Correct Wrong Path Simulation

Chrysanthos Pepi
Texas A&M University
College Station, TX, USA
cpepis@tamu.edu

Krishnam Tibrewala
Texas A&M University
College Station, TX, USA
krishnamtibrewala@tamu.edu

Bhargav Reddy Godala
AheadComputing
Hillsboro, OR, USA
bhargav.godala@aheadcomputing.com

Sankara Prasad Ramesh
University of California, San Diego
San Diego, CA, USA
spramesh@ucsd.edu

Alberto Ros
University of Murcia
Murcia, Spain
aros@ditec.um.es

Daniel A. Jiménez
Texas A&M University
College Station, TX, USA
djimenez@acm.org

Gilles A. Pokam
AMD Inc.
Santa Clara, CA, USA
gilles.pokam@amd.com

Paul V. Gratz
Texas A&M University
College Station, TX, USA
pgratz@tamu.edu

***Abstract*—Modern OoO CPUs employ deep pipelines with high branch misprediction recovery penalties. Instructions speculatively executed along mispredicted paths can significantly alter microarchitectural state. During design space exploration, architects often rely on trace-driven simulators, which are significantly faster than execution-driven models but trade accuracy for speed. Despite this benefit, trace-driven simulation often fails to adequately model the effects of wrong-path execution because traces are typically collected only from the correct-path. While prior work can accurately model wrong-path effects on the instruction stream, it often makes unrealistic assumptions when modeling the impact on the data stream.**

In this work, we examine the effects of wrong-path execution and present an infrastructure for enabling its modeling in a trace-driven simulator. Our analysis shows that wrong-path execution extensively affects structures on both the instruction and data sides, yielding performance variations ranging from -3.6% to 85.7% compared to a baseline that ignores these effects. To benefit the research community and enhance the accuracy of simulators, we provide our traces and tracing utility. We aim for this to encourage industry to provide wrong-path traces generated by internal simulators, enabling fast academic research without exposing industry proprietary IP.

I. Introduction

Accurate modeling of modern Out-of-Order (OoO) CPUs presents a significant bottleneck in design space exploration due to high simulator complexity and long simulation times of execution-driven simulators. Consequently, architects often use simpler and faster, though less accurate, trace-driven simulators, thereby trading accuracy for speed. The goal of our work is to bridge this gap by enabling trace-driven simulators to model important characteristics of execution-driven simulation without sacrificing performance.

Typically, the instruction traces used in trace-driven simulation are generated on real machines leveraging instrumentation tools, such as Intel PIN [34], DynamoRIO [2], and Spike [5]. These tools capture the sequence of committed instructions, which form the correct-path (CP), from a given program and save them to a trace file for the trace-driven simulator to use during execution. In the event of a branch misprediction, however, while a real machine will execute wrong-path (WP) instructions (with an impact on caches and various predictors' and prefetchers' states), a trace-driven simulator will typically stall the front-end of the machine until the branch misprediction is resolved and CP execution can be resumed. Ignoring the impact of WP execution on microarchitectural state can distort performance estimates, as we demonstrate in this work.

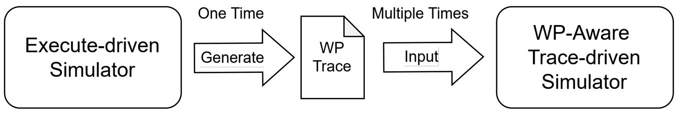


Fig. 1: Proposed flow: an execution-driven simulator generates WP-aware traces for rapid, accurate, trace-driven simulation.

Prior works [13], [49] have proposed solutions to model WP effects in trace-driven simulators. However, existing solutions primarily capture WP effects on the instruction stream; modeling WP memory accesses and their impact on the data-cache hierarchy remains limited. To our knowledge, no existing approach encompasses *both* the pollution and prefetching impact on *both* the instruction and data streams. The alternative is to use more "accurate[1]" execution-driven models, but they are substantially slower than trace-driven simulation [21], [33].

The success of trace-driven simulation can be attributed to its ability to support rapid modeling of novel microarchitectural ideas. Its simplicity and speed make it an attractive choice during the initial design exploration phase. Notably, the frequent appearance of trace-driven simulation [12], [26], [49], [50], [59] in top-tier conference publications underscores the ongoing need for such tools. We improve the fidelity of trace-driven simulator by introducing a WP-augmented trace format that encodes both WP and CP instruction execution. Fig. 1 illustrates our workflow, in which a one-time execution-driven run produces WP-augmented traces that enable rapid WP-aware trace-driven design-space exploration. To realize this workflow, we extend an execution-driven simulator [33] to generate WP-augmented traces and enhance a trace-driven simulator [21] to execute them.

[1]While some execution-driven simulators do model wrong-path, there is no guarantee that they model other aspects of the core more accurately than trace-driven simulators.

This paper makes the following contributions:

- A new methodology for more accurate trace-driven simulation that fully incorporates the prefetching and pollution effects of WP on both instructions and data caches.
- An open implementation built on gem5 and ChampSim, including a new trace format, tracing utility, and WP-aware simulator.
- An open set of WP-aware traces of widely used data center, graph, and SPEC workloads ready to be used.
- Extensive analysis quantifying the impact of WP execution on cache state and performance.

We show that with minor increases in trace size and limited impact on simulator runtime, WP instructions can be accurately modeled without sacrificing the benefits of trace-driven simulation.

II. Motivation

There is a critical need to reconcile the accuracy of high-fidelity simulation with the speed and scalability of trace-driven analysis. A trace-driven simulator models the impact of executing a stream of instructions on various microarchitectural components, such as caches, branch predictors, prefetchers, and memory systems. These instruction streams typically encode register dependencies and memory operation addresses but omit the actual register or memory values, focusing instead on the structural behavior of the program.

Modeling WP execution in such simulators presents unique challenges. When a branch is mispredicted, the simulator must determine the sequence of instructions that fall along the incorrect control path, which together constitute the WP. Accurately simulating WP behavior involves propagating these instructions through the early pipeline stages (Fetch, Decode, Rename) and through Execution. Once the misprediction is detected and resolved, the pipeline must be flushed to discard the WP instructions, resuming CP execution.

Capturing WP instructions in traces is not possible with existing dynamic binary instrumentation frameworks, such as Pin [34], because they record only committed instructions that have retired and updated the architectural state. Since WP instructions are speculatively executed and ultimately squashed before commit, they are typically invisible to such tracing tools. This absence limits the ability of trace-driven simulators to model speculative side effects such as cache pollution or branch predictor state changes caused by WP execution.

We propose a new methodology to improve the accuracy of existing trace-driven simulators. In particular, we use an execution-driven simulator, gem5 [33], to capture WP instructions and encode them in the trace. We also modify an existing, popular trace-driven simulator, ChampSim [21], to support the execution and squashing of WP instructions. This allows us to perform Execution-Driven Wrong-Path Simulation (ED-WP).

We choose ChampSim as the target simulator for ED-WP due to several compelling reasons. ChampSim is widely used in the computer architecture community, it has been used for several architecture competitions, and it is the basis of many top conference publications. This is partly due to ChampSim providing significantly faster runtimes, often an order of magnitude faster than full-system simulators such as gem5. Its codebase is smaller and more accessible, and it already includes a wide range of prefetchers, replacement policies, and other microarchitectural components. By extending ChampSim to support WP behavior, we aim to preserve its speed while greatly enhancing its accuracy.

While we use ChampSim as our experimental platform, our core contribution extends beyond any specific tool or implementation. Our work fundamentally examines how WP execution influences the behavior and accuracy of trace-driven and execution-driven simulation, the two predominant methodologies for microarchitectural evaluation. ChampSim exemplifies trace-driven simulation in our study, while gem5 serves as a representative of execution-driven models. However, the insights we present are broadly applicable to any framework following these paradigms. The central message we aim to convey is a deeper understanding of when trace-driven simulation is appropriate, where it falls short, and most importantly, how its limitations can be mitigated without sacrificing the performance and scalability that make it so attractive. By elevating the discussion from specific tools to general simulation methodology, we hope to ensure that the broader relevance of our work is not obscured by implementation-specific details.

