2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) | 979-8-3315-5103-2/26/$31.00 ©2026 IEEE | DOI: 10.1109/ISPASS69572.2026.00024

Compiler and System Optimizations for Gem5 Simulator

Haneul Park[1*], Siddharth Agarwal[1*], Pradyun Narkadamilli[1], Kiung Jung[2], Yongjun Park[2], Ipoom Jeong[2], Nam Sung Kim[1]

[1]*University of Illinois at Urbana–Champaign, Urbana, USA*
[2]*Yonsei University, Seoul, Republic of Korea*

{hnpark2, sa10, pradyun2, nskim}@illinois.edu, {kiung, yongjunpark, ipoom}@yonsei.ac.kr

***Abstract*—Architectural simulators are indispensable for modern computer architecture research, but they remain notoriously slow due to their event-driven, cycle-level execution model. In this work, we present a set of software- and system-level optimizations to accelerate large-scale design-space exploration with gem5. First, we reduce per-instance simulation time via compiler-level optimization. We demonstrate that although gem5 suffers severe frontend stalls on modern CPUs stemming from its large instruction footprints, naïve Profile-Guided Optimization (PGO) is impractical in this setting because it requires frequent reprofiling and recompilation. To address this, we challenge the conventional reliance on self-profiling and instead construct a universal, performance-driven profile that generalizes across simulation inputs. Second, we improve aggregate simulation throughput by strengthening performance isolation using Sub-NUMA clustering (SNC). Finally, we show that a simple co-scheduling heuristic has great potential for reducing resource stranding and boosting multi-instance efficiency. Together, these techniques improve single simulation speed by 17% and aggregate throughput by 27%, making large-scale design-space exploration more practical and efficient.**

I. Introduction

Software simulators are essential for computer architecture research, allowing for exploration of complex design spaces without the prohibitive costs and time of physical prototyping. Open-source simulators such as gem5 [20], Sniper [10], and ZSim [24] are widely used in both academia and industry. Among them, gem5 provides a modular framework that supports diverse Instruction Set Architectures (ISAs) and full system simulation. As an event-driven, cycle-level simulator, gem5 enables detailed analysis of microarchitectural behavior and system interactions.

However, this high fidelity comes at a steep performance cost: detailed out-of-order core simulation typically runs at around 0.3 MIPS. The slowdown is not solely due to the fundamental work required for cycle-level simulation, but also due to inefficiencies in how the simulator executes on the host CPU. Prior work shows that gem5 attains only a fraction of the peak performance of the host CPU [28]. In particular, gem5 is often dominated by the frontend stalls (*i.e.*, L1-I cache misses), attributed to its event-driven control flow that heavily relies on function pointers and virtual dispatch across a large codebase. As gem5 processes events, the instruction stream repeatedly jumps among disparate components—from CPU models to memory controllers and interconnects—leading to poor instruction locality.

Profile-Guided Optimization (PGO) is a powerful compiler technique that can reduce such frontend stalls [22], but has received little attention in the gem5 community. PGO leverages runtime profiles to drive transformations such as code layout optimization, function inlining, and indirect-call promotions [18]. However, applying PGO effectively is challenging because profiles are highly input-dependent: collecting a representative profile typically requires an additional compile–profile–recompile cycle for each new combination of guest binary and microarchitectural configuration, since different simulation inputs stress different regions of the simulator. Because architecture simulations typically evaluate each configuration only once, this cost is difficult to amortize.

In this work, we show that the conventional assumption that PGO must rely on *self-profiling*—*i.e.*, profiling with the exact target input to maximize performance—does not hold for gem5. Instead, we demonstrate that performance-driven profile selection can consistently outperform self-profiling. Across a wide range of simulations, gem5 shares common execution paths, and PGO's benefits are largely concentrated in a narrow but performance-critical subset of the codebase, namely memory simulation. Optimizations applied to less critical components provide little benefit while unnecessarily inflating code size, ultimately diluting the gains from memory-centric optimizations. Guided by this analysis, we construct an extremely memory-intensive profile that selectively preserves only the most effective optimizations and generalizes across inputs, enabling robust speedups without input-specific tuning. Using this *universal* profile, gem5 consistently outperforms self-profiling and achieves 17% performance gain on SPEC CPU2017.

Beyond per-instance speed, effective design-space exploration also depends on multi-instance throughput. When many simulation instances run concurrently, the large working set size of gem5 induces substantial interference in the shared Last-Level Cache (LLC), limiting scalability. As a result, naïve scaling achieves only a 32× throughput improvement when increasing from a single instance to running 80 concurrent instances. We show that this sublinear scaling arises primarily

*These authors contributed equally.

979-8-3315-5103-2/26/$31.00 ©2026 IEEE
DOI 10.1109/ISPASS69572.2026.00024

from LLC contention and evaluate a simple yet effective set of system-level optimizations to mitigate it. By combining PGO with careful allocation of LLC capacity across different sub-NUMA domains, we improve aggregate throughput to 37× that of a single instance on an 80-thread system.

This work makes the following contributions:

- We show that PGO can substantially accelerate event-driven simulators like gem5, while also demonstrating that PGO's hotness-driven decisions do not necessarily translate into end-to-end performance improvements
- We propose a practical universal profiling methodology for gem5 that focuses PGO on the most performance-critical paths and minimizes code-inflation side effects.
- We identify two scalability challenges in multi-instance simulation execution—shared resource contention and CPU core stranding—and provide system-level guidelines to mitigate them.

II. Background

A. Event-Driven Simulation

In architectural simulators, it quickly becomes infeasible to encode all component control and timing behavior within a single flat control structure. Such monolithic control logic scales poorly as new components or protocols are introduced, especially when their timing semantics differ from those of existing logic. Consequently, simulators such as gem5 [20] commonly adopt an event-driven simulation model that decouples component behavior. In this model, the simulator maintains a global priority queue of events scheduled to execute at specific simulation times, where priority is determined by the time of execution. Time-varying behavior—CPU pipeline stages, cache and memory hierarchy timing, and interconnect activity—is expressed as events. Each event carries a scheduled execution time and a callback function that is invoked when the event is dequeued. As events can schedule additional events immediately or at future times, developers can extend existing components or add new ones with their own timing semantics without significantly increasing the complexity of the overall control and scheduling logic. Such a design impacts the simulator's performance characteristics, in particular its instruction throughput and code locality, as we discuss in Section IV.

B. SimPoint Analysis

SimPoint is a phase-based sampling technique that identifies a small set of representative intervals for detailed architectural simulation. Program execution is first partitioned into fixed-length intervals (*e.g.*, 10–100 million instructions), and each interval is characterized by a Basic Block Vector (BBV)—a high-dimensional vector whose entries record the normalized execution counts of each basic block within that interval [29]. Intervals with similar BBVs exhibit similar control-flow and microarchitectural behavior and are thus treated as belonging to the same phase. SimPoint clusters these BBVs (*e.g.*, using k-means) and, for each cluster, selects a single interval—the one closest to the cluster centroid—as the simulation point (SimPoint) representing that phase [9]. Each SimPoint is then assigned a weight proportional to the fraction of dynamic instructions covered by its cluster, and weighted averages over detailed simulations at these SimPoints yield accurate estimates of whole-program metrics (*e.g.*, IPC, miss rates) at a fraction of the cost of simulating the entire execution.

