ROIperf: A Framework to Rapidly Validate Workload Sampling Methodologies

Alen Sabu
alen@u.nus.edu
National University of Singapore

Harish Patil
harish.patil@intel.com
Intel Corporation

Wim Heirman
wim.heirman@intel.com
Intel Corporation

Trevor E. Carlson
tcarlson@nus.edu.sg
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 regions of interest (ROIs) or the representative sample found can be proven to accurately represent the behavior of the full workload. One standard way to validate the regions of interest is to measure the prediction error, which is the difference in the performance of the full workload and the predicted performance obtained using the representative. Performance is typically obtained using simulation. However, simulation of long-running workloads is infeasible, taking months to years.

In this work, we propose ROIperf, a framework that assesses the quality of workload sampling methodologies based on hardware performance counters by evaluating both the full and representative workloads using real hardware systems. ROIperf allows for the accurate validation of regions of interest by measuring the performance using real hardware instead of simulation. The work aims to present a methodology for long-running programs for which the prevailing simulation-based validation technique is not feasible. We demonstrate the efficacy of ROIperf by evaluating various sample selection methodologies across a wide range of workloads. We evaluate single-threaded and multi-threaded SPEC CPU2017 benchmarks as well as the NAS Parallel Benchmarks to test the proposed technique. ROIperf provides a significant speedup in validating regions selected for simulation, allowing for more widespread use of sample verification, especially for long-running workloads.

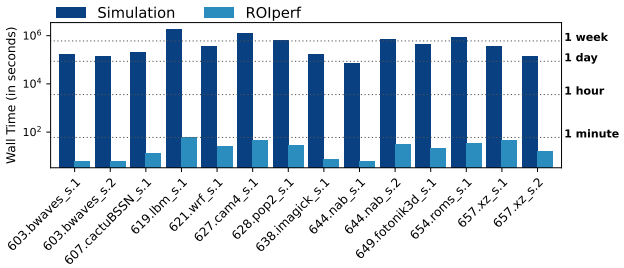

Figure 1: A comparison of the total wall-time required to validate the representativeness of the sample identified for multi-threaded SPEC CPU2017 benchmarks using *train* inputs (the gap is expected to increase for *ref* inputs). The bars show the time taken to validate LoopPoint [22] methodology using a cycle-accurate simulator and using the ROIperf tool, assuming infinite resources.

1 INTRODUCTION

Cycle-accurate, detailed simulation of computer systems can be rather slow, with simulation speeds of complex, modern processor designs as low as a few thousand instructions per second, or 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, simulating *regions of interest* (ROIs) from large application runs and extrapolating whole-program performance is a standard technique employed [26, 23, 30, 28, 3, 7, 6, 22]. To gain confidence in the extrapolated results, it is necessary to validate that the ROIs selected closely represent whole-program behavior [11, 29, 9]. Traditionally, such validation is done by comparing the simulated performance of the entire program with the performance extrapolated from ROI simulations. However, since whole-program simulation for most realistic applications are impractical, to begin with, such simulation-based validation is limited to either short-running programs and/or using fast but inaccurate simulators.

A much faster alternative to using architecture simulators for validation is performance monitoring on native hardware. Figure 1 shows how the validation of representative regions using our proposed ROIperf methodology can be much faster than simulation-based validation. Selecting the most representative regions in an application requires tuning the parameters used for region selection and re-validating the newly selected regions. This is an iterative process that is impractical without extremely fast techniques like the one proposed in this work, thus validating the efficacy of a workload sampling methodology.

While measuring the whole-program performance on native hardware today can be straightforward [20, 12], performing these measurements for regions of interest can be tricky. To keep simulation times in check, ROIs are typically very small compared to the whole-program execution: a few million instructions running for mere milliseconds on real hardware. Precisely gathering performance information for just the ROIs on native hardware with high accuracy can be challenging. For example, the loop-based representation of ROIs is considered highly accurate and reproducible [22]. However, identifying such representation of the regions on native hardware is not straightforward. In an attempt to address this

challenge, we present ROIperf, a methodology that incorporates a lightweight instrumentation technique that provides the required control and precision for this methodology. A Pin probe tool [14] is very low overhead as it works by patching an in-memory image of the application instead of using just-in-time (JIT) compilation that can exhibit high overheads [4] that interfere with the workload itself. 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 using the ROI specification.

