IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 30 November 2023, accepted 25 December 2023, date of publication 15 January 2024, date of current version 6 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3354069

RESEARCH ARTICLE

Viper: Utilizing Hierarchical Program Structure to Accelerate Multi-Core Simulation

ALEN SABU, (Graduate Student Member, IEEE),
CHANGXI LIU, (Graduate Student Member, IEEE),
AND TREVOR E. CARLSON, (Senior Member, IEEE)
School of Computing, National University of Singapore, Singapore 117417

Corresponding author: Trevor E. Carlson (tcarlson@comp.nus.edu.sg)

This work was supported in part by Intel and VMware.

ABSTRACT Pre-silicon performance evaluation is a crucial component of computer systems research and development. While simulation has long been the de facto standard in this context, it can be prohibitively time-consuming for long-running, realistic workloads. To expedite this process, researchers have traditionally turned to sampling techniques. However, these techniques typically rely on fixed-length intervals for analysis, which can often be out of sync with the periodicity of program execution. Additionally, since an application's phase behavior is strongly correlated to the code it executes, it can exhibit a hierarchy of phase behaviors that can be observed at various interval lengths, rendering conventional sampling techniques inadequate. To address these limitations, we propose Viper – a novel sampled simulation methodology that applies to single-threaded and multi-threaded workloads by leveraging the hierarchical structure of program execution. Viper takes into account both application periodicity and inter-thread synchronization in order to achieve better sampling accuracy and smaller regions, which enables faster register-transfer level (RTL) simulations. We evaluate Viper with the multi-threaded SPEC CPU2017 benchmarks and demonstrate a significant simulation speedup (up to 2,710×, 358× on average for the train input set) while maintaining an average sampling error of just 1.32%. The source code of Viper is available at https://github.com/nus-comparch/viper.

INDEX TERMS Multi-core simulation, performance estimation, RTL evaluation, workload sampling.

I. INTRODUCTION

As we approach the limits of technology scaling, there is a growing emphasis on efficient and high-performance processor designs. Exploring and evaluating the design space of these next-generation architectures is an essential part of this research. However, the traditional dependence on extremely time-consuming microarchitectural simulations for large, realistic workloads proves impractical in addressing this challenge. For multithreaded workloads, this issue is further exacerbated by the complex interactions between multiple threads and the synchronization techniques employed to achieve scalable performance. One solution to address this issue is sampled simulation, which selects a representative subset of regions to simulate in detail and interpolates the performance of the entire application based on this. Prior works [1], [2], [3], [4], [5], [6] have demonstrated that, due to the repetitive behavior of workloads, sampling can often reduce the simulation time by orders of magnitude while preserving the original program characteristics.

SimPoint [1] reduces the simulation time by leveraging the application's phase behavior for single-threaded workloads. It does so by splitting the application into *fixed-size* regions, clustering them based on their execution behavior, and then simulating a representative element from each cluster in detail to extrapolate the performance of the entire application. However, a major drawback of this method is that it uses fixed-size regions for analysis, which do not often align with the actual periodicity [7] of program execution. Simpoint 3.0 [8] introduces variable length regions but does not address application periodicity. Moreover, since an application's phase behavior [9], [10] is strongly correlated to the code it executes, it can exhibit a hierarchy of phase behaviors that can be observed at different interval lengths [11]. Consequently, a single fixed region size cannot effectively capture the full spectrum of phase behaviors and often leads to suboptimal phase classification [12].

Later works, such as BarrierPoint [4], TaskPoint [13], and LoopPoint [6], address this shortcoming by utilizing

The associate editor coordinating the review of this manuscript and approving it for publication was Tomas F. Pena.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

the program structures and constructs within the application code to split the application into a series of independently analyzable regions to build a representative sample. Unfortunately, however, both BarrierPoint and TaskPoint only apply to specific classes of applications. BarrierPoint targets applications that use global barriers for synchronization, whereas TaskPoint targets task-based applications. While LoopPoint applies to generic multi-threaded applications, the regions it selects do not necessarily align with the application's phase behavior. Moreover, all these techniques use large region sizes (≈100 million instructions or more per thread), suitable for microarchitecture-level simulations (which take a few hours) but not for RTL-level simulations, which may take weeks to months for completion. In addition, no previous methodology provides a solution to detect small regions needed for RTL-level simulation, as they would typically result in aliasing [11], leading to unpredictable results. In this work, we propose a solution to solve both of these issues to achieve high performance and accuracy.

The goal of this work is to address the generic sampling problem by selecting representative regions that align with the application phases for simulation. Utilizing the innate program structures instead of fixed-length intervals allows for flexible region sizes that are more likely to be aligned with the application periodicity, thereby reducing the possibility of aliasing [11]. To do this, we present a novel methodology, Viper, that enables fast and efficient analysis prior to sampled simulation. In short, we make the following contributions to this work:

- We propose a novel methodology, Viper, that goes beyond prior state-of-the-art sampled simulation techniques to allow for fine-grained region selection and accurate performance reconstruction.
- We present a methodology that meets the requirements for RTL-level simulations for accurate performance estimations. We show this by performing experiments on microarchitecture-level and RTL-level simulators, enabling the detailed evaluation of large benchmarks.
- We provide an extensive evaluation of Viper and demonstrate best-in-class accuracy (average sampling error of just 1.32%) and speedup of up to 2,710×, with an average of 358× for the train input set of SPEC CPU2017 benchmarks. We also explore the accuracy and performance trade-offs of Viper in comparison with prior works.

The rest of the paper is organized as follows. In Section II, we discuss the relevant background and the challenges involved in the simulation of multi-threaded applications. Section III presents the Viper methodology in detail. We then discuss the experimental infrastructure in Section IV, followed by an extensive evaluation of Viper in Section V showcasing the applicability of the proposed methodology. Finally, we discuss prior work in Section VI and conclude the paper in Section VII.

II. BACKGROUND AND MOTIVATION

In this section, we present the background to understand the key features of sampled simulation. We also discuss the challenges in simulating large workloads and how the existing sampling methodologies are insufficient to address them.

A. CHARACTERIZING PROGRAM EXECUTION

A basic block is a sequence of instructions that has single entry and exit points with no branches or jumps within the sequence. A basic block vector (BBV) is a data structure that represents a set of basic blocks, storing counts for each executed basic block, and forms a fingerprint of a region's execution. It provides a compact representation of the program's control flow. Typically, BBVs are collected at regular intervals during the program execution. Each of these BBVs represents a region of an application that correlates to region performance [14]. BBVs provide information about how the program execution behavior changes over time.