C. Profile-Guided Optimizations

Profile-Guided Optimization (PGO) is a feedback-directed compilation technique that leverages dynamic execution profiles to guide inlining, code layout, and other optimizations that static heuristics often tune poorly. Modern toolchains (*e.g.*, Clang/LLVM and the Intel oneAPI C/C++ compiler) typically follow a two-phase workflow: they first build an instrumented binary, run representative workloads to collect profiles, and then recompile the program using the collected profile data [14], [18]. In LLVM, profiles are stored in a `.profdata` file, created by merging one or more raw profiles collected at runtime [17]. In this work, we use either a single raw profile obtained from one SimPoint or a merged profile constructed from multiple SimPoint-level profiles. The profile records entry counts and basic-block/edge counts for each function, which the optimizer translates into branch probabilities and hot/cold path information; optional value profiling further captures frequently observed indirect-call targets and other hot values [19]. Guided by this information, the inliner prioritizes hot call sites, layout passes place hot paths contiguously, and hot-cold splitting moves rarely executed code out of the main text segment, reducing the effective instruction footprint. Value profiles also enable indirect-call promotion, converting hot indirect calls into guarded direct calls that unlock additional inlining and improve branch prediction. Collectively, these PGO-driven transformations reduce I-cache and iTLB pressure, which is particularly beneficial for large, branch-heavy codebases like gem5.

D. Sub-NUMA Clustering

Recent server CPUs increasingly adopt tiled or chiplet-based designs, in which cores, LLC slices, and memory controllers are grouped into on-die clusters that provide lower intra-cluster latency than cross-cluster accesses. For example, 4th-generation Intel Xeon Processors (codenamed Sapphire Rapids) employ a four-tile Extreme Core Count (XCC) layout, with each tile integrating its own partition of LLC and memory-controller resources [12]. As a result, memory access latency can vary substantially depending on where data resides, since requests may traverse multiple on-die hops over the interconnect when accessing remote tile resources.

Sub-NUMA Clustering (SNC) exposes these internal clusters as separate NUMA nodes (*e.g.*, SNC2 or SNC4), effectively binding cores with a subset of LLC capacity and local memory controllers into each NUMA domain [13]. In SNC4, for instance, a socket is partitioned into four NUMA domains, each owning roughly one quarter of the cores and LLC capacity. This organization can improve performance isolation and reduce effective memory access latency for workloads

whose working sets fit within a single domain's LLC and local memory resources.

III. Methodology

Simulation setup. We configure `gem5` with the X86 ISA in System-call Emulation (SE) mode. The simulated baseline is a single-core O3 CPU with a two-level cache hierarchy. The O3 CPU models an out-of-order superscalar core derived from the Alpha 21264 design. The cache hierarchy includes private 32 KiB L1 instruction and data caches and a unified 64 KiB L2 cache. We use SPEC CPU2017 as the workload suite to cover a diverse range of CPU behaviors; brief descriptions are provided in Table I, and we refer to benchmarks by their numeric identifiers hereafter. Because full-program simulation is prohibitively expensive, we apply SimPoint analysis [9] to identify representative execution phases. Unless otherwise stated, we simulate all SimPoints for each benchmark; in a subset of analyses (*e.g.*, Figs. 1, 2, and 10), we report results only for the most prominent SimPoint per benchmark.

Evaluation platform. All simulations run on the evaluation platform summarized in Table II. The system provides 12.8 GB of memory per core (6.4 GB per hardware thread with Hyper-Threading enabled) and can be configured with one, two, or four NUMA nodes. We use Intel VTune [11] and Linux `perf` [5] to collect performance metrics from hardware performance counters.

Compiler setup. We compile gem5 using clang 18.1.3 and select PGO settings that deliver the best performance. First, we collect profiles using instrumentation rather than sampling. Although instrumentation adds noticeable profiling overhead, it yields more accurate profiles than sampling-based PGO, which relies on statistical estimation. Second, among the three instrumentation-based PGO modes—Frontend PGO (FE-PGO), IR-level PGO (IR-PGO), and context-sensitive IR-PGO (CS-IR-PGO)—we use IR-PGO. FE-PGO is a legacy approach that typically provides lower-fidelity profiles, while CS-IR-PGO can increase instruction-cache pressure, which is undesirable for gem5 given its front-end bottleneck. We report speedups of PGO-optimized gem5 binaries relative to `gem5.fast`, compiled with `-O3` and `-flto` flags.

TABLE I: SPEC CPU2017 benchmark suite [25].

Workload	Description
`600.perlbench_s`	Perl interpreter
`602.gcc_s`	GNU C compiler
`605.mcf_s`	Route planning
`620.omnetpp_s`	Discrete event simulation – computer network
`623.xalancbmk_s`	XML to HTML conversion via XSLT
`625.x264_s`	Video compression
`631.deepsjeng_s`	AI: alpha-beta tree search (Chess)
`641.leela_s`	AI: Monte Carlo tree search (Go)
`648.exchange2_s`	AI: recursive solution generator (Sudoku)
`657.xz_s`	General data compression

TABLE II: Evaluation platform.

CPU	Intel® Xeon® Platinum 8460H CPU @ 3.80 GHz (Sapphire Rapids), 40 cores (80 threads)
NUMA nodes	1 NUMA node, 2/4 sub-NUMA nodes
OS (kernel)	Ubuntu 24.04.3 LTS (6.14.0-35-generic)
Cache hierarchy	32 KiB L1-I$, 48 KiB L1-D$, 2 MiB L2$ 105 MiB shared LLC (15 ways, non-inclusive)
Main memory	2×32 GB DDR5 DRAM per channel (total: 512 GB, 8 channels)

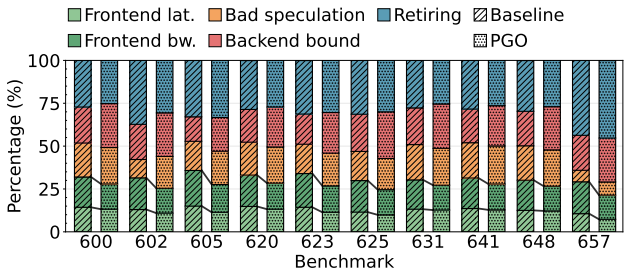


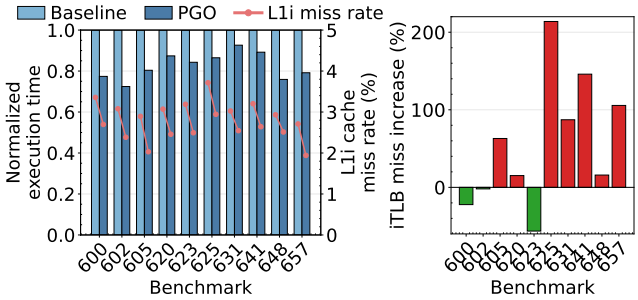
Fig. 1: Profiling results of simulation bottlenecks without (left bar) and with (right bar) PGO for each workload.