The repeatability challenge

With profile-based simulation region selection techniques such as SimPoint [23], at least two executions of the test program are involved. In the first execution of the program, the profiling results are collected, which are used for selecting ROIs. The second execution uses the regions selected for simulation. The assumption is that the two runs are identical in all respects. However, such strict repeatability is hard to guarantee in practice, especially for multi-threaded programs. This can be due to machine differences (for example, different ISA support leading to scalar vs. vectorized runs), different system libraries (because machines in a simulation pool have different operating systems or even slightly different patch levels), or timing-dependent control flow (for example, work stealing in parallel applications).

The PinPlay work [18] proposes a record and replay framework to guarantee the repeatability of program executions during multiple analyses. The authors show how to enable repeatable analyses of parallel programs by recording the entire program execution, and then, the profiling and simulation are done using a deterministic replay of the whole-program recording. However, the overhead of PinPlay replay is high ($\approx 50\times$), so performance counter-based evaluation becomes inaccurate. Other efforts to improve repeatability include using static binaries, checkpoints [8], or ELFies [17]. Still, none of these techniques can guarantee fully repeatable execution, especially in multi-threaded scenarios where timing-dependent control flow and the resulting execution divergence can happen more often [19, 2]. We, therefore, need tools that can detect the divergence and allow for mitigation (by debugging machine/library differences, retrying the run, selecting a different and more reliable representative from a cluster, etc.).

With ROIperf, we perform the quality evaluation of the ROI using a native program run. We, therefore, cannot guarantee that the program run under ROIperf is identical to the earlier run during region selection either. We can minimize the effects of the repeatability challenge by performing ROI selection, and ROIperf runs in the exact same environment. We describe tests for the applicability of ROIperf prior to the measurement in Section 5. In our evaluations, we find that ROIperf succeeds in finding the regions accurately in most cases.

Contributions

In this paper, we present ROIperf, a framework to evaluate the efficacy of workload sampling methodologies quickly. We show its usage in validating ROIs of long-running single and multi-threaded workloads. The ROIs were selected using the PinPoints [15] (for single-threaded executions) and LoopPoint [22] (for multi-threaded programs) methodologies. Both these selection methodologies use deterministic replay [18] based profiling for region selection. For accurate ROI validation, it is ideal to have the control flow during the ROIperf-based run be exactly the same as during the profiling run during ROI selection. This is very hard to achieve, especially for multi-threaded programs. We use sanity tests (see Section 5.1) to verify reproducibility and to rule out extreme cases of control flow diversion. We will open-source the ROIperf tools for use in the community.

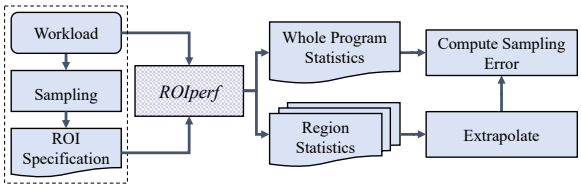

Figure 2: An overview of the working of ROIperf tool.

2 BACKGROUND

2.1 Hardware Performance Counters

Modern microprocessors have special hardware registers called hardware performance counters for monitoring various performance-related metrics [21]. 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 [24]. 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.

2.2 Workload Sampling Techniques

Single-threaded Workloads. SimPoint [23] is a well-established technique for sampling single-threaded programs for simulation. It requires the collection of signature vectors (typically basic-block-vectors, BBVs) every sampling period (slice-size). The BBVs generated are then clustered using the k-means clustering algorithm to find up to *maxk* phases. 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 toolkit was first introduced in [15], where Pin was used for BBV generation of x86 applications. Noticing the problem of repeatability of some regions, a deterministic replay-based version of the PinPoints kit (for x86) was introduced in [18]. The overall flow of the PinPlay-based x86 PinPoints toolchain is described in Figure 3.

