LoopPoint: Checkpoint-driven Sampled Simulation for Multi-threaded Applications

Alen Sabu
National University of Singapore

Harish Patil
Intel Corporation

Wim Heirman
Intel Corporation

Trevor E. Carlson
National University of Singapore

***Abstract*—Generic multi-threaded sampled simulation has been a long-standing, challenging problem with the potential to help change how researchers study modern, complex computing systems. Yet, a practical solution for reducing complex multi-threaded applications into a representative sample has been elusive. Existing techniques either do not provide significant speedups to be useful (Time-based Sampling techniques can show less than a 10× speedup compared to a fully-detailed simulation) or apply only to particular synchronization types (BarrierPoint for barrier-based workloads). In addition, workload-specific solutions can be rigid with respect to region selection, which can limit the overall simulation speedup when regions are large. A solution is needed that both supports generic multi-threaded applications, no matter the synchronization primitives used, as well as allows for ease of deployment and fast evaluation.**

In this work, we aim to solve these challenges and propose a novel sampling technique for multi-threaded applications, called LoopPoint, that is both agnostic to the type of synchronization primitives used and scales by the similarity exhibited by the application. The proposed methodology combines several vital features, including (1) repeatable, up-front application analysis, (2) a novel clustering approach to take into account run-time parallelism, and (3) the use of loop-based simulation markers to divide the work into measurable chunks, even in the presence of spin-loops. LoopPoint identifies representative simulation regions that can be simulated in parallel to achieve speedups of up to 801× for the train input set of the multi-threaded SPEC CPU2017 benchmarks with an average simulation error of just 2.33%. For the ref inputs of CPU2017, we calculate the speedup with LoopPoint to be 11,587× on average (for parallel simulation), and up to 31,253×, demonstrating how the identification of application regularity and loops can lead to significant simulation improvements compared to state-of-the-art solutions.

***Index Terms*—checkpointing; multi-threaded; record-and-replay; sampling; simulation**

I. Introduction

Sampling is a well-known application workload reduction technique that traces its roots back decades. From the earliest works [1], [2], researchers have been able to identify regularity in single-threaded applications and exploit that to sample large applications into smaller application representatives. Because of the repeated execution of regions with similar behavior, these techniques have been shown to accurately predict the original workload behavior, and significantly reduce the simulation time needed [1], [2].

Apart from sampling, researchers have developed a number of complementary techniques to reduce the overall amount of work required to simulate applications in detail, including input size reduction [3] and benchmark synthesis [4]. While each

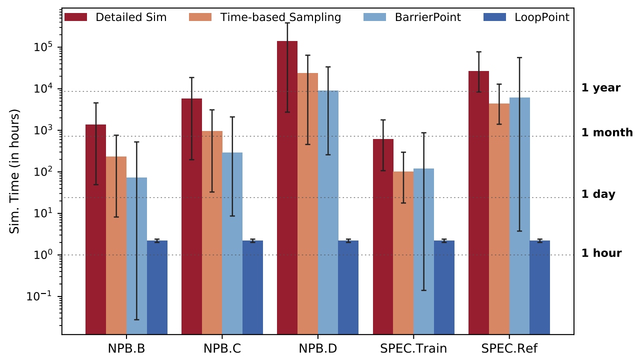


Fig. 1: Approximate time to evaluate the performance of multi-threaded benchmarks with different methodologies. *The average result and error bars represent the estimated simulation time for all benchmarks in the corresponding suite and input sets, assuming infinite simulation resources (the longest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

technique presents its benefits and challenges, sampling has emerged as a straightforward way to maintain the original application characteristics and accurately extrapolate performance while reducing the overall simulation burden.

With the increasing number of cores in modern processors, multi-threaded applications can exploit a large amount of compute through task and loop parallelism. Simulating these large, multi-threaded applications is extremely difficult, even on modern simulators. Ultra-fast FPGA-based simulators [5] require detailed implementations and are capacity-limited, preventing the simulation of large processors and large parallel systems, and fast software-based simulators [6], [7] still require a significant amount of time to run an entire large, parallel workload to completion. Multi-threaded applications are inherently difficult to analyze [8] as the threads can go to sleep at any time, threads interfere with one another, and complex behavior emerges from regular application parameters like misalignment of threads to cores and unequal cache distribution.