LRU stack distance is the number of distinct cache accesses between consecutive accesses of the same data item [15]. LRU stack distance vectors (LDVs) are data structures that are used to keep track of the LRU stack distances. LDVs consist of integers associated with each cache line, representing the number of cache lines accessed between the current cache line and its most recent access. Shen et al. [16] showed that LDVs can be used to characterize program behavior. While BBVs focus on analyzing control flow patterns, LDVs provide insights into memory access patterns and cache behavior. By combining BBVs and LDVs, a more comprehensive understanding of program behavior can be achieved [4].

B. PROGRAM SAMPLING

Sampling is the process of selecting a minimal subset or a sample from a population to represent the entire population. The attributes or characteristics of the population are estimated using the selected sample. We employ this technique to reduce the simulation time of large workloads by simulating a representative sample from the entire program execution. Prior works [1], [2] split an application into a series of execution slices and cluster these slices with similar execution features into groups. These techniques demonstrate high performance by simulating selected representative slices from each group to represent the entire cluster of software slices.

Single-threaded sampling is largely considered to be a solved problem, whereas multi-threaded sampling has been a long-standing problem due to the complexity of the workload behavior: threads that sleep, synchronize, or are being delayed in spin-loops, among other issues. Alameldeen et al. [17] demonstrated the limitations of non-determinism with multi-threaded workloads and demonstrated that IPC can be a poor performance indicator [18], leading to inaccurate estimation of speedup or run time.

While initial works on multi-threaded sampling [19] focused on handling applications with uncorrelated thread behaviors, subsequent research [3], [11] considered time

as the sampling unit which applies to synchronizing multi-threaded workloads. However, a major drawback of this approach is that the whole application needs to be simulated sequentially (i.e., it cannot be parallelized), and thus, the maximum attainable simulation speedup is limited by the number of instructions in the whole application. Techniques like BarrierPoint [4] and LoopPoint [6] consider application barriers and loops, respectively, to define a unit-of-work [18]. BarrierPoint works on inter-barrier regions that can be so large that it is infeasible to simulate them, limiting scalability. LoopPoint divides the application into similarly sized regions enclosed within loop entries, ensuring size limits. However, the regions may not align with application phases. While LoopPoint regions are large enough to ensure accuracy and prevent aliasing, they are often too long for RTL-level simulation.

C. CHECKPOINTING TECHNIQUES

Checkpointing is a widely used technique to save the state of a simulation at a particular point in time, which can then be restored later, allowing for further simulation or debugging. Checkpointing is often used to parallelize simulation as well as to improve performance by reducing the amount of time that needs to be spent re-simulating portions of an application that have already been executed. For example, Checkpoint/Restore In Userspace (CRIU) [20] is a well-known checkpointing mechanism on Linux. CRIU has been integrated with major container engines like docker [21]. In addition, gem5 [22], [23] uses its own checkpointing format that is useful to create microarchitecture-level snapshots of simulation that can be restored later. For x86 systems, the PinPlay infrastructure [24] supports storing the application state as architectural checkpoints, called Pinballs, which can be replayed on PinPlay-enabled tools and simulators. Recent works on executable checkpoints, like ELFies [25], are promising in terms of usability and portability, as it is supported on popular microarchitecture simulators like gem5 [22] and Sniper [26].

D. MICROARCHITECTURAL STATE WARMUP

Modern processors employ various techniques to improve performance, such as branch prediction, caching, and speculative execution. These techniques can have a significant impact on the workload execution run time. While simulating the key parts of an application, it is important to rebuild or warm up the microarchitectural state of the system. This ensures that subsequent simulations or performance measurements accurately reflect the behavior of the processor. Methodologies like LoopPoint [6] rely on simulating a large region right before the start of the simulation region to warm up the microarchitectural state, while SMARTS [2] or time-based sampling techniques [3], [11] enable functional warming during the entire simulation. TurboSMARTS [27] uses a microarchitecture-level checkpointing mechanism to handle warmup that captures and stores the functionally warmed system state before each simulation region. Checkpoint-based warmup techniques require a large amount of storage. Moreover, it may not always be suitable for microarchitecture design-space exploration that runs experiments altering the memory hierarchy configuration, like cache sizes or the number of cache levels, because it would invalidate the checkpoint for those regions, requiring new memory checkpoints for each cache configuration.

E. THE QUEST FOR ADVANCED AND EFFICIENT SAMPLING

With the widening gap between simulator performance and the processors they model, running a cycle-accurate full-system simulation of large designs can be extremely time-consuming. Current sampling solutions are primarily targeted for microarchitecture-level simulations. Some recent works [28] attempted to adapt these solutions for RTL-level simulations on Verilator [29] using smaller region sizes aiming to improve simulation efficiency, which, however, resulted in accuracy that is typically not acceptable. The result is that it is currently infeasible to evaluate the performance of large workloads on the RTL level. Recent works [30], [31], [32] addressed the problem of accelerating RTL simulation by leveraging techniques like batch processing, task-level dataflow execution, low-level parallelism, and selective execution. These orthogonal techniques to speed up simulation may not scale well for very large workloads. In addition, while FPGA simulation infrastructures, such as Diablo [33] or FireSim [34], offer a faster alternative for simulation, FPGAs are specialized devices with inherent limitations in terms of memory capacity and logic capacity. This means that it is often not possible to fit large, realistic processor models on FPGAs. This highlights the need for developing specialized workload sampling methodologies that can be flexibly applied to both microarchitecture-level and RTL-level simulations. These methodologies should support finer region granularities that align with the dynamic phase behavior exhibited by the application. By tailoring the sampling approach to capture the specific characteristics and phases of the workload, more accurate and efficient sampled simulations can be performed.

III. THE VIPER METHODOLOGY

In this section, we describe the details of our proposed sampled simulation methodology, Viper (shown in Figure 1). Viper consists of four main steps: (i) Pre-profile Analysis which marks the region boundaries at which we split the application, (ii) Region Profiling, where the profiling information in the form of feature vectors is collected for each region, (iii) Clustering, which groups together regions with similar execution behavior based on the profiling information, and (iv) Simulation, where each application region is simulated either in Detailed Mode or Fast-forward Mode based on the clusters formed. The full application performance is reconstructed from the performance of each region. In the subsequent subsections, we provide details on how each of these stages operates.