Multi-threaded Workloads. The problem of sampling multi-threaded workloads is even more challenging, and naive extensions of single-threaded workload sampling techniques cannot be applied directly due to the presence of thread interactions, spin-loops, etc. There are several techniques proposed to sample multi-threaded workloads [3, 7, 6, 22], each having its own limitations. In this

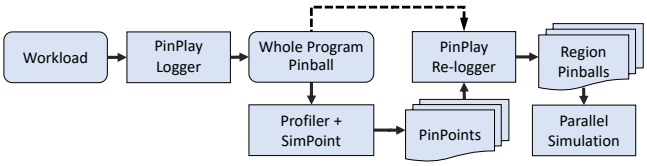

Figure 3: The overview of PinPoints methodology with PinPlay and SimPoint applicable to single-threaded workloads.

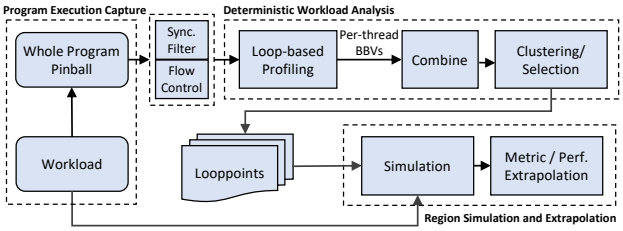

Figure 4: The workflow of LoopPoint sampled simulation methodology that applies to multi-threaded workloads.

work, we evaluate a recently proposed generic multi-threaded sampled simulation methodology, LoopPoint [22]. LoopPoint identifies regions bounded by loop entries and can achieve high speedups without compromising on the sampling accuracy. Figure 4 shows the overall flow of the LoopPoint methodology. The analysis and region identification is done using a deterministic analysis of the workload. The simulation of the identified representative regions is performed to estimate the performance of the individual regions, which are used along with the corresponding multiplier to estimate the performance of the whole workload.

3 IMPLEMENTATION

We describe the details of implementing the ROIperf methodology in this section. We also present and compare the different ways to use the tool for single-threaded and multi-threaded applications.

3.1 Pin instrumentation modes

Pin [14] is a well-known dynamic instrumentation and analysis framework. It offers an application programming interface (API) for adding extra code at various program points. The API differs based on the mode specified during Pin initialization. Pin supports two modes: (1) a just-in-time (JIT) mode which translates the test program in memory and adds instrumentation during the translation, and (2) a probe mode which merely patches an in-memory copy of the program with extra code. The JIT mode API is quite rich and allows for very 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 uses Pin in probe mode due to its low overhead, which minimizes perturbation during performance measurement of the target application.

3.2 Region of Interest (ROI) specification

We select the simulation regions of single-threaded applications using a variation of SimPoint [23] methodology where an application is profiled to generate signature vectors, typically basic-block vectors, after every execution slice (indicating *unit of work*), and the resulting vectors are clustered to form multiple phases. A representative ROI is selected for each phase, and the ROI carries a weight proportional to the size of the phase it represents. The selected ROIs are then used to drive architectural simulation. The assumption is that the ROIs selected are reached precisely during the simulation as they were during the initial profiling run.

We factor in the repeatability aspect of the programs while evaluating ROIperf so that the ROIs remain representative of the original application. Single-threaded program runs may not always be repeatable [18]. The main reasons for non-repeatability for single-threaded programs are changes in the shared library version and the load and stack locations. Therefore, to ensure ROI validity, one needs to make sure the exact same shared libraries are used, and the library load and stack start location are made to match those during the profiling run used to find ROIs. On Linux, that can be achieved by turning off 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 1) of single-threaded programs. For multi-threaded programs, a significant source of non-repeatability is the synchronization among threads [18]. Instruction counts are, therefore, not a reliable way to specify ROI boundaries. The LoopPoint [22] methodology ensures that the ROIs found are repeatable by choosing units of work that start and end at a *worker* loop (not-spinloop) entry, a loop that is not doing synchronization and representing ROI start and end as (PC, count) pairs for the corresponding loop entry addresses. With ROIs, starting/stopping at worker loop entries ensures that the start and end counts will be invariant across executions. ASLR needs to be turned off during profiling and measurements for ROI boundary PCs to be repeatable.