A. Prior Approaches

Some prior work exists for modeling WP execution [26], [42], [49] in trace-driven simulators; however, this has certain limitations. Protean's [26] method introduces WP prefetching, primarily focusing on branches. As a result, it does not account for other types of mispredictions or fully capture cache impacts. EIP's [49] use of instruction history helps model dependencies, but it may miss dynamic address changes, leading to some inaccuracies. UDP's [42] Scarab [4], while replaying previously recorded WP instructions, has challenges with accurately modeling data cache pollution due to address reuse. We refer to these approaches as Trace-Inferred Wrong-Path Simulation (TI-WP). Broadly, while these approaches make useful strides, no existing approach models both the prefetching and pollution effects in both the instructions and data caches comprehensively.

III. Trace-Driven Simulation of Wrong-Path Instructions

This section describes our system to enable execution-driven WP simulation in a trace-driven simulator. First, we describe our WP trace format along with the modifications to the execution-driven simulator. Then, we detail the modifications to the trace-driven simulator to support the execution of those traces. Finally, we discuss other implementation aspects.

A. WP Trace

We first introduce a new trace format designed to explicitly capture the stream of instructions executed along a mis-speculated path, whether the mis-speculation arises from a branch misprediction or a memory ordering violation due to incorrect load-store disambiguation. In both cases, speculative execution leads the processor down an incorrect control or data path before the mis-speculation is detected and corrected.

In a traditional execution model, WP instructions are speculatively fetched and potentially executed but are eventually squashed once the mis-speculation is resolved. These instructions do not commit and hence are typically excluded from conventional traces. Our format extends the trace structure to include WP instructions, preserving their order and context following the mis-speculated instruction. Additional metadata fields are used to differentiate WP instructions from those on the CP, enabling the trace-driven simulator to model their side effects more accurately without affecting architectural state.

Modern processors use decoupled front-ends often using a form of Fetch Directed Instruction Prefetching (FDIP) [47], where the branch predictor runs ahead of the fetch unit, queuing predicted targets in the Fetch Target Queue (FTQ). Mispredictions, whether control-based or memory-based, cause the front-end to redirect and flush pending fetch entries, but instructions prefetched before the redirect may still enter caches and consume bandwidth. Our trace format captures these prefetched-but-non-executed WP entries so ChampSim can model the resulting front-end pressure.

InstrID	Instruction	WP instr.	FTQ-prefetch
x-2	instr < ..... >	0	0
x-1	instr < ..... >	0	0
x	branch < ..... > (miss predicted)	0	0
.	instr < ..... >	1	0
.	instr < ..... >	1	0
x+n-2	NOP	1	1
x+n-1	NOP	1	1
x+n	instr < ..... > (target of InstrID x)	0	0
x+n+1	instr < ..... >	0	0

Fig. 2: WP traces format illustrating newly added instruction & prefetches on WP in gray.

Fig. 2 illustrates WP trace format in the context of FDIP. Instructions `x-2` and `x-1` occur prior to the mispredicted branch and execute normally on the CP. The instruction at `x` is a mispredicted branch — it is itself on the CP, but it causes the Instruction Fetch Unit (IFU) to temporarily follow the WP. The two subsequent instructions (denoted by `.`) are fetched and pushed into the pipeline along this WP. These are speculatively executed and are ultimately discarded, and therefore marked with `WP instr.` = 1 and `FTQ-prefetch` = 0. The instructions at `x+n-2` and `x+n-1` are designated "NOPs" because, despite being actual instructions, they were never executed in the original simulator, instead they were prefetched via FDIP before the pipeline was redirected but flushed before execution. Although they were not executed,

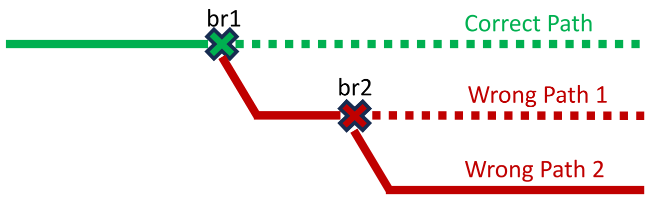


Fig. 3: Speculative Execution with Nested Mispredictions.

they reside on the WP and influence the microarchitectural state by consuming fetch bandwidth and polluting the instruction cache, hence, `WP instr.` = 1 and `FTQ-prefetch` = 1. The instruction at `x+n` is the first correct-path instruction at the actual target of the mispredicted branch and is on CP. Finally, `x+n+1` is the next sequential instruction on the CP.

By including both executed WP instructions and FTQ-prefetched instructions, our extended trace format provides a more complete representation of speculative instruction flow, allowing ChampSim to more faithfully reproduce the dynamics of real-world out-of-order processors.

B. WP Integration

In conventional trace-driven simulators (e.g., ChampSim), each branch instruction triggers a query to the Branch Prediction Unit (BPU) to determine its predicted direction and target. The simulator then compares the BPU's prediction against the ground truth in the trace to detect mispredictions. During branch resolution, the front-end stalls; once resolved, a fixed *pipeline repair* penalty is incurred before fetch resumes along the CP. Since WP instructions never enter the pipeline, there is no need to squash structures or perform recovery actions, and they do not introduce any cache pollution. To enable modeling of WP instructions within ChampSim, we modify stages of the pipeline as follows: The **Fetch** stage continues consuming the trace after a misprediction, issuing I-cache requests along the WP and stopping only when a resteer redirects it to the CP. The **Decode** stage identifies mispredicted unconditional direct CP branches, squashes younger instructions, flushes front-end buffers, and issues a resteer that advances the trace to the correct target. The **Execute** stage handles all remaining CP mispredictions by marking younger Reorder Buffer (ROB) instructions as WP, flushing them along with associated pipeline structures, and updating rename mappings to eliminate speculative dependencies.

C. WP Mispredictions

To simplify the modeling of nested mispredictions in a trace-driven simulator like ChampSim, we restrict branch resolution to branches on the CP only (e.g., `br1` in Fig. 3). This choice maintains implementation simplicity while remaining consistent with the behavior of execution-driven simulators. Because both branch predictions and targets are embedded within the trace, the effects of WP branches (e.g., `br2`) are inherently captured, regardless of whether their predictions were correct or not.

Field / Model	Alder Lake like
ISA	ARM
Private L1 Cache	I$-32kB (8-way), D$-64kB (16-way), 64B
Private L2 Cache	1MB (16-way), 64B
Shared L3 Cache	2MB (16-way), 64B
Branch Predictor	TAGE-SC-L [53] (64KB), ITTAGE [52] (64KB)
BTB Size	16K Entries (8-way)
FTQ / ROB Entries	24 / 512 Entries
Decode / Retire	12 Wide
Issue / Load / Store Q.	194 / 144 / 112
Int / Vec. Registers	448 / 400

TABLE I: Processor configurations

Feature	No-WP	TI-WP	ED-WP (Our Work)
FDIP	✓	✓	✓
WP Inst	✗	✓[2]	✓
WP Data	✗	✗	✓

TABLE II: Comparison of Simulation Methodologies

This methodology is applicable to a broad range of microarchitectural exploration tasks. However, integrating gem5's branch prediction outcomes limits the ability of the resulting tool to explore branch prediction optimizations, since the trace must be regenerated whenever the branch predictor is modified to ensure that gem5 and ChampSim follow the same WPs. For future work, we are investigating mechanisms to collect deeper WP information that would enable evaluating different branch predictors directly in ChampSim without regenerating traces.

IV. Simulation Methodology

This section outlines the simulation infrastructure, the different models of WP execution evaluated, and the benchmarks used to evaluate these models.

A. Simulation Infrastructure

We extended ChampSim [21] to incorporate support for WP instructions, as described in the previous section. We configured the simulated CPU microarchitecture to reflect Intel's Golden Cove core, which is featured in Alder Lake processors. To generate traces with WP information, we use the gem5 simulator [33] configured in O3CPU mode with a decoupled front-end. Both gem5 and ChampSim are configured with a 10 million-instruction warm-up phase followed by 100 million instructions of detailed simulation, using the microarchitectural parameters listed in Table I.

B. Simulation Models

To evaluate the impact of WP execution on application behavior, we compare three simulation models in ChampSim, as summarized in Table II.

