the $PICS_D$ are dominated by 'Base' (see Figures 7a and 7b). The key events that delay loads are long-latency cache misses, but these affect the execution of a μop *after* it has dispatched and thus cannot explain its dispatch latency. In other words, these events cause egress issues and are thus better understood by looking at $PICS_C$. In contrast, DIP and the golden reference explain that the dispatch latency of `ld|34` is dominated by front-end bubbles. The specific reason is that `ld|34` is the first instruction in a function and not aligned to an instruction cache block, in which case the BOOM's fetch stage prioritizes sending a partially filled fetch packet downstream rather than stalling until it can be filled completely.

Sensitivity to sampling frequency. We now examine the effects of sampling frequency on profiling. Figure 8 shows average profile errors for DT, DT-M, and DIP across all benchmarks, as we sweep sampling frequencies from 100 Hz to 20 kHz. The key takeaway is that DIP is more accurate than DT and DT-M across all sampling frequencies. All profilers exhibit diminishing returns beyond 4 kHz, demonstrating that 4 kHz is a sweet spot because runtime overhead increases roughly linearly with sampling frequency, also shown in Figure 8. Performance overhead is practically the same for DT, DT-M, and DIP because it is mainly due to context switching (see Section 3.2). Increasing the sampling frequency reduces the statistical component of profile error. However, systematic error, due to wrong instruction or component attribution, does not decrease.

6 Combining PICS at Dispatch and Commit

Prevalence of ingress and egress issues. The key benefit of using $PICS_D$ and $PICS_C$ for performance analysis is that they collectively provide insight into instruction window ingress and egress. This is most useful when applications exhibit both ingress and egress issues, which, as we will show, is the common case. Using DIP and TEA [27], respectively, we aggregate the cycles attributed to each component in the $PICS_D$ and $PICS_C$ across all 10 hottest instructions into

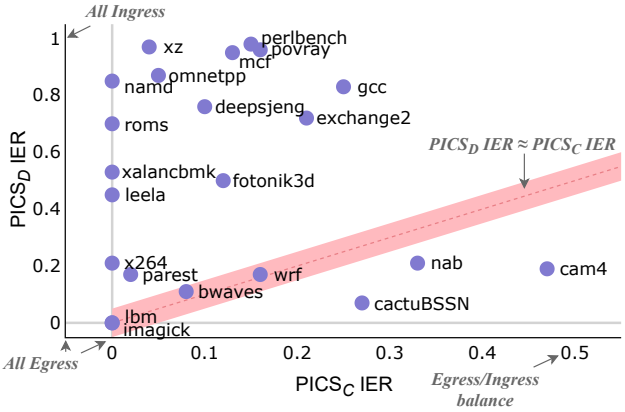


Figure 9. Ingress to Egress Ratio (IER) for $PICS_D$ and $PICS_C$ of each benchmark's 10 hottest instructions. *18 out of 22 benchmarks require both $PICS_D$ and $PICS_C$.*

ingress cycles c_i or egress cycles c_e. For $PICS_D$, *Front-end* and *Misspeculation* states are accounted as ingress, whereas *Back-end* and *Base* states as egress. For $PICS_C$, we classify components with at least one ingress event (i.e, TEA's drain and flush events) as ingress and all other components as egress; signatures that contain both ingress and egress events account for only 1.6% of total cycles. We account *Base* as egress for DIP to achieve consistent classification across $PICS_D$ and $PICS_C$. The reason is that TEA can attribute significant time to instructions that are not subject to any events. For example, all time attributed to the 10 hottest instructions of imagick and namd is due to late-issuing arithmetic instructions, and incurs no events.

