Accelerating the Simulation of Parallel Workloads using Loop-Bounded Checkpoints

ALEN SABU*, Arm Ltd, UK
ZHANTONG QIU, University of California, Davis, USA
HARISH PATIL, Intel Corporation, USA
CHANGXI LIU, National University of Singapore, Singapore
WIM HEIRMAN, Intel Corporation, Belgium
JASON LOWE-POWER, University of California, Davis, USA
TREVOR E. CARLSON, National University of Singapore, Singapore

Efficient sampled simulation of multi-threaded applications remains a long-standing challenge with significant implications for evaluating modern computing systems. Existing methodologies are either limited in speedup (Time-based Sampling) or restricted to specific synchronization types (BarrierPoint). Workload-specific techniques tend to be rigid with respect to region selection, which may limit the overall speedup.

In this work, we aim to solve these challenges and propose a novel sampling technique for multi-threaded applications, called LoopPoint, that is both agnostic to the type of synchronization primitives used and scales with the similarity exhibited by the application. The methodology combines several vital features, including (a) repeatable, up-front loop-based analysis of the workload, (b) a novel clustering approach to take into account run-time parallelism, and (c) the use of simulation markers to divide the execution into measurable chunks based on the amount of work done, even in the presence of spin-loops. LoopPoint identifies representative regions that can be simulated in parallel to achieve speedups of up to 801× for the train input set of the multi-threaded SPEC CPU2017 benchmarks with an absolute geometric mean sampling error of just 1.48%. For the ref inputs, we estimate speedups up to 31,253×, demonstrating how the identification of application regularity and loops can lead to significant simulation improvements.

We further propose ROIperf, a hardware-based framework to enable rapid correlation of representative regions. Instead of long-running simulations, ROIperf allows for the performance measurement of full workloads and the representative regions directly on the hardware itself. This presents a practical methodology for large, realistic workloads where the prevailing simulation-based validation techniques are prohibitively slow. We demonstrate the efficacy of ROIperf across SPEC CPU2017 and NPB benchmark suites, showing strong correlation between hardware measurements and simulation predictions.

CCS Concepts: • **Computing methodologies** → **Simulation tools**; **Simulation evaluation**; • **Computer systems organization** → **Multicore architectures**.

Additional Key Words and Phrases: Sampled Simulation, Multi-threaded Workloads, Checkpointing, Performance Estimation

1 Introduction

Sampling is a well-known workload reduction technique that traces its roots back decades. From the earliest works [64, 70], researchers identified regularity in single-threaded applications and exploited that to reduce large applications

[*] This work was completed while the author was at the National University of Singapore.
Extension of Conference Paper [58]. In this work, we demonstrate the full-system simulation of LoopPoint on gem5 with implementation details, propose ROIperf – a novel technique for sample validation, and release the representative executable checkpoints or ELFies [52] of SPEC CPU2017 benchmarks.

Authors' Contact Information: Alen Sabu, alen.sabu@arm.com, Arm Ltd, UK; Zhantong Qiu, ztqiu@ucdavis.edu, University of California, Davis, USA; Harish Patil, harish.patil@intel.com, Intel Corporation, USA; Changxi Liu, National University of Singapore, Singapore, changxi_liu@u.nus.edu; Wim Heirman, wim.heirman@intel.com, Intel Corporation, Belgium; Jason Lowe-Power, jlowepower@ucdavis.edu, University of California, Davis, USA; Trevor E. Carlson, National University of Singapore, Singapore, tcarlson@comp.nus.edu.sg.

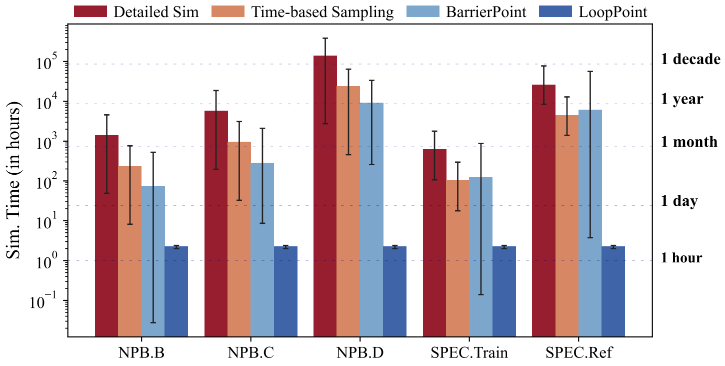


Fig. 1. Estimated time to evaluate the performance of multi-threaded benchmarks with different methodologies. The average result and error bars represent the estimated simulation time for all benchmarks in the corresponding suite and input sets, assuming infinite simulation resources (the longest simulation region determines the overall simulation time). Benchmarks were configured with 8-threads and passive OpenMP wait policy, assuming a total simulation speed of 100 KIPS.

into smaller application representative subsets. Because of the repeated execution of regions with similar behavior, these techniques have been shown to accurately predict the original workload behavior while significantly reducing the simulation time [64, 70].

In addition to workload sampling, researchers have developed several other techniques to reduce the overall amount of work required for detailed application simulation, including input size reduction [37] and benchmark synthesis [12, 22, 40, 41, 48]. Although each technique presents its benefits and challenges, sampling has emerged as a straightforward way to maintain the original application characteristics, accurately estimate performance, and reduce the overall simulation burden.

With the increasing number of cores in modern processors, multi-threaded applications can exploit a large amount of compute through task and loop parallelism. Simulating these large, multi-threaded applications is extremely difficult, even on modern simulators, due to the long runtime required (see Figure 1). Ultra-fast FPGA-based simulators [33] require detailed implementations and are capacity-limited, preventing the simulation of large processors and large parallel systems. Moreover, fast software-based simulators [13, 15, 61] still require a significant amount of time to simulate an entire large, parallel workload to completion. Multi-threaded applications are inherently difficult to analyze [1] due to the non-deterministic nature of thread scheduling, inter-thread contention, and complex behaviors emerging from system-level factors such as thread-to-core mapping and non-uniform cache distribution.

Some of the earliest multi-threaded sampling solutions prove effective when the threads themselves do not synchronize but can still interact with the memory hierarchy [68]. Any amount of synchronization requires thread progress to be measured in time (instead of by instruction count) to track the amount of progress or parallelism in the application. The move toward a time-based sampling methodology has led to the development of sampling techniques for synchronizing multi-threaded applications. These techniques [5, 16] describe one of the first generic sampling solutions for multi-threaded applications. However, the overall simulation speed is still bound to the total application length, which dominates the simulation time of this methodology. Later proposals introduced application- or synchronization-specific methodologies [18, 23, 24], which exceeded the performance of time-based sampling and allowed for the simulation complexity to be bound to application diversity rather than application length. Unfortunately, these methodologies are tied to specific application characteristics (the use of barriers [18] or tasks [23, 24]), and therefore do not represent a general sampling solution that covers all application types. In practice, the simulation overhead of large-scale multi-threaded applications using time-based sampling or BarrierPoint (assuming sufficient inter-barrier regions are available for simulation) can become computationally prohibitive, often resulting in unfeasible turnaround times, as shown in Figure 1. Clearly, current methodologies are insufficient for simulating the largest, most realistic benchmarks like the multi-threaded SPEC CPU2017 with the `reference` input set.

In this work, we aim to overcome the limitations of these prior works to enable synchronization-agnostic application sampling for multi-threaded workloads while still scaling the amount of work based on the representative nature of the application. To accomplish this goal, we present the *LoopPoint* methodology that reduces an application to a few representative regions, called *looppoints*, by taking into account several key factors like understanding (1) *where to simulate* which requires (1a) an accurate analysis methodology that can provide for reproducible analysis, and (1b) using a precise clustering mechanism that partitions the regions to reduce the workload into its representative components. In addition, our methodology presents (2) *how to simulate* the regions to allow the application to take advantage of the underlying hardware, while not constraining execution to a deterministic path [3] that might not exhibit true application behavior.

Beyond the simulation efficiency challenges, it is necessary to validate that the representative regions or regions of interest (ROIs) closely represent full-program behavior [27, 34, 69]. Traditionally, such validation is done by comparing the simulated performance of the entire program with the performance extrapolated from ROI simulations. However, since full-program simulation for most realistic applications is impractical to begin with, such simulation-based validation is limited to either short-running programs or using fast but inaccurate simulators.

Performance monitoring on native hardware offers a significantly faster alternative. Accurately identifying representative regions within an application typically involves iterative parameter tuning and re-validation. For example, SimPoint [64] demonstrates that applications like `gcc` may require up to five times more representative regions than others. Without extremely fast techniques, it becomes impractical to validate the efficacy of sampling methodologies for large-scale applications.

Although measuring full-program performance on native hardware is well-established [36, 55], isolating and measuring the performance of specific regions of interest presents a significant challenge. To keep simulation times in check, ROIs are often significantly smaller than the full-program, typically only a few million instructions, corresponding to milliseconds of execution on real hardware. Gathering precise, high-fidelity performance data solely for such small windows on native hardware is challenging. Loop-based ROI representations offer high accuracy and reproducibility [43, 57, 58], but hardware lacks native support for directly identifying such regions.

In an attempt to address this challenge, we present *ROIperf*, a methodology that incorporates lightweight instrumentation to achieve the necessary control and precision for isolating regions of interest. ROIperf utilizes Pin in probe mode [45], which has low overhead as it operates by patching an in-memory image of the application instead of using just-in-time (JIT) compilation, which can introduce significant performance overheads [7] that interfere with the workload behavior. ROIperf uses the Pin probe to merely hook into the application execution at the beginning and register callbacks based on hardware performance counters guided by the ROI specification.

Profile-based sample selection techniques, like SimPoint [64], assume identical program behavior across profiling and simulation runs, which is difficult to guarantee, especially for multi-threaded programs. Heterogeneity in hardware environments (for instance, varying ISA support leading to scalar versus vectorized runs), inconsistencies in system libraries, and timing-dependent control flow (for example, work stealing in parallel applications) can introduce discrepancies among multiple runs. PinPlay [53] guarantees identical execution through record-and-replay framework, but incurs significant overhead (≈ 50× slowdown), rendering performance counter-based evaluation inaccurate. Other approaches to improve repeatability include using static binaries, checkpoints [17], or ELFies [52]. However, none of these techniques guarantee complete reproducibility, particularly in multi-threaded scenarios where timing-dependent control flow and the resulting execution divergence happen more often [1, 54].

