ROIperf: Rapid Validation and Iterative Tuning of Workload Sampling Methodologies

Alen Sabu
National University of Singapore

Harish Patil
Intel Corporation

Wim Heirman
Intel Corporation

Trevor E. Carlson
National University of Singapore

***Abstract*—Estimating workload performance for a future processor is a daunting task, as traditional cycle-accurate simulation techniques used by architects are extremely slow. Workload sampling can significantly speed up this process, assuming the identified regions of interest (ROIs) or the representative regions can be proven to accurately represent the behavior of the full workload. One standard way to validate the ROIs is to measure the sampling error, which is the difference in the performance of the full workload and the extrapolated performance obtained using the ROIs. The performance of a custom microarchitecture is typically measured using simulation. However, the full-program simulation of long-running workloads is infeasible for sample validation, as it defeats the purpose of sampling.**

In this work, we propose ROIperf, a framework to quickly validate the representative regions of large workloads. ROIperf allows for the evaluation of both the full workload and the regions of interest by measuring the performance using real hardware systems instead of long-running simulations. ROIperf presents a methodology for long-running, realistic workloads for which the prevailing simulation-based validation techniques are not viable in a reasonable amount of time. We demonstrate the efficacy of ROIperf by evaluating state-of-the-art sample selection methodologies across a wide range of workloads. We use the SPEC CPU2017 suite, including both single-threaded and multi-threaded benchmarks, alongside the NAS Parallel Benchmarks to evaluate the proposed technique. Our results indicate that ROIperf provides a significant speedup in validating representative simulation regions, enabling more extensive use of sample verification at near-native speeds, particularly for large workloads.

***Index Terms*—Hardware performance counters, performance projection and validation, region of interest, workload sampling**

I. Introduction

Cycle-accurate, detailed simulation of computer systems tends to be extremely slow, with simulation speeds of complex, modern processor designs can be as low as a few thousand instructions per second, that is, more than 100,000× slower than native speeds. Simulating large modern applications with trillions of instructions in their entirety is, therefore, not practical when using these methods directly. Instead, simulation of *regions of interest* (ROIs) from large application executions and extrapolating the full-program performance is a standard technique employed [1]–[8]. To gain confidence in the extrapolated results, it is necessary to validate that the ROIs selected closely represent full-program behavior [9]–[11]. 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 and/or using fast but inaccurate simulators.

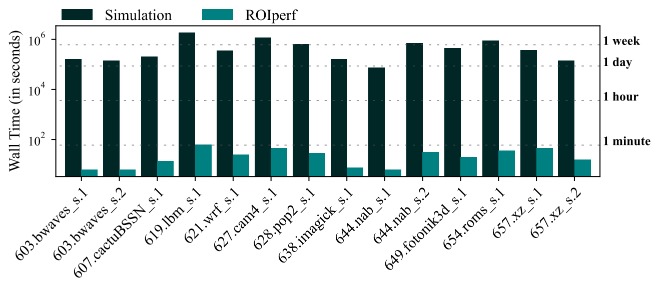

Fig. 1. A comparison of the total wall-time required to validate the representative regions identified for the multi-threaded SPEC CPU2017 benchmarks using *train* inputs (the gap is expected to increase for *ref* inputs). The bars show a comparison of the minimum wall time taken to validate the regions (selected using LoopPoint [8] methodology) on a cycle-level simulator and the ROIperf framework.

Performance monitoring on native hardware offers a significantly faster alternative for sample validation compared to traditional architecture simulators. Figure 1 shows that the validation of representative regions using our proposed ROIperf methodology can be performed at near-native speed, while simulation-based validation can take weeks or months. Accurately identifying representative regions within an application typically involves iterative parameter tuning and revalidation. For example, applications like `gcc` may require up to five times more representative regions than other applications, as shown in previous works [2]. Without extremely fast techniques, it becomes impractical to validate the efficacy of workload sampling methodologies for large-scale applications. In this work, we aim to provide a solution to this challenge to enable rapid sample validation without the need for long-running simulations.

Although measuring full-program performance on native hardware is well-established [12], [13], 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 execution. These ROIs might only consist of a few million instructions, running for just milliseconds on real hardware.