- **No-WP** is the baseline: our enhanced ChampSim with FDIP support but no WP modeling. On a misprediction, the front-end stalls until the branch is resolved.
- **TI-WP** (Trace-Inferred Wrong-Path) models WP execution by replaying previously recorded instructions, as in EIP [49] and UDP [42]. Since WP instructions are drawn from the original CP trace, their data addresses reflect prior CP behavior rather than actual wrong-path addresses, resulting in data accesses that predominantly hit in the existing cache state.
- **ED-WP** (Execution-Driven Wrong-Path simulation mode), our proposed approach using our modified trace format to encode WP instructions obtained from an execution-driven simulator.

Key	Source	Benchmarks
LCF	DaCapo [14]	cassandra, kafka, tomcat
	Renaissance [46]	finagle-chirper, finagle-http
	OLTP Bench [17]	tpcc, wikipedia
	Tailbench [28]	specjbb, web-search
	Cloudsuite V4 [19]	data-serving, media-streaming
	Chipyard [6]	verilator
	Browser Bench [1]	speedometer2.0
GAP	GAP [11]	bc, bfs, cc, ccsv, pr, prspmv, sssp, tc
SPEC	CPU 2017 [15]	505.mcf, 525.x264, 531.deepsjeng 541.leela, 548.exchange2, 557.xz

TABLE III: Benchmarks used in our evaluation.

C. Benchmarks

To evaluate the impact of WP execution across a diverse set of application behaviors, we selected benchmarks from several widely used suites. The full list is shown in Table III.

First we assembled a selection of 13 widely used client-side and server-side multi-threaded workloads with substantial code footprints, specifically curated to stress the instruction cache and front-end pipelines. These workloads were selected from various benchmark suites, as listed in Table III [1], [6], [14], [17], [46]. We selected benchmarks with an L1I MPKI exceeding 10. In the rest of this section, we refer to these workloads as "Large Code Footprint" or LCF workloads. These benchmarks have also been utilized in prior related studies [22], [29], [30], [39], [44].

Next, we incorporate 8 benchmarks from the GAP benchmark suite [11], including irregular and memory-intensive applications designed to highlight pressure on the memory hierarchy and speculative execution mechanisms (these benchmarks are denoted as "GAP"). For both of these suites, since the standard simpoints methodology does not apply, we used Intel's VTune Profiler [3] on a modern Alder Lake Intel processor to run benchmarks on a real machine to determine when a given workload reaches its steady state, creating a gem5 checkpoint at this point for our simulations.

Finally, we use a subset of six SPEC CPU 2017 [15] benchmarks, to round out the suite of workloads examined. We use the simpoints methodology [25] to identify representative program phases for this suite. We evaluate all six workloads that our simulation toolchain currently supports. This combination of compute, memory, and front-end bound benchmarks enables a comprehensive analysis of WP effects across a wide performance spectrum.

In order to fully understand the impact of WP through the core modeling, we examined the relative performance changes for a number of prefetchers and replacement policies at different cache levels. The full list is shown in Table IV.

[2]Only if present in the original trace.

Key	Prefetcher
L1I	Next Line [9], BARÇA [23], BIP [24], DJolt [40], TAP [20] EIP [48], [49], FNL+MMA [54], Mana [7], PIPS [35]
L1D	Next Line, IP Stride [58], Berti [41], IPCP [43]
L2C	Next Line, IP Stride, SPP [31], Bingo [10] MLOP [56], SMS [57], Streamer [16]
LLC Repl.	SHIP [60], SRRIP [27], DRRIP [27], Mockingjay [55]

TABLE IV: Prefetchers and replacement policies examined

V. Evaluation

This section presents a performance analysis of simulating WP execution through ED-WP mode and compares it with No-WP and TI-WP mode on all benchmarks detailed in Section IV-C. We also examine the impact of TI-WP and ED-WP modes on the cache hierarchy, providing measurements for the L1 instruction (L1I), L1 data (L1D), and L2C caches [3].

A. *WP Impact on Instructions processed*

Modern processors have aggressive decoupled front ends, large structures, and deep ROBs, allowing a substantial number of WP instructions to be fetched and executed before a branch misprediction is detected and resolved.

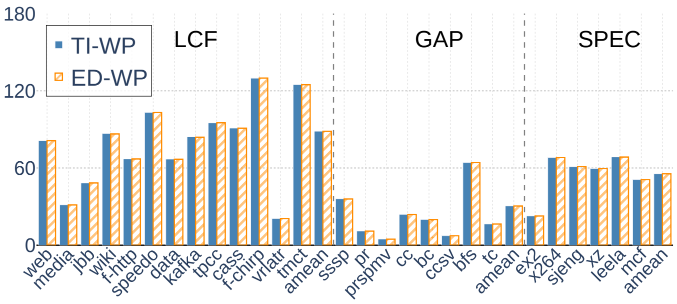


Fig. 4: Increase in L1I footprint under TI-WP and ED-WP, relative to No-WP.

Fig. 4 shows the increase in unique L1I cache lines accessed under TI-WP and ED-WP relative to No-WP. Both WP-aware modes show comparable footprint growth, as the additional accesses stem from WP instruction fetches absent in No-WP. LCF workloads exhibit the largest increase at 88.6%, reflecting their wide instruction footprint and frequent mispredictions that expose new cache lines along the WP. GAP and SPEC benchmarks show 30.4% and 55.5% increases, respectively. These results demonstrate that No-WP significantly underestimates the instruction working set under speculative execution.

Fig. 5 shows the data brought into L1D by WP execution, expressed as a percentage of CP L1D misses in the No-WP baseline. This metric captures how much additional data WP loads introduce into the cache hierarchy. For GAP and SPEC workloads, ED-WP produces meaningful L1D fills, averaging 8.8% and 18.0% of No-WP CP misses respectively. Individual benchmarks such as `ccsv` (35.4%), `cc` (22.9%), and `505.mcf` (22.2%) show particularly high fill rates. In

[3]We also examined the LLC, both for cache misses and prefetching, but found results largely similar to L2C and thus omitted them for brevity.

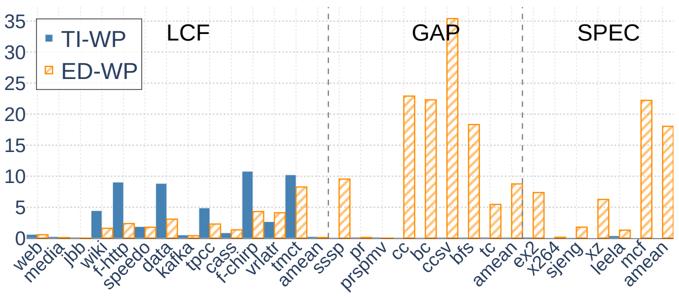


Fig. 5: L1D cache lines filled by WP execution, as a percentage of CP L1D misses in the No-WP baseline.

contrast, TI-WP produces near-zero fills, as its WP loads almost entirely hit in L1D. This demonstrates that ED-WP's WP execution acts as an implicit data prefetcher, an effect entirely absent in TI-WP. For LCF workloads, the overall fill rates remain low (under 11%) for both modes, indicating that WP data fills are not the primary driver of performance differences in instruction-intensive workloads.

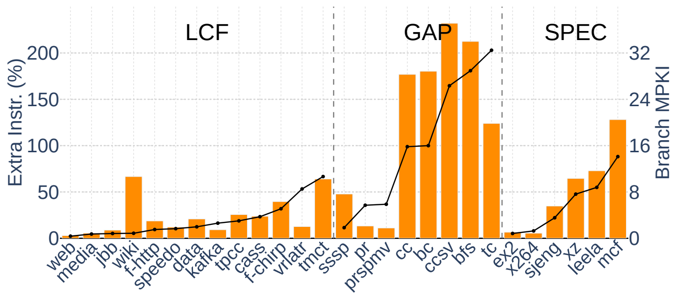


Fig. 6: Comparison of extra instructions executed and MPKI. The bars represent the percentage of extra instructions executed due to wrong-path execution (ED-WP), while the lines show the MPKI for each benchmark in the respective suite.

Fig. 6 reports the additional instructions along the WP in ED-WP, showing significant variation across workloads. WP instruction volume correlates with branch MPKI, which determines how frequently execution diverges onto the wrong path. However, the relationship is not strictly linear, structural limits such as ROB and FTQ sizes bound how far speculation can proceed. This explains why LCF workloads, despite moderate MPKI, produce relatively few WP instructions: their wide instruction footprint causes frequent L1I misses that stall the front-end, limiting speculation depth. GAP workloads, in contrast, have compact instruction footprints that rarely miss in L1I, allowing deep speculation along mispredicted paths. To aid in the interpretation of this behavior, all benchmarks within each workload suite are sorted in ascending order by branch MPKI in all figures.