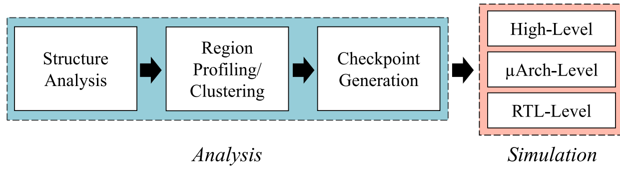


FIGURE 1. The workflow of Viper showing region identification, clustering, and simulation. The hierarchical structure of an application is used to identify regions. Sampled simulation is performed based on the clustering information of the regions. The simulation can be performed on various kinds of simulators depending on the level of detail required.

A. EXPLORING THE HIERARCHICAL STRUCTURE OF PROGRAM EXECUTION

Multi-threaded applications typically execute in a hierarchical flow, exhibiting different cyclic behavior patterns at varying interval lengths. These repeating patterns, often referred to as *phases*, are strongly correlated to the code executed by the application [11], [14], [35]. Thus, by analyzing the inherent program structures in an application's code, one can effectively capture the variations in its phase behavior. In Viper, we utilize this principle to identify *phase markers* [35] – the points within a program that correspond to change in the application's phase behavior. Phase markers can be used to split the application into a series of independently analyzable regions.

There are several kinds of program constructs in a parallel multi-threaded code region, such as barriers and loops, which can serve as *potential phase markers* of the application. Choosing barrier counts or loop counts over instruction counts to represent work can accurately demarcate multi-threaded regions over several runs.

- Barriers: Multi-threaded applications include single-threaded and multi-threaded code regions, with thread synchronization at boundaries using barriers that can be detected by compiler-generated instructions or functions to mark new code regions in machine code. In OpenMP-enabled applications, the GCC compiler generates the `_omp_fn` identifier that can be used to detect barriers.
- Loops: Typically, generic multi-threaded applications consist of various levels of nested loops. In our analysis, we use the application's dynamic control-flow graph (DCFG) [36] to identify the loops in the outermost level of the code region as *task loops* and the remaining as inner loops or *ordinary loops*. The DCFG is utilized to identify loop headers, and for each loop, information about their outer loops and associated subroutines is then collected. This helps to determine whether a loop is the outermost one in the current subroutine and if the current subroutine is the outermost in the given multi-threaded region.

After identifying potential phase markers in the application, we prioritize them for use as region boundaries. Barriers receive the highest priority due to their natural alignment of threads. Prior studies [4] support this, highlighting that partitioning at barrier boundaries prevents aliasing issues and increases accuracy. Task loops within a code region receive the next highest priority, marking boundaries between parallel tasks. Lastly, inner loops or ordinary loops are considered for finer granularity, albeit with lower priority, as ordinary loops typically do not act as phase markers in large applications. We then select a subset of these potential markers as region boundaries, considering their priorities. We also ensure that the resulting region sizes are suitable for analysis, meeting both a minimum (δ_{min} = 10,000,000) instruction threshold to capture variations in phase behavior and avoid aliasing issues [26] and a maximum (δ_{max} = 100,000,000) threshold for efficient simulation in a reasonable amount of time.

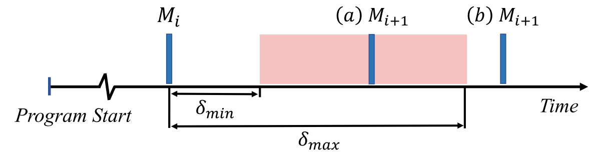


FIGURE 2. The selection of region boundaries (or markers) in an application using Viper. Marker M_i signifies the beginning of the current region with expected region lengths to be between δ_{min} and δ_{max} instructions. M_{i+1} is finally identified in accordance with case (a) or (b) (described in section III-A), which marks the end of the current region.

1) REGION BOUNDARIES

Once the list of potential phase markers is identified, the next step is to collect the highest-priority phase marker from every T ($T \approx$ 1,000,000) instructions. From this highest-priority list, we further select a subset of phase markers to serve as the region boundaries, subject to the constraints that the resulting region sizes approximately fall within the range of $[\delta_{\min}, \delta_{\max}]$ instructions as illustrated in Figure 2. This is done by employing a greedy algorithm that selects only the highest priority potential phase marker available beyond an interval of δ_{min} instructions but within the next δ_{max} instructions as the next region boundary (Figure 2a). If no such marker exists, the first potential phase marker encountered is selected as the next region boundary, regardless of its priority (Figure 2b). Region boundaries are represented as triplets: (*Image*, PC_{offset}, *Count*), denoting the object/library, instruction address offset from the Image's base address, and the address's count.

Figure 3 shows the classification of all the markers identified by Viper in SPEC CPU2017 applications, along with the chosen markers that serve as the region boundaries. We observe that applications like `638.imagick_s.1`, `657.xz_s.1`, and `657.xz_s.2` have a few or no barriers. Therefore, most of the selected markers are ordinary loops that serve as region boundaries. On the other hand, Viper selects as many barrier-bounded regions as possible, as observed in cases such as `607.cactuBSSN_s.1`, `621.wrf_s.1`, `644.nab_s.1`, `654.roms_s.1`, etc.

B. REGION PROFILING

Accurately capturing the execution behavior of a multi-threaded code region can be complex as threads synchronize at different points using various synchronization primitives, and the execution pattern of each thread may vary across multiple runs due to differences in memory access patterns [17].

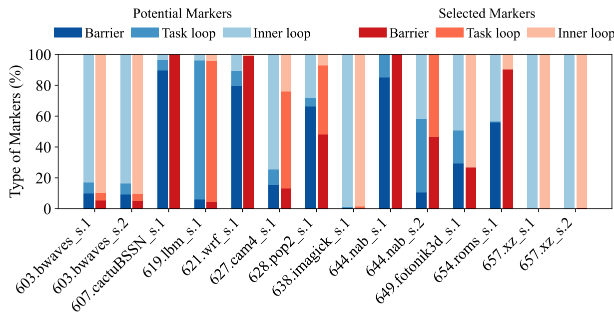


FIGURE 3. The percentage distribution of the type of markers (barriers, task loops, and inner loops) identified in the 8-threaded SPEC CPU2017 benchmarks using train inputs. Potential Markers denote all the available markers in the application, while Selected Markers signify the markers that serve as the boundaries of regions.