We then compute the Ingress-to-Egress Ratio (IER) by dividing the ingress cycles c_i by the total cycles, i.e., $IER = c_i/(c_i + c_e)$. IER hence equals 1 when all cycles are attributed to ingress, and 0 when all cycles are attributed to egress. Figure 9 plots the IER obtained from the $PICS_D$ and $PICS_C$ of the 10 hottest instructions of our SPEC 2017 benchmarks (i.e., IER_{PICS_D} and IER_{PICS_C}, respectively). We consider $PICS_D$ and $PICS_C$ to be aligned if IER_{PICS_D} and IER_{PICS_C} are within 0.05 of each other, i.e., fall within the red region in Figure 9. $PICS_D$ and $PICS_C$ which are aligned report a similar balance between ingress and egress cycles (i.e., $IER_{PICS_D} \approx IER_{PICS_C}$). Interestingly, only lbm, imagick, bwaves, and wrf fall within this region, meaning that $PICS_D$ and $PICS_C$ are only aligned for 4 out of our 22 benchmarks. This demonstrates that $PICS_D$ and $PICS_C$ are not commonly aligned. Importantly, even when aligned, $PICS_D$ and $PICS_C$ may report different hot instructions. For example, although the 7 hottest instructions for the $PICS_D$ and $PICS_C$ of bwaves are loads, only 3 of 7 are the same. The reason is that when a long-latency load stalls at commit and blocks egress, the instruction window will fill up and eventually stall dispatch and block ingress. Although the stalling μops at commit and dispatch in this

case can belong to different dynamic invocations of the same static instruction, it is unlikely and therefore uncommon.

The majority (15 out of 22) of our benchmarks are above the diagonal red region in Figure 9, i.e., the region in which IER_{PICS_D} is larger than IER_{PICS_C} and the difference between them is greater than 0.05. These benchmarks represent the common case in which $PICS_D$ highlight ingress issues, and $PICS_C$ highlight egress issues. Consider `mcf`, which is a known memory-bound workload. `mcf` has many branch conditions depending on long-latency loads missing in the cache. Delayed branch resolution increases the impact of misspeculation due to branch misprediction, which is shown by IER_{PICS_D} close to 1 (i.e., dispatch dominated by ingress issues). Meanwhile, the long-latency loads stalling at the head of the ROB manifests as IER_{PICS_C} closer to 0 (i.e., commit exposing egress issues).

Figure 9 also points out that $PICS_C$ barely shows any ingress issues (i.e., IER_{PICS_C} is at most 0.47 for cam4). Therefore, the developer needs $PICS_D$ to identify them and understand which instructions are to blame. The best examples are the 7 benchmarks with IER_{PICS_C} equal to 0, while IER_{PICS_D} ranges from 0 (imagick and lbm) to 0.85 (namd) (i.e., all benchmarks present on the y-axis). According to the $PICS_C$, these benchmarks are fully egress-bound. However, the $PICS_D$ shows if this is true (i.e., imagick and lbm), or only one side of the story (e.g., roms and namd). For example, namd's $PICS_D$ report instructions that are hot due to misspeculation, whereas its $PICS_C$ report long-latency floating-point instructions. Looking at $PICS_C$ alone is hence misleading.

The benchmarks cactuBSSN, nab, and cam4 are below the diagonal. These benchmarks hence have lower IER_{PICS_D} than IER_{PICS_C}, meaning the $PICS_D$ highlights an egress issue while the $PICS_C$ highlights an ingress issue. This perhaps counterintuitive situation occurs when a performance issue empties the ROB. For cactuBSSN, the specific reason is that the LSQ is full of committed stores that have yet to be sent to the memory system. In the BOOM, this creates a stall at dispatch, which DIP considers an egress issue, but causes the ROB to drain because no new μops are dispatched, which TEA considers an ingress issue. The execution window is hence (practically) empty. For nab and cam4, the hottest instructions are serialized, i.e., they execute alone and thereby empty the execution window.