3.3 ROI handling in ROIperf

The goal of the ROIperf methodology is to output interesting hardware performance counter values at the beginning and the end of each ROI. ROIperf uses the Linux function call `perf_event_open()` to program hardware performance counters. The performance counters of interest are specified based on the Linux header file `/usr/include/linux/perf_event.h`. The specification is done by setting an environment variable ROIPERF_LIST as a comma-separated list of number pairs perftype:counter, where perftype is either `0` (indicating a hardware counter) or `1` (indicating a software counter). Other perftypes are currently not supported. The `counter` values are set based on the information in the `perf_event.h` file. For example, the perftype:counter pair `0:0` indicates hw_cpu_cycles, and `1:2` indicates sw_page_faults.

[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`.

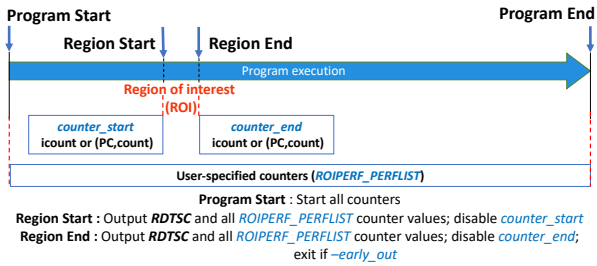

Figure 5: The Execution flow of an application under the ROIperf tool.

ROIperf works on an application and its ROI specification, as shown in Figure 2. Hardware performance counters are programmed in two ways: (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 call-back function name. For an instruction-count-based ROI specification, two counters are programmed to measure PERF_COUNT_HW_INSTRUCTIONS in user mode with an overflow value equalling the start and end instruction count for the ROI. When the counter overflows, the call-back function registered gets called. The callback function outputs the current value system-wide time-stamp counter using the Read Time-Stamp Counter (RDTSC), which reads the current value of the processor's time-stamp counter. It also outputs the current values of the interesting performance counters programmed in the continuous mode. Various actions taken by the ROIperf tool are described in Figure 5. For (PC, count) ROI specification, two counters are programmed with perftype PERF_TYPE_BREAKPOINT with the PCs of the start and the end of the ROI. Overflow values are set to the `count` values for the start and end specifications. The call-back function outputs RDTSC values and the values of performance counters from ROIPERF_LIST.

We noticed that PERF_COUNT_HW_INSTRUCTIONS counting and overflow handling are very efficiently handled in all the x86 processors we tested. PERF_TYPE_BREAKPOINT, on the other hand, is quite expensive to use. Every time the programmed PC is executed, there is a trap in the operating system kernel, which checks the overflow using a software counter. This causes a lot of perturbation in performance measurements, especially if the PCs involved in the ROI specification are frequently executed. We, therefore, think using (PC, count) specification only for the start of the ROI and then using a relative instruction-count-based PERF_COUNT_HW_INSTRUCTIONS for getting the ROI end is the best compromise. Since we are only interested in the delta between starting and ending performance metrics for an ROI, reaching the ROI start with a precise but expensive (PC, count) mechanism and reaching the ROI end with a fast albeit imprecise (particularly for multi-threaded programs) is acceptable.

In the case of multi-threaded programs, only the main thread (thread 0) is running at this point; hence ROIperf only starts hardware performance counters for the core/processor where the main thread is running. Pin probe does not monitor thread creation; hence there is no callback to the ROIperf tool when threads are created later during program execution. Therefore ROIperf cannot monitor any children threads in a multi-threaded program. RDTSC values that ROIperf outputs do, however, capture time for the entire ROI, including all threads. As long as the main thread (thread 0) is active inside all ROIs being tested, ROIperf's approach of monitoring only thread 0 execution will suffice. The counters specified with ROIPERF_PERFLIST will only be counted for the core/processor where the main thread runs through.

4 EXPERIMENTAL SETUP

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.

Sample Selection. For single-threaded sampling, we use PinPlay-based profiling methodologies involving the PinPoint [15] tool, derived from the SimPoint [23] 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 [13] 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.