B. *WP Impact on Cache State*

1) Instruction Cache State: Modeling the execution of WP instructions significantly alters the microarchitectural state,

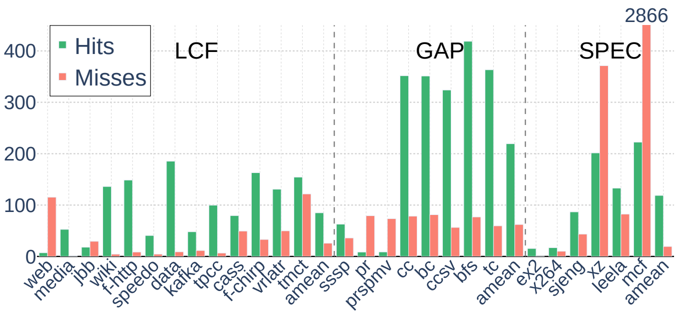


Fig. 7: Relative increase in L1I hits and misses under ED-WP compared to No-WP.

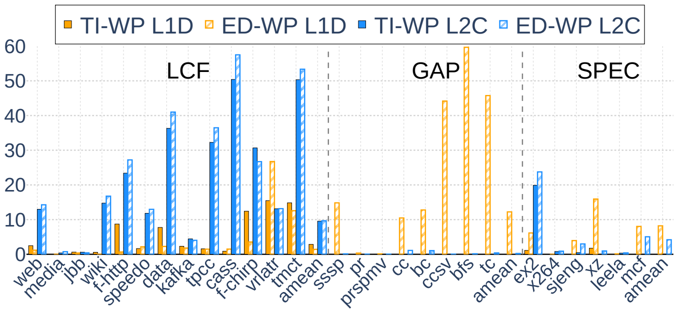


Fig. 8: Relative increase in L1D and L2C misses under ED-WP and TI-WP compared to No-WP.

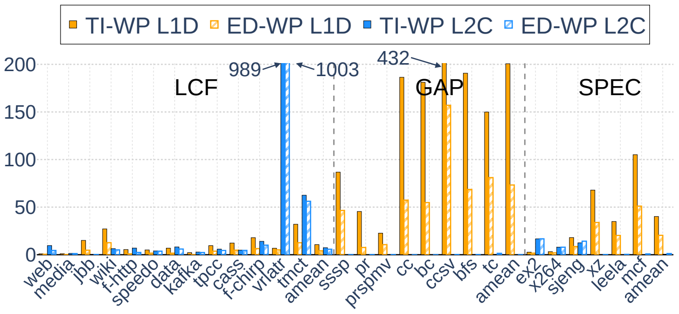


Fig. 9: Relative increase in L1D and L2C hits under ED-WP and TI-WP compared to No-WP.

with the L1I being the most affected component. In particular, ED-WP mode introduces a notable transformation in L1I behavior compared to No-WP mode. This is illustrated in Fig. 7, which shows that there is an average 84.8% increase in L1I hits and a 25.7% increase in L1I misses across the LCF benchmarks in ED-WP mode. This trend suggests that WP instructions increase instruction reuse within the pipeline, possibly due to re-fetching along mispredicted paths or revisiting instruction sequences during recovery.

The GAP suite exhibits the most dramatic changes. On average, GAP benchmarks show 3.2× increase in L1I hits, with individual benchmarks such as `bfs` and `cc` showing 5.2× and 4.5× respectively. Despite being speculative, these WP-influenced accesses populate the instruction cache with high-locality sequences, reducing cold and capacity misses during CP execution. This also leads to an average 62.1% increase in L1I misses, indicating some degree of pollution leading to the eviction of useful lines and subsequent new misses. In contrast, the SPEC benchmarks show relatively modest variations. The average increase in L1I hits is around 118.6%, while misses increase by 19.3%, driven primarily by memory-intensive workloads such as `mcf` and `xz`.

Cache lines brought in due to WP instruction execution can be either useful (eventually used on the CP) or useless (evicted without being used). WP-induced useless fills reduce effective cache occupancy and may evict critical lines, while WP-induced useful fills effectively prefetch lines later accessed by CP instructions. On average, 74% of WP fills in L1I cache are useful, demonstrating a strong prefetching effect.

2) Data Cache State: Fig. 8 shows the relative increase in L1D and L2C misses when WP execution is modeled. At the L1D level, ED-WP increases misses for GAP (12.2%) and SPEC (8.2%) due to wrong-path loads accessing new data, while TI-WP shows near-zero L1D miss changes for these suites. LCF workloads show modest L1D miss increases under both modes. At the L2C level, LCF workloads show the largest miss increases (9.7% for both modes), driven by the additional instruction and data traffic propagating down the hierarchy. GAP shows negligible L2C impact, while SPEC shows a modest 4.2% increase under ED-WP.

Fig. 9 shows the relative increase in L1D and L2C hits. TI-WP consistently shows larger L1D hit increases than ED-WP — 200.7% vs 73.3% for GAP, 40.2% vs 20.4% for SPEC, and 10.6% vs 4.2% for LCF. This is because TI-WP's wrong-path loads predominantly access data already present in L1D, inflating the hit count without bringing new data into the cache. ED-WP's wrong-path loads, in contrast, access speculated addresses that more frequently miss in L1D, producing fewer hits but more fills as shown in Fig. 5. At the L2C level, hit increases are modest for both modes across all suites.

This distinction has direct performance consequences: ED-WP's new fills act as implicit prefetches for subsequent CP accesses, whereas TI-WP's WP loads do not alter cache state.

C. WP Impact on the Core Backend

To understand how ED-WP affects the backend, we report the occupancy of the ROB. Since ROB usage fluctuates constantly during execution, we choose to measure it at the point of branch misprediction resolution, because it offers a consistent and meaningful point for comparison across simulation modes. By the time a misprediction is resolved, speculative instructions have often accumulated, including those on the WP, giving us a representative snapshot of backend pressure.

As shown in Fig. 10, ED-WP sees substantially higher ROB occupancy, averaging 208 entries compared to just 94 in No-WP—a 2.2× increase. This is expected: ED-WP continues executing instructions along the mispredicted path, filling the ROB more aggressively. While these instructions are eventually squashed, the higher occupancy more accurately reflects real hardware behavior, where deep speculation puts additional pressure on backend resources like register rename

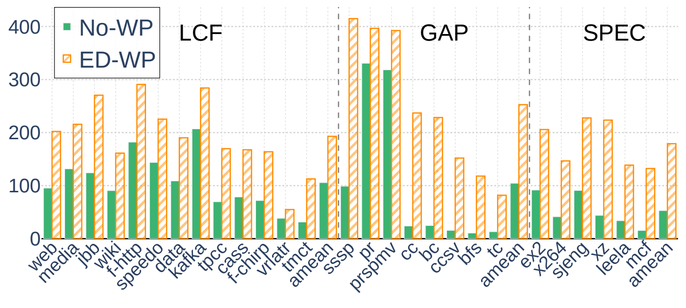


Fig. 10: Average ROB occupancy at branch misprediction.

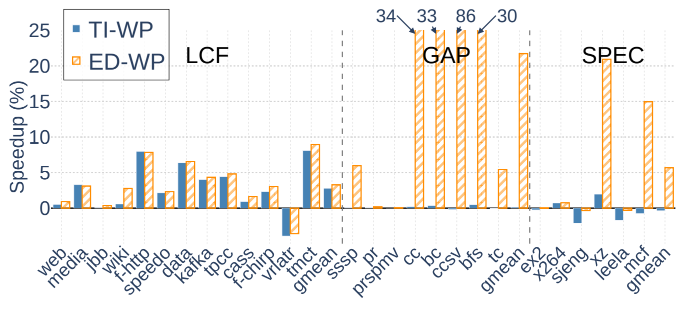


Fig. 12: IPC of TI-WP and ED-WP relative to the No-WP.

tables, issue queues, and execution units. Moreover it also increases the window of in-flight instructions, enabling more opportunities for memory loads and stores to overlap and prefetches to be issued earlier.