Precisely gathering performance data solely for the ROIs on native hardware with high fidelity is challenging. Loop-based representations of ROIs, for instance, offer high accuracy and reproducibility [8]. However, current hardware architectures lack native support for directly identifying such representation of 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 [14], which is 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 [15] that interfere with the workload behavior. While the instrumentation capability of a Pin probe tool is limited, its low overhead makes it ideal as a building-block for ROIperf. 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.

The Repeatability Challenge

Profile-based simulation region selection techniques, like SimPoint [2], typically require at least two program executions. The first run gathers profiling data to identify the ROIs for simulation. Subsequently, the second execution simulates the selected ROIs. Profile-based sample selection methodologies assume identical program behavior across executions, which is difficult to guarantee, especially for multi-threaded programs. Heterogeneity in hardware environments (for instance, varying ISA support leading to scalar vs. vectorized runs), inconsistencies in system libraries, and timing-dependent control flow (for example, work stealing in parallel applications) can all introduce discrepancies between profiling and simulation runs.

Several works have been proposed to ensure repeatable program execution during profiling and simulation. PinPlay [16] utilizes a record-and-replay framework, guaranteeing identical behavior across analyses by capturing the entire program execution and then performing profiling/simulation using a deterministic replay. However, PinPlay's replay incurs significant overhead ($\approx$ 50$\times$ slowdown), rendering performance counter-based evaluation inaccurate. Other efforts to improve repeatability include using static binaries, checkpoints [17], or ELFies [18]. However, none of these techniques guarantee fully repeatable execution, particularly in multi-threaded scenarios where timing-dependent control flow and the resulting execution divergence happen more often [19], [20].

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 a strictly controlled environment. We describe tests for the applicability of ROIperf prior to the measurement in Section V. In our evaluations, we find that ROIperf is effective in identifying the regions accurately in most cases.

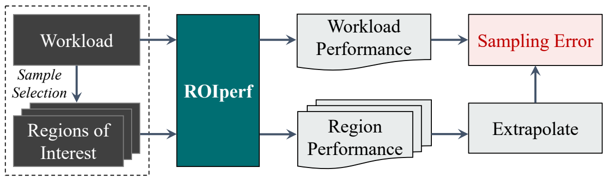

Fig. 2. An overview of the working of ROIperf framework to validate the regions of interest (ROIs). The performance of the full workload and the ROIs are measured on the native hardware. The extrapolated performance is compared with the performance of the full runs to quantify the sampling error.

Contributions and Organization of the Paper

To summarize, the paper makes the following contributions:

1) We introduce ROIperf, a framework designed to rapidly evaluate the effectiveness of workload sampling methodologies. We demonstrate its application in validating regions of interest (ROIs) for long-running single-threaded, and multi-threaded workloads.
2) We leverage previously proposed PinPoints [21] (for single-threaded programs) and LoopPoint [8] (for multi-threaded programs) methodologies for ROI selection. Both methodologies rely on profiling based on deterministic replay [16] for region selection.
3) We also introduce sanity tests (detailed in Section V-A) to assess reproducibility and identify extreme deviations in control flow in program execution. For accurate ROI validation, achieving identical control flow between the ROIperf run and the profiling run used for ROI selection is essential.
4) We will open-source the ROIperf infrastructure for use in the community, enabling the sample validation of large, realistic applications at near-native speeds, which was not possible before.

The rest of the paper is organized as follows. In Section II, we discuss the background on workload sampling methodologies and techniques for sample validation. Section III presents the ROIperf methodology and implementation details. We then describe the experimental infrastructure in Section IV, followed by an extensive evaluation of ROIperf in Section V to demonstrate the applicability of the proposed methodology. Finally, we present the related work in Section VI and conclude the paper in Section VII with some possible future directions.

II. Background

In this section, we provide a background on the existing workload sampling techniques, particularly those leveraged in this paper for sample selection. We also provide an overview of previously proposed sample validation techniques.

A. Sample Selection

1) Single-threaded Workloads: SimPoint [2] and SMARTS [3] are two well-established techniques for sampling single-threaded applications to accelerate simulation. SimPoint is a profile-driven methodology that identifies representative regions for simulation, whereas SMARTS employs statistical sampling for fast simulation. We employ SimPoint to select representative regions of single-threaded applications. SimPoint requires the collection of basic block vectors or BBVs at every sampling period (typically 100 million instructions). The BBVs are then clustered using the k-means clustering algorithm to identify the number of phases within the application. A representative region is selected from each cluster that is assigned a weight proportional to the number of regions that belong to the cluster. PinPoints methodology was introduced in [16], [21], where Pin was used for the BBV generation of x86 applications.