IV. Analyzing PGO Benefits in gem5

A. Frontend Bottlenecks in gem5

In this section, we first reevaluate the frontend bottleneck behavior of gem5 simulations—previously reported on Intel Cascade Lake [28]—on a newer CPU microarchitecture, Sapphire Rapids. Fig. 1 breaks down CPU cycles into five categories using VTune's top-down analysis [30]: frontend latency bound, frontend bandwidth bound, bad speculation, backend bound, and retiring. Despite substantial increases in per-core L2 capacity across CPU generations, the L1-I cache capacity has remained largely unchanged; consequently, the baseline gem5 binary continues to be dominated by frontend stalls and exhibits a pronounced frontend bottleneck, as shown in the left stacked bars for each workload.

We then apply PGO to the baseline gem5 binary using profiles collected from the same simulated benchmarks (*i.e.*, *self-profiling*). As shown in the right stacked bars of Fig. 1, PGO substantially alleviates the front-end bottleneck, reducing frontend stalls from 31.7% to 26.3% on average and



(a) Execution time and L1-I miss rate (b) iTLB miss increase

Fig. 2: Execution time and frontend bottlenecks without and with PGO for each workload.

shifting the dominant limitation toward the backend. This improved frontend efficiency directly translates into faster simulation. Specifically, Fig. 2 shows that PGO reduces L1-I cache misses by 33.2% on our evaluation platform and lowers overall execution time by 11.3% on average across all benchmarks. Our analysis shows that most of the speedup stems from standard compiler optimizations (but strengthened by PGO's profile information), rather than from PGO-specific transformations such as indirect-call promotion (ICP) or hot–cold splitting. Optimizations applied by the compiler include function inlining (90.6%), loop-invariant code motion (4.6%), global value numbering (1.8%), and loop unrolling (0.4%). Although inlining can improve frontend efficiency by reducing call/return overhead and increasing instruction locality, it may also inflate code size by aggressively inlining large, hot functions. Consistent with this trade-off, we observe a noticeable increase in iTLB misses in Fig. 2b, which can partially offset the gains from PGO.

We find that ablations of PGO—disabling either indirect-call promotion (ICP) or hot–cold splitting—produce identical performance to the full PGO configuration, indicating that these transformations contribute negligibly in our setting. This result is somewhat surprising, as ICP has been shown to deliver significant speedups in codebases with frequent indirect calls [1]. While the gem5 event queue would appear to be a natural beneficiary of ICP because events are dispatched through function pointers, our analysis shows that 84% of event callbacks resolve to *O3::tick()*, making the call target highly predictable for the hardware. This observation also reflects a design trade-off in gem5's O3 CPU model: prioritizing programmability over simulation speed. In particular, advancing the pipeline by invoking a per-tick handler—even when the pipeline is idle—resembles cycle-stepping rather than "pure" event-driven simulation, but it simplifies implementing complex interactions among microarchitectural components. As a result, the event queue becomes dominated by *O3::tick()* calls, many of which execute only a few lines of code and return without materially changing the simulated state when the pipeline is idle or stalled.

(O1) PGO effectively accelerates gem5 simulations by alleviating frontend stalls by increasing L1-I cache hits.
(O2) Compiler optimizations for gem5 are concentrated on function inlining, which inflates the binary footprint and increases iTLB pressure.

B. Rethinking Profile Selection in PGO

Despite the promising benefits of PGO, a widely accepted assumption—*PGO achieves the highest performance when the inputs used for profiling closely match those used at execution* [7], [18], [21], [26]—has long hindered its adoption in gem5 simulations. In gem5, the profiling inputs include not only the simulated workload (*e.g.*, SPEC CPU2017), but also the checkpoint and the simulation configuration, such as the target microarchitecture parameters. Consequently, collecting profiles and recompiling gem5 for every simulation configuration is prohibitively expensive. Moreover, in typical design-space exploration, each simulation input is executed only once due to the deterministic nature of architectural simulators, leaving little opportunity for profile reuse.

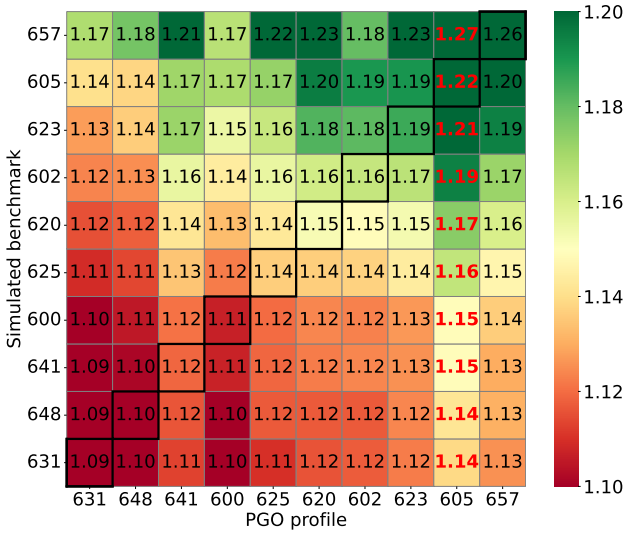


Fig. 3: Speedup of PGO over gem5.fast across all combinations of guest binaries and profiling workloads. Benchmarks are ordered by increasing average speedup. The maximum speedup in each row is highlighted in red text.

In this section, we present results that directly challenge this long-standing assumption. We evaluate all combinations of SPEC CPU2017 benchmarks (as guest binaries) and PGO-optimized gem5 binaries, where each PGO binary is compiled using a profile obtained by merging all SimPoint-granular profiles of a single benchmark. Fig. 3 shows a heatmap of speedups relative to the baseline gem5 binary, with benchmarks ordered by increasing average speedup.

Counterintuitively, self-profiling—using a benchmark's own profile to optimize gem5—does not consistently yield the best performance. The black-boxed diagonal entries in the heatmap are often inferior to off-diagonal ones, indicating that binaries optimized with other benchmarks' profiles can perform better. This behavior stems from the fact that PGO is hotness-driven, not performance-driven: it prioritizes frequently executed paths rather than paths that dominate end-to-end simulation time. As a result, benchmarks like 631.deepsjeng_s, whose hottest paths are not performance-critical, achieve limited speedups even under self-profiling. Furthermore, PGO lacks the ability to evaluate the trade-offs associated with individual optimizations. Each optimization exhibits two opposing effects (improved instruction locality but increased code size), yet PGO has no visibility into how these effects interact or converge in terms of overall performance.

More strikingly, some profiles are universally superior. For example, gem5 binaries optimized with the 605.mcf_s profile consistently achieve the highest performance across all guest benchmarks, whereas profiles derived from 631.deepsjeng_s perform the worst. This observation suggests that the hot

paths exercised during 605.mcf_s simulations overlap substantially with performance-critical paths in other workloads—even when their high-level execution characteristics differ. In other words, although the relative hotness of paths varies across benchmarks, gem5 simulations share a common set of performance-critical code paths.

The clear clustering of red and green regions in the heatmap further supports this conclusion. Benchmarks in the upper-right region exhibit strong alignment between path hotness and performance criticality. As this alignment improves, two effects emerge: (1) the simulated benchmark benefits more from PGO because a larger fraction of execution occurs on optimized paths, and (2) the resulting profile becomes more effective at optimizing gem5 itself, as it better captures globally performance-critical behavior.