Simulators Used. We use two different simulators for our experiments. For our experiments with the SPEC CPU2017 benchmark, we use an in-house simulator derived from Sniper [5], called CoreSim, for evaluations. CoreSim allows for rapid yet fairly accurate simulation of x86 many-core systems that use SDE [10] as the simulation front-end. We simulated both Intel Skylake and Intel Cascade Lake microarchitectures using CoreSim. We also use Sniper multi-core simulator [5, 25] version 8.0 (using Pin [14] front-end)

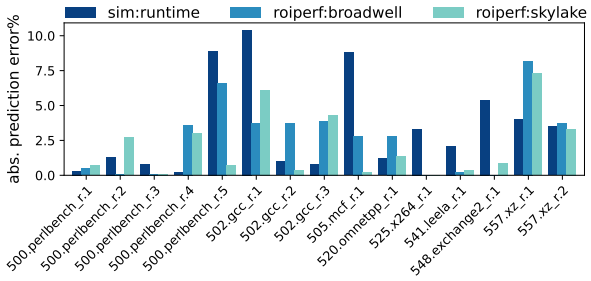

Figure 6: Sampling error in predicting cycles-per-instructions (CPI) for single-threaded SPEC CPU2017 benchmarks that use train inputs

for our evaluations with NPB benchmarks. We configured Sniper to simulate a microarchitecture similar to Intel Gainestown.

5 EVALUATION

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

5.1 Testing ROIperf applicability

As discussed in Section 1, 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) ***Does the thread-0 instruction count match between the region-selection run and the ROIperf run?***
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) ***Do the regions described using (PC, count) specification get reached 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. (PC, count) specified ROIs not getting reached implies subtle control flow diversion between the region-selection and ROIperf runs. Any cases with a substantial number of ROIs missed were ruled out. Note that instruction-count-based ROI specification will not catch control flow divergence as the (PC, count) specification does.

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.

5.2 Single-threaded evaluation

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.

SPEC CPU2017 with train input. We first simulated the binaries running train input with CoreSim in two ways (1) for the entire program execution and (2) once each for each ROI selected by PinPoints (instruction-count-based specification). Prediction error for each benchmark was computed using the simulated runtime, whole-program, and region-projected. The longest-running whole-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 whole-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 6 reports the prediction errors for simulation and ROIperf-based validation. We see that while the absolute prediction error values differ, the trends in prediction errors are the same between simulation-based and ROIperf-based validation. This gives us confidence in using ROIperf as a much faster alternative to simulation-based ROI validation.

SPEC CPU2017 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 whole-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 7 reports the prediction errors for ROIperf-based validation of SPEC CPU2017 ref input runs on running Broadwell and Skylake servers. The missing entries (✕) are where we encountered the repeatability challenge described in Section 1. ROIperf applicability testing (described earlier in Section 5.1) shows test machine native-run instruction count shows more than 15% variation from the profiling run instruction count for the ✕ cases. On the Skylake machine, all runs of `503.bwaves` 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. The *libm* library on the Broadwell machine had an optimization that removed the canonical input check from `pow()`, which led to a 2.4× reduction in the instruction count.

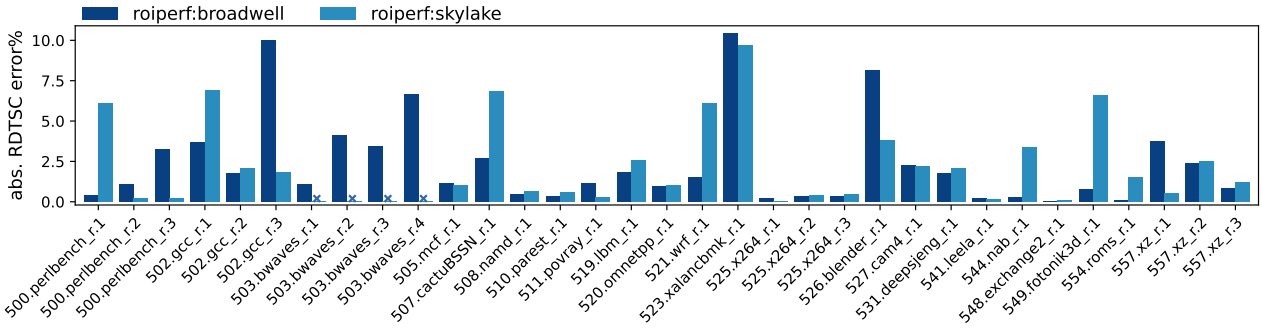

Figure 7: Prediction errors in cycles-per-instructions (CPI) computation for the single threaded SPEC benchmarks using ref input. For the cases marked ×, ROIperf is not applicable due to repeatability challenges.