2) Multi-threaded Workloads: Accurately sampling multi-threaded workloads presents a significant challenge. Applying naive extensions of single-threaded techniques directly proves ineffective due to factors like thread interactions and spin-loops [22]. There are several techniques proposed to sample multi-threaded workloads [4]–[8], each having its own limitations. SimFlex [4] proposed a technique for the sampled simulation of multi-process workloads. Time-based sampling techniques [5], [6] were the first to address the sampled simulation of multi-threaded workloads, which considers time as the sampling unit. While these methodologies could not achieve practically useful speedups, BarrierPoint [7] achieves speedups in order of magnitudes as compared to time-based sampling techniques. This methodology, however, is applicable only to those workloads that use barriers for thread synchronization and, therefore, is not applicable to generic multi-threaded workloads. In this work, we evaluate LoopPoint [8] methodology that applies to generic multi-threaded workloads. LoopPoint identifies regions bounded by loop entries and can achieve high simulation speedups without compromising on sampling accuracy.

B. Sample Validation

Validating the representative regions identified for a large application can be tedious. This typically requires the full simulation of the application and the representative regions. While FPGA-based simulation infrastructures like FireSim [23] offer faster execution compared to traditional cycle-level software simulators, their turnaround time is still not negligible. Moreover, the hardware implementation of each component within the limited memory of the FPGAs is challenging.

Perelman et al. [24] propose a technique to select statistically valid representative regions early in the application to reduce the fast-forward time to reach the simulation regions. Gottschall et al. [25] propose SimPoint validation with TraceDoctor, an instrumentation framework attached to FireSim. This technique can be used to validate SimPoints for a RISC-V model running on FPGAs at high speeds. While significantly faster than detailed simulation, it remains slower than hardware validation and is limited to FPGA-based models.

C. Hardware Performance Counters

Modern microprocessors have special hardware registers called hardware performance counters for monitoring various performance-related metrics [26]. These counters provide the ability to measure performance in real-time and are typically used by software performance tools to measure metrics like the number of instructions executed, cache misses, page faults, etc [27]. By measuring these metrics, performance tools can help developers identify bottlenecks and other performance issues in their software. Because these counters are built into the hardware, they are able to provide measurements of performance-related metrics with very low overhead.

D. Instrumentation using Pin

Pin [14] is a well-known dynamic instrumentation and analysis framework for x86 applications. It offers an application programming interface (API) for adding extra code at various points within a program. The API differs based on the mode specified during Pin initialization. Pin supports two modes: (a) a just-in-time (JIT) mode which translates the test program in memory and adds instrumentation during the translation, and (b) a probe mode which merely patches an in-memory copy of the program with extra code. The JIT mode API allows for sophisticated run-time analyses but at the cost of translation overhead. The probe mode API is limited to adding extra code only at specific program points, although the overhead of such an addition is very low. ROIperf leverages Pin in probe mode due to its low overhead, which minimizes perturbation during performance measurement of the target application.

III. Methodology and Implementation Details

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.

A. 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 [21]. This method builds upon SimPoint methodology [2] 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 [8] 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.

B. ROI Specification

The evaluation using ROIperf considers program repeatability to ensure ROIs remain representative. However, single-threaded programs can often exhibit non-repeatable behavior [16]. 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).[1]

A single-threaded ROI can be simply represented by the retired instruction count at the beginning and the end of the region. As long as regions are repeatable, ensured by using fixed shared libraries and by turning off ASLR, capturing hardware performance counter values at ROI boundaries is sufficient for performance projection (Figure 3) of single-threaded programs. For single-threaded programs, an ROI can be represented by the retired instruction count at the beginning and end of the region. Assuming repeatable program execution (achieved through fixed shared libraries and similar microarchitecture), capturing hardware performance counter values at ROI boundaries suffices for performance measurements (as shown in Figure 3). For multi-threaded programs, a major source of non-repeatability is the timing and behavior of thread synchronization [16], [20], [22]. Instruction counts are, therefore, not a reliable way to specify ROI boundaries. The LoopPoint methodology [8] 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.

C. 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

[1]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`.