(O3) PGO's hotness-driven decisions do not necessarily target the most performance-critical code paths.
(O4) gem5 simulations exhibit a largely shared set of performance-critical execution paths across different guest binaries, enabling some profiles to generalize well beyond the workload used for profiling.

C. Memory Simulation is the Primary Bottleneck

We find that memory simulation constitutes the dominant performance-critical path in gem5. By comparing the PGO transformations induced by the worst profile (631.deepsjeng_s) and the best profile (605.mcf_s), we observe that over 60% of the optimizations unique to the best profile fall within memory-simulation code paths—*e.g.*, L2 miss handling, MSHR management, crossbar communication, memory-packet queueing, and the memory controller. These optimizations deliver an additional 5–10%p speedup across diverse guest binaries. Notably, even when simulating 631.deepsjeng_s, where memory-simulation paths are not profiled as "hot", optimizing these paths still improves performance by 5%p, reinforcing that PGO hotness does not always align with end-to-end criticality.

Furthermore, the fraction of execution spent in memory simulation largely determines the attainable PGO speedup. Fig. 4 reports L2-miss and memory-writeback counts for each benchmark (normalized to the minimum), with benchmarks ordered by increasing average speedup as in Fig. 3. Benchmarks on the right generate substantially more memory traffic

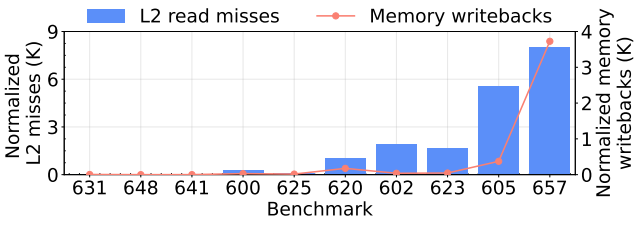


Fig. 4: Simulation statistics of L2-miss and memory-writeback counts across all guest binaries. Each metric is normalized to the minimum value among the binaries.

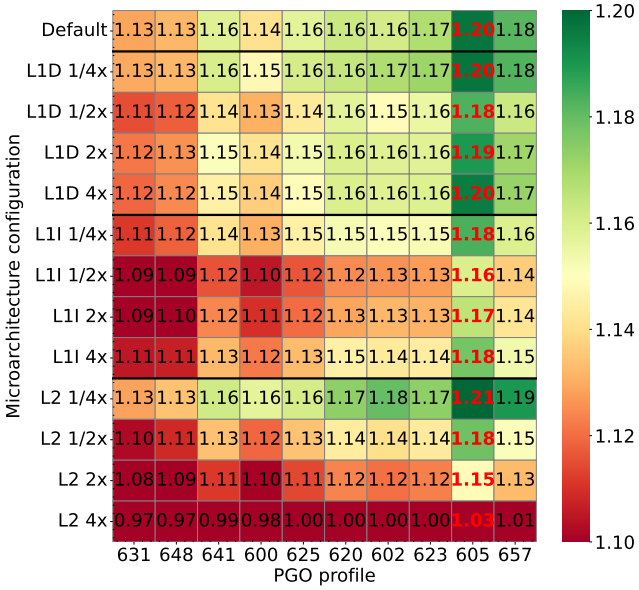


Fig. 5: Speedup of PGO over gem5.fast running 602.gcc_s while sweeping microarchitecture configurations. The highest speedup in each row is highlighted in red text.

and therefore see larger speedups, consistent with Amdahl's Law. In addition, because these workloads exercise memory-simulation paths more frequently, their profiles optimize those paths more aggressively, yielding higher-quality profiles that generalize better to other guest binaries.

This trend is further corroborated by sweeping cache-hierarchy parameters (Fig. 5). The PGO speedup is strongly governed by the L2 cache size—which directly controls the amount of memory-simulation work—and is comparatively insensitive to other microarchitectural settings. Smaller L2 caches increase memory traffic, amplifying the benefit of PGO; in contrast, once the L2 cache is increased by 4× and memory simulation becomes rare, PGO's gains largely vanish and can even reverse. In this regime, reduced I-cache misses are offset by increased iTLB misses, resulting in little net improvement (or slight slowdowns) in overall simulation time.

(O5) PGO is most effective when optimizing gem5's memory-related simulation codes.

D. Simpoint-Granular Analysis

We also perform a detailed SimPoint-level analysis to understand which additional components of the simulation can be improved with PGO. SimPoints provide the finest granularity that still preserves two essential properties: (1) a complete execution path with full runtime context for a single representative interval, and (2) distinct phase behavior. Unlike coarse aggregations (*e.g.*, grouping profiles by source files such as CPU-related objects), SimPoint profiles retain deep control-flow interactions across simulator components. In addition, it naturally captures coherent microarchitectural

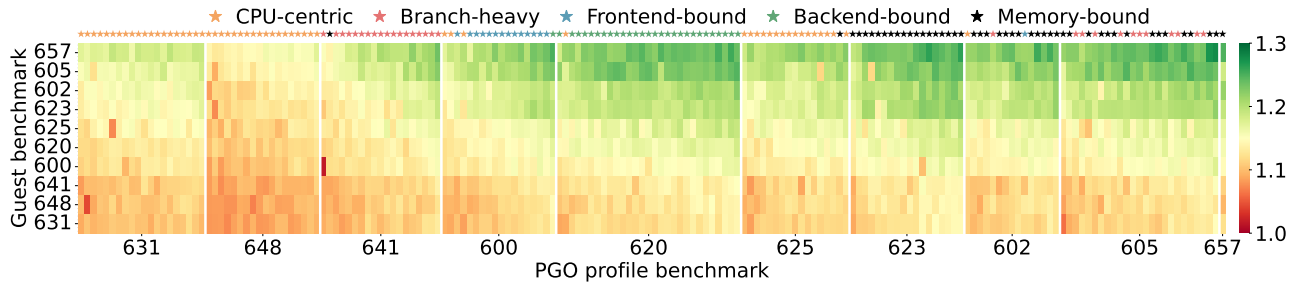


Fig. 6: Speedup of PGO at SimPoint granularity. Benchmarks are ordered by increasing average speedup when simulated, and SimPoint cluster memberships are annotated with colored stars.

behavior as each SimPoint corresponds to a distinct application phase.

To characterize SimPoints by architectural behavior, we cluster them into five categories. For each SimPoint, we construct a 23-dimensional feature vector comprising core metrics (*e.g.*, ROB-full rate, branch-miss behavior, FU utilization), cache metrics (*e.g.*, MPKI, prefetch activity), and memory metrics (*e.g.*, bandwidth, average load latency). We then cluster SimPoints using cosine similarity between feature vectors, producing five architectural clusters. TABLE III summarizes these clusters, their dominant characteristics, and the top-5 performance-critical SimPoints in each cluster; within each cluster, the SimPoint achieving the highest speedup is highlighted in bold, and clusters are ordered by decreasing average speedup.