Some of the earliest multi-threaded sampling solutions prove effective when the threads themselves do not synchronize but can still interact with the memory hierarchy [9]. Any amount of synchronization requires thread progress to be measured in time to track the amount of progress or parallelism in the application. The move toward a time-based sampling methodology has led to the development of sampling techniques for synchronizing multi-threaded applications. These techniques [10], [11] describe one of the first generic sampling solutions for multi-threaded applications. However, the overall simulation speed is still bound to the total application length, which dominates the simulation time of this methodology. Later proposals, in the form of application and synchronization-specific methodologies [12]–[14], exceeded the performance of time-based sampling and allowed for the simulation complexity to be bound to application diversity, not application length. Unfortunately, these methodologies are tied to specific application characteristics (the use of barriers [12] or tasks [13], [14]), and therefore do not represent a general sampling solution that covers all application types. In fact, as Figure 1 demonstrates, both time-based sampling, and BarrierPoint (when inter-barrier regions exist to simulate), **approach a simulation time of one year to simulate the sample** when considering large, multi-threaded applications. Clearly, current methodologies are insufficient for simulating the largest, most realistic benchmarks like the multi-threaded SPEC CPU2017 with the `ref` input set.

In this work, we aim to overcome the limitations of these prior works to enable synchronization-agnostic application sampling for multi-threaded workloads while still scaling the amount of work based on the representative nature of the application. To accomplish this goal, we present the *LoopPoint* methodology that reduces an application to a few representative regions, called *looppoints*, by taking into account several key factors like understanding (1) *where to simulate* which requires (1a) an accurate analysis methodology that can provide for reproducible analysis, and (1b) using a precise clustering mechanism that partitions the regions to reduce the workload into its representative components. In addition, our methodology presents (2) *how to simulate* the regions to allow the application to take advantage of the underlying hardware, while not constraining execution to a deterministic path [15] that might not exhibit true application behavior.

We make the following contributions in this work:

1) A representative simulation region selection methodology called *LoopPoint* suitable for the performance projection of multi-threaded programs (more details on supported workloads in Section III-K) based on using loop iterations as the unit of work.
2) A technique to enable multi-threaded sampled simulation by filtering out spin-loops during region identification, selecting repeatable loop boundaries of a practical region size, and accurately extrapolating performance characteristics.
3) The development of a process to record a constrained application checkpoint for accurate analysis and subsequently simulate the workload's unconstrained behavior during simulation.
4) A comprehensive evaluation of the LoopPoint methodology to demonstrate the potential for speedup while maintaining accuracy using the OpenMP-based multi-threaded subset of SPEC CPU2017 benchmark suite and NAS Parallel Benchmarks (NPB).

In the following sections of this work, we first provide an overview the LoopPoint methodology, results, and evaluation. In Section II, we detail each of the components needed for a fast, accurate, and generic multi-threaded sampled simulation. In Section III, we describe the LoopPoint methodology. We then detail the experimental infrastructure and setup in Section IV, evaluate the LoopPoint methodology in Section V. Finally, we compare to related work (Section VI) and conclude the paper (Section VII).

II. Fast and Generic Multi-threaded Simulation Requirements

Time-based sampling methodologies [10], [11] present the first workable solution to sample generic multi-threaded applications. However, the speed-ups achieved (up to $5.8\times$) using these methodologies are limited by the need to visit the entire application. To achieve high speed-up while maintaining accuracy during multi-threaded workload sampling, we need to consider the inherent application regularity and the amount of parallelism present in the workload at any particular time. We need to define a unit-of-work that is suitable to exploit the application regularity and, at the same time, is applicable across a variety application and synchronization types. The key is the ability to (1) recognize representative regions in a generic way across multi-threaded workload types, and to (2) classify these regions considering application parallelism. To this end, we present a new application sampling methodology called *LoopPoint* that (a) uses loop iterations as the main unit of work, (b) utilizes constrained *pinballs* [16] (user-level checkpoints that allow for reproducible analysis), (c) employs heuristics to remove synchronization during analysis, but use them during simulation, and (d) performs unconstrained simulation of the selected simulation regions allowing for fast and accurate workload evaluation. Figure 2 shows the overall methodology.

Sampling methodologies that rely on instruction counting can perform poorly when dealing with multi-threaded applications [17]. We demonstrate this by performing a naive adaptation of Simpoint [1] for multi-threaded applications of SPEC CPU2017 using 8 threads. With this methodology, the average error obtained for the applications using active wait policy is 25% and as high as 68.44%, whereas errors for the passive wait policy are as high as 20%.

Previous works like the BarrierPoint [12] methodology use inter-barrier regions as the unit of work, whereas the TaskPoint [13] methodology applies only to task-based applications that use task instances as the unit of work. Unfortunately, BarrierPoint, when used to sample large applications with a small number of barriers, can yield negligible simulation speedups. This can be common, especially while sampling realistic workloads for which the length of inter-barrier regions is a bottleneck. BarrierPoint, therefore, is not practical for such workloads. Figure 1 shows how the instruction count (and therefore simulation time) of an inter-barrier region grows