While ROIperf leverages native program execution to validate the samples or ROI, we acknowledge the inherent challenge of guaranteeing perfect repeatability across runs. However, the effects of this challenge can be minimized by executing both the sample selection and ROIperf measurements in controlled environments. In our evaluation, we show that ROIperf is effective in identifying and validating the regions accurately in most cases.

We make the following contributions in this work.

- We propose a representative simulation region selection methodology called *LoopPoint* suitable for the performance projection of multi-threaded programs based on using loop iterations as the unit of work.
- We develop a process to record a constrained application checkpoint for accurate analysis and subsequently simulate the unconstrained behavior of the workload during simulation.
- We introduce ROIperf, a lightweight framework for rapid evaluation of representativeness, and demonstrate its applicability to long-running workloads.
- We release the executable checkpoints, known as ELFies [52], of the SPEC CPU2017 benchmarks selected using the LoopPoint methodology and validated using ROIperf [60].

2 Fast and Generic Multi-threaded Simulation Requirements

Time-based sampling methodologies [5, 16] present the first workable solution to sample generic multi-threaded applications. However, the speed-ups achieved (up to 5.8×) using these methodologies are limited by the need to visit the entire application. To achieve high speed-up while maintaining accuracy during multi-threaded workload sampling, the key is the ability to (1) recognize representative regions in a generic way across multi-threaded workload types, and to (2) classify these regions considering application parallelism. To this end, we present a new application sampling methodology called *LoopPoint* that (a) uses loop iterations as the main unit of work, (b) utilizes constrained *pinballs* [53] (user-level checkpoints that allow for reproducible analysis), (c) employs heuristics to remove synchronization during analysis, but use them during simulation, and (d) performs unconstrained simulation of the selected simulation regions allowing for fast and accurate workload evaluation. Figure 2 shows the overall methodology.

Previous works like the BarrierPoint [18] methodology use inter-barrier regions as the unit of work, whereas the TaskPoint [24] methodology applies only to task-based applications that use task instances as the unit of work. Unfortunately, BarrierPoint, when used to sample large applications with a small number of barriers, can yield negligible simulation speed-ups. This can be common, especially while sampling realistic workloads for which the length of inter-barrier regions is a bottleneck. BarrierPoint, therefore, is not practical for such workloads. Figure 1 shows how the instruction count (and therefore simulation time) of an inter-barrier region grows with larger input sets of SPEC CPU2017 and NAS Parallel Benchmarks (NPB) [8] with 8 threads. BarrierPoint works well for NPB with the A input size [18], but as the input sizes grow, for classes C, D and E, inter-barrier regions become so large that it becomes impractical to use BarrierPoint for those input sets. The same is the case with SPEC CPU2017 using `ref` inputs.

LoopPoint instead uses loop iterations as the unit of work with the goal to apply to generic multi-threaded programs. The idea of using loop iterations as slices for single-threaded programs was proposed earlier [38]. Similarly, Kimura et

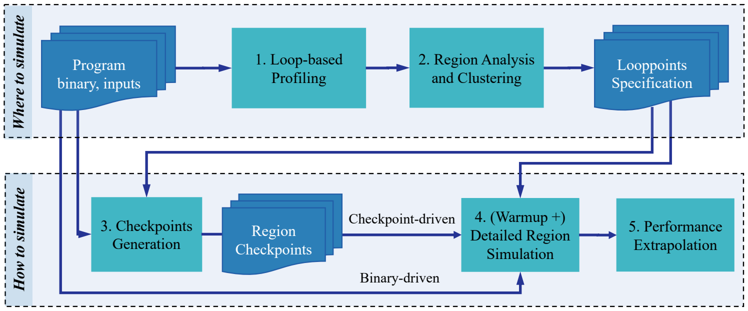


Fig. 2. LoopPoint-based region selection and simulation for multi-threaded workloads. The workload is captured for analysis and region selection based on loop information. The representative regions are simulated using a checkpoint-driven method as well as in a binary-driven unconstrained way, allowing for extrapolation of performance and other metrics of interest.

al. [35] applied loop iteration sampling to parallelized programs, but their approach relies on manual loop identification and approximates multi-core performance by extrapolating single-core real-machine profiles. With loop entries as slice boundaries, the simulation regions can then be specified using a *(PC, count)* pair for the starting and ending loop entry for each simulation region. By monitoring the amount of work, as represented by loops, and not instructions or barriers, we can isolate multi-threaded application representatives and understand the amount of global work completed. For multi-threaded programs, one additional constraint is that the loop entries that are chosen to start and end slices should be those doing meaningful work. Automatically separating loops doing real work from synchronization can be a daunting task. However, we can use application knowledge or synchronization mechanism details to filter out synchronization loops. For example, the Intel OpenMP run-time uses functions in the `libiomp5.so` library for synchronization; hence loops from that library should not be counted towards *work done* while profiling the application.

Where to simulate. As detailed cycle-accurate simulation can be time-consuming, researchers often use sampling to decide where to simulate by choosing small portions or regions of long-running program executions for simulation. Sampling requires (a) choosing the regions so that they are representative of the whole program behavior and (b) projecting the whole-program performance based on the simulation results of the selected regions. SimPoint [64] is a popular simulation region selection approach. It works by dividing the program execution into smaller slices and collecting an execution signature for each slice. *K-means* clustering is used to determine phases from slice signatures. One representative per cluster is then chosen with the weight corresponding to the cluster size. Since these representatives are designed to be micro-architecture independent, the signature collected for each slice needs to be dependent only on the program execution and not based on any micro-architecture dependent metric. Typical signatures used include the *BBV* (Basic Block Vector) which contains execution counts of various *basic blocks* (single-entry/exit code blocks). How to slice a program's execution into regions is an important decision. In our work, we keep slices of approximately similar sizes demarcated by loop entries. The region selection is based on the replay of a previously recorded whole-program execution as a pinball. According to the micro-architecture of the recording machine, the synchronization seen there can be different from the synchronization seen during unconstrained simulation. We, therefore, augment our region selection methodology to make a selection only on the real computation or work done.

How to simulate. A critical decision that the simulator developers need to make is how to simulate, i.e., how to connect the application in consideration to a simulator. The most commonly used methods are (1) *binary-driven* where a program binary is executed during simulation feeding instructions to the simulator, (2) *checkpoint-driven* where a snapshot of selected region memory/register state and a list of injection events are used to drive the simulator, and (3) *trace-driven* where an instruction-by-instruction recorded state is fed to a timing-only simulator. The choice of how to simulate depends on several factors, such as ease of deployment, cost of generation, and flexibility of the evaluation. For this work, we use both binary-driven and checkpoint-driven simulations for our evaluation, although the implementation itself is generic and supports any of these simulation methods. Checkpoints are easier to share among multiple users than program binaries whose execution might require complex setup and input availability. We propose to capture regions selected by LoopPoint as pinball [50] checkpoints to drive PinPlay-based simulators.

By default, PinPlay supports *constrained* replay of pinballs where the shared memory accesses among threads are repeated in the order captured during recording. Simulation based on such constrained replay will repeat the thread ordering based on the micro-architecture of the machine on which the pinballs were generated. However, we ideally want the target, simulated micro-architecture to decide the thread behavior during simulation. To achieve that, we also use binary-driven simulation of the regions selected by LoopPoint using stable (PC, count)-based boundaries defining those regions. Therefore, the simulation proceeds as though the region was executed natively on the simulated micro-architecture. Another technique to achieve unconstrained simulation using pinballs is to convert them to executable checkpoints, called ELFies [52].

3 The LoopPoint Methodology

In this section, we explain the different parts of the proposed methodology, LoopPoint. We start with an upfront analysis of the application to determine its behavior and to identify loops, as shown in Figure 2. This is a one-time step and we use the information collected here for clustering regions to choose representatives. The representative regions are then simulated with sufficient warmup. The simulation results enable us to reconstruct the overall application performance.

3.1 Selecting a Unit of Work

LoopPoint proposes a strategy that identifies regions (the unit of work) of interest in terms of work done by each thread. For an unmodified application with the same input set, the unit of work chosen needs to remain the same for each application execution regardless of the properties of the underlying hardware, although the number of instructions executed may vary each time. The generality of the chosen unit of work is crucial for application sampling as this determines the amount of simulation speedup achieved.

We consider the number of loop iterations as the unit of work done. Program loops are ubiquitous across application domains and the number of iterations of any particular loop doing real computation as opposed to synchronization can remain constant over multiple executions for an unmodified application and for a fixed input size. In a multi-threaded environment, we consider loop execution, ignoring spin-loops (one form of active synchronization), to compute the amount of work done. Spin-loops contribute to the IPC of the application and consume CPU cycles, however, they do not contribute to the meaningful work done by the particular thread (waiting cannot be considered work completed).

3.2 Understanding Parallelism

Program phase behavior is an important aspect to consider while sampling applications. A phase is a set of slices in a program's execution that shows similar behavior, regardless of where they appear within the execution.

Capturing BBVs is an essential way to understand the fingerprint of an application execution region. We consider the slice-size to be approximately $N \times 100$ million global (all-threads) instructions, that align with loop boundaries, for a N-threaded application. We ignore the instructions executed in spin-loops or any other synchronization code while collecting the BBVs. The end of a region specified by a BBV is the next loop entry once the instruction-count target is achieved. Although this can be implemented in several ways (as described in[38]), we do not currently differentiate between inner and outer loop markers, and do not restrict specific threads to indicate loop boundaries. The loop entries that serve as region markers need to be worker loops and not spin-loops. We assume that the spin-loops are found only in the synchronization library (for example, OpenMP) and therefore we end a region only at a loop entry that is present in the main image of the application. The per-region BBVs of each thread are concatenated into a longer, global BBV that represents a multi-threaded region. This guides the clustering phase when there are regions that exhibit non-homogeneous thread behavior.

There are a number of reasons to maintain sufficiently large per-thread slices (approximately 100 million instructions). If a smaller slice-size is chosen, a large number of simulation points may be required, and such regions are highly sensitive to warmup and aliasing issues [16]. At the same time, we also need to make sure that there are enough intervals in the application for the clustering algorithm to work efficiently [25].