Fig. 6 reports PGO speedups for every SimPoint from each benchmark. Benchmarks on both x/y-axes follow the same ascending order of average PGO speedup on simulation; within each benchmark, SimPoints are ordered by the average speedup they produce when used as a profile. The resulting heatmap shows that profile effectiveness is strongly correlated with cluster type: the best PGO binaries are overwhelmingly derived from memory-bound SimPoints (8 of the top 10), whereas CPU-centric SimPoints—exhibiting opposite behavior—do not appear even among the top 100. Moreover, SimPoints that serve as high-quality profiles also tend to achieve high speedups when simulated, yielding a concentrated high-speedup region (green cells) in the upper-right, consistent with the coarser benchmark-level trend in Fig. 3. Finally, SimPoints from a given benchmark typically fall into one or two clusters, depicted by stars atop each column, indicating that each benchmark exhibits a consistent architectural behavior trend and that SimPoint-derived profiles from the same benchmark yield similar performance with small variation.

TABLE III: Clusters and representative simpoints.

Cluster	Architectural aspect (key metrics)	Top-5 performance-critical SimPoints
Memory-bound *(47 SimPoints)*	High L1D MPKI; high memory bw; frequent LSQ_full	623: 1, 14, 16, **17** 605: 5
Backend-bound *(29 SimPoints)*	High ROB_full rate; low IPC and FU_busy	620: **6**, 8, 18, 28 600: 3
Frontend-bound *(15 SimPoints)*	High L1I / ITLB MPKI; high idle-cycle ratio	600: **6**, 13, 15, 18 602: 15
Branch-heavy *(30 SimPoints)*	High branch MPKI; high squash rate	605: 13, 16, 20, **23** 641: 2
Compute-centric *(59 SimPoints)*	High IPC / FU_busy; low branch density	625: 2, 3, **4**, 8, 9

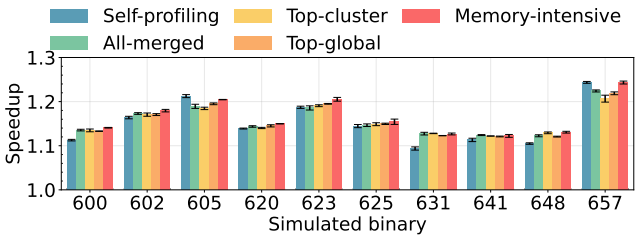
(O6) Memory-bound SimPoint profiles deliver the best performance, followed by backend-bound, frontend-bound, branch-heavy, and compute-centric profiles.
(O7) SimPoints within the same benchmark exhibit limited behavioral variation and are consistently classified into one or two clusters.

V. Universal Profile for gem5 Optimization

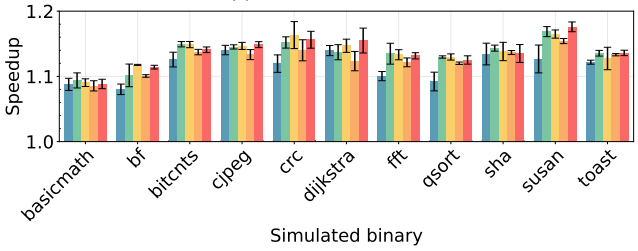
The observations in §IV motivate us to construct a single, optimal profile applicable across all gem5 simulations. This approach has several benefits: (1) it eliminates the need to recollect profiles and recompile gem5 for each simulation input (guest binary, checkpoint, and microarchitecture), substantially reducing the end-to-end cost of applying PGO in design-space exploration; (2) it enables *profile reuse* across one-off, deterministic simulation runs, making PGO practical even when each configuration is evaluated only once; (3) it captures and prioritizes the simulator's shared performance-critical paths (*e.g.*, memory simulation), improving robustness and delivering consistent speedups across diverse workloads and configurations; and (4) it simplifies deployment and evaluation by providing a single PGO-optimized gem5 binary that can be used as a drop-in replacement across experiments, improving reproducibility and usability.

A. Universal Profile Candidates

Based on the analysis in §IV-D, we construct three candidate profiles that aim to generalize across gem5 simulations.
Top-cluster. We select the highest-performing SimPoint from each of the five *clusters* in TABLE III. Each selected profile is intended to capture a complementary performance-critical execution region across the benchmark suite, rather than repeatedly optimizing similar microarchitectural behavior.



(a) SPEC CPU2017



(b) MiBench

Fig. 7: Speedups of gem5 simulations optimized with different profile candidates.

Top-global. We merge the *top-10* profiles across all benchmarks. As these profiles are predominantly memory-bound, the resulting merged profile is expected to emphasize common memory-centric optimizations while also incorporating profile-specific opportunities exposed by individual workloads.

Memory-intensive. Motivated by the observation that memory simulation is the primary bottleneck, we construct a *memory-intensive* profile by profiling the most memory-bound SimPoint in SPEC CPU2017 (*i.e.*, 623.xalancbmk_s) under a shallow cache hierarchy to amplify memory-system activity.

B. Evaluation

We compare the proposed universal-profile candidates against two baselines: *self-profiling* and *all-merged*. Self-profiling represents the conventional PGO workflow in which profiling and execution use the same benchmark; we construct each self-profile by merging all SimPoint-level profiles of the corresponding benchmark. All-merged uses a single profile formed by merging all SimPoints across SPEC CPU2017. We evaluate five PGO-optimized gem5 binaries using both SPEC CPU2017 and MiBench, as shown in Fig. 7a.

For SPEC CPU2017, all three universal-profile candidates consistently outperform self-profiling, except for heavily memory-bound benchmarks (*e.g.*, 605.mcf_s and 657.xz_s). Among the candidates, the *memory-intensive* profile delivers the best overall performance, exceeding self-profiling by 1.5–3.0% for most benchmarks and matching it within the error bars for 605.mcf_s and 657.xz_s. The largest gains occur for CPU-centric workloads such as 631.deepsjeng_s and 648.exchange2_s, where self-profiling tends to under-optimize shared, memory-simulation bottlenecks.

We observe little synergistic benefit from merging multiple SimPoint-level profiles (*all-merged*). Moreover, the *memory-intensive* profile—derived from a single SimPoint of 623.xalancbmk_s—outperforms even its own *self-profile*. These results suggest that profile merging, whether within a benchmark or across benchmarks, can dilute the most effective SimPoint-level signal—either by overriding performance-critical transformations or by introducing additional optimizations that bloat code size (**(O2)**). This trend is also reflected when moving from *top-cluster* and *top-global* to *memory-intensive*: as non-memory-bound profiles are progressively excluded, performance improves monotonically.

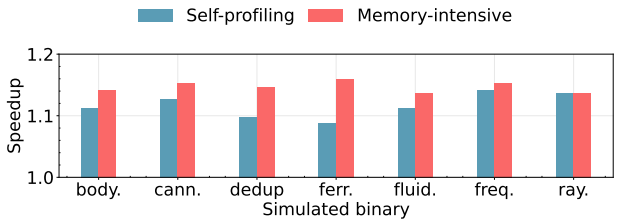


Fig. 8: Speedup of full system gem5 simulation

TABLE IV: Memory intensive profiles used in Fig. 9.