Case study. We now turn our attention to optimizing fotonik3d because Figure 9 shows that it exhibits a balance of ingress and egress cycles in both $PICS_D$ and $PICS_C$, making it an interesting example. Fotonik3d simulates the propagation of electromagnetic waves through photonic waveguides, and spends nearly all its execution time on floating-point computations over 3D arrays. For this example, Figure 10a presents a snippet of assembly code containing the hottest instructions at dispatch, while fotonik3d's $PICS_D$ and $PICS_C$

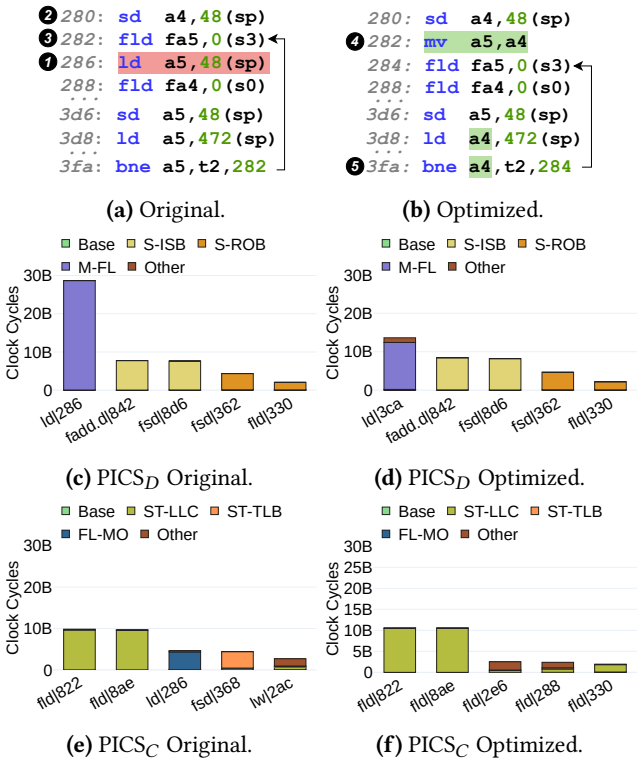


Figure 10. Fotonik3d case study. *Optimization enabled by $PICS_D$ yields an 8.5% speed-up over the original benchmark.*

are shown in Figures 10c and 10e, respectively. The hottest instruction at dispatch is `ld|286`, see ❶, which incurs a dispatch latency of nearly 29 billion cycles. By contrast, the two hottest instructions in the $PICS_C$ (Figure 10e) are floating-point loads (`fld|822` and `fld|8ae`), with `ld|286` ranked third, incurring less than half the execution time impact of the `fld` instructions. Looking at the $PICS_C$ alone hence easily leads the developer to conclude that the only optimization option is to improve the locality of the `fld` operations, thereby improving egress. This is challenging however because (i) fotonik3d's arrays are large and often exceed cache capacity, and (ii) its access patterns are irregular and strided due to use of a Yee grid [64].

Luckily, the $PICS_D$ helps uncover an alternative optimization strategy. Examining the $PICS_D$ of `ld|286` reveals a memory ordering exception (M-FL). Specifically, `ld|286` typically executes before an older store (`sd|280`), see ❷ in Figure 10a. The problem is that `ld|286` speculatively reads data that `sd|280` later changes, thus requiring the core to flush `ld|286` and any younger μops before re-executing `ld|286`. Although spilling register `a4` to the stack and then reading it back again may appear unnecessary, the compiler does this because the intervening instruction `fld|282` is a branch target, see ❸. The assembly code shows that register `a4` is not used on the execution path up to where register `a5` is used, and we can

hence avoid the load entirely. The optimized code in Figure 10b changes `ld|282` to a move instruction which copies `a5` to `a4` and swaps the order of the `mv` instruction with the preceding `fld`, see ❹. This stops the `mv` from being executed when coming from the branch at ❺, and the previous consumers of `a5` now use `a4` instead.