In Viper, we achieve this by using basic block vectors or BBVs as described in prior works [1], [4], [6]. A BBV is the execution fingerprint of a particular interval represented using basic blocks and their counts. BarrierPoint [4] showed that using LRU stack distance vectors (LDVs) along with BBVs can result in better clustering results. An LDV represents a fingerprint of the LRU-stack distance vector for a particular interval, which helps distinguish the regions that execute the same code but have different memory access patterns. We combine BBVs and LDVs on a per-thread level for each region to form per-thread signature vectors or SVs [4]. In order to represent a multi-threaded region, we concatenate the per-thread SVs to form a multi-threaded SV, which captures the amount of parallelism among the threads. The multi-threaded SVs are used for clustering to determine the similarity among the identified regions. We collect all the signature vectors using a high-level emulator.

C. DETERMINING THE REGION SIMILARITY

Once the application regions are identified and profiled, the next step is to determine the regions with similar execution characteristics in order to group them together and determine representative regions from among them. This is done based on the profiling information collected for each region which consists of multi-threaded SVs derived from the BBVs and LDVs of all threads, which are projected down to a smaller, fixed dimension. In our experiments, we use 1024 dimensions which could result in higher sampling accuracy and is a good trade-off with respect to the performance. The resulting SVs are then clustered using the k-means [37] clustering algorithm to group similar regions. We use the SimPoint [1] infrastructure to perform the clustering.

D. FAST AND ACCURATE FAST-FORWARDING

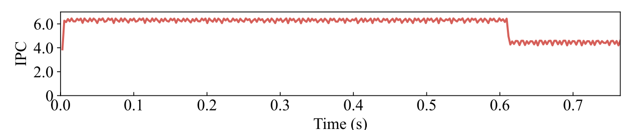
To speed up the simulation, representative regions of the application identified in the clustering stage are simulated in detail, whereas all the other regions are fast-forwarded. Note that this is applicable only for microarchitecture-level simulators, and for RTL-level simulators, we create simulation checkpoints as discussed in Section III-F. During the fast-forwarding phase, we ensure that all of the application threads make similar forward progress in time at regular intervals. This is particularly important because both the functional and timing simulations are disabled during this phase, which can lead to thread orderings that would not typically occur.

E. THE WARMUP CHALLENGE

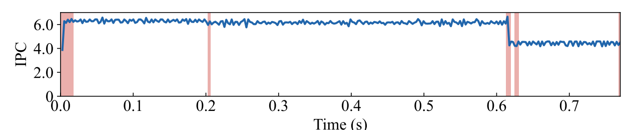
One major challenge in sampled simulation is building an accurate microarchitectural state before the start of every region to be simulated in detail. It is essential to choose a method that is flexible enough to support different cache configurations and can quickly build the right state, as this can significantly impact the overall speedup achieved. In this work, we choose the memory timestamp record (MTR) [38] warmup technique that can quickly build the cache state at run time. From our experiments, we observed that the harmonic mean of the slowdown due to MTR reconstruction is just 7.97% for SPEC CPU2017 benchmarks using train inputs. We implement MTR to collect the cache line information accessed by each `load` and `store` instruction during simulation, ordered in LRU fashion per set. The requests are then injected into the cache in the right order to rebuild the appropriate cache state before the simulation. We focus explicitly on cache warming in simulation, as smaller structures like prefetchers tend to warm up rapidly. For our RTL-level simulations, we simulate a warmup region right before the start of detailed performance measurements of the simulation region.

F. GENERATING SIMULATION CHECKPOINTS

Checkpointing is a widely used technique to capture the system state as a checkpoint and later restore it. We use the application binaries to guide the microarchitecture-level simulations. In order to guide RTL simulations, we create RISC-V full-system checkpoints using the MINJIE infrastructure [28]. MINJIE is an open-source platform that integrates a set of tools for pre-silicon validation and verification. MINJIE provides an instruction set interpreter/emulator called NEMU, which is used for checkpoint generation. The checkpoints are restored to simulate them in parallel on the RTL implementation of XiangShan [28] RISC-V processor using Verilator [29].



(a) The aggregate IPC of the full run.



(b) The aggregate reconstructed IPC using Viper.

FIGURE 4. Plot (a) shows the aggregate IPC of the full run, and plot (b) shows the reconstructed IPC of the `644.nab_s.1` benchmark (SPEC CPU2017) using Viper. This example shows the benchmark running with *test* inputs using eight threads. The shaded regions in the plot (b) represent the regions simulated in detail.

G. SIMULATION OF REPRESENTATIVE REGIONS

We assume that the region that lies the closest to the cluster centroid is the representative of that cluster. Once the cluster representatives are identified, these regions are simulated in parallel using the generated checkpoints. To demonstrate the applicability of the proposed methodology to both microarchitecture-level and RTL-level simulations, we perform simulations on Sniper (microarchitecture-level) and Verilator (RTL-level). We use the MTR warmup technique to rebuild the right micro-architecture state and inject it into the simulator before the detailed simulation, as discussed in Section III-E. The performance of the overall workload is estimated from the performance obtained from the simulation of the representative regions. Figure 4 shows Viper's IPC reconstruction from representatives of `644.nab_s.1` benchmark using test inputs.

IV. EXPERIMENTAL SETUP

In this section, we discuss the experimental setup to evaluate the Viper methodology. We describe the workloads and the platform used to conduct the experiments.

A. SIMULATION TOOLS

We implemented the support for Viper on Sniper [26] version 7.4, which is configured to model Intel's Gainestown microarchitecture, which is the latest hardware-validated microarchitecture available on Sniper. More details on the configuration used for the simulation are shown in Table 1. We modified the front-end of Sniper to support Viper's region specification. However, we expect that implementing this region specification support on other software simulators like gem5 [22], [23] or ZSim [39] is possible. For RTL-level simulations, we use Verilator [29] to simulate the RISC-V processor XiangShan [28] using the checkpoints generated using the MINJIE platform. In this work, we generate the simulation checkpoints using NEMU [28]. The methodology is also applicable to other RTL simulators (like VCS [40]) if corresponding checkpoints are generated.

TABLE 1. The configuration of Gainestown microarchitecture.