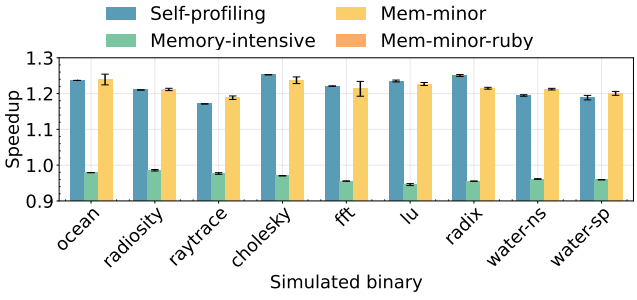
Profile name	Simulated binary	Simulated CPU core	Memory configuration
Memory-intensive	623.xalancbmk_s:17	Single O3	Classic
Mem-minor	623.xalancbmk_s:17	Single Minor	Classic
Mem-minor-ruby	623.xalancbmk_s:17	Single Minor	Ruby

C. Generalization

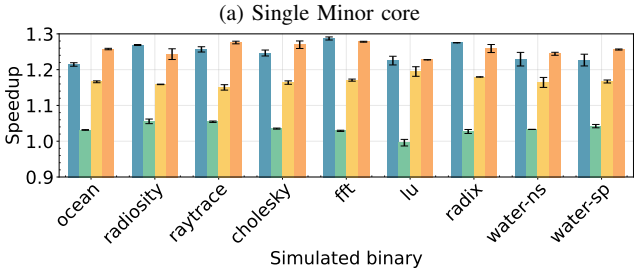
In Fig. 7b, we evaluate universal-profile candidates with MiBench to validate generalization beyond SPEC. MiBench is a set of small benchmarks targeting embedded systems, consisting of six categories: automotive, consumer, network, office, security, and telecomm [8]. Because these benchmark programs are small, we simulate each benchmark in full without SimPoint analysis, which made the variance across iterations larger than SPEC CPU2017.

All candidate profiles perform as well as or better than *self-profiling*. Among them, the *memory-intensive* profile still achieves the best performance on MiBench, outperforming self-profiling by 1.9% on average. Interestingly, profiles formed by merging across different clusters (*i.e.*, *top-cluster* and *all-merged*) improve performance compared to the *top-global* profile. We attribute this behavior to the smaller instruction working set of MiBench simulations relative to SPEC CPU2017. As a result, code inflation caused by diverse optimizations is less detrimental. At the same time, distinct optimization paths captured by different clusters can be combined synergistically, leading to additional performance gains.

We also evaluate the generalization of the profiles to a multi-core gem5 full system simulation, running the PARSEC [4] benchmark suite. Fig. 8 shows that our *memory-intensive* profile collected from a single core SE mode simulation works effectively for a multi-core FS mode simulation.



(a) Single Minor core



(b) Four Minor cores with Ruby cache coherency

Fig. 9: Speedups of gem5 simulations optimized with memory-intensive profile variants.

We further evaluate generalization beyond benchmarks by varying core types and memory configurations. Different cores and memory systems (*e.g.*, O3 vs. Minor cores, Classic vs. Ruby memory) use substantially different code paths, making a single universal profile insufficient across configurations. However, once a target simulation configuration is fixed, a corresponding memory-intensive profile can be used. Since the number of such configurations is limited, the overall profiling overhead can be effectively amortized. For example, we obtain different variants of memory-intensive profiles running the same memory-bounded workload (623.xalancbmk_s) on a single-core, shallow cache hierarchy as shown in TABLE IV.

We assess generalization of memory-intensive profiles using the Splash-3 multicore benchmark suite [23], as shown in Fig. 9. In Fig. 9a, we show simulation results on a single Minor core configuration. Applying an O3-derived memory-intensive profile degrades performance below the baseline, as it incorrectly prioritizes O3-specific code paths. In contrast, a Minor-specific profile (*mem-minor*) achieves a 21.6% speedup over the baseline, matching self-profiling performance across benchmarks and cache hierarchies.

In Fig. 9b, we simulate multiple Minor cores with the Ruby memory model. Again, the O3-based profile slightly outperforms the baseline. The *mem-minor* profile also fails to generalize to Ruby, achieving 6.36% lower speedup than *self-profiling*. However, a Ruby-specific profile (*mem-minor-ruby*) achieves a 25.7% speedup, matching self-profiling across benchmarks, cache hierarchies, and core counts. This improvement exceeds that of Fig. 7, suggesting that detailed memory modeling of Ruby provides additional optimization opportunities.

VI. Throughput-Oriented System Optimizations

In this section, we examine system-level concerns that emerge when running multiple concurrent simulations. While prior sections focused on optimizing single-instance performance, real-world use cases often involve executing multiple simulations simultaneously to maximize throughput for tasks such as regression testing or design space exploration.

A. Opportunities in Running Multiple Simulator Instances

The first concern is how to minimize interference on shared resources such as the LLC; the second is how to maximize core utilization by packing as many instances as possible within a fixed memory budget.

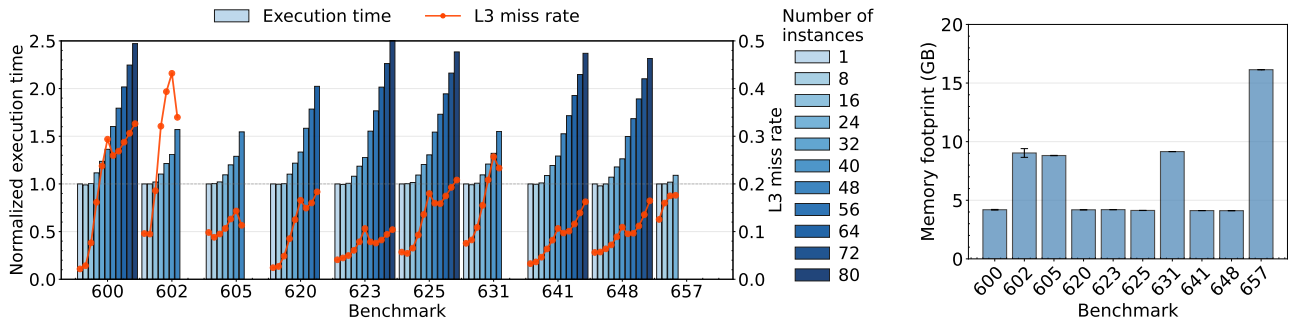
Shared resource. As the number of instances increases, contention on shared resources grows and throughput scales sub-linearly. Figure 10a shows the degradation in single-instance execution time and LLC hit rate as we increase the number of concurrent identical instances. Missing bars indicate cases where insufficient memory prevents launching additional simulations. As the system scales from 1 to 80 instances, execution time and LLC miss rate increase up to 2.5× and 15×, respectively. Beyond 40 instances, SMT activity, private-cache sharing, and core-level contention compound the slowdown, sharply reducing scalability.

CPU core stranding. Furthermore, naive scheduling of simulations frequently leads to resource stranding, where available CPU cores remain idle due to exhausted memory capacity. As shown in Figure 10b, SPEC 2017 CPU workloads require 4–10 GB of memory per simulation, imposing substantial demands on main memory. Large variance in per-instance memory footprints further exacerbates the stranding problem. Although resource management is well-established in cluster management systems and job schedulers, such systems rely on user-specified memory requirements, which are often conservatively overestimated [3], [6]. Moreover, even these systems cannot effectively manage intra-job scheduling within large allocations, as they depend solely on user-provided scripts.