Figures 10d and 10f show the $PICS_D$ and $PICS_C$ of the optimized fotonik3d code in Figure 10b. As expected, `ld|286` is now gone from the $PICS_D$, but `ld|3ca` shows a similar problem, albeit to a much lesser extent (i.e., incurring about 14 billion cycles compared to nearly 29 billion cycles in Figure 10c). The remaining hot instructions at dispatch are practically unchanged. Turning to the $PICS_C$ in Figure 10f, we see that `fld|822` and `fld|8ae`, as expected, are unaffected by the optimization, as they are located in an unrelated code region. However, the bottom 3 hot instructions now incur significantly less execution time, i.e., 6.9 billion cycles in Figure 10f compared to 11.7 billion in Figure 10e, due to the core hiding execution and memory latencies better. Overall, our optimization improves performance by 8.5%.

7 Related Work

The most related works are sampling-based performance profilers that can create PICS. AMD IBS [19], Arm SPE [5], and IBM RIS [33] are most closely related as they can create $PICS_D$. The DT configuration of Section 5 is representative of these profilers, which DIP significantly improves on. TIP [26] and TEA [27] are both time-proportional at commit and therefore create accurate $PICS_C$. Intel PEBS [34] also creates $PICS_C$, but unlike TIP and TEA, it is not time-proportional at commit due to bias in its attribution polices. We showed in Section 6 that developers need both $PICS_D$ and $PICS_C$ to understand instruction window ingress and egress issues. These works represent the current state-of-the-art in a research direction that has been active for decades (e.g., ProfileMe [16] and DCPI [3]).

A favorable early-stage alternative to PICS is application-level cycle stacks, such as those from the Top-Down methodologies of Intel [63], AMD [2], and Arm [45]. These commercial implementations build upon a long line of academic work [1, 20, 21, 38, 44, 50, 65]. The work Multi-Stage CPI Stacks [22, 23] is similar, as it observes that looking at dispatch and commit yields different performance insights. By operating at the level of single instructions, we go further and explain that this difference arises due to ingress versus egress issues. Application-level cycle stacks hence complement PICS for early optimization stages, as they are easier to obtain. Unlike PICS however, they cannot provide instruction-level performance insights.

Another popular approach is to use Performance Monitoring Counters (PMCs) to gain insight into application performance. A wide body of work report challenges with PMCs [8, 9, 11, 14, 15, 41, 46, 54, 57, 58, 60, 62, 66, 68], but despite this they remain popular (e.g., [18, 49]). This paper argues for a different approach, i.e., obtaining $PICS_D$ and $PICS_C$ to understand instruction window ingress and egress at the instruction level. If this does not fully explain application performance, PMCs can be used to fill in the gaps. Analysis frameworks that rely on static [29, 31, 39, 51, 56] or dynamic [10, 43, 47] instrumentation incur a performance overhead that makes them poorly suited to understanding and addressing challenging performance issues.

Statistical sampling has also been applied to other areas of microarchitecture research. Some works propose statistical simulation sampling to enable faster simulation [30, 32, 59, 61]. Azimi et al. [7] use statistical sampling to dynamically multiplex Hardware Performance Counters (HPCs) and increase the set of logical HPCs available. In the power domain, Joseph et al. [36] use sampling to estimate signal activity, while Ardestani et al. [4] sample thermal metrics from thermal simulations at the architectural level.

8 Conclusion

This work demonstrated that advanced performance analysis requires Per-Instruction Cycle Stacks (PICS) at both dispatch (i.e., $PICS_D$) and commit (i.e., $PICS_C$) for developers to fully understand how effectively μops ingress into and egress out of the out-of-order execution window. This requires time-proportional at dispatch $PICS_D$. Therefore, we propose dispatch-time profiling (DIP), which generates $PICS_D$ with an average profile error of only 5.2%. This is a significant improvement over the 26.9% profile error of dispatch-tagging, which is the profiling technique used in state-of-the-art AMD IBS [19], Arm SPE [5], and IBM RIS [33]. DIP is simple to implement and incurs low overheads, i.e., a storage overhead of 49 bytes, a runtime overhead of ~1%, and a power overhead of approximately 12.2 mW.

Acknowledgments

We thank the anonymous reviewers and the shepherd Tanvir Ahmed Khan for their many helpful comments and suggestions. This work has been supported in part by the Research Council of Norway under grant no. 354581, Research Foundation Flanders (FWO) under grant no. G096225N, and gifts from AMD.