Component	Parameters
Processor	8 cores, 2.66 GHz, 128-entry ROB
Branch predictor	Pentium M, 8 cycles penalty
L1-I/D	32KB, 4/8 way, LRU
L2 cache	256KB, 8 way, LRU
L3 cache	8MB per core, 16 way, LRU

B. BENCHMARKS USED

SPEC CPU2017 benchmark suite [41] is a widely used collection of applications used for computer architecture evaluation and exploration. The benchmarks are written in C, C++, Fortran, or a combination of these programming languages. We use the multi-threaded OpenMP-based subset of the SPEC CPU2017 benchmarks that are enabled for multi-threaded execution. We use the *speed* version of these benchmarks configured to use eight threads. Note that these are multi-threaded benchmarks that synchronize and share memory. SPEC CPU2017 benchmarks use three different inputs: *test*, *train*, and *reference* (ref). We configure the SPEC CPU2017 benchmarks to use the train inputs for our evaluation. These benchmarks are compiled for x86-64 architecture using GCC 6.4.0 and gfortran with the `-O3` optimization compiler flag. The multi-threaded benchmarks are configured to use *passive* OpenMP thread wait policy.

C. ANALYSIS TOOLS

We use Intel's Pin [42] to build the analysis and profiling tools (Pintools) that we use for this methodology. We also utilize the Dynamic Control Flow Graph (DCFG) tool [36] included in the Pin distribution to collect potential markers that are used to identify regions. DCFG collects the trace information of the application, which can be utilized by implementing a pintool to detect barriers and loops that can act as region markers.

V. EVALUATION

In this section, we describe the experimental results of the proposed methodology. We also present the key factors that affect the performance of the methodology.

A. COMPARISON WITH STATE-OF-THE-ART

We first show the estimated wall time of full RTL simulation and Viper using XiangShan on Verilator. The,n we evaluate the accuracy and performance of Viper using the Sniper simulator and compare it with LoopPoint, the state-of-the-art sampled simulation methodology for multi-threaded applications [6]. We then conduct detailed studies on how region length affects speedup and accuracy. Unfortunately, it is not possible to evaluate the accuracy of Viper on XiangShan as full RTL simulation may take more than a year for SPEC CPU2017 benchmarks using train inputs.

In our experiments, we calculate the average value by taking the geometric mean of the values across all benchmarks as advised by previous work [43].

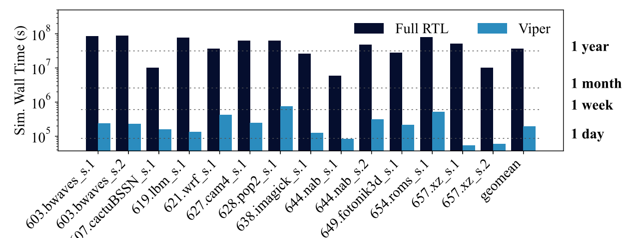


FIGURE 5. A comparison of the estimated wall time to simulate SPEC CPU2017 benchmarks using train inputs and eight threads for the full simulation (Full RTL) and Viper. We use the simulation rate of XiangShan on Verilator and assume parallel simulation of all the representative simulation checkpoints.

1) RTL-LEVEL SIMULATION

Figure 5 shows the estimated total time required to simulate SPEC CPU2017 benchmarks using Verilator. We observe that the sampling efficiency is bounded by the largest region identified by the sampling methodology. It is imperative to

identify smaller regions to significantly reduce the simulation time of these large workloads, which is now possible using Viper.

2) ACCURACY

Viper achieves similar or better error rates as compared to prior multi-threaded sampling methodologies like BarrierPoint or LoopPoint. To measure the sampling accuracy of the proposed methodology, we compare the simulation runtimes T_{full} obtained from the full simulation and T_{sample} obtained from the sampled simulation. The absolute runtime prediction error Δ can be represented as $\Delta = |1 - \frac{T_{sample}}{T_{full}}|$.

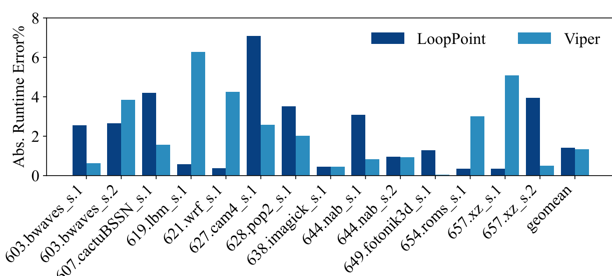


FIGURE 6. A comparison of the absolute runtime prediction error for Viper and LoopPoint for SPEC CPU2017 benchmarks that use train inputs running eight threads.

Figure 6 shows a comparison of absolute runtime prediction errors with Viper and LoopPoint obtained for the 8-threaded SPEC CPU2017 benchmarks using train inputs. Viper performs similarly to LoopPoint while achieving lower maximum and average (1.32%) errors. The results validate that choosing regions that are aligned to application phases, while potentially much smaller in length, can achieve better accuracies.

We evaluate the performance of Viper for 16 threads using the same set of SPEC CPU2017 benchmarks along with train inputs (except for `657.xz_s.1` and `657.xz_s.2`, which run only with one thread and four threads, respectively). For the rest of the benchmarks, we observe an average absolute error in predicting the runtime to be 1.79%. The maximum error that we observe is 5.29% (for `603.bwaves_s.2`), whereas the minimum error is 0.01% (for `638.imagick_s.1`).

3) SPEEDUP

The speedup is the ratio of the wall time required for the full simulation to that of the sampled simulation. We define serial speedup as the speedup achieved when the samples are simulated sequentially, whereas parallel speedup is the speedup achieved when the samples are simulated in parallel. Note that the analysis time of the benchmarks is excluded from the speedup calculation as this is often a one-time cost that is amortized over multiple simulations.

We compare the speedup of the proposed methodology with LoopPoint as shown in Figure 7a (parallel speedup) and Figure 7b (serial speedup). Viper outperforms LoopPoint in all but one case for parallel speedup, as shown in Figure 7a. We observe that Viper samples fewer but larger loop-bounded regions compared to LoopPoint for `627.cam4_s.1` resulting in larger simulation times. This is because `627.cam4_s.1` has larger loops, and unlike LoopPoint, Viper favors larger loops for higher accuracy. In the case of serial simulations, Viper outperforms LoopPoint in most cases (9 out of 14) in Figure 7b. The maximum serial speedup achieved by the proposed methodology is 6.23×as compared to the full simulation. The primary reason behind achieving a larger speedup is that the region size of Viper corresponds to the phase boundaries of the application.

In Figure 8, we compare the estimated serial speedup of Viper with that of LoopPoint. Viper achieves notably higher speedup when compared to LoopPoint for the SPEC CPU2017 benchmarks that use ref inputs. We estimate the speedup achieved for these large benchmarks by computing the ratio of the number of instructions in the full simulation to that of the sampled simulation of the benchmarks.