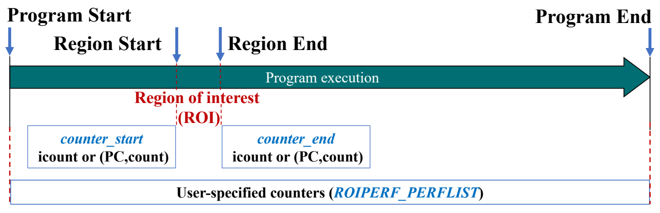

Fig. 3. The high-level execution flow of an application using the ROIperf tool. Upon program start, user-defined performance counters are initialized. Measurements are then activated at the start of ROI and remain active until the end of ROI. Hardware instruction counts or address (PC) counts are employed to identify the ROI.

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, as detailed in Figure 2. ROIperf 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 using 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 performance counters specified in *ROIPERF_LIST* alongside sampled counting. An illustration of these various actions taken by ROIperf can be found in Figure 3.

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 RDTSC 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 core/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 RDTSC 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.

IV. Experimental Setup

A. Workloads Used

We use two benchmarks for our evaluation, SPEC CPU2017 and NAS Parallel Benchmarks (NPB). For our single-threaded evaluations, we use the *rate* version of SPEC CPU2017 benchmarks using training (train) inputs and reference (ref) inputs. For our multi-threaded evaluations, we use the multi-threaded subset of SPEC CPU2017 benchmarks (*speed* version). These benchmarks can spawn several threads that synchronize and share memory. We configure the benchmarks with eight OpenMP threads. We also use NPB version 3.3 (OpenMP-based) for our multi-threaded evaluations that are configured to Class C inputs with eight threads. We present the evaluation results for all but `dc` (data cube) benchmark in the NPB benchmark suite as it generates a huge amount of data. We use *active* thread wait-policy for evaluating the SPEC CPU2017 benchmarks, which means that the threads spin (user-level) at the synchronization point, whereas *passive* policy is used for the NPB benchmarks for which the threads go to sleep while waiting for the other threads at a synchronization point.

B. Sample Selection

For single-threaded sampling, we use PinPlay-based profiling methodologies involving the PinPoint [21] tool, derived from the SimPoint [2] methodology. We split the application every 200 million instructions. We also use a maxk of 50 for k-means clustering. For sampling multi-threaded applications that use eight threads, we use the LoopPoint methodology [28] with default settings. We split the applications targeting multi-threaded regions of size 800 million global (all-threads) instructions, always aligning with a loop entry. The regions are represented as basic block vectors (BBVs), clustered using k-means clustering with a maxk of 50. PinPlay processing, especially logging, is quite expensive, and therefore, running region selection in a controlled environment was not practical. Instead, region selection was done on machines with varying microarchitectures and run-time libraries. But, in an ideal case, we are required to (a) run all the experiments (region selection, simulation, ROIperf validation, etc.) on the same microarchitecture and (b) package and reuse the system libraries so that we are sure we control the simulation. For ROIperf-based evaluations, we chose two machines with Broadwell and Skylake microarchitectures.

C. Simulators Used

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 [29], called CoreSim, for evaluations. 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 [31] and Intel Cascade Lake [32] microarchitectures. We also use the Sniper multi-core simulator [29], [33] version 8.0 (using Pin [14] front-end) for our evaluations with NPB benchmarks. We configured Sniper to simulate the Intel Gainestown microarchitecture.

V. Evaluation

In this section, we aim to demonstrate the effectiveness of the ROIperf methodology across different benchmarks.

A. Testing ROIperf Applicability

As discussed in Section I, the repeatability of application results can be an issue for a number of applications, both single-threaded and multi-threaded. For our evaluations, we selected the applications that were not prone to this issue. We devised a pre-test for the applications for repeatability, which is two-fold:

1) *Do the thread-0 instruction counts from the region selection and ROIperf runs exhibit a close match?*
   The cases where we found a difference of more than 10% were ruled out from ROIperf evaluations. This test works well for single-threaded applications. For multi-threaded programs, where run-to-run variation is expected due to different amounts of synchronization code, the instruction count test may not be adequate.
2) *Are the regions described using (PC, count) specifications executed on the test machine?*
   We tested this with a Pin-based tool to report ROI start and end events based on (PC, count) ROI specification. If the ROIs are not being executed, this implies subtle control flow diversion between the region selection and