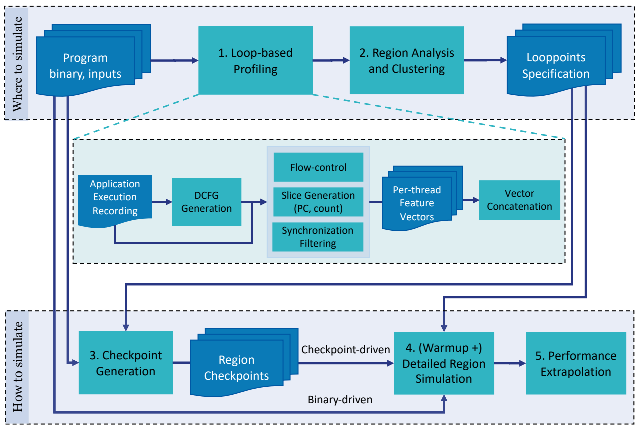


Fig. 2: LoopPoint-based region selection and simulation for multi-threaded workloads. The workload is captured for analysis and region selection based on loop information. The representative regions are simulated using a checkpoint-driven method as well as by binary-driven unconstrained way allowing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

with larger input sets of SPEC CPU2017 and NAS Parallel Benchmarks (NPB) [18] with 8 threads. BarrierPoint works well for NPB with the A input size [12], but as the input sizes grow, for classes C, D and E, inter-barrier regions become so large that it becomes impractical to use BarrierPoint for those input sets. The same is the case with SPEC CPU2017 using `ref` inputs.

Instead, LoopPoint uses loop iterations as the unit of work with the goal to apply to generic multi-threaded programs. The idea of using loop iterations as slices for single-threaded programs was proposed in [19]. With loop entries as slice boundaries, the simulation regions can then be specified using a *(PC, count)* pair for the starting and ending loop entry for each simulation region. By monitoring the amount of work, as represented by loops, and not instructions or barriers, we can isolate multi-threaded application representatives and understand the amount of global work completed. For multi-threaded programs, one additional constraint is that the loop entries that are chosen to start and end slices should be those doing meaningful work. Automatically separating loops doing real work from synchronization can be a daunting task. However, we can use application knowledge or synchronization mechanism details to filter out synchronization loops. For example, the Intel OpenMP run-time uses functions in the `libiomp5.so` library for synchronization; hence loops from that library should not be counted towards *work done* while profiling the application. Alternatively, if the synchronization routines are known before-hand, the code from such routines can likewise be avoided.

Where to simulate. As detailed cycle-accurate simulation can be time-consuming, architects and researchers often use sampling to decide where to simulate by choosing small portions or regions of long-running program executions for simulation. Sampling requires (a) choosing the regions so that they are representative of the whole program behavior and (b) projecting the whole-program performance based on the simulation results of the selected regions. SimPoint [1] is a popular simulation region selection approach. It works by dividing the program execution into smaller slices and collecting an execution signature for each slice. *K-means* clustering is used to determine phases from slice signatures. One representative per cluster is then chosen with the weight corresponding to the cluster size. Since these representatives are designed to be micro-architecture independent, the signature collected for each slice needs to be dependent only on the program execution and not based on any micro-architecture dependent metric. Typical signatures used include the *BBV* (Basic Block Vector) which contains execution counts of various *basic blocks* (single-entry/exit code blocks). How to slice a program's execution into regions is an important decision. For single-threaded programs, using a fixed instruction count called the *slice size* has been shown to work well [1]. In our work, we keep slices of approximately similar sizes demarcated by loop entries. The region selection is based on the replay of a previously recorded whole-program execution as a pinball. According to the micro-architecture of the recording machine, the synchronization seen there can be different from the synchronization seen during unconstrained simulation. We, therefore, augment our region selection methodology to make a selection only on the real computation or work done. The heuristics described earlier to avoid synchronization loop entries as region boundaries can also be used to filter out (to execute but not count) synchronization code during profiling for region selection.

How to simulate. A critical decision that the simulator developers need to make is how to simulate, i.e., how to connect the application in consideration to a simulator. The most commonly used methods are (1) *binary-driven* where a program binary is executed during simulation feeding instructions to the simulator, (2) *checkpoint-driven* where a snapshot of selected region memory/register state and a list of injection events are used to drive the simulator, and (3) *trace-driven* where an instruction-by-instruction recorded state is fed to a timing-only simulator. The choice of how to simulate depends on several factors, such as ease of deployment, cost of generation, and flexibility of the evaluation. For this work, we use both binary-driven and checkpoint-driven simulations for our evaluation, although the implementation itself is generic and supports any of these simulation methods. Checkpoints are easier to share among multiple users than program binaries whose execution might require complex setup and input availability. We propose to capture regions selected by LoopPoint as pinball [20] checkpoints so they can be used to drive PinPlay-based simulators.