B. Performance Isolation with Sub-NUMA Clustering

To mitigate contention on shared resources, we leverage Sub-NUMA Clustering (SNC) [13], which partitions a single physical NUMA node into multiple smaller NUMA nodes. In our experiments, we enable SNC-4 mode, which divides each physical NUMA node into four logical NUMA nodes, each with its own dedicated partition of the LLC and memory controllers. Moreover, SNC also reduces the average memory access latency by shortening the distance between cores and the LLC slices they access. However, it also reduces the available memory bandwidth per partition, as each partition has access to only a subset of the total memory channels.

Figure 11 compares throughput as a function of the number of concurrent simulation instances, with and without Sub-NUMA Clustering (SNC), and with and without Profile-Guided Optimization (PGO) applied to the simulator binary. On our test platform with 40 physical cores (80 hardware threads), throughput increases with instance count and peaks



(a) Execution time and overall LLC miss rates

(b) Memory footprint per instance

Fig. 10: System metrics with multiple simulation instances.

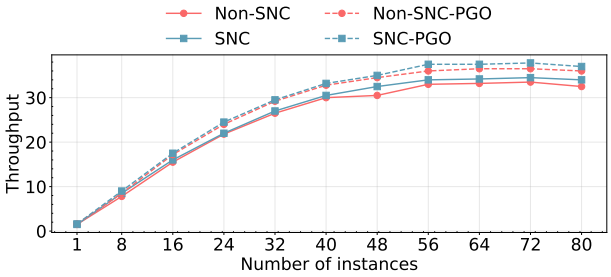


Fig. 11: Impact of PGO and SNC on throughput of multiple simulations as a function of the number of instances

at 72 concurrent instances. Beyond this point, performance degrades due to increased resource contention, as corroborated by the system-level metrics shown in Figure 10, which indicate a sharp rise in LLC contention when running more than 72 instances.

SNC has a limited impact on single-instance performance, where its primary benefit is reduced memory access latency. In contrast, SNC provides a substantial throughput benefit when multiple instances are run concurrently. In this regime, SNC reduces inter-instance interference by partitioning the LLC, allowing each SNC domain to operate with a dedicated cache slice. As a result, the peak throughput achieved with SNC enabled is, on average, 6 % higher than without SNC.

Although SNC also partitions memory bandwidth across domains, this does not negatively affect performance. The simulator is not strongly memory-bandwidth-bound, and the reduction in LLC contention dominates any bandwidth effects.

PGO further improves throughput in both SNC and non-SNC configurations, as discussed in Section IV. Notably, the benefits of PGO are more pronounced when SNC is enabled, indicating that improved instruction-level efficiency compounds the gains from reduced cache contention. Overall, the combination of SNC and PGO yields the highest throughput, achieving an average improvement of 20 % over the baseline configuration without SNC and PGO at the optimal concurrency of 72 instances.

C. Scheduling for Efficient Design Space Explorations

Deriving an optimal job schedule for large-scale design-space exploration is impractical. A full exploration typically consists of a three-level nested loop over benchmarks, SimPoints, and microarchitectural designs, which makes it infeasible to obtain exact execution times and memory footprints for every simulation instance in advance. In this subsection, we show that substantial throughput improvements can be achieved without such information—specifically, without per-instance execution-time estimates, complex optimization formulations, or dynamic scheduling—by simply reordering the loop structure.

In contrast to execution time, memory footprint is accurately predictable and primarily benchmark-dependent. Figure 10b reports the memory footprint of each SPEC CPU2017 benchmark, averaged across SimPoints and microarchitectural configurations. The low variance within a benchmark indicates that memory usage is largely insensitive to SimPoints and microarchitectural choices. Instead, it is dominated by the benchmark itself, as the memory footprint is determined by the size of the simulated memory state captured at checkpoint time.

This observation motivates batching simulations from different benchmarks to balance memory usage. In practice, simulation throughput degrades sharply once the system hits the memory limit, even when CPU resources remain available. Avoiding such resource stranding therefore requires preventing large-memory instances from being scheduled together. While accurately predicting execution time is difficult, approximate memory usage can be inferred from the benchmark identity alone. Consequently, interleaving simulations from different benchmarks effectively balances aggregate memory demand and significantly mitigates resource stranding.

We compare two scheduling policies. The baseline places the benchmark loop at the outermost level, followed by simpoints and microarchitectural configurations. In contrast, our proposed approach moves the benchmark loop to the innermost level, after the SimPoint and microarchitecture loops. Our simple scheduler launches jobs sequentially until it reaches either the CPU core or memory limit, with a small delay to

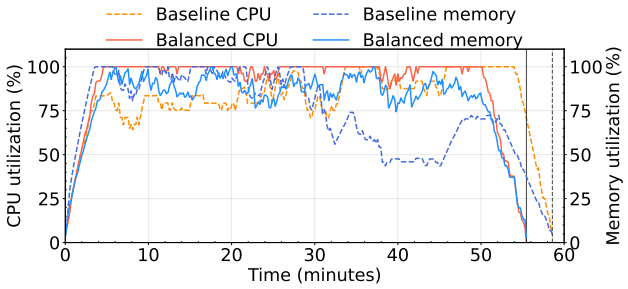


Fig. 12: Comparing scheduling policies of design space exploration.

avoid out-of-memory errors caused by instantiating multiple instances simultaneously. We do not pin the processes to specific cores and rely on the operating system scheduler. Figure 12 compares CPU utilization and memory usage of two policies over time. By balancing memory demand across concurrently running instances, our heuristic maintains high CPU utilization and achieves an additional 6% higher throughput compared to naive scheduling.

VII. Related Work

PGO-based compiler optimizations are effective for programs that experience front-end stalls, where large instruction footprints and indirect, configuration-dependent control flow defeat hardware prediction and static heuristics. Several works show that datacenter workloads benefit from PGO because their combined application, runtime, and kernel code easily reach multi-megabyte instruction working sets, causing frequent I-cache and I-TLB misses [2], [15], [16]. Specifically, Ugur *et al.* [27] exploits the fact that diverse datacenter applications traverse largely overlapping hot paths in the Linux kernel, and therefore applies a single *universal* kernel profile to improve performance across many services. Similarly, in event-driven frameworks, prior work shows that GUI systems and protocol stacks exhibit highly repetitive event and handler sequences, and that profile-directed specialization of dispatch paths and handlers is an effective way to cut control-flow overheads and front-end inefficiencies [22]. Our work brings this line of thinking to architectural simulation. We quantitatively demonstrate, for the first time, the effectiveness of PGO on an architectural simulator and propose a methodology that can be readily used in practice.

Prior work, such as *par-gem5* [31], reduces simulation time by aggressively parallelizing the simulator. However, tight control dependencies and synchronization overheads limit scalability, resulting in sub-linear speedups with respect to the core counts (*i.e.*, 25× speedup with 128 cores). Because architectural evaluation typically involves running hundreds of independent simulations, we instead prioritize maximizing the throughput of multiple single-threaded simulator instances.

VIII. Conclusion

This work accelerates slow gem5-based design-space exploration through practical compiler- and system-level optimizations. We show that the common assumption that self-profiling is optimal for PGO is flawed, and that performance-driven profile selection can outperform it in practice. By focusing PGO on shared, memory-simulation–critical paths, we make PGO effective without input-specific tuning. We further improve multi-instance throughput by mitigating shared-resource contention and CPU core stranding using simple system-level techniques. Together, these optimizations significantly improve both single-instance performance and aggregate throughput up to 27%.