References

[1] Anastassia Ailamaki, David DeWitt, Mark Hill, and David Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 266–277.

[2] AMD Advanced Micro Devices. 2024. *AMD uProf User Guide* (revision 4.2 ed.). https://www.amd.com/content/dam/amd/en/documents/developer/version-4-2-documents/uprof/uprof-user-guide-v4.2.pdf Accessed: August 18, 2025.

[3] Jennifer Anderson, Lance Berc, Jeffrey Dean, Sanjay Ghemawat, Monika Henzinger, Shun-Tak Leung, Richard Sites, Mark Vandevoorde, Carl Waldspurger, and William Weihl. 1997. Continuous Profiling:

Where Have All the Cycles Gone?. In *ACM Transactions on Computer Systems (TOCS)*. 357–390. doi:10.1145/265924.265925

[4] Ehsan K. Ardestani, Francisco J. Mesa-Martínez, Gabriel Southern, Elnaz Ebrahimi, and Jose Renau. 2013. Sampling in Thermal Simulation of Processors: Measurement, Characterization, and Evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1187–1200. doi:10.1109/TCAD.2013.2253156

[5] Arm. 2023. Arm Statistical Profiling Extension: Performance Analysis Methodology. https://developer.arm.com/documentation/109429/latest/df Accessed: August 18, 2025.

[6] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott Baden, and Dean Tullsen. 2012. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE Micro* (2012), 4–16. doi:10.1109/MM.2012.57

[7] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. 2005. Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters. In *Proceedings of the International Conference on Supercomputing (ICS)*. 101–110. doi:10.1145/1088149.1088163

[8] Subho Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar 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)*. 832–844. doi:10.1145/3445814.3446739

[9] William Bowman, Swaha Miller, Vincent St-Amour, and Kent Dybvig. 2015. Profile-Guided Meta-Programming. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 403–412. doi:10.1145/2737924.2737990

[10] Derek Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph. D. Dissertation. AAI0807735.

[11] 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 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 553–568. doi:10.1145/1297027.1297068

[12] Cadence. 2025. Genus Synthesis Solution. https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[13] Cadence. 2025. Joules RTL Power Solution. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/power-analysis/joules-rtl-power-solution.html.

[14] Thomas Conte, Kishore 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. doi:10.1109/MICRO.1996.566448

[15] Thomas Conte, Burzin Patel, and Stan Cox. 1994. Using Branch Handling Hardware to Support Profile-Driven Optimization. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 12–21. doi:10.1145/192724.192726

[16] Jeffrey Dean, James Hicks, Carl Waldspurger, William 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)*. 292–302. doi:10.1109/MICRO.1997.645821

[17] Shuwen Deng, Bowen Huang, and Jakub Szefer. 2022. Leaky Frontends: Security Vulnerabilities in Processor Frontends. In *Proceedings of The International Symposium on High Performance Computer Architecture (HPCA)*. 53–66. doi:10.1109/HPCA53966.2022.00013

[18] Joao Mario Domingos, Tiago Rocha, Nuno Neves, Nuno Roma, Pedro Tomás, and Leonel Sousa. 2023. Supporting RISC-V Performance Counters Through Linux Performance Analysis Tools. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 94–101. doi:10.1109/ASAP57973.2023.00027

[19] Paul Drongowski. 2007. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. Technical Report. AMD. https://alessandropellegrini.it/theses/IBS.pdf Accessed: August 18, 2025.