By default, PinPlay supports *constrained* replay of pinballs where the shared memory accesses among threads are repeated in the order captured during recording. Simulation based on such constrained replay will repeat the thread ordering based on the micro-architecture of the machine on which the pinballs were generated. However, we ideally want the target, simulated micro-architecture to decide the thread behavior during simulation. To achieve that, we also use binary-driven simulation of the regions selected by LoopPoint using stable (PC, count)-based boundaries defining those regions. Therefore, the simulation proceeds as though the region was executed natively on the simulated micro-architecture. Another technique to achieve unconstrained simulation using pinballs is to convert them to executable checkpoints, called ELFies [21].

III. The LoopPoint Methodology

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

A. *Selecting a Unit of Work*

Multi-threaded applications may use different execution paths with different runs, and therefore the use of IPC to evaluate the performance of multi-threaded workloads is infeasible [17]. LoopPoint proposes a strategy that identifies regions of interest in terms of work done by each thread. We define the unit of work as the actual amount of compute done within a slice of an application. For an unmodified application with the same input set, the unit of work chosen needs to remain the same for each application execution regardless of the properties of the underlying hardware, although the number of instructions executed may vary each time. The generality of the chosen unit of work is crucial for application sampling as this determines the amount of simulation speedup achieved. We would want the chosen unit of work to be large in number within the program, to be one that repeats itself, and to remain unchanged over multiple executions.

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

B. *Understanding Parallelism*

One of the fundamental requirements of a multi-threaded sampling methodology is the ability to understand how the parallelism of an application changes, over time, and to use that information to drive the representative selection process. In fact, understanding parallelism in a generic way is one of the main insights of this work. To accomplish this, we continue to use worker loop instructions as the key metric for work completed.

Program phase behavior is an important aspect to consider while sampling applications. A phase is a set of slices in a program's execution that shows similar behavior, regardless of where they appear within the execution. The locations in source code whose executions correlate to a phase change in the application are called software phase markers [19]. The software phase markers can accurately identify the phase changes which occur in an application execution irrespective of the underlying microarchitecture. These are execution points which can act as simulation region boundaries that are invariant across multiple application executions. We identify source level program loops as possible checkpoints which form the basic building blocks of a program.

Capturing BBVs is an essential way to understand the fingerprint of an application execution region. We consider the slice-size to be approximately $N \times 100$ million global (all-threads) instructions, that align with loop boundaries, for a N-threaded application. For example, we collect BBVs in intervals of approximately 800 million instructions for an 8-threaded application. We ignore the instructions executed in spin-loops or any other synchronization code while collecting the BBVs. The end of a region specified by a BBV is the next loop entry once the instruction-count target is achieved. Although this can be implemented in several ways (as described in [19]), we do not currently differentiate between inner and outer loop markers, and do not restrict specific threads to indicate loop boundaries. The loop entries that serve as region markers need to be worker loops and not spin-loops. We assume that the spin-loops are found only in the synchronization library (for example, OpenMP) and therefore we end a region only at a loop entry that is present in the main image of the application. The per-region BBVs of each thread are concatenated into a longer, global BBV that represents a multi-threaded region. This guides the clustering phase when there are regions that exhibit non-homogeneous thread behavior. Figure 3 shows the ratio of the number of instructions executed by each thread as the application progresses. The application `657.xz_s.2`, as an example, clearly exhibits a non-homogeneous thread behavior.

There are a number of reasons to maintain sufficiently large per-thread slices (approximately 100 million instructions). If a smaller slice-size is chosen, a large number of simulation points may be required, and such regions are highly sensitive to warmup and aliasing issues [11]. At the same time, we also need to make sure that there are enough intervals in the application for the clustering algorithm to work efficiently [22]. Prior analyses [1] on single-threaded applications showed that fixed size (of 100 million instructions) intervals of execution can be used to identify phase behavior. Using varying length intervals [23] corresponding to the application periodicity can help mark the phases more accurately. In LoopPoint, we use approximately similar interval lengths, however, the methodology can also be used with varying length intervals.

While we profile an application for BBVs or any feature

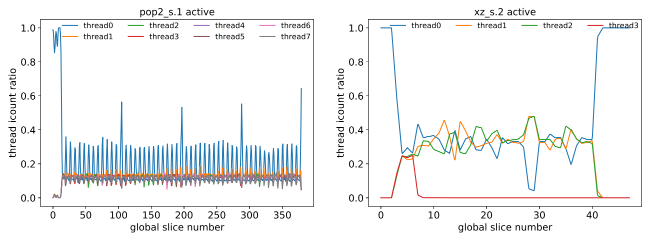


Fig. 3: The above graphs show the variation in the share of the per-thread instruction count on a per-slice (with a slice size of 800M global instructions) basis as the application progresses. If we consider a multi-threaded region, the basic-block share is different for all threads. This is subtly captured by concatenating the per-thread execution fingerprints.

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

C. Marking Region Boundaries

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

D. Identifying Loops using DCFG

Loops are found often in typical applications and the number of loop iterations can remain constant for an unmodified application for a particular input over multiple executions. This is the key to our generic methodology which is explained below in detail.