While we profile an application for BBVs or any feature vectors, we make sure that all threads in the application make the same amount of forward progress during analysis. This is to stabilize the collected profile for any thread imbalance that is caused by external events on the host processor (and is unrelated to the analysis environment). We call this method to enforce equal progress between threads flow-control.

3.3 Marking Region Boundaries

Every region in an application has its boundaries at a loop entry. In the case of single-threaded applications, instruction count can be used to define regions reliably. However, for multi-threaded applications, this does not hold. We describe the start and end of each region as an ordered-pair (PC, count), where the *PC* is the address of the corresponding region boundary marker instruction and the *count* is the execution count of the marker. The value of count for a particular region size is invariant across multiple executions which represents the unit of work done.

3.4 Identifying Loops using DCFG

Loops are found often in typical applications and the number of loop iterations can remain constant for an unmodified application for a particular input over multiple executions. We employ a Dynamic Control-Flow Graph (DCFG) to identify the regions that represent loops. A DCFG is similar to a classical control-flow graph with a primary difference: Each edge of a DCFG is augmented with a trip count to indicate the number of times the edge was traversed. The source code locations whose executions correlate with a phase change are called a software phase markers [38]. The software phase markers identify the phase changes that occur in an application execution irrespective of the underlying microarchitecture. These phase markers need to repeat in number and order across multiple program executions so that this can meaningfully act as simulation region boundaries.

We choose headers of loops that are in the main image of the program assuming that the synchronization loops are in the libraries. The number of iterations of synchronization loops may vary across different program executions. The DCFG of the whole program execution is instrumented for loop header instructions to identify a subset of loops from the main image. Loop header instructions are instrumented to emit Basic Block Vectors (BBVs) after *slice-size* number of instructions. LoopPoint considers all loops, barring spin loops, regardless of their size. However, this heuristic can be modified to exclude innermost loops as region boundaries. One potential drawback of this approach is that it may produce regions with significant variations in interval sizes. Figure 3 shows a region identified using the DCFG. The region is from the `638.imagick_s.1` application, executed with train inputs and 8 threads.

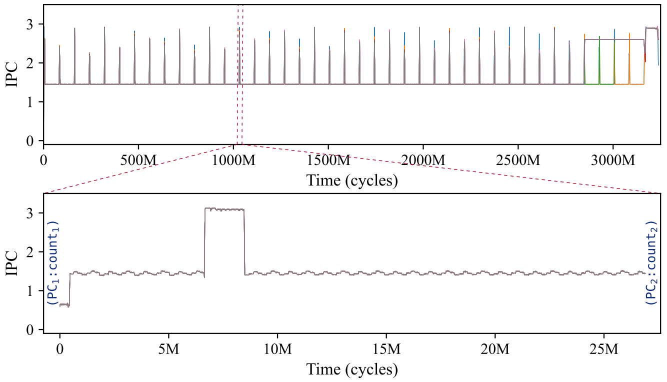


Fig. 3. An example of a representative region identified by LoopPoint. The top graph shows the variation of IPC over time for the full application run, while the bottom graph shows that of the chosen region. The boundaries of the region are marked using PC and count as shown inside the IPC graph of the region.

3.5 Clustering Representative Regions

Once an application is profiled, and region boundaries marked, we will have a collection of variable-length regions. These BBVs represent the state of the application, and also allow one to understand the amount of work accomplished by each thread. The amount of time the application executes becomes the combination of the amount of work executed in one quantum, together with the runtime attributed to that quantum. These quanta can then be clustered in order to identify similar work, and therefore identify similar runtime behavior. Although BBVs are used in this work, other feature vector information such as LRU stack distance vectors or LDVs can be concatenated [18] on a per-thread basis and can be used in this methodology. The BBVs are projected down to 128 dimensions by random linear projection to bring down the computing requirements for the clustering algorithm. We use K-means clustering technique [21] along with a BIC goodness criteria [63] to select clustering in a method similar to previous work [64]. The K-means algorithm requires specifying the maximum number of clusters, denoted as $maxK$, which we set to $maxK = 50$. Based on our experiments, this value is sufficient to identify the clusters for the applications we evaluate. However, some applications may exhibit a larger number of clusters, in which case this limit could underrepresent the full program behavior.

Since BBV data accounts for both parallelism and computational work, we can now cluster the regions, and use the resulting clusters to extrapolate the performance of the workload. We choose the BBV that is closest to the centroid of each cluster to be the representative of the cluster. We generate the checkpoints of the representative region from the original application based on the PC-based region boundaries and call them *looppoints*. The number of representative regions primarily reflects the phase diversity captured by the clustering process, whereas sampling error depends on how well the selected representative regions approximate the execution characteristics of the full program. Generating more regions (or increasing $maxK$) does not necessarily reduce error if those regions correspond to insignificant characteristics that are already captured.

3.6 Warmup

To maximize simulation performance, we will want to simulate each looppoint separately, in parallel, given enough resources. For accurate results, the microarchitectural state needs to be warmed up at the start of simulation of each region. There are several techniques [10, 46, 67] proposed to warmup cache state. For binary-driven simulation, we warm up each region from the start of the application to minimize warmup error. Likewise, for checkpoint-driven constrained simulation, we use a sufficiently large warmup region preceding the simulation region.

3.7 Runtime Extrapolation

Once the representatives are simulated, we can estimate the overall application execution time through the use of weight-based extrapolation. In this methodology, we use the percentage of work that this region represents, based on the instruction count of the entire collection of representatives that have been clustered together relative to the total amount of work done in the original application (quantum multiplier), to extrapolate the final runtime performance. The instructions that contribute to spin-loops are not considered here. The final step of this methodology uses the simulation results of these identified representatives, along with the multiplier, to reconstruct the overall workload runtime.

Our runtime extrapolation uses the below mentioned formula considering N looppoints identified as rep_1 to rep_N,

$$total\ runtime = \sum_{i=rep_1}^{rep_N} runtime_i \times multiplier_i.$$

The *multiplier* of a looppoint is the ratio of the sum of the filtered instruction counts from all of the regions that are represented by the looppoint to the filtered instruction count of that looppoint.

$$multiplier_j = \frac{\sum_{i=0}^{m} inscount_i}{inscount_j},$$

where m is the number of regions that are represented by the j^{th} looppoint.

We evaluate our region selection methodology by comparing the extrapolated runtime based on region simulation with the actual runtime based on the whole-application simulation to compute the prediction error. We demonstrate runtime extrapolation using the above formula, but this methodology can be used for any event of interest, such as cache and branch miss counts, for example.

3.8 Reproducible Application Execution for Accurate Analysis

The execution path of a multi-threaded application can vary from run to run due to several factors. One requirement to use this methodology is the ability to analyze a multi-threaded application in a repeatable way. Traditional execution environments do not support this type of execution to allow for reliable, reproducible execution. We leverage Intel's Pin [45] and Pinplay [53] tools to generate reproducible, constrained, multi-threaded execution snapshots, called pinballs, to allow for repeatable analysis. Pinballs are more advanced than a trace file in that they contain a snapshot of the execution state of an application (registers and memory). By replaying the pinball, we can analyze the properties of an application to collect the microarchitecture-independent execution signatures of the application. Note that these pinballs are intended strictly for architectural simulation and are not executed natively on the hardware.

3.9 Putting it All Together

Together, the combination of reproducible replay of applications, along with the identification and clustering of workload characteristics, allows us to build an end-to-end methodology to identify workload representatives for performance extrapolation. Previous works [18] have shown that extrapolation in this manner does apply to runtime, as well as other metrics of interest. The insights with respect to the identification of application parallelism, as well as the constrained, reproducible execution of the workloads allow us to analyze, cluster, and extrapolate multi-threaded workloads across a number of synchronization types.

One of the most significant benefits of a checkpoint-based methodology is the ability to substantially reduce the amount of work that needs to be simulated to estimate the entire application performance. Simulator performance relates directly to the required length and number of regions to simulate. In addition, checkpoints can be simulated in parallel, with enough resources available, speeding time-to-results significantly.

3.10 Workload Applicability

The methodology that we present here enables fast simulation of statically scheduled, generic multi-threaded workloads, irrespective of their synchronization mechanisms. This approach, however, is not suited for dynamically scheduled multi-threaded applications, as their thread interactions can vary from initial execution, potentially leading to inaccurate simulation results. Such applications could benefit from alternative methodologies [43] for sampling as they might not be able to be analyzed or sampled based on an upfront analysis of the application. Like other checkpoint-based methods like BarrierPoint, LoopPoint requires an upfront application analysis and assumes a static hardware configuration. This configuration is free from any run-time-dependent configuration changes or unexpected events that trigger a configuration change while the application is running. An example of a dynamic event is thermal throttling resulting in a dynamic voltage and frequency scaling (DVFS) event, which can affect the application performance and is runtime- and hardware-dependent. We address the problem of workload imbalance among the threads (a heterogeneous workload) by keeping per-thread information intact while clustering the individual regions.

4 Implementation Details of LoopPoint Methodology on gem5

Full-system simulation involves the entire software stack from applications through the operating system down to microarchitecture and devices. It evaluates novel designs and provides deeper insights into the system within a realistic environment. However, full-system simulation introduces new challenges to the sampling methodology due to the involvement of the operating system. To ensure accurate profiling, the address space must remain stable throughout the execution of the application. Kernel-level instructions, which can vary significantly across microarchitectures, should be excluded from the application phase analysis.

To evaluate LoopPoint methodology on the full-system simulation, we utilize the gem5 [13, 44], a cycle-level simulator that supports full-system simulation and is widely used in both academia and industry. In order to support the methodology, we introduce two new probe modules in gem5 to support the implementation of the LoopPoint methodology in full-system simulation: the LoopPoint Analysis module and the PcCount Tracker module.

The LoopPoint methodology involves three primary steps: phase analysis, checkpoint creation, and representative region simulation, as detailed in Section 3. In gem5, the phase analysis of the application is performed using the LoopPoint Analysis module to determine its underlying behavior and identify recurring loops. The representative regions are selected based on offline clustering results. We further utilize markers to create gem5 checkpoints of these regions. Finally, we run detailed simulations (with appropriate warmup) of these checkpoints using the PcCount Tracker module. As of gem5 version 24.1, LoopPoint functionality has been upstreamed and is readily available.