[20] Joel Emer and Douglas Clark. 1984. A Characterization of Processor Performance in the vax-11/780. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 301–310. doi:10.1145/800015.808199

[21] 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. doi:10.1145/1168857.1168880

[22] Stijn Eyerman, Wim Heirman, Kristof Du Bois, and Ibrahim Hur. 2018. Extending the Performance Analysis Tool Box: Multi-stage CPI Stacks and FLOPS Stacks. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 179–188. doi:10.1109/ISPASS.2018.00031

[23] Stijn Eyerman, Wim Heirman, Kristof Du Bois, and Ibrahim Hur. 2018. Multi-Stage CPI Stacks. *Computer Architecture Letters (CAL)* (Oct. 2018), 55–58. doi:10.1109/LCA.2017.2761751

[24] Filanovsky, Vadim Filanovsky and Sane, Harshad. 2022. Seeing through hardware counters: A journey to threefold performance increase. https://netflixtechblog.com/seeing-through-hardware-counters-a-journey-to-threefold-performance-increase-2721924a2822. Accessed: August 18, 2025.

[25] Björn Gottschall, Silvio Campelo de Santana, and Magnus Jahre. 2023. Balancing Accuracy and Evaluation Overhead in Simulation Point Selection. In *IEEE International Symposium on Workload Characterization (IISWC)*. 43–53. doi:10.1109/IISWC59245.2023.00019

[26] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2021. TIP: Time-proportional instruction profiling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 15–27. doi:10.1145/3466752.3480058

[27] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2023. TEA: Time-proportional event analysis. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 1–13. doi:10.1145/3579371.3589058

[28] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2024. Per-Instruction Cycle Stacks Through Time-Proportional Event Analysis. *IEEE Micro* (2024), 27–33. doi:10.1109/MM.2024.3407377

[29] Susan Graham, Peter Kessler, and Marshall Mckusick. 1982. Gprof: A Call Graph Execution Profiler. *ACM SIGPLAN Notices* (1982), 120–126. doi:10.1145/872726.806987

[30] Qi Guo, Tianshi Chen, Yunji Chen, and Franz Franchetti. 2016. Accelerating Architectural Simulation Via Statistical Techniques: A Survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016), 433–446. doi:10.1109/TCAD.2015.2481796

[31] 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)*. 145–155. doi:10.1109/ICSE.2012.6227198

[32] J.W. Haskins and K. Skadron. 2001. Minimal Subset Evaluation: Rapid Warm-up for Simulated Hardware State. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*. 32–39. doi:10.1109/ICCD.2001.955000

[33] IBM. 2018. POWER9 Performance Monitor Unit User's Guide. https://ibm.ent.box.com/s/8kh0orsr8sg32zb6zmq1d7zz6hud3f8j. Accessed: August 18, 2025.

[34] 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. Accessed: August 18, 2025.

[35] Intel. 2023. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-

reference-manual-volume-1.html Accessed: December 29, 2025.

[36] Russ Joseph and Margaret Martonosi. 2001. Run-time Power Estimation in High Performance Microprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. 135–140. doi:10.1109/LPE.2001.945389

[37] 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)*. 29–42. doi:10.1109/ISCA.2018.00014

[38] Kimberly Keeton, David Patterson, Yong Qiang He, Roger Raphael, and Walter Baker. 1998. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 15–26. doi:10.1145/279358.279364

[39] 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)*. 75–86. doi:10.1109/CGO.2004.1281665

[40] Charles Leiserson, Neil Thompson, Joel Emer, Bradley Kuszmaul, Butler Lampson, Daniel Sanchez, and Tao Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* (2020). doi:10.1126/science.aam9744

[41] Ao Li, Marion Sudvarg, Zihan Li, Sanjoy Baruah, Chris Gill, and Ning Zhang. 2025. Tintin: A Unified Hardware Performance Profiling Infrastructure to Uncover and Manage Uncertainty. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 575–593.

[42] Linux. 2020. perf. https://perfwiki.github.io/main/. Accessed: August 18, 2025.

[43] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 190–200. doi:10.1145/1065010.1065034

[44] Yue Luo, Juan Rubio, Lizy John, Pattabi Seshadri, and Alex Mericas. 2003. Benchmarking Internet Servers on Superscalar Machines. *Computer* (Feb. 2003), 34–40. doi:10.1109/MC.2003.1178044

[45] Jumana Mundichipparakkal, Yibo Cai, Sam Ellis, Darin Greene, Michael Williams, and Tushar Chouhan. 2023. *Arm Neoverse V1 Core: Performance Analysis Methodology*. White Paper. Arm Limited. https://developer.arm.com/documentation/109199/latest/ Accessed: August 18, 2025.