We employ a Dynamic Control-Flow Graph (DCFG) to identify the regions that represent loops. A DCFG is similar to a classical control-flow graph with a primary difference: Each edge of a DCFG is augmented with a trip count to indicate the number of times the edge was traversed. The source code locations whose executions correlate with a phase change are called a software phase markers [19]. The software phase markers identify the phase changes that occur in an application execution irrespective of the underlying microarchitecture. These phase markers need to repeat in number and order across multiple program executions so that this can meaningfully act as simulation region boundaries. We choose headers of loops that are in the main image of the program assuming that the synchronisation loops are in the libraries. The number of iterations of synchronisation loops may vary across different program executions. The DCFG of the whole program execution is instrumented for loop header instructions to identify a subset of loops from the main image. Loop header instructions are instrumented to emit Basic Block Vectors (BBVs) after *slice-size* number of instructions. Figure 4 shows a region identified using DCFG. The region is contained in the `638.imagick_s.1` application with train inputs and 8 threads.

E. Clustering Representative Regions

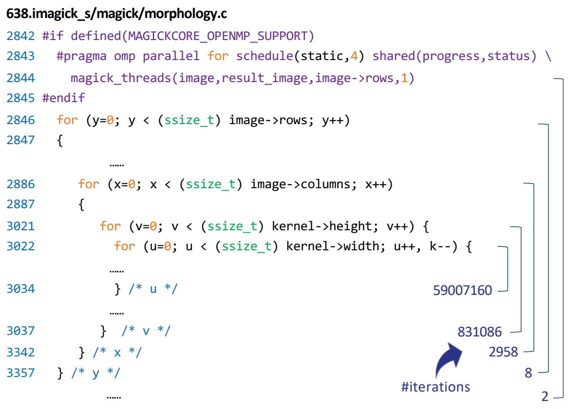
Once an application is profiled, and region boundaries marked, we will have a collection of variable-length regions. These BBVs (with spin-loops filtered) represent the state of the application, and also allow one to understand the amount of work accomplished by each thread. For example, in regions where a single thread is active, the thread will no longer interfere with memory requests from other threads, potentially leading to faster single-thread execution. But, a fully populated system with N threads would continue to interfere, potentially slowing overall progress. The amount of time the application executes becomes the combination of the amount of work executed in one quantum, together with the runtime attributed to that quantum. These quanta can then be clustered in order to identify similar work, and therefore identify similar runtime behavior. Although BBVs are used in this work, other feature vector information [12] can be concatenated on a per-thread basis and can be used in this methodology.

The BBVs are projected down to 100 dimensions by random linear projection to bring down the computing requirements for the clustering algorithm. We use K-means clustering technique [24] along with a BIC goodness criteria [25] to select clustering in a method similar to previous work [1]. The K-means algorithm requires the selection of the maximum number of clusters that we can expect, $maxK$, for which we use $maxK = 50$.

Because we use BBV data that represents both parallelism and work executed, we can now cluster the regions, and use the resulting clusters for workload extrapolation. We choose the BBV that is closest to the centroid of each cluster to be the representative of the cluster. We generate the region that represents each cluster from the original application based on the region boundaries and call them *looppoints*.

F. Warmup

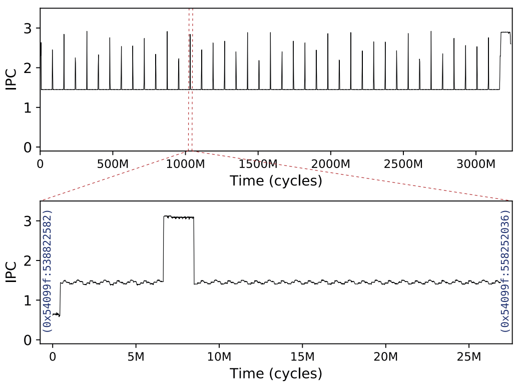
For high-performance, we will want to simulate each looppoint separately, in parallel, given enough resources. For accurate results, the microarchitectural state needs to be warmed up at the start of simulation of each region. There are several techniques [26]–[28] proposed to warmup cache state. For binary-driven simulation, we warm up each region from the start of the application to minimize warmup error.



638.imagick_s/magick/morphology.c

```
2842 #if defined(MAGICKCORE_OPENMP_SUPPORT)
2843   #pragma omp parallel for schedule(static,4) shared(progress,status) \
2844     magick_threads(image,result_image,image->rows,1)
2845 #endif
2846   for (y=0; y < (ssize_t) image->rows; y++)
2847   {
             ......
2886     for (x=0; x < (ssize_t) image->columns; x++)
2887     {
3021       for (v=0; v < (ssize_t) kernel->height; v++) {
3022         for (u=0; u < (ssize_t) kernel->width; u++, k--) {
             ......
3034         } /* u */
             ......
3037       }  /* v */
3342     } /* x */
3357   } /* y */
            ......
```

(a)



(b)

Fig. 4: An example of a representative region identified by LoopPoint. (4a) The numbers represent iterations of the corresponding loops that form the 8-threaded region. The start point and end point of the chosen region is at line 3022, the entry point of loop u. (4b) The top graph shows the variation of IPC over time for the full application run, while the bottom graph shows that of the chosen region. The (PC, count) boundaries are marked inside the IPC graph of the region.