4.1 Phase Analysis

To accurately capture application phases using the LoopPoint methodology, we need to ensure the stability of the address space of the application and exclude all kernel instructions [68]. Before starting the simulation, we disable Address Space Layout Randomization (ASLR) in the kernel and obtain the address mapping of the application. This ensures the right address mapping of the application, which is especially important for the LoopPoint methodology, as virtual addresses are used to mark region boundaries and exclude synchronization libraries. The entire phase analysis is conducted in gem5 functional mode, which employs the atomic CPU.

For the LoopPoint Analysis module in gem5, we ignore all kernel instructions to prevent them from affecting the analysis of application phases. We also ignore synchronization library instructions since they do not directly contribute to the application's phase behavior, as detailed in Section 3. We only consider loops inside the application code base as reliable candidates for region markers. The analysis phase produces a list of regions along with their BBVs, and the region boundaries as (PC, count) pairs that can be used as the marker for each region.

4.2 Checkpoint Creation and Representative Region Simulation

After clustering the BBVs and selecting representative regions, we create gem5 checkpoints using the respective (PC, count) pairs. These pairs serve as markers to identify the warm-up start, simulation start, and end points of each representative region. In gem5, we maintain a list of these markers and monitor the committed instructions using the PcCount Tracker module. This module accurately detects markers during simulation, ensuring precise control over checkpoint creation and representative region simulation.

Typically, checkpoints are placed at the start of the warmup region. Upon restoring a checkpoint, gem5 executes the warm-up instructions before collecting statistics in the representative region. By the time simulation reaches the ROI, key microarchitectural structures (such as caches and branch predictors) will have been warmed up, resulting in more accurate performance measurements.

Since checkpoints are self-contained, simulations of representative regions can execute in parallel, significantly reducing the overall simulation time. The PcCount Tracker module continuously monitors committed instructions to precisely detect the end of warmup region (marking the start of representative region) and end of each representative region, enabling accurate performance measurements.

5 Sample Validation using ROIperf

This section describes the implementation details of the ROIperf methodology. We will further present and compare the usage models of the methodology for single-threaded and multi-threaded applications.

5.1 ROI Selection using Sampling

To select representative regions of interest (ROIs), we employ phase-based and loop-based approaches. For single-threaded applications, we leverage the PinPoints methodology [51]. This method builds upon SimPoint methodology [64] where an application is profiled to generate basic block vectors at every execution slice (indicating *unit of work*), and the resulting vectors are clustered to identify multiple phases in the application. A representative ROI is chosen for each phase, weighted proportional to the size of the phase it represents. Similarly, we use LoopPoint methodology [58] to identify the representative ROIs of multi-threaded applications. LoopPoint demarcates application regions based on loop iterations (instead of instruction counts) and clusters these regions to select ROIs. The ROIs are then used to guide architectural simulations. However, this approach relies on the assumption that the execution of ROIs can be reproduced precisely during the simulation as they were during profiling.

5.2 ROI Specification

The evaluation using ROIperf considers program repeatability to ensure ROIs remain representative. However, even single-threaded programs can often exhibit non-repeatable behavior [53]. One of the main reasons for this behavior is the differences in the microarchitecture that are used for profiling and performance measurements. Other reasons include changes in shared library versions and memory allocation patterns (load and stack locations). To address this and maintain ROI validity, ROIperf enforces identical shared libraries and memory allocation during measurement as observed during profiling. For example, the loading addresses of the shared libraries and the starting address of their stacks should remain the same. On Linux, this can be achieved by temporarily disabling Address Space Layout Randomization (ASLR).

A single-threaded ROI can be uniquely identified by the retired instruction count at its entry and exit points. Under deterministic execution (guaranteed through fixed libraries and disabled ASLR), capturing hardware performance counter values at these boundaries provides a sufficient basis for performance projection. In multi-threaded applications, the primary source of non-deterministic execution is the varying timing and interleaving of thread synchronization [1, 2, 53]. Consequently, dynamic instruction counts do not provide a stable or repeatable metric for defining ROI boundaries across different executions. The LoopPoint methodology [58] guarantees ROI repeatability by selecting units of work that begin and end at *worker* loop entries to avoid synchronization overhead and ensure consistent behavior across executions. LoopPoint defines ROIs using pairs of (PC, count), where PC represents the program counter address of the corresponding worker loop entry and count signifies the number of loop iterations. This approach ensures the invariant nature of worker loops to establish reliable ROI boundaries across executions.

5.3 ROI Handling in ROIperf

ROIperf aims to capture relevant hardware performance counter values at the boundaries of each region of interest (ROI). It achieves this by leveraging the Linux function `perf_event_open()` to program specific hardware performance counters. The required performance counters can be specified through an environment variable *ROIPERF_LIST*. This variable expects a comma-separated list of number pairs in the format *perftype:counter*. Here, *perftype* indicates the counter type (0 for hardware, 1 for software). The specific counter selection is based on values defined within the Linux header file `/usr/include/linux/perf_event.h`. For example, the *perftype:counter* pair `0:0` corresponds to *hw_cpu_cycles* (hardware counter for CPU cycles), while `1:2` refers to *sw_page_faults* (software counter for page faults).

ROIperf operates on an application along with its designated ROIs, and utilizes two primary methods to program hardware performance counters: (a) sampled counting of retired instructions or program counters and (b) continuous monitoring of performance counters specified with *ROIPERF_LIST*. The sampled counting is programmed using an overflow value and a callback function. When using instruction count-based ROIs, two counters monitor user-mode *PERF_COUNT_HW_INSTRUCTIONS*, with overflow values set to the start and end instruction counts of the ROI. Upon overflow, the callback function triggers, capturing the current system-wide time as Time-Stamp Counter (TSC) via the Read Time-Stamp Counter (RDTSC) and the values of all the performance counters programmed for continuous monitoring (defined by the *ROIPERF_LIST* environment variable). This continuous monitoring mode allows tracking

This can typically be done globally by modifying `/proc/sys/kernel/randomize_va_space`, or on a per-process basis by prepending the command line with `setarch x86_64 --addr-no-randomize`.

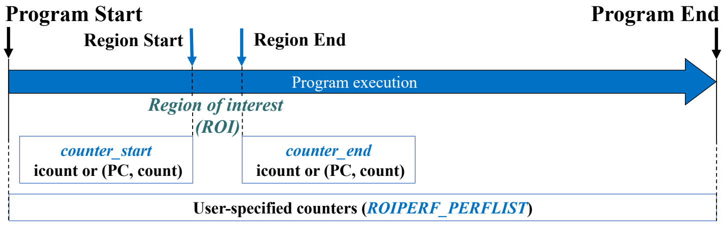


Fig. 4. The high-level operation of the ROIperf tool. At program start, user-defined performance counters are initialized. Measurements are activated at the start of ROI and remain active until ROI end. The ROI is demarcated using either hardware instruction counts obtained via *PERF_COUNT_HW_INSTRUCTIONS* or address (PC) counts implemented using *PERF_TYPE_BREAKPOINT*.

performance counters specified in *ROIPERF_LIST* alongside sampled counting. An illustration of the working of ROIperf is shown in Figure 4.

For ROIs defined by program counter (PC) and `count` values, a different approach is employed. Here, two counters with perftype set to *PERF_TYPE_BREAKPOINT* target the start and end PCs of the ROI. The overflow values are set to the corresponding `count` values specified for the ROI boundaries. Similar to sampled counting, the callback function upon overflow outputs TSC values and the values of performance counters from *ROIPERF_LIST*.

Our experiments revealed a significant performance difference between the techniques for programming performance counters used in ROIperf. While *PERF_COUNT_HW_INSTRUCTIONS* with overflow handling proved highly efficient across all tested x86 processors, *PERF_TYPE_BREAKPOINT* exhibited higher overhead. This overhead derives from the operating system trapping into the kernel on every execution of the programmed PC to check for overflow using a software counter. This frequent trapping can significantly perturb performance measurements, especially for ROIs with frequently executed PCs. To address this trade-off, we propose a hybrid approach for ROI specification. We recommend using *PERF_TYPE_BREAKPOINT* only for the ROI start, leveraging its precise triggering mechanism. For the ROI end, we suggest employing a relative instruction count-based *PERF_COUNT_HW_INSTRUCTIONS* approach. This combination prioritizes a precise start point while achieving faster monitoring for the ROI end (albeit slightly imprecise, particularly for multi-threaded scenarios). Since ROIperf ultimately focuses on the performance measurements between the start and end of the ROI, this approach offers an acceptable solution.

ROIperf exhibits limitations when dealing with multi-threaded programs, as it focuses on monitoring only the main thread (thread 0). Hence ROIperf starts hardware performance counters for the processor where the main thread is running. Pin in probe mode cannot monitor thread creation events. Consequently, there is no callback to ROIperf when child threads are spawned during program execution. Therefore ROIperf cannot monitor any children threads in a multi-threaded program. While ROIperf cannot directly monitor child threads, the captured TSC values still reflect the total execution time for the entire ROI, including the work done by child threads. This approach hinges on the assumption that the main thread remains active throughout the ROIs, which means the counters specified using *ROIPERF_PERFLIST* will be counted for the core/processor where the main thread is active.

Table 1. The primary characteristics of the simulated system used on Sniper.

Component	Features
Processor	8 & 16 cores, Gainestown-like
Core	2.66 GHz, 128 entry ROB
Branch predictor	Pentium M
Cache Hierarchy	32KiB L1i/L1d, 256KiB L2, 8MiB L2

Table 2. The configuration used for the full-system simulation on gem5.

Component	Features	
Architecture	x86	Arm
Core	8-core OoO @ 4GHz	8-core OoO @ 3GHz
Cache Hierarchy	64KiB L1d, 64KiB L1i, 1MiB L2	64KiB L1d, 64KiB L1i, 1MiB L2, 32MiB SLC
Memory	DDR4, 2400 MHz, 38.4GB/s	DDR4, 2400 MHz, 76.8GiB/s
OS	Ubuntu 24.04	Ubuntu 24.04

6 Experimental Setup

6.1 Simulation Infrastructure

In this work, we use Sniper multicore simulation infrastructure [15] (version 8.0) that supports PC-based simulation region specification. We configured Sniper to model a multicore out-of-order processor resembling the Intel Gainestown microarchitecture using an 8 or 16-core processor model to simulate 8 or 16-threaded (respectively) applications. The simulated system characteristics that we use are detailed in Table 1. We use the gem5 simulator [13] to evaluate the applicability of LoopPoint methodology for full-system workloads.