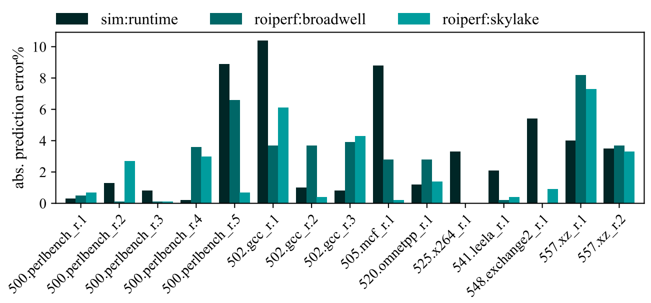

Fig. 4. Sampling error in predicting cycles-per-instructions (CPI) for single-threaded workloads from the SPEC CPU2017 suite using train inputs. The errors were measured using both a cycle-level simulator and the ROIperf tool running on Broadwell and Skylake hardware platforms.

ROIperf runs. Any cases with a substantial number of ROIs missed were ruled out.

We demonstrate the ROIperf methodology using simulation-based region validation as the base case and compare the prediction errors reported by the simulation to those reported by ROIperf. We do this for relatively shorter train input for SPEC CPU2017 runs as the simulation of ref inputs is otherwise not practical.

B. *Evaluation of Single-threaded Applications*

We use the *rate* setup from SPEC CPU2017 benchmarks. The binaries used were compiled using GCC to use the AVX vector instructions. The simulator used was, CoreSim, an SDE-based simulator modeling an Intel Skylake processor. ROIperf evaluations were done on two test machines, one with a Broadwell processor and another with a Skylake processor. The region selection was done using the PinPoints methodology with a slice-size of 200 million instructions and a maximum cluster count (maxk) of 50.

1) SPEC CPU2017 with train input: 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 as shown in Figure 2. The measurement was repeated several times, and the average values were considered. The entire evaluation took a few hours, which is a significant improvement over the simulation-based validation methodology. Figure 4 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.

2) SPEC CPU2017 with ref input: 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 as the simulator, the evaluation times are much shorter. Figure 5 reports the prediction errors for ROIperf-based validation of SPEC CPU2017 ref input runs on running Broadwell and Skylake servers. ROIperf applicability testing (Section V-A) revealed a significant variation (¿15%) in the instruction count between the native run on the test machine and the profiling run. On the Skylake machine, all runs of `503.bwaves_r` showed more than 50% difference between instruction count during profiling and during ROIperf run. We observed the Skylake machine happened to have a different version of the math library than the Broadwell machine, and the code executed on the two machines was quite different, as measured by the instruction mixes on both machines. Security updates often add input checks that can lead to significant slowdowns. For example, the *libm* library on the Broadwell machine uses an optimization that removed the canonical input check from `pow()`, which led to a 2.4× reduction in the instruction count for `503.bwaves_r`[2].

C. *Evaluation of Multi-threaded Applications*

Evaluating synchronizing multi-threaded applications can be quite challenging [22]. Tools like PinPlay [34] offer deterministic analysis of multi-threaded applications. While using ROIperf, we estimate the performance using native hardware. For multi-threaded evaluation, we used the OpenMP subset of the speed version of SPEC CPU2017 benchmarks. The regions of interest (ROIs) of the benchmarks were selected using LoopPoint methodology using the settings as described in Section IV-B.

1) SPEC CPU2017 with train input: Figure 6 shows a comparison between the RDTSC 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 very similar trends in the estimation errors, especially for applications like `627.cam4_s.1`.

2) NPB using Class C inputs: We repeat the comparison of prediction errors from ROIperf and simulation for NAS Parallel Benchmarks (NPB) Class C input size. Figure 7 shows the runtime prediction errors obtained from simulation (Sniper:Gainestown), and prediction errors for user-level hardware CPU cycles and RDTSC using ROIperf. Again the error bars show similar trends which signify the reliability of the results obtained using ROIperf.

[2]The results shown here have been filtered to exclude these specific cases.