Likewise, for checkpoint-driven constrained simulation, we use a sufficiently large warmup region preceding the simulation region. Determining the appropriate amount of warmup required for each representative region falls outside the scope of this work.

G. *Runtime Extrapolation*

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

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

$$total\ runtime = \sum_{i=rep_1}^{rep_N} runtime_i \times multiplier_i \qquad (1)$$

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

$$multiplier_j = \frac{\sum_{i=0}^{m} inscount_i}{inscount_j} \qquad (2)$$

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

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

H. *Reproducible Application Execution for Accurate Analysis*

The execution path of a multi-threaded application can vary from run to run due to several factors. One requirement to use this methodology is the ability to analyze a multi-threaded application in a repeatable way. Traditional execution environments do not support this type of execution to allow for reliable, reproducible execution. We leverage Intel's Pin [29] and Pinplay [16] tools to generate reproducible, constrained, multi-threaded execution snapshots, called pinballs, to allow for repeatable analysis. Pinballs are more advanced than a trace file in that they contain a snapshot of the execution state of an application (registers and memory). By replaying the Pinball, we can analyze the properties of an application to collect the microarchitecture-independent execution signatures of the application.

I. *Putting it All Together*

Together, the combination of reproducible replay of applications, along with the identification and clustering of workload characteristics, allows us to build an end-to-end methodology to identify workload representatives for performance extrapolation. Previous works [12] have shown that extrapolation in this manner does apply to runtime, as well as other metrics of interest. The insights with respect to the identification of application parallelism, as well as the constrained, reproducible execution of the workloads allow us to analyze, cluster and extrapolate multi-threaded workloads across a number of synchronization types.

J. Speed-up Potential

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

K. Workload Applicability

The methodology that we present here targets statically scheduled generic multi-threaded workloads regardless of the synchronization mechanisms used in order to simulate them in a faster way that was not possible before. Dynamically scheduled multi-threaded applications could require a different methodology for sampling as such workloads might not be able to be analysed or sampled based on an upfront analysis of the application. This is because such applications can interact with other threads in ways that were not seen in the initial execution of the application, potentially leading to an incorrect simulation run-time extrapolation. All checkpoint-based methodologies (including BarrierPoint) require upfront application analysis. We address the problem of workload imbalance among the threads (a heterogeneous workload) by keeping per-thread information intact while clustering the individual regions. Like other checkpoint-based methodologies, we also assume that the hardware configuration is known up-front. This configuration is free from any run-time-dependent configuration changes or unexpected events that trigger a configuration change while the application is running. An example of a dynamic event is thermal throttling resulting in a dynamic voltage and frequency scaling (DVFS) event, which can affect the application performance and is runtime- and hardware-dependent.

IV. Experimental Setup

In this section, we describe the setup on which we conducted our experiments to evaluate our generic multi-threaded sampling methodology.

A. Simulation Infrastructure

In this work, we use Sniper multicore simulation infrastructure [6] (version 7.4) with modifications to support PC-based simulation region specification. We configured Sniper to model a multicore out-of-order processor resembling the Intel Gainestown microarchitecture using an 8 or 16-core processor model to simulate 8 or 16-threaded (respectively) applications. The simulated system characteristics that we use are detailed in Table I.

B. Workloads

In order to evaluate the proposed methodology, we consider the SPEC CPU2017 [31] benchmark suite. SPEC CPU2017 is available in two different versions depending on the evaluation purpose: `rate` and `speed` [32]. The `rate` version is used to estimate the throughput of the underlying system whereas the `speed` version is used to estimate the runtime of the benchmark on the system. Unlike prior versions of SPEC benchmarks, CPU2017 includes a set of synchronizing multi-threaded programs that share memory consisting of OpenMP-compatible multi-threaded applications. We use the `speed` version of SPEC CPU2017 with `train` inputs and eight threads (See Table II for application descriptions) for our evaluation. The `train` input set is used so as to keep the full program simulation time to a reasonable length. As the detailed simulation of the full SPEC CPU2017 applications with `ref` inputs is not practical, computing the sampling error is also not feasible. Therefore, we utilize the `ref` inputs to estimate the potential speedup of the methodology in the paper. The benchmarks we use include OpenMP directives, with a summary of the primitives used described in (Table III).

TABLE I: The primary characteristics of the simulated system.