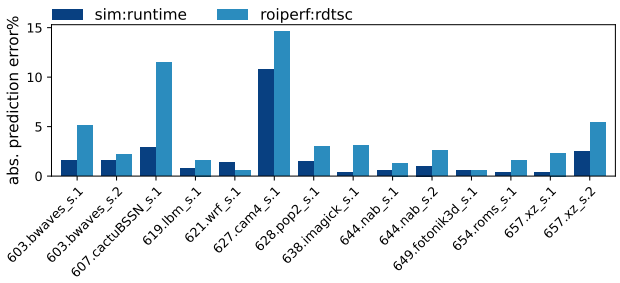

Figure 8: A comparison of RDTSC estimation error using ROIperf and runtime estimation error using CoreSim simulator. The benchmark suite is SPEC CPU2017, and the benchmarks use 8 threads, train inputs, and active wait policy. The ROIs are identified using LoopPoint methodology.

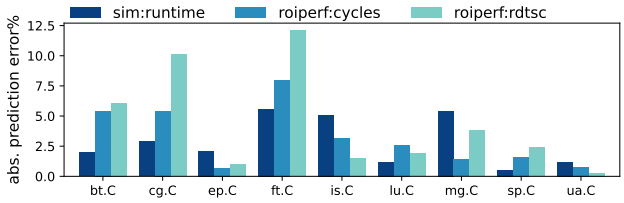

Figure 9: A comparison of simulation-based prediction errors with ROIperf results for both HW_CPU_CYCLES and RDTSC projections on a Skylake Server. We use NPB benchmarks that use Class C inputs, 8 threads and passive wait policy.

5.3 Multi-threaded evaluation

Evaluating synchronizing multi-threaded applications can be quite challenging [1]. Tools like PinPlay [16] 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 4.

SPEC CPU2017 with train input. Figure 8 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`.

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 9 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.

6 RELATED WORK

The overhead of the Linux perf_event counter interface that ROIperf uses is described in prior works [27]. 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 [15] for single-threaded SPEC2000 Itanium programs. Region selection was done with SimPoint [23] 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.

7 CONCLUSION

We present ROIperf, a technique to estimate the quality of workload sampling methodologies. ROIperf validates the representativeness of a chosen workload sample utilizing hardware performance counters, which can be used in several ways to study the workload characteristics, core interactions, cache behavior, etc., without the need for a simulator.

Our analyses show that the program behavior needs to be repeatable across multiple executions of the workload to obtain a stable measurement. Simulators provide a controlled environment for performance estimation and, especially in the case of multi-threaded applications, control the thread progress. Using ROIperf, the quick validation of regions chosen for large workloads, like SPEC CPU2017 benchmarks using ref inputs, is now possible, which enables to estimate the efficacy of several workload sampling methodologies.

REFERENCES

[1] A.R. Alameldeen and D.A. Wood. 2006. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26, 4, 8–17. DOI: 10.1109/MM.2006.73.

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

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

[4] Moshe Bach et al. 2010. Analyzing parallel programs with pin. *Computer*, 43, 3, 34–41.

[5] T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* Article 52. (Nov. 2011), 52:1–52:12.

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

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

[8] Trevor E. Carlson, Wim Heirman, Harish Patil, and Lieven Eeckout. 2014. Efficient, accurate and reproducible simulation of multi-threaded workloads. eng. In *Workshop on Reproducible Research Methodologies (REPRODUCE)*. Orlando, FL, USA, (Feb. 2014).

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

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

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

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

[13] 2022. LoopPoint source code. https://github.com/nus-comparch/looppoint. (2022).

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

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

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

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

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

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

[20] 2012. Perf: linux profiling with performance counters. https://perf.wiki.kernel.org/. (2012).

[21] [n. d.] 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. ().

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

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

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

[25] [n. d.] The Sniper multi-core simulator. https://snipersim.org. https://github.com/snipersim/snipersim. ().

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

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

[28] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. 2006. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26, 4, 18–31. DOI: 10.1109/MM.2006.79.

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

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