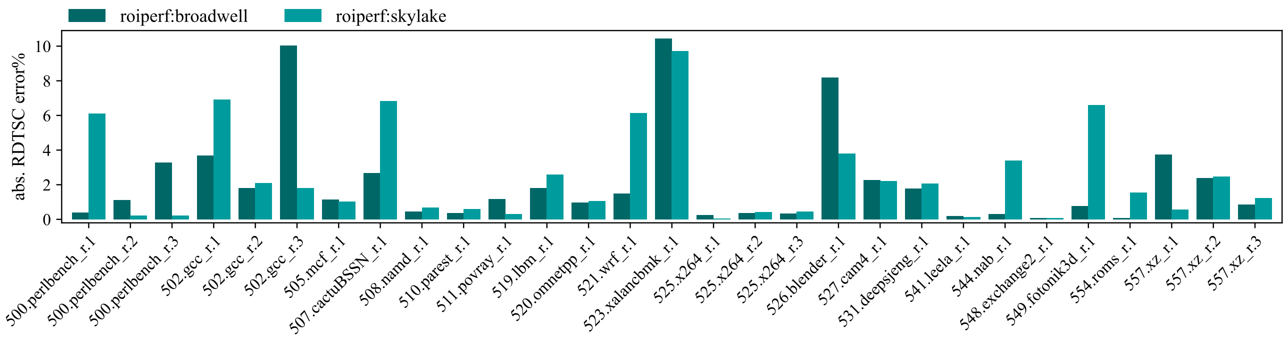

Fig. 5. Sampling error in predicting the RDTSC values of the single-threaded SPEC benchmarks using ref input.

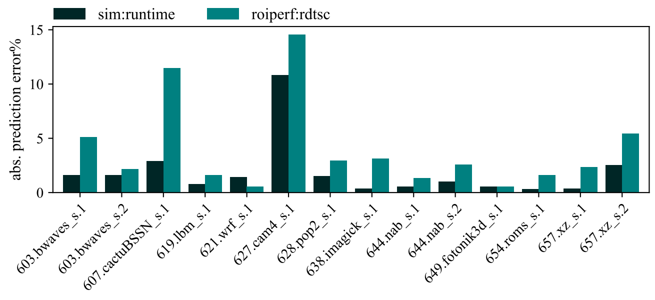

Fig. 6. A comparison of RDTSC estimation error using ROIperf and runtime estimation error using CoreSim simulator. The benchmark suite is SPEC CPU2017, and the benchmarks use 8 threads, train inputs, and active wait policy. The ROIs are identified using LoopPoint methodology.

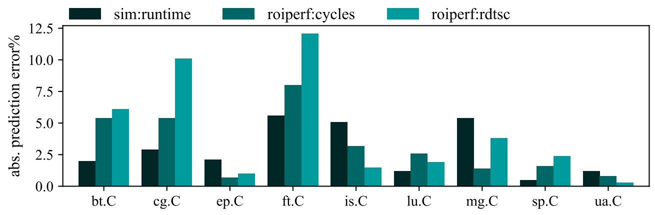

Fig. 7. A comparison of simulation-based prediction errors with ROIperf results for both HW_CPU_CYCLES and RDTSC projections on a Skylake Server. We use NPB benchmarks that use Class C inputs, 8 threads and passive wait policy.

VI. Related Work

The overhead of the Linux perf_event counter interface that ROIperf uses is described in prior works [35]. ROIperf uses the self-monitoring interface as described earlier and hence is prone to various overheads, namely overheads for performance counter starting, reading, reading multiple times, and stopping. The paper suggests turning off dynamic frequency scaling to avoid affecting the RDTSC instruction results. We did that for our test machines. They also suggest using static linking to avoid dynamic link overhead of the `read()` system call used to read performance counters.

Hardware performance counter-based simulation region validation was reported in [21] for single-threaded SPEC2000 Itanium programs. Region selection was done with SimPoint [2] with a fixed region length of slice-size instructions. For evaluation, a JIT-mode Pin tool was used that ran till the beginning of an ROI using instruction count and then detached from the underlying application. Thus, the run till the beginning of the ROI was under Pin and on native hardware afterward. The Pintool run was launched with a performance monitoring Linux tool sampling specified hardware performance counters at slice-size intervals. Thus, the first sample after Pin detached from the application roughly corresponded with the ROI. ROIperf does not detach from the application, but since it is a Pin probe tool, it has negligible overhead. ROIperf can monitor ROI boundaries more precisely, especially if (PC, count) specifications are used. ROIperf also handles variable-sized ROIs from both single and multi-threaded programs.