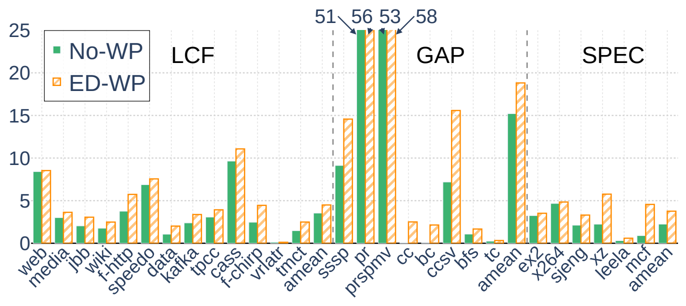


Fig. 11: ROB Full Events per Kilo Instructions (PKI).

In addition to average occupancy, we track the total number of times the ROB becomes full throughout execution, as shown in Fig. 11. When the ROB is full, dispatch stalls until space becomes available, directly impacting pipeline throughput. Under ED-WP, the ROB becomes fully occupied much more frequently, with 4.50 full ROB events per kilo instructions for the LCF workload, 18.82 for GAP, and 3.76 for SPEC. In comparison, No-WP sees fewer full ROB events: 3.54 for LCF, 15.20 for GAP, and 2.24 for SPEC. The increased ROB pressure under ED-WP underscores the importance of modeling WP execution, as overlooking it risks underestimating core-level contention that affects real-world performance. Despite this pressure, ED-WP sustains higher IPC than No-WP for most benchmarks, as the prefetching benefits of speculative execution offset the backend contention.

D. *Performance Impact*

As discussed previously, the performance impact of WP instructions cannot be accurately captured by analyzing cache behavior, branch mispredictions, or back-end effects in isolation. WP instructions interact with these components in different ways depending on the workload, and only a comprehensive, unified analysis can reflect their true impact. Fig. 12 presents the IPC speedup of the ED-WP and TI-WP modes relative to No-WP configuration across a range of benchmarks.

Overall, both WP-aware models deliver performance improvements over No-WP, indicating that the beneficial prefetching effects of WP execution outweigh the associated cache pollution. For instruction-heavy LCF workloads, TI-WP performs on par with ED-WP, as both models incorporate WP instruction behavior similarly. However, on data-intensive benchmarks such as `505.mcf`, `557.xz` from SPEC, and `cc`, `bc`, `ccsv`, and `bfs` from GAP suite, ED-WP shows clear advantages. GAP workloads have low baseline IPC (0.21–0.69), so small absolute IPC gains yield large relative improvements. ED-WP's advantage stems from its faithful modeling of WP data cache accesses.

Within LCF workloads, `wikipedia` illustrates the importance of data-side modeling: ED-WP achieves a 2.8% speedup over No-WP, while TI-WP captures only 0.6%, with the 2.2% gap attributable to WP data effects that TI-WP cannot model. `x264`, with low MPKI (1.3), shows negligible difference between ED-WP and TI-WP (0.04%), as few mispredictions leave little room for WP effects. `verilator-bolted` has high MPKI but limited WP instruction growth and many early-resolved unconditional branches, leading to performance degradation due to squashed WP instructions (cf. Fig. 6).

The prefetching effect of WP instructions dominates in a majority of benchmarks. Fig. 13 shows the percentage reduction in CP cache misses when running in TI-WP and ED-WP mode, relative to No-WP. The L1I miss reduction is the most pronounced, with ED-WP reducing misses by 47.1% for LCF, 14.7% for GAP, and 38.4% for SPEC on average. Notably, `mcf` achieves 100% L1I miss reduction, as its compact instruction footprint is entirely prefetched by WP instruction fetches. TI-WP achieves nearly identical L1I reductions, confirming that both modes capture the instruction-side prefetching effect of WP execution equally well. The critical difference between ED-WP and TI-WP emerges on the data side. ED-WP reduces L1D CP misses by 11.7% for GAP and 21.8% for SPEC, with individual benchmarks such as `ccsv` (41.4%), `cc` (33.5%), and `mcf` (27.1%) showing substantial reductions. TI-WP, in contrast, achieves near-zero L1D miss reductions for these suites, as its WP loads hit in L1D without prefetching new data. This gap explains the IPC differences observed in Fig. 12. This pattern extends to L2C, where ED-WP reduces misses for SPEC by 28.2% while TI-WP shows no improvement. Comparable reductions in L1I misses do not

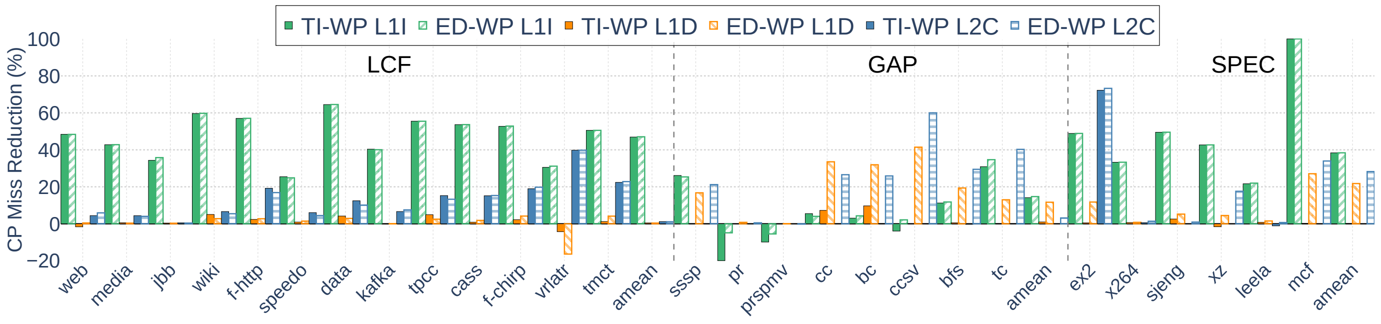


Fig. 13: CP miss reduction under ED-WP and TI-WP relative to No-WP, for L1I, L1D, and L2C.

yield the same performance improvement as similar reductions on the data side. This aligns with expectations for modern OoO processors, where aggressive front-end designs are better equipped to tolerate instruction cache inefficiencies, whereas performance is highly sensitive to data cache behavior.

E. Takeaways

Culminating the results from Sections V-A - V-D, we find: 1) WP execution introduces substantial additional instructions, with significant prefetching effects and execution resource pressure throughout the system; 2) Prior work techniques that only model the inferred WP (TI-WP) miss substantial changes due to the data prefetching effects of WP execution.

F. WP Execution Validation

Fig. 14 shows a per-cycle breakdown of execute-stage activity. Each bar is partitioned into: **CP** (only Correct-Path instructions executed), **WP** (only Wrong-Path instructions executed), **CP+WP** (both), **ROB Empty** (no instructions in the ROB), **ROB Empty (No WP)** (ROB is empty and no wrong-path instructions are available from the trace), and **ROB Not Ready** (instructions are present but blocked on unresolved dependencies). This breakdown helps expose where execution stage state spends its cycles in ED-WP mode.

We use the ROB Empty (No WP) fraction to validate the completeness of our WP traces. A high fraction would indicate that the simulator frequently lacks WP instructions when it needs them, suggesting insufficient trace coverage. This fraction is low across all suites: 7.6% for LCF, 3.2% for GAP, and 3.2% for SPEC, confirming that the traces provide sufficient WP instruction coverage. These results show that the simulator is not significantly impacted by a lack of WP instructions for execution. Notable exceptions are `verilator` (51.0%), `tc` (21.7%), and `tomcat` (16.7%), all of which have high branch MPKI, leading to frequent pipeline flushes that outpace the WP instructions available in the trace. It might be possible to increase the WP trace depth dynamically depending on branch MPKI to try and address this issue for those workloads, we will explore this in future work.

G. Case Studies: Impact of ED-WP vs. No-WP in Evaluating Microarchitectural Techniques

In this section, we perform a set of case studies examining the impact of WP instruction simulation when modeling new microarchitectural techniques. In particular, we examine the relative performance gain of different L1I, L1D, and L2C prefetchers when executed in No-WP only versus ED-WP in simulation. Here, we use the complete set of prefetchers listed in Table IV. Most of the L2C prefetchers used in this study were adapted from the Pythia framework [12].

1) L1I Prefetchers: Fig. 15a shows the geomean performance gains of different L1I prefetchers when only CP instructions (No-WP) are executed, versus when WP instructions are included. Note that the figure shows data only for the LCF benchmarks, as the other benchmarks showed little performance impact for L1I prefetching.