[46] Todd Mytkowicz, Peter 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. doi:10.1109/MICRO.2007.27

[47] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices* (June 2007), 89–100. doi:10.1145/1273442.1250746

[48] Chrysanthos Pepi, Bhargav Reddy Godala, Krishnam Tibrewala, Gino A. Chacon, Paul V. Gratz, Daniel A. Jiménez, Gilles A. Pokam, and David I. August. 2025. Skia: Exposing Shadow Branches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1091–1106.

[49] Daniel Puckett, Tyler Tomer, Paul Gratz, Jiang Hu, Galen Shipman, Jered Dominguez-Trujillo, and Kevin Sheridan. 2025. Estimating CPI Stacks From Multiplexed Performance Counter Data Using Machine Learning. *Computer Architecture Letters (CAL)* (April 2025), 1–4. doi:10.1109/LCA.2025.3556644

[50] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. 1998. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 307–318. doi:10.1145/291069.291067

[51] Tao Schardl, Tyler Denniston, Damon Doucet, Bradley Kuszmaul, I-Ting Angelina Lee, and Charles Leiserson. 2017. The CSI Framework for Compiler-Inserted Program Instrumentation. *Proceedings of the Measurement and Analysis of Computing Systems (POMACS)* (2017), 1–25. doi:10.1145/3154502

[52] Sawan Singh, Arthur Perais, Alexandra Jimborean, and Alberto Ros. 2022. Exploring Instruction Fusion Opportunities in General Purpose Processors. In *Proceedings of The International Symposium on Microarchitecture (MICRO)*. 199–212. doi:10.1109/MICRO56248.2022.00026

[53] Standard Performance Evaluation Corporation. 2019. https://www.spec.org/cpu2017/ Accessed: February 17, 2025.

[54] Peter 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)*. 5.

[55] Vertical Research Group. 2021. Microbench. https://github.com/VerticalResearchGroup/microbench. Vertical Research Group, University of Wisconsin–Madison. Accessed: August 18, 2025.

[56] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI)*. 473–486.

[57] Vincent Weaver and Sally McKee. 2008. Can Hardware Performance Counters be Trusted?. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 141–150. doi:10.1109/IISWC.2008.4636099

[58] Vincent 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. doi:10.1109/ISPASS.2013.6557172

[59] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. 2005. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 408–409. doi:10.1145/1064212.1064278

[60] John Whaley. 2000. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of the Conference on Java Grande (JAVA)*. 78–87. doi:10.1145/337449.337483

[61] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 84–97. doi:10.1145/859618.859629

[62] 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. doi:10.1145/3330345.3330371

[63] Ahmad 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. doi:10.1109/ISPASS.2014.6844459

[64] Kane Yee. 1966. Numerical Solution of Initial Boundary Calue Problems Involving Maxwell's Equations in Isotropic Media. *IEEE Transactions on Antennas and Propagation* (1966), 302–307. doi:10.1109/TAP.1966.1138693

[65] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. 1996. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of the Conference on Supercomputing (SC)*. 16–es. doi:10.1145/369028.369059

[66] 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. doi:10.1109/ISPASS.2009.4919635

[67] 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.

[68] 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)*. Association for Computing Machinery, 433–450. doi:10.1145/2858965.2814281

[69] Kan Zhu, Yilong Zhao, Yufei Gao, Peter Braun, Tanvir Ahmed Khan, Heiner Litz, Baris Kasikci, and Shuwen Deng. 2025. From Optimal to Practical: Efficient Micro-op Cache Replacement Policies for Data Center Applications. In *Proceedings of The International Symposium on High Performance Computer Architecture (HPCA)*. 716–731. doi:10.1109/HPCA61900.2025.00060