We compare the effectiveness of ROIperf in sample validation against performance evaluation using simulators. For our experiments with the SPEC CPU2017 benchmarks, we use an in-house simulator derived from Sniper [15], called CoreSim. CoreSim allows for rapid yet fairly accurate simulation of x86 many-core systems that use Intel SDE [30] as the simulation front-end. We configured CoreSim to simulate both Intel Skylake [19] and Intel Cascade Lake [4] microarchitectures.

6.2 Workloads Used

In order to evaluate the proposed methodology, we consider the SPEC CPU2017 [14] benchmark suite. SPEC CPU2017 is available in two different versions depending on the evaluation purpose: `rate` and `speed` [42]. The `rate` version is used to estimate the throughput of the underlying system whereas the `speed` version is used to estimate the runtime of the benchmark on the system. Unlike prior versions of SPEC benchmarks, CPU2017 includes a set of synchronizing multi-threaded programs that share memory consisting of OpenMP-compatible multi-threaded applications. We use the `speed` version of SPEC CPU2017 with `train` inputs and eight threads. The `train` input set is used so as to keep the full program simulation time to a reasonable length. As the detailed simulation of the full SPEC CPU2017 applications with `ref` inputs is not practical, computing the sampling error is also not feasible. Therefore, we utilize the `ref` inputs to estimate the potential speedup of the methodology in the paper.

All SPEC CPU2017 workloads except `657.xz_s` runs are 8-threaded. `657.xz_s.2` runs with 4-threads whereas `657.xz_s.1` runs as a single-threaded application. All the benchmarks in the SPEC CPU2017 benchmark suite are compiled using the Intel compiler toolchain (Intel Parallel Studio XE, version 2019 Update 2) with optimizations enabled (-O2) and debug information available for binary to source-level mapping.

We also use NAS Parallel Benchmarks (NPB) [9, 11] version 3.3 with OpenMP based parallelization [32] that use class C inputs. We evaluate all benchmarks in the suite with both 8 and 16 threads, but do not evaluate the npb-dc (data cube) benchmark because of the large amount of data generated by that application. These benchmarks are compiled using GCC 5.5 for applications in C and GFortran for Fortran applications with -O3 optimizations for the x86-64 architecture.

We consider both `active` and `passive` wait policies for thread synchronization of the SPEC CPU2017 OpenMP applications. We use the `passive` OpenMP wait policy to configure NPB benchmarks. In passive wait policy, the threads do not spin while waiting for other threads. Meanwhile in the case of active wait policy, the threads remain active and they consume processor cycles while waiting by executing spin-loops. The use of (PC, count) region specification can accurately represent a region over multiple runs even in the presence of spin-loops, which is not possible if the region specification is based on global or per-thread instruction counts.

6.3 ROIperf Settings

We show the applicability of ROIperf for single-threaded and multi-threaded applications. For the single-threaded case, we use PinPlay-based profiling methodologies involving the PinPoint [51] tool, derived from the SimPoint [64] methodology for sampling. For the multi-threaded case, we use the LoopPoint infrastructure [60] with default settings to identify loop-bounded regions. The regions are represented as basic block vectors (BBVs) and clustered using k-means clustering with a maxk of 50. For ROIperf-based evaluations, we chose two machines with Broadwell and Skylake microarchitectures.

6.4 DCFG and Basic Blocks

The Dynamic Control Flow Graph (DCFG) [71] is created by executing the program via a pin-tool enabled with the DCFG library [31]. Internally, the pin-tool *hooks* the control-flow instructions and records a count of each of the resulting edges throughout the execution of the workload on a per-thread basis. At the end of the execution, *fall-through* edges are created to ensure non-overlapping basic blocks. These basic blocks are guaranteed to have only one entry and one exit point and not overlap with each other. In this way, they differ from the basic block structures in Pin, which do not have these guarantees. The resulting basic blocks and the edges that connect them thus create a connected graph. From this graph, routine boundaries are identified based on call edges and heuristics to handle non-standard routines that are sometimes found in non-compiled code. Inside the sub-graph of each routine, the immediate dominators of each node are found. Loops are then identified using the immediate dominator relationships. The graph, including the identified routines and loops are recorded.

6.5 Handling Synchronization

OpenMP active runs, enabled by setting the environment variable OMP_WAIT_POLICY to ACTIVE [49], have threads busy-waiting at user-level (as opposed to using futex() in the passive runs). We replay a pinball that was recorded earlier for reproducible analysis for the generation of BBVs. If we directly use the recording we encounter the busy-waiting code that was originally executed by the application. However, the busy-waiting code can differ if the application is executed another time with different conditions. While busy-waiting consumes processor cycles, they do not contribute to the *real* work done by the program. Therefore, we ignore busy-waiting during BBV profiling, yet include it during simulation. Identifying busy-waiting code automatically [39] can be a challenge and is yet another research problem. In our methodology, we ignore the entire code from the relevant synchronization library (`libiomp5.so` in our case). Note that this idea can easily be extended to other compilers and threading libraries. For example, in the case of applications using pthread synchronization, we can ignore the code from the `libpthread` library. The filtered instruction count is up to 40% (for `657.xz_s.2`) fewer than the original instruction count for the active runs.

7 Evaluation

In this section, we present the evaluation results of LoopPoint methodology. We analyse the effect of various model parameters that make up the methodology. We also evaluate the accuracy and the speedup achieved using LoopPoint. Finally, we demonstrate the effectiveness of the ROIperf methodology.

7.1 Accuracy

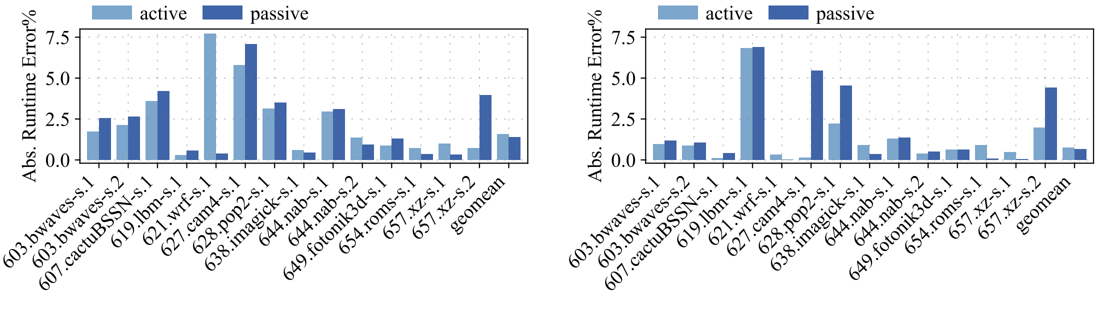
We show the accuracy of LoopPoint methodology by comparing the predicted runtime and the actual runtime of the application. The prediction error of our methodology is the percentage difference in the simulation performance of the whole application and the extrapolated performance making use of the performance of all the representative regions identified for the application.

Constrained and unconstrained simulations. The LoopPoint methodology is tested for applications using the `active` and `passive` wait policies, and the simulation results are given here. Synchronizing multi-threaded applications with `active` wait policy uses spinloops to synchronize the threads. Sampling such an application can be considered a difficult problem to solve. We ignore the instructions of spin-loops during BBV generation and clustering phases.

We perform binary-driven unconstrained simulations of the whole application as well as the representatives to measure the performance. In order to mark the region boundaries using (PC, count) correctly, we need to keep spin-loops away from being the region boundaries as mentioned earlier. We limit the region boundaries to be from the application code and not from any of the library code. Here, we make an assumption that the synchronization code can only be present in the libraries.

The region checkpoints are generated as pinballs which can be used for constrained simulation. We assume a large enough warmup region added to the representative region while generating the pinball checkpoint. However, using constrained simulation introduces artificial thread delays and are therefore not reliable for performance extrapolation. There are several ways to simulate these pinball checkpoints in an unconstrained way. One such method is to convert them to ELF binaries, called ELFies, as discussed in a prior work [52]. In this paper, however, we are not evaluating ELFies. Instead, we consider the region boundaries specified as (PC, count) to perform unconstrained simulation using the application binaries by providing perfect warmup before the start of detailed simulation. One caveat that we want to mention is that not all region boundaries specified using (PC, count) can provide stable regions. For instance, applications can have certain code blocks that are selectively executed with respect to the underlying microarchitecture. Such code blocks or PCs cannot serve as stable (PC, count) region boundaries.

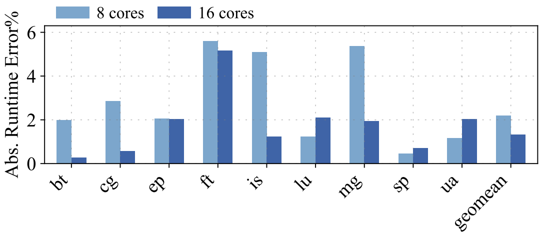
Results when simulating constrained simulation can be misleading and can lead to high errors. For example, we observe a runtime error for `657.xz_s.2` of up to 19.6% while simulating in a constrained environment. One of the reasons that using constrained simulation infrastructure can result in high error rates is that the simulation itself does not properly mimic the real application run. Instead, the application tries to replicate the behavior that was recorded previously on a specific machine. For instance, constrained execution forces spin-loops to be replayed even though this would not occur in a real execution. This introduces high error for applications, like `657.xz_s.2`, that have fewer



(a) Gainestown core

(b) Inorder core

Fig. 5. The runtime prediction errors of SPEC CPU2017 applications (train inputs) using active and passive wait policies that use 8 threads for unconstrained simulation. The y-axis represents the percent error in predicting the runtime of each of the applications.



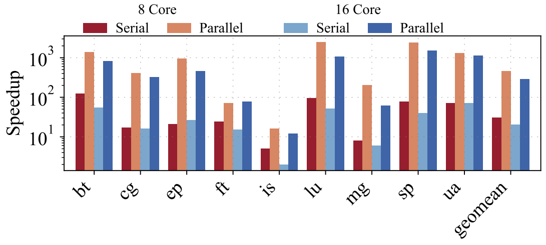


Fig. 6. The sampling errors in terms of extrapolating the runtime of NPB benchmarks that use 8 and 16 threads using passive wait policy and class C inputs.