Component	Features
Processor	8 & 16 cores, Gainestown-like microarch.
Core	2.66 GHz, 128 entry ROB
Branch predictor	Pentium M
L1-I cache	32K, 4-way, LRU
L1-D cache	32K, 8-way, LRU
L2 cache	256K, 8-way, LRU
L3 cache	8M, 16-way, LRU

TABLE II: SPEC CPU2017 speed application attributes. F=Fortran, KLOC=thousand lines of code. From [30]

Application	Lang.	KLOC	Application Area
603.bwaves	F	1	Explosion modeling
607.cactuBSSN	F, C++	257	Physics: relativity
619.lbm	C	1	Fluid dynamics
621.wrf	F, C	991	Weather forecasting
627.cam4	F, C	407	Atmosphere modeling
628.pop2	F, C	338	Wide-scale ocean modeling
638.imagick	C	259	Image manipulation
644.nab	C	24	Molecular dynamics
649.fotonik3d	F	14	Comp. Electromagnetics
654.roms	F	210	Regional ocean modeling

TABLE III: SPEC CPU2017 speed synchronization primitives used. sta4=static for, dyn4=dynamic for, bar=barrier, ma=master, si=single, red=reduction, at=atomic, lck=lock.

Application	sta4	dyn4	bar	ma	si	red	at	lck
603.bwaves	Y					Y	Y	
607.cactuBSSN	Y	Y	Y			Y	Y	
619.lbm	Y							
621.wrf		Y		Y				
627.cam4	Y	Y	Y	Y				
628.pop2	Y		Y	Y				
638.imagick	Y		Y	Y	Y			Y
644.nab		Y	Y			Y	Y	
649.fotonik3d	Y							
654.roms	Y							

All SPEC CPU2017 workloads except `657.xz_s` runs are 8-threaded. `657.xz_s.2` runs with 4-threads whereas `657.xz_s.1` runs as a single-threaded application.

All the benchmarks in the SPEC CPU2017 benchmark suite are compiled using the Intel compiler toolchain (Intel Parallel Studio XE, version 2019 Update 2) with optimizations enabled (-O2) and debug information available for binary to source-level mapping, and built for the 64-bit x86 instruction-set architecture.

We also use NAS Parallel Benchmarks (NPB) [33], [34] version 3.3 with OpenMP based parallelization [35] that use class `C` inputs. We evaluate all benchmarks in the suite with both 8 and 16 threads, but do not evaluate the npb-dc (data cube) benchmark because of the large amount of data generated by that application. These benchmarks are compiled using GCC 5.5 for applications in C and GFortran for Fortran applications with -O3 optimizations for the x86-64 architecture.

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

For each benchmark, we record the execution path of the whole application and keep it as a pinball so that it can be replayed in both constrained and unconstrained mode later on. We have developed Pintools [29] to generate BBVs of the regions which are fed to Simpoint for clustering the regions to identify the representative regions. We also have employed Pintools to restrict the forward progress of all the threads in a well balanced way thereby avoiding the chances of recording a skewed trace because of CPU load imbalances. The representative regions identified are simulated in parallel. We evaluate the runtime accuracy of the chosen representatives by simulating in constrained and unconstrained modes.

C. Constrained Execution Infrastructure

We use Intel's PinPlay [16] infrastructure that provides tools to record and replay arbitrary regions of a program execution. The recorder captures the execution of an application in a set of files collectively called a pinball [20] which can later be replayed on any machine since pinballs are portable. A pinball consists of a memory file (`.text`), the architecture register values at the beginning of the execution region in per-thread register files (`.reg`), a set of memory and register values in per-thread injection files (`.sel`), and a subset of shared-memory dependencies among various threads in per-thread dependency files (`.race`). A pinball once captured is self-contained, which means that both the application binary and inputs are not needed during replay of the pinball.

The replayer loads the initial memory and register state and starts executing the restored program region like a regularly loaded binary. System calls are skipped and their side-effects are injected. Shared-memory access in all threads are monitored and the threads are artificially delayed as needed to enforce the access order as recorded in the pinball. Finally, the replay is ended gracefully when the exit condition is met. Since system calls are skipped during replay, a pinball can be replayed across different operating systems.

D. DCFG and Basic Blocks

The Dynamic Control Flow Graph (DCFG) is created by executing the program via a pin-tool enabled with the DCFG library [36], [37]. Internally, the pin-tool *hooks* the control-flow instructions and records a count of each of the resulting edges throughout the execution of the workload on a per-thread basis. At the end of the execution, *fall-through* edges are created to ensure non-overlapping basic blocks. These basic blocks are guaranteed to have only one entry and one exit point and not overlap with each other. In this way, they differ from the basic block structures in Pin, which do not have these guarantees. The resulting basic blocks and the edges that connect them thus create a connected graph. From this graph, routine boundaries are identified based on call edges and heuristics to handle non-standard routines that are sometimes found in non-compiled code. Inside the sub-graph of each routine, the immediate dominators of each node are found. Loops are then identified using the immediate dominator relationships. The graph, including the identified routines and loops are recorded.