The figure highlights that the absence of WP modeling has a large impact on estimating prefetcher effectiveness. Seven of the nine policies hurt performance with a more accurate simulation. Among the evaluated prefetchers, only EIP [49] and FNL+MMA [54] see performance improvement. However, their benefit is greatly reduced to only 0.8% and 1.9%, respectively, when simulated in the ED-WP model, compared to 8.9% and 7.4% when considering only the CP. We note that the absence of WP does not just overestimate the effectiveness of instruction prefetchers but also alters their ranking relative to one another, undermining the insights about which prefetchers would perform best in a real system.

2) L1D Prefetchers: Similarly, Fig. 15b shows a comparison of No-WP vs ED-WP for L1D prefetchers. Here we see that No-WP continues to overestimate the effectiveness of data prefetchers, though generally the relative ordering stays broadly the same. For example, Berti drops from 8.6% to 6.3% for GAP and from 2.2% to 0.7% for SPEC. Overall, GAP shows the largest changes between No-WP and ED-WP, consistent with ED-WP's data prefetching impact (Section V-B).

3) L2C Prefetchers: Fig. 15c shows the speedup for various L2C prefetchers on the No-WP and ED-WP. Broadly, prefetching in the L2C has a modest impact for these workloads, with only IP Stride and SPP showing much gain and only for the GAP benchmark suite. LCF workloads show little performance benefit from L2C prefetching, with gains under 1.4% across all evaluated prefetchers. GAP workloads involve a great deal of pointer chasing on the data side, limiting the ability of typical L2C prefetchers to improve performance due to relatively poor accuracy. Several prefetchers cause significant degradation, with Bingo at −23.5% and Next Line at −20.9%

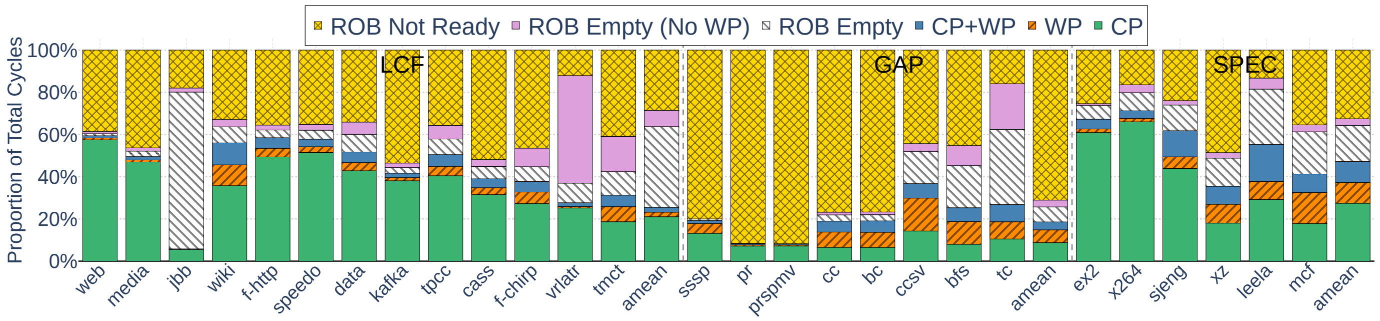


Fig. 14: Breakdown of execute-stage activity as a percentage of total cycles.

under ED-WP. The subset of workloads chosen from SPEC also do not have particularly high L2C misses, which lowers the impact of prefetching on performance. Nevertheless, ED-WP simulation reduces prefetcher benefit without altering their ranking, consistent with L1I and L1D results.

4) LLC Replacement Policies: As shown in Fig. 15d, ED-WP does not significantly alter the overall geomean performance. However, it impacts their relative rankings noticeably—especially for LCF workloads. DRRIP [27], for instance, sees a marked improvement when WP accesses are included, while Mockingjay [55] sees minimal change.

The effect of WP on ranking is particularly evident in LCF workloads, which generate a high volume of speculative accesses to the LLC due to their large footprints. This change in ranking is explained by the fundamental differences in how these policies handle cache management. Mockingjay [55] tries to mimic Belady's optimal replacement by learning and predicting reuse distances using detailed access history and timing information. Because of this, it is more susceptible to interference from speculative accesses, which can throw off its predictions and lead to poorer eviction choices. DRRIP [27], in contrast, uses simpler insertion heuristics and is less sensitive to noise from WP execution due to its set-dueling mechanism's effectiveness at isolating thrashing lines by steering them toward SRRIP. Thus, including WP effects is important to ensure a more accurate picture of system behavior.

H. Simulation Overhead

We observe significant advantages of trace-driven simulation over execution-driven approaches in both performance and usability. For example, simulating 100 million instructions with gem5 [33] takes about 1 hour on our HPC setup. In contrast, ChampSim [21] completes the same workload in roughly 10 minutes. ChampSim traces are also more portable, requiring only a single trace file per workload, whereas gem5 setups involve binaries, input data, and scripts. ED-WP introduces minimal simulation overhead, averaging only 1.13× the runtime of baseline ChampSim with a range of 0.93–1.40× across individual benchmarks. These overheads reflect only the trace-driven simulation; the one-time cost of trace generation using gem5 is not included. WP-augmented traces are larger than their CP-only counterparts, growing by 1.53× for LCF, 3.25× for GAP, and 2.02× for SPEC on average.

VI. Related Work

The effect of modeling WP execution on processor performance has a rich history in the literature. Researchers have explored how to mitigate or exploit WP effects.

A. Quantifying and Exploiting the Effect of WP Execution

Pierce and Mudge introduce an instruction prefetcher that aggressively fetches through all targets, ignoring the branch predictor [45]. Using a trace-driven simulator, they found that WP prefetching outperformed contemporary prefetchers.

Mutlu *et al.* characterize the impact of WP execution on performance, demonstrating that modeling WP effects is essential to accurately estimating performance due primarily to the trade-off between the positive WP cache prefetching effect and the negative WP cache pollution effect [37], [38]. They use an execution-driven simulator capable of measuring the effect of loads and stores along the WP.

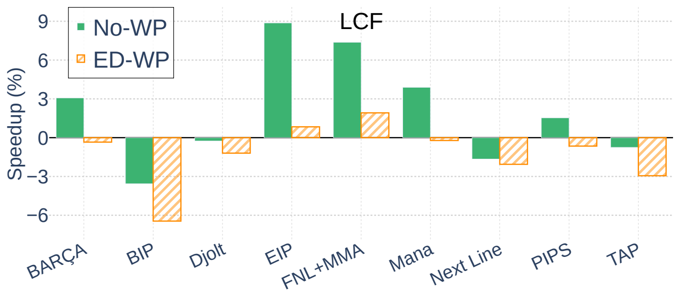
Sendag *et al.* extend this exploration to multiprocessors [51], showing that WP memory accesses cause a significant increase in cache coherence events and proposing an optimization to recover performance in the presence of these events.

Armstrong *et al.* exploit the ability to simulate cache activity on the WP to explore a novel branch prediction technique that uses the anomalous behavior of WP events to detect and correct a misprediction [8]. Again, this work was made possible by using an execution-driven simulator that captures cache activity on the WP. The same group followed this line of work to demonstrate a cache filtering optimization [36] as well as a way to fine-tune fetch throttling based on whether WP effects will be harmful or beneficial [32].

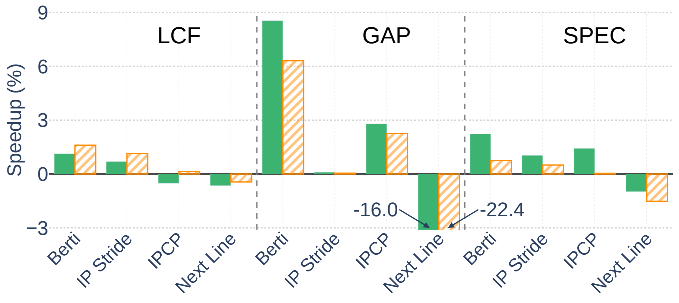
B. Improving Simulation Methodology

Prior work has explicitly explored techniques for accurately modeling WP execution in processor simulators. Bhargava *et al.* [13] addressed this problem by recreating a copy of the application code to insert WP instructions into a trace-driven simulator. Although this is appropriate to know the instructions that will be fetched on the WP, it fails to faithfully reproduce the memory locations of the loads in the WP, and hence cannot accurately show the effects of the WP on the data side.