Acknowledgments

This work was supported in part by grants from Samsung Advanced Institute of Technology (SAIT), the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-09942968 and RS-2024-00456287)), and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2025-02214322 and RS-2024-00405857).

References

[1] G. Aigner and U. Hölzle, "Eliminating virtual function calls in c++ programs," in *European conference on object-oriented programming*. Springer, 1996, pp. 142–166.

[2] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 462–473.

[3] J. Bader, F. Skalski, F. Lehmann, D. Scheinert, J. Will, L. Thamsen, and O. Kao, "Sizey: Memory-efficient execution of scientific workflow tasks," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2024, pp. 370–381.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[5] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, no. 1, 2010.

[6] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[7] Google. (2024) Profile-guided optimization — android ndk. Accessed: 2025-12-03. [Online]. Available: https://developer.android.com/ndk/guides/pgo

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[9] 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.

[10] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: Scalable and accurate parallel multi-core simulation," in *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance and Embedded Architecture and Compilation Network of ..., 2012, pp. 91–94.

[11] Intel, "Intel VTune Profiler," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, Accessed on 2025.

[12] Intel Corporation, "Fourth-generation intel® xeon® processor scalable family overview," https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html, 2022.

[13] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, https://cdrdv2-public.intel.com/821613/355308-Optimization-Reference-Manual-050-Changes-Doc.pdf, Intel Corporation, 2024, chapter 10: Introducing Sub-NUMA Clustering.
[14] Intel Corporation, "Profile guided optimization options," https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-2/profile-guided-optimization-options.html, 2024, intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference, accessed: 2025-12-09.
[15] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, "Whisper: Profile-guided branch misprediction elimination for data center applications," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 19–34.
[16] 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.
[17] LLVM Project, "llvm-profdata — profile data tool," https://llvm.org/docs/CommandGuide/llvm-profdata.html, 2019.
[18] ——, "Clang compiler user's manual: Profile-guided optimization," https://clang.llvm.org/docs/UsersManual.html, 2025.
[19] ——, "Instrumentation profile format," https://llvm.org/docs/InstrProfileFormat.html, 2025.
[20] 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.
[21] Oracle. (2024) Profile-guided optimization (pgo) for native image. [Online]. Available: https://www.graalvm.org/latest/reference-manual/native-image/optimizations-and-performance/PGO/
[22] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting, "Profile-directed optimization of event-based programs," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 106–116. [Online]. Available: https://doi.org/10.1145/512529.512543
[23] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 101–111.
[24] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.
[25] Standard Performance Evaluation Corporation (SPEC), "SPEC CPU2017 documentation," https://www.spec.org/cpu2017/Docs/, 2018.
[26] The Go Authors. (2024) Profile-guided optimization (pgo). Accessed: 2025-12-03. [Online]. Available: https://go.dev/doc/pgo
[27] M. Ugur, C. Jiang, A. Erf, T. Ahmed Khan, and B. Kasikci, "One profile fits all: Profile-guided linux kernel optimizations for data center applications," *ACM SIGOPS Operating Systems Review*, vol. 56, no. 1, pp. 26–33, 2022.
[28] J. Umeike, N. Patel, A. Manley, A. Mamandipoor, H. Yun, and M. Alian, "Profiling gem5 simulator," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 103–113.
[29] *BBV: Basic Block Vector Generation Tool*, https://valgrind.org/docs/manual/bbv-manual.html, Valgrind Project, 2010.
[30] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 35–44.
[31] N. Zurstraßen, J. Cubero-Cascante, J. M. Joseph, L. Yichao, X. Xinghua, and R. Leupers, "par-gem5: Parallelizing gem5's atomic mode," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

ARTIFACT APPENDIX

A. Abstract

This artifact contains a set of scripts to collect the profiles described in our work and apply them to the gem5 binary to produce the key results shown in the paper. Specifically, we include scripts to reproduce Figure 2 and Figure 3, which show the impact of PGO using self-profile, Figure 4, which shows the likely cause of the difference in the profiles among various inputs, and finally Figure 7, which shows the performance of different profile candidates.

B. Artifact check-list (meta-information)

- **Program:** gem5.
- **Compilation:** LLVM 18.
- **Run-time environment:** Ubuntu 24.04 with Kernel Version 6.14.
- **Hardware:** Intel 4th Generation Xeon CPU (Our Results on 8460), with at least 256 GB of DRAM.
- **Output:** Figures (stored in `results/figs`), Raw Data (stored in `results/data`).
- **How much disk space required (approximately)?:** 250 GB.
- **How much time is needed to prepare workflow (approximately)?:** 5 hours to generate checkpoints. Approximately 5 hours to collect required profiles, 10–12 hours to re-compile.
- **How much time is needed to complete experiments (approximately)?:** 24 hours to run all simulations required for a single iteration
- **Publicly available?:** Yes, at https://github.com/ece-fast-lab/ISPASS-2026-Gem5-PGO (except for the SPEC CPU 2017 benchmarks, which require a license).
- **Archived (provide DOI)?:** Will be provided in the camera-ready version.

C. Description

1) How to access: The set of scripts and detailed instructions for using them can be accessed at the public GitHub repository (https://github.com/ece-fast-lab/ISPASS-2026-Gem5-PGO).

2) Hardware dependencies: While our numbers reflect a Sapphire Rapids platform, the trends should hold on any relatively recent system. At least 256 GB of memory is required for the throughput experiments.

3) Software dependencies: The SPEC CPU 2017 benchmark is required.

From the directory where you installed the suite, run the following commands.

```
$ runcpu --config=x86 --tune=base --action=
    runsetup intspeed
$ export SPEC_BUILT_DIR=<PATH TO BENCHSPEC>
```

D. Installation

For the most up-to-date installation instructions, please refer to the README in the repository. The steps are reproduced here for reference.

```
$ git clone https://github.com/ece-fast-lab/
    ISPASS-2026-Gem5-PGO
$ cd ISPASS-2026-Gem5-PGO
$ source setup/init.sh
$ ./setup/gen-simpoints.sh
$ ./setup/gen-ckpts.sh
$ ./setup/gen-pgobin.sh
```

E. Experiment workflow

There is a corresponding script for each figure, with the details on how to run each script in the README in the repository.

F. Evaluation and expected results

Given that we are measuring the performance on real hardware, there may be some variance in the results across platforms. However, the key takeaways from each plot should remain as expected.

- **Figure 2:** Shows L1-I miss rate decreases under PGO, resulting in a 10–20% speedup with self-profile.
- **Figure 3:** Shows the relation between profiled input and the resulting speedup. This experiment is a sweep of running every SPEC benchmark workload with a gem5 profile from each one. Results should show that the profile from `605.mcf` performs the best across most benchmarks, even outperforming the self-profile.
- **Figure 4:** Shows the results from the simulated system to illustrate the nature of the workloads, and the source of the differences between profiles.
- **Figure 7:** Shows a key result from the work, the performance of the different profiles across a wide set of benchmarks.