The maximum speedup achieved by Viper is approximately 7,490×, which is more than the maximum serial speedup achieved for LoopPoint. The primary reason behind the improved speedup is that Viper selects a more fine-grained representative sample of the application as the regions are identified based on the hierarchical structure. Benchmarks like `627.cam4_s.1` and `628.pop2_s.1` perform slightly better for LoopPoint as the hierarchical structure of these benchmarks is not easy to detect.

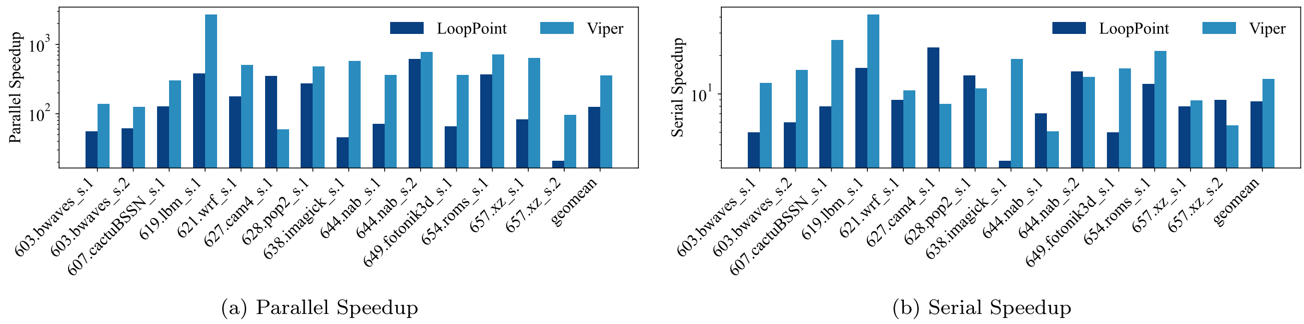
B. VARYING REGION SIZES

We use Viper methodology to illustrate the experimental results using different region sizes to show their effect on error rates. We also show the importance of choosing regions inherent to the application structure instead of fixed-size slices. We use Viper to select fixed-size regions of 10 million, 20 million, 50 million, and 100 million instructions.

1) ACCURACY

We show the accuracy in predicting the runtime of each of the benchmarks. In general, larger region sizes can improve accuracy, but there are a number of outliers, as shown in Figure 9. For example, in the case of `628.pop2_s.1`, `638.imagick_s.1`, or `654.roms_s.1`, the error decreases with an increase in region size. However, larger region size does not always yield better accuracy in some other cases. For example, benchmarks like `603.bwaves_s.1`, `621.wrf_s.1` and `627.cam4_s.1` achieve their best accuracies when the region size is around 50 million. We infer from the experiment that there is no general region size that can be used for every application due to the differences in loop behavior. This motivates us to choose region sizes that are application-dependent.

The average error of Viper-100M (regions of size $\approx$ 100M) is 0.74%, whereas that for Viper is 1.32%. Although using a larger region size yields a slightly better average error,



(a) Parallel Speedup

(b) Serial Speedup

FIGURE 7. A speedup comparison of LoopPoint and Viper for the 8-threaded SPEC CPU2017 benchmarks using train inputs.

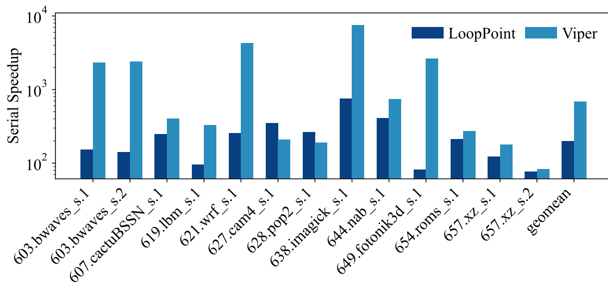


FIGURE 8. A comparison of estimated serial speedup achieved for Viper and LoopPoint for SPEC CPU2017 benchmarks that use ref inputs running eight threads. On average, Viper achieves better speedup (686×) as compared to LoopPoint (200×).

Viper consistently achieves better accuracy when simulating most benchmarks.

2) SPEEDUP

As Figure 9b shows, the serial speedup is larger for smaller atomic region sizes in most cases (although the errors can be higher). For smaller region sizes, clustering allows there to be fewer instructions to be simulated in detail overall, which allows for a larger speedup. However, in certain cases, the number of regions to be simulated in detail can be much more when the region sizes are smaller. For example, `649.fotonik3d_s.1` achieves a smaller speedup at region size *20M* when compared with that of region size *50M*. Comparing Figure 7b with Figure 9b, we observe that the speedup achieved using Viper-100*M* is similar to that of LoopPoint.

VI. RELATED WORK

In this section, we present the relevant prior research that investigates workload sampling and simulation.

A. SINGLE-THREADED SAMPLING

SimPoint [1] uses Basic Block Vectors (BBVs) as unique signatures to represent instruction streams with fixed length intervals based on the fact that code sections that perform similarly should traverse similar sequences of basic blocks. SMARTS [2] proposed a systematic sampling framework that simulated programs by alternating among fast-forward, warm-up, and detailed simulation phases and obtaining IPC samples for each detailed simulation. LiveSim [44] is another simulator that uses statistical sampling with confidence levels to estimate IPC. They extended the framework by using in-memory checkpoints to enable interactive simulations.

B. MULTI-THREADED SAMPLING

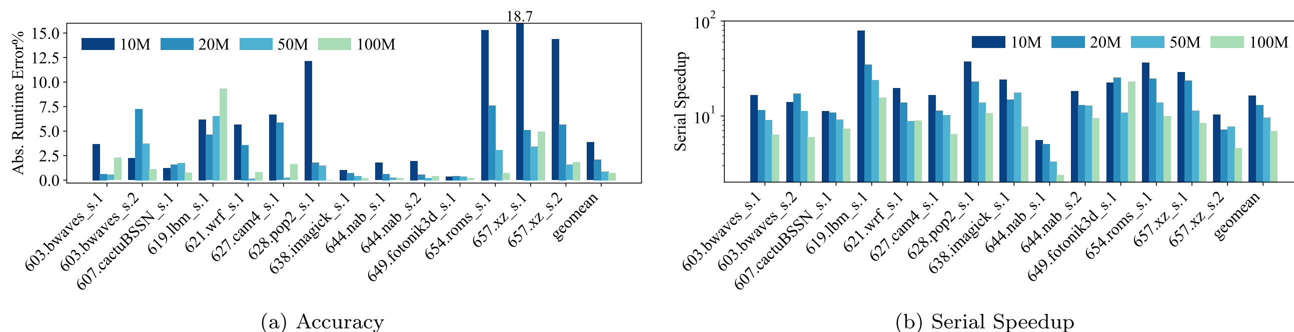
SimFlex [45] proposed a sampling technique for multi-processor workloads by sampling on processors that execute the program's critical path. COTSon [46] targeted the full software stack and complete hardware models to ensure both high performance and accuracy. Time-Based Sampling methodologies [3], [11] introduced a generic simulation framework for multi-threaded applications based on the progressed time. BarrierPoint [4] and TaskPoint [13] leveraged the structures in multi-threaded programs by using barriers and tasks as the unit of work, respectively. LoopPoint [6] proposed to enable a generic profiling-based sampling methodology that uses loop boundaries as the heuristic for demarcating representative regions. LoopPoint identifies similarly sized regions that are, however, not suitable for RTL-level simulations.