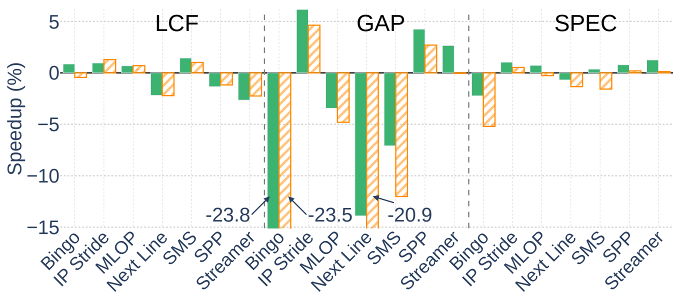
More recently, Eyerman *et al.* explore three different ways of simulating WP instructions in the context of functional simulation [18]: 1) reconstructing the instructions along the



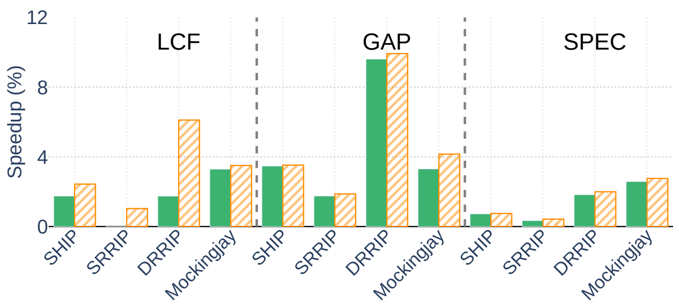
(a) Comparison of L1I Prefetchers



(b) Comparison of L1D Prefetchers



(c) Comparison of L2C Prefetchers



(d) Comparison of LLC Replacement Policies

Fig. 15: Geomean IPC speedup of various prefetchers and LLC replacement policies under No-WP and ED-WP.

WP using previously recorded CP information, 2) using more aggressive and costly simulation to attempt to reconstruct memory addresses along the WP, and 3) combining the first idea with doing a less aggressive reconstruction of memory addresses by exploiting control reconvergence. They report that this third technique provides the best balance between simulation speed and accurate estimation of performance. Unfortunately, this approach is still execution-driven rather than trace-driven. It requires having the binary executables as well as inputs, making the simulation infrastructure more complex and possibly infeasible if the goal is to study proprietary software for which traces but not binaries can be made available. Trace-driven simulation also has the advantage of complete determinism, while functional simulation can behave nondeterministically based on e.g. using time-of-day information to seed random number generators.

Ros and Jimborean [49] propose a WP model based on information collected from a trace. While accurate for modeling the front end, we show in this work that it fails to accurately model data accesses on the WP. In this paper, we model this approach as TI-WP. Oh *et al.* [42] propose a similar model. Execution-driven simulation is used whenever possible, while trace-driven simulation is leveraged for complex, multi-process, and Java-VM-based applications that cannot be directly executed in their simulator. The trace data is collected using DynamoRIO and Intel PT, ensuring a precise reconstruction of dynamically executed basic blocks and memory accesses for accurate performance modeling. To provide a credible methodology while using a trace-driven simulator, when a branch is mispredicted leading to a WP, they replay previously recorded traces from when that path was the CP, selecting traces likely to have similar memory behavior, or NOP instructions if no such path is available. They find that in 99% of cases, there are traces available for which that path was previously correct. However, again, the sequence of memory addresses generated from this methodology is not the same as the actual WP. Replayed load/store instructions reuse prior memory addresses, potentially affecting WP L1D pollution.

VII. Conclusions

Modern OoO CPUs speculatively execute a substantial volume of instructions along mispredicted paths, significantly affecting cache state and overall performance. Trace-driven simulators, though widely used for their speed and simplicity, typically ignore these effects or model them only on the instruction side. We introduced ED-WP, which incorporates execution-driven wrong-path behavior into trace-driven simulation. Our evaluation shows IPC variations of -3.6% to 85.7% compared to ignoring WP effects. TI-WP captures the instruction-side prefetching effect but misses the data side entirely, where ED-WP reduces L1D misses by up to 41.4%. WP modeling also alters prefetcher and replacement policy rankings. We also provide tools and traces to help improve simulation accuracy.

Acknowledgments

We thank the reviewers for their valuable suggestions. We thank the Texas A&M High Performance Research Computing Center for providing computation resources.

References

[1] "Browserbench," https://browserbench.org/.
[2] "Dynamorio," https://dynamorio.org/.
[3] "Intel vtune profiler," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[4] "Scarab simulator (litz-lab/scarab)," https://github.com/Litz-Lab/scarab.

[5] "Spike risc-v isa simulator," https://github.com/riscv-software-src/riscv-isa-sim.

[6] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *Ieee Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[7] A. Ansari, F. Golshan, R. Barati, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Mana: Microarchitecting a temporal instruction prefetcher," *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 732–743, 2022.

[8] D. N. Armstrong, H. Kim, O. Mutlu, and Y. N. Patt, "Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery," in *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 2004, pp. 119–128.

[9] J.-L. Baer, *Microprocessor architecture: from simple pipelines to chip multiprocessors*. Cambridge University Press, 2009.

[10] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 399–411.

[11] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[12] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1121–1137.

[13] R. Bhargava, L. K. John, and F. Matus, "Accurately modeling speculative instruction fetching in trace-driven simulation," in *1999 IEEE International Performance, Computing and Communications Conference (Cat. No. 99CH36305)*. IEEE, 1999, pp. 65–71.

[14] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006, pp. 169–190.

[15] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[16] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE transactions on computers*, vol. 44, no. 5, pp. 609–623, 1995.

[17] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, 2013.

[18] S. Eyerman, S. Van Den Steen, W. Heirman, and I. Hur, "Simulating wrong-path instructions in decoupled functional-first simulation," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 124–133.

[19] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *Acm sigplan notices*, vol. 47, no. 4, pp. 37–48, 2012.

[20] N. Gober, G. Chacon, D. A. Jiménez, and P. Gratz, "Temporal ancestry prefetcher," *The 1st Instruction Prefetching Championship (IPC1)*, 2020.

[21] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," *arXiv preprint arXiv:2210.14324*, 2022.

[22] B. R. Godala, S. P. Ramesh, G. A. Pokam, J. Stark, A. Seznec, D. Tullsen, and D. I. August, "Pdip: Priority directed instruction prefetching," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 846–861.

[23] D. A. J. P. V. Gratz and G. C. N. Gober, "Barca: Branch agnostic region searching algorithm," *The First Instruction Prefetching Championship*, 2020.

[24] V. Gupta, N. S. Kalani, and B. Panda, "Runjump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching," *The First Instruction Prefetching Championship*, 2020.

[25] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[26] M. Hassan, C. H. Park, and D. Black-Schaffer, "Protean: Resource-efficient instruction prefetching," in *Proceedings of the International Symposium on Memory Systems*, 2023, pp. 1–13.

[27] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 60–71, Jun. 2010. [Online]. Available: https://doi.org/10.1145/1816038.1815971

[28] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[29] T. A. Khan, N. Brown, A. Sriraman, N. K. Soundararajan, R. Kumar, J. Devietti, S. Subramoney, G. A. Pokam, H. Litz, and B. Kasikci, "Twig: Profile-guided btb prefetching for data center applications," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 816–829.

[30] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-guided instruction cache replacement for data center applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 734–747.

[31] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[32] C. J. Lee, H. Kim, O. Mutlu, and Y. N. Patt, "Performance-aware speculation control using wrong path usefulness prediction," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 2008, pp. 39–49.

[33] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[35] P. Michaud, "Pips: Prefetching instructions with probabilistic scouts," in *IPC-1-First Instruction Prefetching Championship*, 2020, pp. 1–4.

[36] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt, "Cache filtering techniques to reduce the negative impact of useless speculative memory references on processor performance," in *16th Symposium on Computer Architecture and High Performance Computing*. IEEE, 2004, pp. 2–9.

[37] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt, "Understanding the effects of wrong-path memory references on processor performance," in *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, 2004, pp. 56–64.

[38] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt, "An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1556–1571, 2005.