Fig. 7. A comparison of actual speedups achieved by LoopPoint when the applications use 8 and 16 cores. Speedups listed for the NPB suite using the C input set and a passive wait policy.

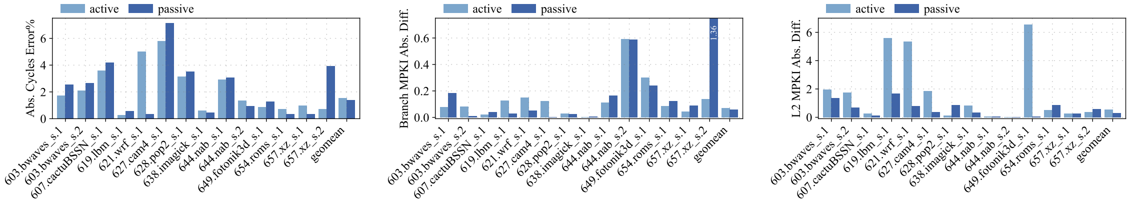
synchronization points compared to other applications in the SPEC CPU2017 benchmark suite, and therefore can see high variability from run to run.

The runtime prediction results (Figure 5a) using the unconstrained simulation of active applications yield geometric mean absolute error of just 1.58%, whereas that of passive applications is 1.40%. These error rates are comparable to previous sampling methodologies [18].

The looppoints identified are representative of the application across microarchitectural configurations. Our up-front analysis is solely based on architecture-level details, not microarchitectural settings or simulation details. Figure 5b shows the error in predicting the runtime of the same applications while simulated for an inorder core instead of the out-of-order Gainestown-like core, while keeping all other simulation parameters the default as in Table 1. The graph clearly shows that looppoints can be portable across microarchitectures. The out-of-order execution of Gainestown with large buffers, speculation, and aggressive prefetching make it more sensitive to active waiting, which can increase resource contention and coherence traffic, leading to higher errors for applications such as `621.wrf_s`. In contrast, the inorder core with simpler, non-speculative pipeline limits such interactions, resulting in lower and more stable errors under both wait policies, though its limited memory-level parallelism causes higher errors for memory-intensive workloads like `619.lbm_s`.

Varying the number of threads. We show that LoopPoint supports varying the number of application threads. Figure 6 shows the error rates while predicting the runtime of the NPB benchmarks. The applications are evaluated using 8 threads and 16 threads. Note that the applications using a different number of threads need to be profiled separately, as discussed in Section 3. We observe that the average absolute error obtained is 2.19% for 8-threaded applications while for the 16-threaded applications it is as low as 1.32%.

Comparison of hardware events. Figure 8 shows the performance prediction of several hardware events while simulated on an unconstrained environment for applications using `active` and `passive` wait policies. It is possible to extrapolate microarchitectural metrics like the number of cycles as shown in Figure 8a, branch miss rate or misses per kilo-instruction (MPKI) as shown in Figure 8b, and the miss rates or MPKI of different components in the memory hierarchy (Figure 8c) using LoopPoint. In Figure 8b and Figure 8c, we show the absolute differences in the metrics predicted, rather than the percentage error in prediction, because those metrics have small absolute values and a small difference can result in a high percentage error. Previous research [18, 51] has presented differences in a similar manner.



(a) Number of Cycles

(b) Branch MPKI

(c) L2 MPKI

Fig. 8. The prediction errors of various metrics for SPEC CPU2017 benchmarks using LoopPoint. The benchmarks use active and passive wait policies with train inputs and 8 threads, and are simulated in realistic unconstrained mode.

7.2 Speedup

We consider speedup in two different ways: theoretical speedup and actual speedup. Theoretical speedup is the reduction in the number of instructions (ignoring the instructions that contribute to spinloops) to be simulated in detail when using LoopPoint methodology. We also define the actual speedup as the reduction in the simulated runtime using LoopPoint with respect to the simulated runtime of the whole application. Serial speedup is the speedup achieved when all the representatives are simulated back-to-back. It is the overall reduction in work given the serial execution of both the full, and reduced workload. Parallel speedup assumes sufficient parallel resources, and evaluates the speedup given the execution of all regions in parallel.

In Figures 9 and 10, we see both the serial and parallel speedups for these applications. We obtain a maximum speedup of 801× for the applications with `train` inputs and 31,253× for the applications with `ref` inputs. The geometric mean serial speedup for applications using `train` inputs and `ref` inputs are respectively 8.74× and 195.94× whereas the geometric mean parallel speedup for the applications are 125.64× and 9,529.77× respectively for `train` and `ref` inputs. Consequently, the LoopPoint methodology allows for a significant reduction in simulation time and resources, bringing the turnaround time from months or years to hours.

In Figure 10, we compare the theoretical simulation speedup using LoopPoint and BarrierPoint for the benchmarks using `ref` inputs. Note that we do not plot the actual speedup values using the `ref` inputs. We first validate our

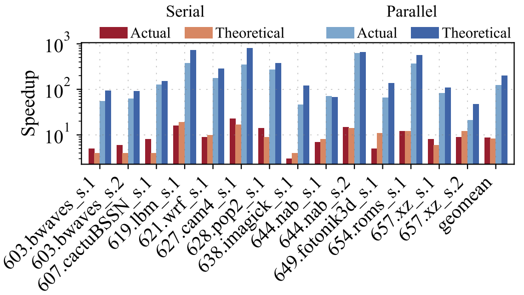


Fig. 9. A comparison of theoretical and actual speedups obtained by LoopPoint. The workload used is SPEC CPU2017 applications (active wait policy) using train inputs.

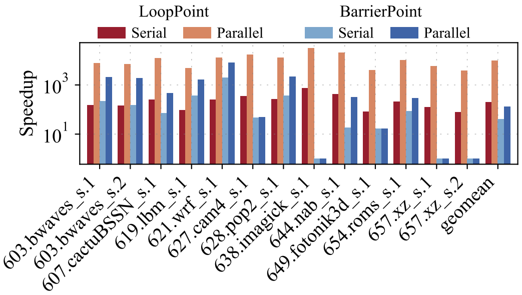


Fig. 10. A comparison of the theoretical speedups obtained for LoopPoint and BarrierPoint using SPEC CPU2017 applications (passive wait policy) using ref inputs.

methodology with `train` inputs, and by extension, we analyze and simulate `ref` input representatives to estimate the performance of the larger application with confidence. Unfortunately, it is not possible to validate the error rates for applications with `ref` inputs because the full runs take too long to simulate (a few months to years as shown in Figure 1).

We observe that LoopPoint consistently achieves good speedup whereas BarrerPoint lags behind for a number of applications. LoopPoint is able to reduce the application into representative regions that can finish simulation in a reasonable time. Additionally, with the BarrierPoint methodology, there is no guarantee on the size of a representative region. For example, the 8-threaded `638.imagick_s.1` benchmark has a very large inter-barrier region (93.06 B instructions) that is comparable to the size of the entire application (93.35 B instructions), defeating the purpose of sampling. However, there are a few applications for which BarrierPoint outperforms LoopPoint. Those applications have a large number of barriers and the inter-barrier regions are typically smaller than the LoopPoint regions. BarrierPoint is unsuitable to evaluate both of the `657.xz_s` applications as they do not contain barriers at all. Overall, a hybrid approach can be chosen to speed up smaller applications, but LoopPoint provides the first methodology to allow for generic sampling of applications that results both in a high simulation speedup and accuracy.

We also show the speedup achieved using NPB applications in Figure 7. LoopPoint achieves significant simulation speedups while the applications are evaluated for 8 threads and 16 threads. The maximum parallel speedup achieved while evaluating the 8-threaded applications is 2,503× with a geometric mean of 462.81×, whereas for the 16-threaded applications, the maximum speedup achieved is 1,498× and a geometric mean of 285.47×. Do note that NPB applications are less complex and more repetitive in nature than SPEC CPU2017 applications. Therefore, the error rates are lower and the speedups achieved are larger when compared to the `train` inputs of the SPEC CPU2017 suite.

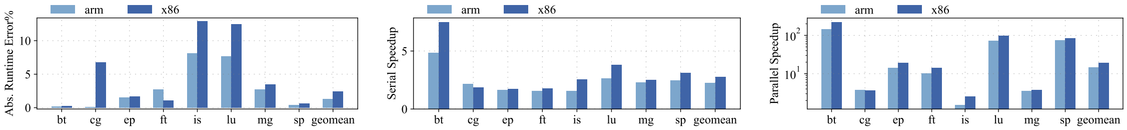
7.3 Full-System Simulation

To evaluate the effectiveness of the LoopPoint methodology in full-system simulation, we utilized both the LoopPoint Analysis and PcCount Tracker modules implemented in gem5. Specifically, we demonstrate this evaluation by applying the LoopPoint sampling process to the NPB benchmark suite. We used the input class A benchmarks for our experiments, as it would be impractical to run the baseline full-system simulations of class B or larger benchmarks with 8 threads to completion within a reasonable amount of time. In addition to evaluating in a full-system environment, we also tested the methodology across different ISAs and various realistic system configurations, as detailed in Table 2. In all experiments, Ubuntu 24.04 was used as the guest operating system.

The phase analysis and checkpoint creation of the selected representative regions are conducted in gem5 functional mode. The checkpoints are then simulated in detailed mode, using the configurations as shown in Table 2. Finally, we evaluated the accuracy of the LoopPoint methodology by comparing the predicted runtimes against baseline simulations. Figure 11a illustrates the relative prediction error for the NPB class A benchmarks.

The prediction errors are slightly larger for the class A benchmarks because there are too few representatives to build an accurate sample. For example, the methodology selected two representative regions for the benchmark *cg* and one for *is* as only eight total regions were discovered for *cg* and three for *is*, respectively.

We also evaluated the speedup for this particular experiment in Figure 11. The speedup is calculated based on the reduction in the number of instructions that must be simulated in detailed mode compared to the full baseline simulation. Due to the limited regions discovered for class A benchmarks, the selected representative regions constitute a large fraction of the total regions discovered. Therefore, the speedup for this particular experiment is limited. It is important to note that the speedup observed with the LoopPoint methodology primarily depends on the number of samples required to achieve a target accuracy, rather than on the simulator itself.