C. ANALYTICAL MODELING

Eyerman et al. [47] proposed a model to divide the dynamic instruction stream into long-latency miss events that limit the scope of out-of-order behaviors. RPPM [48] takes into account synchronization overheads by identifying critical paths to project multi-threaded performance. Statstack [49] and Linear branch entropy [50] are proposed to model the cache miss ratio of a fully associative cache and the branch miss rate of any branch predictor, respectively.

D. WARMUP TECHNIQUES

There are primarily three kinds of warmup techniques: statistical warming, checkpoint-based warming, and functional warming. Statistical warming techniques [38], [51], [52] reconstruct the cache state by collecting all the memory access information. Unlike prior works, DeLorean [53] collects only a selected number of key reuse distances to speed up the statistical warming. CoolSim [54] uses virtualized fast-forwarding to speed up the performance of collecting memory reuse information. Memory Hierarchy State [55] is a checkpoint-based technique that saves the state of major microarchitecture components into a touched memory image.



(a) Accuracy

(b) Serial Speedup

FIGURE 9. The runtime prediction error and speedup achieved for the 8-threaded SPEC CPU2017 benchmarks using train inputs. Viper is used to identify regions of fixed sizes.

E. SIMULATION INFRASTRUCTURES

Gem5 [22] is a cycle-accurate simulator that models CPU pipelines and cache protocols in fine granularity. Sniper [26] and ZSim [39] are fast multi-core simulators that use binary instrumentation to speed up functional simulations. Apart from microarchitecture-level simulators, RTL-level simulators like Verilator [29], VCS [40], etc., are commonly used for performance evaluation, correctness checking, and validation. These software-based RTL simulators are extremely slow for large-scale designs as compared to FPGA-based RTL simulators like FireSim [34].

VII. CONCLUSION

In this work, we propose a novel sampled simulation methodology and infrastructure called Viper that shows significant improvement in performance over the existing methodologies which is applicable to both microarchitecture-level and RTL-level simulators. Viper is both a fast (358× speedup on average) and an accurate (with an average error of just 1.32%) simulation methodology as evaluated with the multi-threaded subset of SPEC CPU2017 benchmarks using train inputs.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable feedback. The work was funded, in part, by grants from Intel and VMware. *(Alen Sabu and Changxi Liu contributed equally to this work.)*

REFERENCES

[1] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. 10th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Oct. 2002, pp. 45–57.

[2] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2003, pp. 84–97.

[3] E. K. Ardestani and J. Renau, "ESESC: A fast multicore simulator using time-based sampling," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2013, pp. 448–459.

[4] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, "BarrierPoint: Sampled simulation of multi-threaded applications," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2014, pp. 2–12.

[5] X. Zheng, H. Vikalo, S. Song, L. K. John, and A. Gerstlauer, "Sampling-based binary-level cross-platform performance estimation," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 1709–1714.

[6] A. Sabu, H. Patil, W. Heirman, and T. E. Carlson, "LoopPoint: Checkpoint-driven sampled simulation for multi-threaded applications," in *Proc. IEEE Int. Symp. High-Performance Comput. Archit. (HPCA)*, Apr. 2022, pp. 604–618.

[7] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, 2001, pp. 3–14.

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

[9] T. Sherwood and B. Calder, "Time varying behavior of programs," Univ. California San Diego, San Diego, CA, USA, Tech. Rep. CS99-630, 1999.

[10] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2003, pp. 336–349.

[11] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2013, pp. 2–12.

[12] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for variable length intervals and hierarchical phase behavior," in *Proc. Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2005, pp. 135–146.

[13] T. Grass, T. E. Carlson, A. Rico, G. Ceballos, E. Ayguadé, M. Casas, and M. Moreto, "Sampled simulation of task-based programs," *IEEE Trans. Comput.*, vol. 68, no. 2, pp. 255–269, Feb. 2019.

[14] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder, "The strong correlation between code signatures and performance," in *Proc. Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2005, pp. 236–247.

[15] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[16] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, 2004, pp. 165–176.

[17] A. Alameldeen and D. Wood, "Variability in architectural simulations of multi-threaded workloads," in *Proc. Int. Symp. High-Performance Comput. Archit. (HPCA)*, 2003, pp. 7–18.

[18] A. R. Alameldeen and D. A. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, pp. 8–17, Jul. 2006.

[19] M. Ekman and P. Stenstrom, "Enhancing multiprocessor architecture simulation speed using matched-pair comparison," in *Proc. Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2005, pp. 89–99.

[20] (Nov. 13, 2023). *Checkpoint/Restore in Userspace*. [Online]. Available: http://criu.org

[21] (Oct. 12, 2021). *CRIU Integration With Docker*. [Online]. Available: https://criu.org/Docker

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

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

[24] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Apr. 2010, pp. 2–11.

[25] H. Patil, A. Isaev, W. Heirman, A. Sabu, A. Hajiabadi, and T. Carlson, ''ELFies: Executable region checkpoints for performance analysis and simulation,'' in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2021, pp. 126–136.

[26] T. E. Carlson, W. Heirman, and L. Eeckhout, ''Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation,'' in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2011, pp. 1–12.

[27] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe, ''TurboSMARTS: Accurate microarchitecture simulation sampling in minutes,'' in *Proc. Int. Conf. Meas. Model. Comput. Syst. (SIGMETRICS)*, 2005, pp. 408–409.