[39] N. P. Nagendra, B. R. Godala, I. Chaturvedi, A. Patel, S. Kanev, T. Moseley, J. Stark, G. A. Pokam, S. Campanoni, and D. I. August, "Emissary: Enhanced miss awareness replacement policy for l2 instruction caching," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[40] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-jolt: Distant jolt prefetcher," *The 1st Instruction Prefetching Championship (IPC1)*, 2020.

[41] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 975–991.

[42] S. Oh, M. Xu, T. A. Khan, B. Kasikci, and H. Litz, "Udp: Utility-driven fetch directed instruction prefetching," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 1188–1201.

[43] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 118–131.

[44] C. Pepi, B. R. Godala, K. Tibrewala, G. A. Chacon, P. V. Gratz, D. A. Jiménez, G. A. Pokam, and D. I. August, "Skia: Exposing shadow branches," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 1091–1106.

[45] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 1996, pp. 165–175.

[46] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon *et al.*, "Renaissance: Benchmarking suite for parallel applications on the jvm," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 31–47.

[47] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999, pp. 16–27.

[48] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 99–111.

[49] A. Ros and A. Jimborean, "Wrong-path-aware entangling instruction prefetcher," *IEEE Transactions on Computers*, vol. 73, no. 2, pp. 548–559, 2023.

[50] A. Saxena and M. Qureshi, "Start: Scalable tracking for any rowhammer threshold," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 578–592.

[51] R. Sendag, A. Yilmazer, J. J. Yi, and A. K. Uht, "Quantifying and reducing the effects of wrong-path memory references in cache-coherent multiprocessor systems," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 10–pp.

[52] A. Seznec, "A 64-kbytes ittage indirect branch predictor," in *JWAC-2: Championship Branch Prediction*, 2011.

[53] A. Seznec, "Tage-sc-l branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[54] A. Seznec, "The fnl+ mma instruction cache prefetcher," in *IPC-1-First Instruction Prefetching Championship*, 2020, pp. 1–5.

[55] I. Shah, A. Jain, and C. Lin, "Effective mimicry of belady's min policy," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 558–572.

[56] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," *The Third Data Prefetching Championship*, no. 2019, 2019.

[57] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 252–263, 2006.

[58] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys (CSUR)*, vol. 32, no. 2, pp. 174–199, 2000.

[59] G. Vavouliotis, M. Torrents, B. Grot, K. Kalaitzidis, L. Peled, and M. Casas, "To cross, or not to cross pages for prefetching?" in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 188–203.

[60] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, "Ship: signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: Association for Computing Machinery, 2011, p. 430–441. [Online]. Available: https://doi.org/10.1145/2155620.2155671

APPENDIX

A. Abstract

This artifact contains the complete experiment workflow for reproducing all results presented in the paper. It includes pre-generated ChampSim traces, pre-computed simulation results, and gem5 checkpoints for all benchmarks. Singularity/Apptainer container definitions are provided for both ChampSim and gem5 to ensure a reproducible environment. Three evaluation paths are supported: a quick path that regenerates all paper figures from pre-computed results ($\sim$10 min), a full reproduction path that rebuilds ChampSim and re-runs all simulations from traces ($\sim$35 binary configurations $\times$ 3 simulation modes $\times$ $\sim$27 benchmarks), and an optional trace generation path that regenerates traces from SPEC CPU 2017, GAP, and LCF benchmarks using gem5.

B. Artifact check-list (meta-information)

- **Program:** gem5 (full-system simulator) and ChampSim (trace-driven microarchitectural simulator)
- **Compilation:** GCC 13, CMake, Make (all provided by the container)
- **Traces:** Pre-generated ChampSim traces for SPEC CPU 2017, GAP, and LCF benchmarks
- **Binary:** Precompiled gem5.opt; ChampSim binaries built during setup ($\sim$35 configurations)
- **Data set:** Pre-computed simulation results for all benchmark/configuration/mode combinations
- **Run-time environment:** Singularity/Apptainer containers (Ubuntu 22.04) for both ChampSim and gem5
- **Hardware:** Machine with $\geq$ 32 GB RAM
- **Metrics:** IPC, cache hit/miss rates, cache pollution, branch MPKI, ROB occupancy, prefetcher accuracy, simulation overhead
- **Output:** Simulation statistics files; paper figures as PDFs and interactive HTML
- **Experiments:** Bash scripts for all configurations (local and SLURM); Jupyter notebook for analysis
- **How much disk space required (approximately)?:** $\sim$50 GiB for traces; $\sim$1.2 TiB for gem5 checkpoints (uncompressed)
- **How much time is needed to prepare workflow (approximately)?:** $\sim$15 min
- **How much time is needed to complete experiments (approximately)?:** $\sim$10 min for quick path; $\sim$hours on HPC / $\sim$days locally for full reproduction; $\sim$weeks for SimPoint generation, $\sim$hours per benchmark for trace generation
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.18914161

C. Description

1) How to access:

- Download artifact from Zenodo: https://doi.org/10.5281/zenodo.18914161
- Download size: $\sim$38 GB

2) Hardware dependencies:

- At least 32 GB memory.
- At least 350 GB disk space ($\sim$1.5 TiB for full trace regeneration).

3) Software dependencies:

- Singularity or Apptainer (install guide)
- All other dependencies (GCC 13, CMake, Python 3, Jupyter) are provided by the container
- Building the container requires `sudo` (or equivalent). On HPC systems without root access, build on a local machine and copy the `.sif` file to the cluster.

D. Installation

Extract the artifact tarball and run the setup script:

```
tar -xzvf cwps-ae.tar.gz -C cwps-ae
cd cwps-ae/ChampSim
bash setup.sh
```

The script builds the Singularity container, compiles ChampSim inside it, and runs a smoke test. All subsequent commands assume you are in the `ChampSim/` directory. For optional trace generation:

```
cd ../trace-generation
bash setup.sh
```

E. Experiment workflow

Three evaluation paths are provided:

1) **Quick Path** ($\sim$10 min): Regenerate all paper figures from pre-computed results.

```
bash scripts/generate_figures.sh \
  --results-dir paper_results
```

2) **Full Reproduction** ($\sim$hours on HPC, $\sim$days locally): Build all $\sim$35 binary configurations, run simulations across 3 modes (CP, ED-WP, TI-WP) for all benchmarks, and regenerate figures.

```
bash scripts/generate_binaries.sh
bash scripts/run_simulations.sh \
  --traces-dir \
  ../trace-generation/output/traces \
  --mode local --suite ALL
bash scripts/generate_figures.sh
```

For SLURM execution:

```
SLURM_ACCOUNT=your_account \
bash scripts/run_simulations.sh \
  --traces-dir \
  ../trace-generation/output/traces \
  --mode slurm --suite ALL
```

3) **Trace Generation** (optional, $\sim$weeks for SimPoint generation on HPC, $\sim$hours per benchmark for trace generation): Regenerate ChampSim traces from SPEC CPU 2017, GAP, and LCF benchmarks using gem5. See `trace-generation/TRACE_GENERATION.md`.

F. Evaluation and expected results

The artifact reproduces all figures presented in the paper. Figures are generated as PDFs in `analysis/pdf/` and as interactive HTML in `analysis/html/`. Key results to verify:

- **IPC speedup** of ED-WP over No-WP across all benchmark suites
- **Cache pollution**: reduction in correct-path misses at L1I, L1D, L2C, and LLC
- **ROB occupancy** increase at misprediction points under wrong-path simulation
- **Simulation overhead** of ED-WP vs. gem5 full-system simulation
- **Prefetcher and replacement policy** sensitivity under wrong-path effects

Using the quick path with pre-computed results produces figures identical to those in the paper. Full reproduction using the provided traces should also produce identical results. If new traces are generated from checkpoints, results may exhibit minor variations, but overall trends and conclusions remain consistent.

G. Experiment customization

Individual suites can be selected with `--suite`:

```
bash scripts/run_simulations.sh \
--traces-dir ../trace-generation/output/ \
traces --mode local --suite SPEC
```

Valid options: `SPEC`, `GAP`, `LCF`, or `ALL`. Run `bash scripts/run_simulations.sh --help` for all options. ChampSim configurations in `configs/` can be modified to experiment with different parameters; rebuild with `bash scripts/generate_binaries.sh`.