(a) Prediction error

(b) Serial Speedup

(c) Parallel Speedup

Fig. 11. LoopPoint prediction error and speedup comparison for NPB class A with eight threads in full-system mode.

7.4 Validation using ROIperf

7.4.1 Single-threaded Applications. We first simulated the binaries running train input with CoreSim in two ways: (a) for the entire program execution and (b) once each for each ROI selected by PinPoints (specification based on instruction count). Prediction error for each benchmark was computed using the simulated runtime, full-program, and region-projected. The longest-running full-program simulation took five weeks to finish. We then used ROIperf using the exact region specification and found prediction errors on two different test machines, one with a Broadwell x86 processor and another with a Skylake x86 processor. We evaluated ROIperf with the full-program and each region and computed prediction error based on cycles-per-instruction (CPI) values reported. We conducted multiple measurement trials and used the average values for our analysis. By executing the workloads directly on hardware using performance counters rather than a simulator, the entire evaluation was finished in hours. Figure 12 reports the prediction errors for simulation and ROIperf-based validation. We see that while the absolute prediction error values differ, the trends in prediction errors are the same between simulation-based and ROIperf-based validation. This gives us confidence in using ROIperf as a much faster alternative to simulation-based ROI validation.

SPEC CPU2017 runs with *ref* input are much longer running compared to train input runs. Simulation-based validation for ref input is therefore not practical as it would take a number of months to finish full-program ref runs simulations with CoreSim. This is where ROIperf-based simulation adds value. Since we are using native hardware

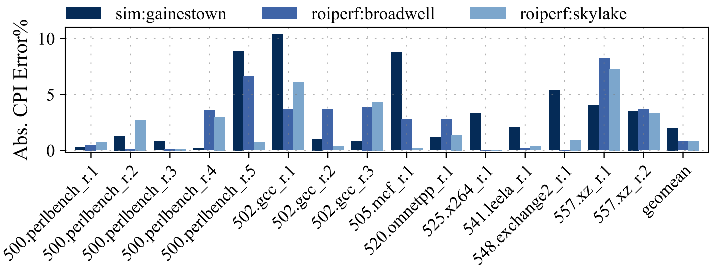


Fig. 12. Sampling error in predicting cycles-per-instructions (CPI) for single-threaded workloads from the SPEC CPU2017 suite using train inputs. The errors were measured using both a cycle-level simulator and ROIperf tool on Broadwell and Skylake machines.

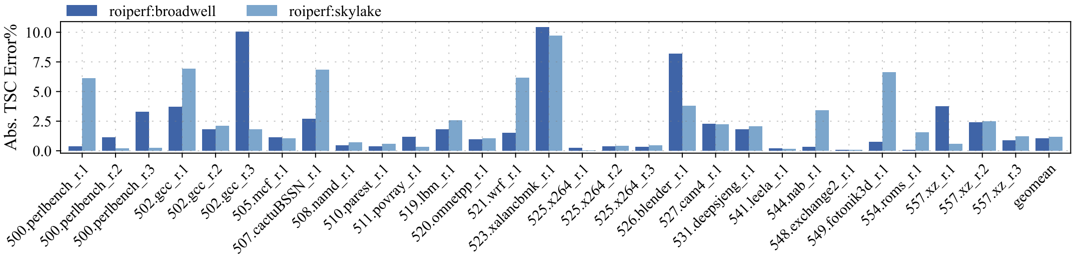


Fig. 13. Sampling error in predicting the TSC values of the single-threaded SPEC CPU2017 benchmarks using ref input. The measurements were conducted on Broadwell and Skylake machines.

as the simulator, the evaluation times are much shorter. Figure 13 reports the prediction errors for ROIperf-based validation of SPEC CPU2017 ref input runs on running Broadwell and Skylake servers.

7.4.2 Multi-threaded Applications. Figure 14 shows a comparison between the TSC prediction error using ROIperf and runtime prediction error using CoreSim. The ROIs were simulated on CoreSim with Cascade Lake microarchitecture specifications. We use 8-threaded SPEC CPU2017 benchmarks that use train inputs for this evaluation. The benchmarks use active thread wait policy. We can observe similar trends in the estimation errors.

We repeat the comparison of prediction errors from ROIperf and simulation for NAS Parallel Benchmarks (NPB) class C input size. Figure 15 shows the runtime prediction errors obtained from simulation (Sniper:Gainestown), and prediction errors for user-level hardware CPU cycles and TSC using ROIperf. Again the error bars show similar trends which signify the reliability of the results obtained using ROIperf.

8 Related Work

While various architectural simulators [13, 15, 61] enable detailed performance estimation, the computational expense of cycle-accurate simulation remains prohibitive for large-scale workloads. Consequently, workload reduction techniques are employed to reduce the total number of instructions that require timing-accurate simulation while maintaining statistical confidence. Workload sampling has been an active research area for decades, with several methodologies

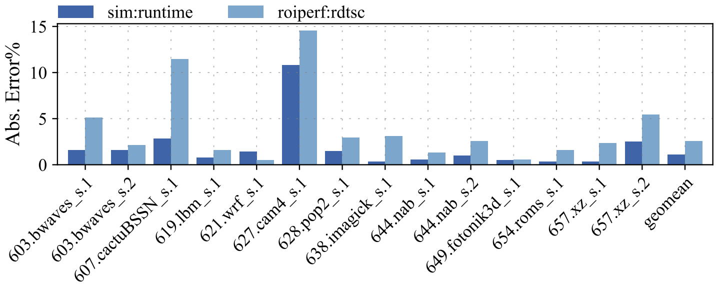


Fig. 14. A comparison of TSC estimation error using ROIperf and runtime estimation error using CoreSim simulator. We use SPEC CPU2017 benchmark with 8 threads, train inputs, and active wait policy. The ROIs are identified using LoopPoint methodology.

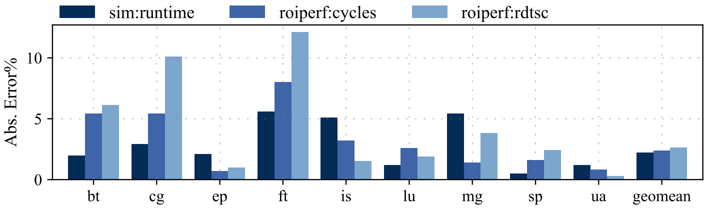


Fig. 15. A comparison of simulation-based prediction errors with ROIperf results for both HW_CPU_CYCLES and TSC projections on a Skylake Server. We use NPB benchmarks that use class C inputs, 8 threads and passive wait policy.

proposed to reduce the simulation time and resources for CPU workloads [5, 6, 16, 18, 24, 26, 29, 56–59, 62, 64, 65, 68, 70]. SimPoint [64] and SMARTS [70] are widely used single-threaded sampled simulation methodologies. SimPoint leverages microarchitecture-independent clustering of code signatures, whereas SMARTS employs rigorous statistical sampling to capture the dynamic interaction between the binary, microarchitecture, and operating system. SimFlex [68] introduces a full-system multiprocessor simulation infrastructure that uses checkpoint-based statistical sampling, while time-based sampling [5, 16] and BarrierPoint [18] introduce sampled simulation methodologies for multi-threaded workloads. Accurate sampled simulation necessitates the restoration of microarchitectural state, particularly for large structures like caches and branch predictors. To this end, various warmup methodologies [10, 20, 28, 46, 47, 66] have been proposed to minimize the bias introduced by cold-start effects.

9 Conclusion

The need to understand larger, more complex multi-core processors continues to increase. This becomes even more critical as the multi-core processors (and the serial code) tend to be the bottleneck in highly parallel applications. General-purpose applications are found on embedded devices, mobile phones, and back-end data center servers. While each platform has its requirements and demands, the need for an accurate understanding of the applications at hand is clear. Simulation solutions alone are insufficient because of the significant slowdown (10,000× or more [13]) seen when simulating applications with industrial-quality simulators. Simulation solutions today require alternatives like sampling to reduce the workloads to realistic simulation times. However, current sampling solutions either target single-threaded workloads or are only applicable to specific workload types.

In this work, we present a generic multi-threaded sampling methodology, one that considers the inherent parallelism of the application and allows for the automatic reduction of workloads to sizes that are on the order of the representatives of the workloads themselves. We demonstrate how our classification methodology automatically partitions the workload into representatives and allows us to predict the performance of the workloads at hand with high accuracy. We also introduce ROIperf, a technique to rapidly assess the quality of representative regions. ROIperf leverages hardware performance counters to validate the representativeness of selected samples, which can be used in several ways to study the workload characteristics, core interactions, and cache behavior without requiring a simulator.

Acknowledgments

This work has benefited greatly from the constructive feedback from several individuals, particularly our colleagues at the National University of Singapore and Intel Corporation. This research was supported by the Ministry of Education, Singapore, under Tier 3 Award MOE-MOET32024-0003, and by a grant from Intel Corporation. Additional support was provided by the U.S. National Science Foundation under Grant No. 2311888 and by the U.S. Department of Energy.

References

[1] A.R. Alameldeen and D.A. Wood. 2003. Variability in architectural simulations of multi-threaded workloads. In *International Symposium on High-Performance Computer Architecture (HPCA)*. 7–18.

[2] A.R. Alameldeen and D.A. Wood. 2006. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro* 26, 4 (2006), 8–17.

[3] Alaa R Alameldeen, Carl J Mauer, Min Xu, Pacia J Harper, Milo MK Martin, Daniel J Sorin, Mark D Hill, and David A Wood. 2002. Evaluating non-deterministic multi-threaded commercial workloads. In *Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*.

[4] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. 2019. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro* 39, 2 (2019), 29–36.

[5] E. K. Ardestani and J. Renau. 2013. ESESC: A fast multicore simulator using Time-Based Sampling. In *International Symposium on High Performance Computer Architecture (HPCA)*. 448–459.

[6] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. 2009. COTSon: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review* 43, 1 (2009), 52–61.

[7] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. 2010. Analyzing parallel programs with pin. *Computer* 43, 3 (2010), 34–41.

[8] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS Parallel Benchmarks 2.0*. Technical Report. NAS-95-020, NASA Ames Research Center.