While repeatability of program execution is crucial for accurate ROI analysis, traditional profiling approaches can suffer from variations across runs. ELFies [18] are user-level executable checkpoints for regions of interest that offer a promising solution. Extracted from deterministic replays of the full-program recording used for profiling, ELFies inherently capture the initial state of the ROI as observed during profiling. This significantly reduces the impact of non-repeatability, as it only affects the native execution of the ELFie itself, not the entire program execution leading up to and within the ROI that ROIperf analyzes. However, imprecise reconstruction of the operating system state during ELFie creation can lead to system call failures or altered behavior within the ELFie. These discrepancies can introduce deviations from the original execution and potentially cause application failures.

VII. Conclusion

We introduce ROIperf, a technique for evaluating the quality of workload sampling methodologies. ROIperf leverages hardware performance counters to validate the representativeness of chosen samples, which can be used in several ways to study the workload characteristics, core interactions, cache behavior, etc., without requiring a simulator.

Our analyses show that the program behavior needs to be repeatable across multiple executions to obtain stable measurements. Simulators provide a controlled environment for performance estimation and, especially in the case of multi-threaded applications, control the thread progress. ROIperf facilitates rapid validation of workload sampling methodologies for large-scale workloads like the SPEC CPU2017 benchmarks with reference inputs. This enables us to efficiently evaluate the effectiveness of sampling approaches.

ROIperf could be expanded to support compatibility beyond specific hardware platforms, particularly towards the increasingly heterogeneous nature of AI and HPC applications. Supporting heterogeneous platforms like CPU-GPU systems would require integrating GPU performance counters, specifically shader clock counters, into the ROIperf framework.

References

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

[2] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, October 2002.

[3] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *International Symposium on Computer Architecture (ISCA)*, pages 84–97, June 2003.

[4] 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*, 26(4):18–31, 2006.

[5] E. K. Ardestani and J. Renau. ESESC: A fast multicore simulator using time-based sampling. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 448–459, February 2013.

[6] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sampled simulation of multi-threaded applications. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12. IEEE, 2013.

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

[8] Alen Sabu, Harish Patil, Wim Heirman, and Trevor E. Carlson. Looppoint: Checkpoint-driven sampled simulation for multi-threaded applications. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2022.

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

[10] Qinzhe Wu, Steven Flolid, Shuang Song, Junyong Deng, and Lizy K John. Invited paper for the hot workloads special session hot regions in spec cpu2017. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 71–77. IEEE, 2018.

[11] Haiyang Han and Nikos Hardavellas. Public release and validation of spec cpu2017 pinpoints. *arXiv preprint arXiv:2112.06981*, 2021.

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

[13] Andi Kleen and Beeman Strong. Intel processor trace on linux. *Tracing Summit*, 2015, 2015.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

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

[16] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *International Symposium on Code Generation and Optimization (CGO)*, pages 2–11, April 2010.

[17] Trevor E. Carlson, Wim Heirman, Harish Patil, and Lieven Eeckhout. Efficient, accurate and reproducible simulation of multi-threaded workloads. In *Workshop on Reproducible Research Methodologies (REPRODUCE)*, February 2014.

[18] Harish Patil, Alexander Isaev, Wim Heirman, Alen Sabu, Ali Hajiabadi, and Trevor E Carlson. ELFies: Executable region checkpoints for performance analysis and simulation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 126–136, February 2021.

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

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

[21] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *International Symposium on Microarchitecture (MICRO)*, pages 81–92, December 2004.

[22] A.R. Alameldeen and D.A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006.

[23] 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 *International Symposium on Computer Architecture (ISCA)*, pages 29–42, June 2018.

[24] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255. IEEE, 2003.

[25] Björn Gottschall, Silvio Campelo de Santana, and Magnus Jahre. Balancing accuracy and evaluation overhead in simulation point selection. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 43–53. IEEE, 2023.

[26] Performance monitoring in the intel 64 and ia-32 architectures software developers manual, volume 3b. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html.

[27] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[28] Alen Sabu, Harish Patil, Wim Heirman, and Trevor E. Carlson. LoopPoint source code. https://github.com/nus-comparch/looppoint, 2022.

[29] T. E. Carlson, W. Heirman, and L. Eeckhout. 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)*, pages 52:1–52:12, November 2011.

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

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

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

[33] T. E. Carlson, W. Heirman, and L. Eeckhout. The Sniper multi-core simulator. https://snipersim.org. https://github.com/snipersim/snipersim.

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

[35] Vincent M. Weaver. Self-monitoring overhead of the linux perf_event performance counter interface. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 102–111, 2015.