[28] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen, L. Gou, Y. Jin, Q. Li, X. Li, and Z. Li, ''Towards developing high performance RISC-V processors using agile methodology,'' in *Proc. Int. Symp. Microarchitecture (MICRO)*, 2022, pp. 1178–1199.

[29] W. Snyder, *Verilator: The Fast Free Verilog Simulator*. (2012). [Online]. Available: http://www.veripool.org

[30] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang, ''From RTL to CUDA: A GPU acceleration flow for RTL simulation with batch stimulus,'' in *Proc. Int. Conf. Parallel Process. (ICPP)*, 2022, pp. 1–12.

[31] F. Elsabbagh, S. Sheikhha, V. A. Ying, Q. M. Nguyen, J. S. Emer, and D. Sanchez, ''Accelerating RTL simulation with hardware–software co-design,'' in *Proc. Int. Symp. Microarchitecture (MICRO)*, 2023, pp. 153–166.

[32] M. Emami, S. Kashani, K. Kamahori, M. S. Pourghannad, R. Raj, and J. R. Larus, ''Manticore: Hardware-accelerated RTL simulation with static bulk-synchronous parallelism,'' in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, Apr. 2024.

[33] Z. Tan, Z. Qian, X. Chen, K. Asanovic, and D. Patterson, ''DIABLO: A warehouse-scale computer network simulator using FPGAs,'' in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, 2015, pp. 207–221.

[34] 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, ''FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud,'' in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 29–42.

[35] J. Lau, E. Perelman, and B. Calder, ''Selecting software phase markers with code structure analysis,'' in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2006, pp. 135–146.

[36] C. Yount, H. Patil, and M. S. Islam, ''Graph-matching-based simulation-region selection for multiple binaries,'' in *Proc. Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2015, pp. 52–61.

[37] E. W. Forgy, ''Cluster analysis of multivariate data: Efficiency versus interpretability of classifications,'' *Biometrics*, vol. 21, no. 3, pp. 768–769, 1965.

[38] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic, ''Accelerating multiprocessor simulation with a memory timestamp record,'' in *Proc. Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2005, pp. 66–77.

[39] D. Sanchez and C. Kozyrakis, ''ZSim: Fast and accurate microarchitectural simulation of thousand-core systems,'' in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2013, pp. 475–486.

[40] Synopsys. (2024). *VCS Functional Verification Solution*. [Online]. Available: https://www.synopsys.com/verification

[41] J. Bucek, K.-D. Lange, and J. V. Kistowski, ''SPEC CPU2017: Next-generation compute benchmark,'' in *Proc. Int. Conf. Perform. Eng. (ICPE)*, Apr. 2018, pp. 41–42.

[42] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, ''Pin: Building customized program analysis tools with dynamic instrumentation,'' in *Proc. Conf. Program. Lang. Design Implement. (PLDI)*, 2005, pp. 190–200.

[43] P. J. Fleming and J. J. Wallace, ''How not to lie with statistics: The correct way to summarize benchmark results,'' *Commun. ACM*, vol. 29, no. 3, pp. 218–221, Mar. 1986.

[44] S. Hassani, G. Southern, and J. Renau, ''LiveSim: Going live with microarchitecture simulation,'' in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 606–617.

[45] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, ''SimFlex: Statistical sampling of computer system simulation,'' *IEEE Micro*, vol. 26, no. 4, pp. 18–31, Jul. 2006.

[46] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, ''COTSon: Infrastructure for full system simulation,'' *ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 1, pp. 52–61, Jan. 2009.

[47] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, ''A mechanistic performance model for superscalar out-of-order processors,'' *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 1–37, May 2009.

[48] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout, ''RPPM: Rapid performance prediction of multithreaded workloads on multicore processors,'' in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2019, pp. 257–267.

[49] D. Eklov and E. Hagersten, ''StatStack: Efficient modeling of LRU caches,'' in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2010, pp. 55–65.

[50] S. De Pestel, S. Eyerman, and L. Eeckhout, ''Micro-architecture independent branch behavior characterization,'' in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2015, pp. 135–144.

[51] J. W. Haskins and K. Skadron, ''Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation,'' in *Proc. Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2003, pp. 195–203.

[52] L. Eeckhout, Y. Luo, K. De Bosschere, and L. K. John, ''BLRL: Accurate and efficient warmup for sampled processor simulation,'' *Comput. J.*, vol. 48, no. 4, pp. 451–459, Jan. 2005.

[53] N. Nikoleris, L. Eeckhout, E. Hagersten, and T. E. Carlson, ''Directed statistical warming through time traveling,'' in *Proc. Int. Symp. Microarchitecture (MICRO)*, 2019, pp. 1037–1049.

[54] N. Nikoleris, A. Sandberg, E. Hagersten, and T. E. Carlson, ''CoolSim: Statistical techniques to replace cache warming with efficient, virtualized profiling,'' in *Proc. Int. Conf. Embedded Comput. Syst., Architectures, Model. Simul. (SAMOS)*, Jul. 2016, pp. 106–115.

[55] M. Van Biesbrouck, B. Calder, and L. Eeckhout, ''Efficient sampling startup for SimPoint,'' *IEEE Micro*, vol. 26, no. 4, pp. 32–42, Jul. 2006.

ALEN SABU (Graduate Student Member, IEEE) is currently pursuing the Ph.D. degree in computer science with the National University of Singapore. Most of his prior works focused on developing techniques to speed up the performance evaluation of multi-core systems and building simulation infrastructures, which were recognized in venues, like HPCA and CGO. His research interests include computer architecture, performance measurements, and simulation methodologies.

CHANGXI LIU (Graduate Student Member, IEEE) received the bachelor's and master's degrees in computer science from Beihang University, Beijing, China. He is currently pursuing the Ph.D. degree in computer science with the National University of Singapore. He has published papers in MICRO, ICS, and ICPP, in the areas of simulation and high-performance computing. His research interests include simulation, compilers, high-performance computing, and computer architecture exploration.

TREVOR E. CARLSON (Senior Member, IEEE) is currently an Assistant Professor with the National University of Singapore (NUS). His research interests include efficient general-purpose processing, secure systems, AI acceleration, and simulation methodologies. He co-develops the Sniper Multi-Core Simulator, which is being used by hundreds of researchers in academia and industry, to evaluate the performance and power efficiency of next-generation systems. He has over 20 years of computer systems and architecture experience in both industry and academia, and his work has received six best papers or best paper nominations at conferences, such as the International Symposium on Microarchitecture (MICRO) and the International Symposium on Performance Analysis of Systems and Software (ISPASS).