[9] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS parallel benchmarks summary and preliminary results. In *Conference on Supercomputing (SC)*. 158–165.

[10] Kenneth C Barr, Heidi Pan, Michael Zhang, and Krste Asanovic. 2005. Accelerating multiprocessor simulation with a memory timestamp record. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 66–77.

[11] E Barszcz, J Barton, L Dagum, P Frederickson, T Lasinski, R Schreiber, V Venkatakrishnan, S Weeratunga, D Bailey, D Browning, et al. 1991. The NAS parallel benchmarks. In *International Journal of Supercomputer Applications*.

[12] Robert H. Bell and Lizy K. John. 2005. Improved Automatic Testcase Synthesis for Performance Model Validation. In *International Conference on Supercomputing (SC)* (Cambridge, Massachusetts). 111–120.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7.

[14] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *International Conference on Performance Engineering (ICPE)*. 41–42.

[15] T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Article 52, 12 pages.

[16] T. E. Carlson, W. Heirman, and L. Eeckhout. 2013. Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.

[17] Trevor E. Carlson, Wim Heirman, Harish Patil, and Lieven Eeckout. 2014. Efficient, Accurate and Reproducible Simulation of Multi-Threaded Workloads. In *Workshop on Reproducible Research Methodologies (REPRODUCE)* (Orlando, FL, USA).

[18] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. 2014. BarrierPoint: Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS).* 2–12.

[19] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro* 37, 2 (2017), 52–62.

[20] Lieven Eeckhout, Yue Luo, Koen De Bosschere, and Lizy K John. 2005. BLRL: Accurate and efficient warmup for sampled processor simulation. *Comput. J.* 48, 4 (2005), 451–459.

[21] Edward W Forgy. 1965. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *biometrics* 21 (1965), 768–769.

[22] Karthik Ganesan and Lizy K John. 2011. Maximum multicore power (mampo) an automatic multithreaded synthetic power virus generation framework for multicore systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–12.

[23] T. Grass, T. E. Carlson, A. Rico, G. Ceballos, E. Ayguadé, M. Casas, and M. Moreto. 2019. Sampled Simulation of Task-Based Programs. *Transactions on Computers (TC)* 68, 2 (2019), 255–269.

[24] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé. 2016. TaskPoint: Sampled simulation of task-based programs. In *International Symposium on Performance Analysis of Systems and Software (ISPASS).* 296–306.

[25] Greg Hamerly, Erez Perelman, and Brad Calder. 2004. How to Use SimPoint to Pick Simulation Points. *ACM SIGMETRICS Performance Evaluation Review* 31, 4 (March 2004), 25–30.

[26] Chenji Han, Huai Xu, Guangyao Guo, Yuxuan Wu, and Fuxin Zhang. 2025. MeMo: Enhancing Representative Sampling via Mechanistic Micro-Model Signatures. In *International Symposium on Performance Analysis of Systems and Software (ISPASS).* 1–13.

[27] Haiyang Han and Nikos Hardavellas. 2021. Public Release and Validation of SPEC CPU2017 PinPoints. *arXiv preprint arXiv:2112.06981* (2021).

[28] John W Haskins and Kevin Skadron. 2003. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. In *International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 195–203.

[29] Sina Hassani, Gabriel Southern, and Jose Renau. 2016. LiveSim: Going live with microarchitecture simulation. In *International Symposium on High Performance Computer Architecture (HPCA).* 606–617.

[30] Intel [n. d.]. Intel Software Development Emulator (Intel SDE). https://www.intel.com/software/sde.

[31] Intel [n. d.]. DCFG generation with PinPlay. https://software.intel.com/content/www/us/en/develop/articles/pintool-dcfg.html.

[32] Haoqiang Jin, Michael Frumkin, and Jerry Yan. 1999. *The OpenMP implementation of NAS parallel benchmarks and its performance.* Technical Report. NAS-99-011, NASA Ames Research Center.

[33] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *International Symposium on Computer Architecture (ISCA).* 29–42.

[34] Humayun Khalid. 2000. Validating trace-driven microarchitectural simulations. *IEEE Micro* 20, 6 (2000), 76–82.

[35] Keiji Kimura, Gakuho Taguchi, and Hironori Kasahara. 2016. Accelerating multicore architecture simulation using application profile. In *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC).* 177–184.

[36] Andi Kleen and Beeman Strong. 2015. Intel processor trace on linux. *Tracing Summit* 2015 (2015).

[37] A. J. KleinOsowski and D. J. Lilja. 2002. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters (CAL)* 1, 1 (2002), 7–7.

[38] Jeremy Lau, Erez Perelman, and Brad Calder. 2006. Selecting Software Phase Markers with Code Structure Analysis. In *International Symposium on Code Generation and Optimization (CGO).* 135–146.

[39] Tong Li, Alvin R Lebeck, and Daniel J Sorin. 2006. Spin detection hardware for improved management of multithreaded systems. *Transactions on Parallel and Distributed Systems (TPDS)* 17, 6 (2006), 508–521.

[40] Mingyu Liang, Wenyin Fu, Louis Feng, Zhongyi Lin, Pavani Panakanti, Shengbao Zheng, Srinivas Sridharan, and Christina Delimitrou. 2023. Mystique: Enabling accurate and scalable generation of production AI benchmarks. In *International Symposium on Computer Architecture (ISCA).* 1–13.

[41] Mingyu Liang, Yu Gan, Yueying Li, Carlos Torres, Abhishek Dhanotia, Mahesh Ketkar, and Christina Delimitrou. 2023. Ditto: End-to-end application cloning for networked cloud services. In *International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.* 222–236.

[42] Ankur Limaye and Tosiron Adegbija. 2018. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *International Symposium on Performance Analysis of Systems and Software (ISPASS).* 190–200.

[43] Changxi Liu, Alen Sabu, Akanksha Chaudhari, Qingxuan Kang, and Trevor E. Carlson. 2024. Pac-Sim: Simulation of Multi-threaded Workloads using Intelligent, Live Sampling. *ACM Trans. Archit. Code Optim.* 21, 4, Article 81 (2024), 26 pages.

[44] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung,

Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

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

[46] Nikos Nikoleris, Lieven Eeckhout, Erik Hagersten, and Trevor E. Carlson. 2019. Directed Statistical Warming Through Time Traveling. In *International Symposium on Microarchitecture (MICRO)*. 1037–1049.

[47] Nikos Nikoleris, Andreas Sandberg, Erik Hagersten, and Trevor E Carlson. 2016. CoolSim: Statistical techniques to replace cache warming with efficient, virtualized profiling. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 106–115.

[48] Siddharth Nilakantan, Karthik Sangaiah, Ankit More, Giordano Salvadory, Baris Taskin, and Mark Hempstead. 2015. Synchrotrace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 278–287.

[49] OpenMP [n. d.]. OpenMP 3.1 API C/C++ Syntax Quick Reference Card. https://www.openmp.org/wp-content/uploads/OpenMP3.1-CCard.pdf.

[50] Harish Patil and Trevor E. Carlson. 2014. Pinballs: portable and shareable user-level checkpoints for reproducible analysis and simulation. In *Workshop on Reproducible Research Methodologies (REPRODUCE)*.

[51] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. 2004. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *International Symposium on Microarchitecture (MICRO)*. 81–92.

[52] Harish Patil, Alexander Isaev, Wim Heirman, Alen Sabu, Ali Hajiabadi, and Trevor E Carlson. 2021. ELFies: Executable Region Checkpoints for Performance Analysis and Simulation. In *International Symposium on Code Generation and Optimization (CGO)*. 126–136.

[53] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *International Symposium on Code Generation and Optimization (CGO)*. 2–11.

[54] C. Pereira, H. Patil, and B. Calder. 2008. Reproducible simulation of multi-threaded workloads for architecture design exploration. In *IEEE International Symposium on Workload Characterization (IISWC)*. 173–182.

[55] Perf 2012. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/.

[56] Zhantong Qiu, Mahyar Samani, and Jason Lowe-Power. 2026. Nugget: Portable Program Snippets. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[57] Alen Sabu, Changxi Liu, and Trevor E. Carlson. 2024. Viper: Utilizing Hierarchical Program Structure to Accelerate Multi-Core Simulation. *IEEE Access* 12 (2024), 17669–17678.

[58] Alen Sabu, Harish Patil, Wim Heirman, and Trevor E. Carlson. 2022. LoopPoint: Checkpoint-driven Sampled Simulation for Multi-threaded Applications. In *International Symposium on High Performance Computer Architecture (HPCA)*. 604–618.

[59] Alen Sabu, Harish Patil, Wim Heirman, Changxi Liu, and Trevor E. Carlson. 2025. TPE: XPU-Point: Simulator-Agnostic Sample Selection Methodology for Heterogeneous CPU-GPU Applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 385–400.

[60] Sabu, Alen [n. d.]. LoopPoint Source Code. https://github.com/nus-comparch/looppoint.

[61] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *International Symposium on Computer Architecture (ISCA)*. 475–486.

[62] Andreas Sandberg, Nikos Nikoleris, Trevor E. Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. 2015. Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed. In *2015 IEEE International Symposium on Workload Characterization*. 183–192.

[63] Gideon Schwarz. 1978. Estimating the dimension of a model. *The Annals of Statistics* (1978), 461–464.

[64] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 45–57.

[65] Rajat Todi. 2001. Speclite: using representative samples to reduce spec cpu2000 workload. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization (WWC-4)*. IEEE, 15–23.

[66] Michael Van Biesbrouck, Brad Calder, and Lieven Eeckhout. 2006. Efficient sampling startup for SimPoint. *IEEE Micro* 26, 4 (2006), 32–42.

[67] Thomas F Wenisch, Roland E Wunderlich, Babak Falsafi, and James C Hoe. 2006. Simulation sampling with live-points. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.

[68] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (2006), 18–31.

[69] Qinzhe Wu, Steven Flolid, Shuang Song, Junyong Deng, and Lizy K John. 2018. Invited paper for the hot workloads special session hot regions in SPEC CPU2017. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 71–77.

[70] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *International Symposium on Computer Architecture (ISCA)* (San Diego, California). 84–97.

[71] C. Yount, H. Patil, and M. S. Islam. 2015. Graph-matching-based simulation-region selection for multiple binaries. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 52–61.