E. Unconstrained Replay

PinPlay's replayer enforces determinism among the threads by injecting recorded system call side-effects and enforcing the recorded shared memory access thread order. We use this mode when analyzing the workload (collecting BBVs and DCFGs to be used in the clustering phase), to ensure different steps of the profiling methodology have a consistent view of the program's execution flow (as recorded during the initial whole-program recording). However, during performance simulation, we want the timing model to control thread progress and synchronization, not PinPlay as this can introduce artificial thread stalls[1].

F. Synchronization Handling

OpenMP active runs, enabled by setting the environment variable OMP_WAIT_POLICY to ACTIVE [39], have threads busy-waiting at user-level (as opposed to using futex() in the passive runs). We replay a pinball that was recorded earlier for reproducible analysis for the generation of BBVs. If we directly use the recording we encounter the busy-waiting code that was originally executed by the application. However, the busy-waiting code can differ if the application is executed another time with different conditions. While busy-waiting consumes processor cycles, they do not contribute to the *real* work done by the program. Therefore, we ignore busy-waiting during BBV profiling, yet include it during simulation. Identifying busy-waiting code automatically [40]

[1]See [38] for a methodology that uses constrained replay during multi-threaded performance simulation, and which can, in limited cases, work around the artificial stalls.

can be a challenge and is yet another research problem. In our methodology, we ignore the entire code from the relevant synchronization library (`libiomp5.so` in our case). Note that this idea can easily be extended to other compilers and threading libraries. For example, in the case of applications using pthread synchronization, we can ignore the code from the `libpthread` library. The filtered instruction count is up to 40% (for `657.xz_s.2`) fewer than the original instruction count for the active runs.

V. Evaluation

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

A. Accuracy

We show the accuracy of LoopPoint methodology by comparing the predicted runtime and the actual runtime of the application. The predicted runtime is calculated by considering the performance of all the representative regions as mentioned in Section III-G. The representative regions are augmented with a warmup region so that the microarchitectural state is warmed when the detailed region starts simulating. The prediction error of our methodology is the percentage difference in the simulation performance of the whole application and the extrapolated performance making use of the performance of all the representative regions identified for the application.

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

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

The region checkpoints are generated as pinballs which can be used for constrained simulation. We assume a large enough warmup region added to the representative region while generating the pinball checkpoint. However, using constrained simulation introduces artificial thread delays and are therefore not reliable for performance extrapolation. There are several ways to simulate these pinball checkpoints in an unconstrained way. One such method is to convert them to ELF binaries, called ELFies, as discussed in a prior work [21]. In this paper, however, we are not evaluating ELFies. Instead, we consider the region boundaries specified as (PC, count) to perform unconstrained simulation using the application binaries by providing perfect warmup before the start of detailed simulation. One caveat that we want to mention is that not all region boundaries specified using (PC, count) can provide stable regions. For instance, applications can have certain code blocks that are selectively executed with respect to the underlying microarchitecture. Such code blocks or PCs cannot serve as stable (PC, count) region boundaries. We assume that the users can choose the appropriate stable regions, and that, while straight-forward to accomplish in an automated way, we leave that analysis to future work.

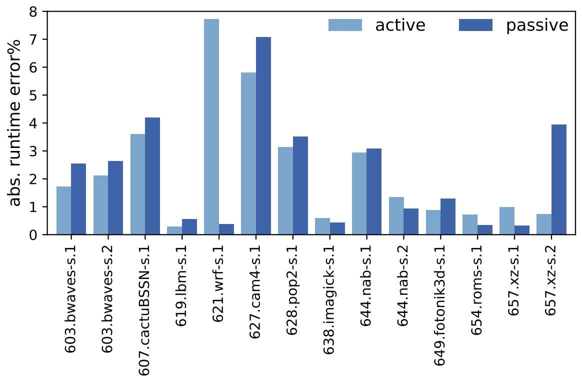
Results when simulating constrained simulation can be misleading and can lead to high errors. For example, we observe a runtime error for `657.xz_s.2` of up to 19.6% while simulating in a constrained environment. One of the reasons that using constrained simulation infrastructure can result in high error rates is that the simulation itself does not properly mimic the real application run. Instead, the application tries to replicate the behavior that was recorded previously on a specific machine. For instance, constrained execution forces spin-loops to be replayed even though this would not occur in a real execution. This introduces high error for applications, like `657.xz_s.2`, that have fewer synchronization points compared to other applications in the SPEC CPU2017 benchmark suite, and therefore can see high variability from run to run.

The runtime prediction results (Figure 5a) using the unconstrained simulation of active applications yield an average absolute error of just 2.33%, whereas that of passive applications is 2.23%. These error rates are comparable to previous sampling methodologies [12].

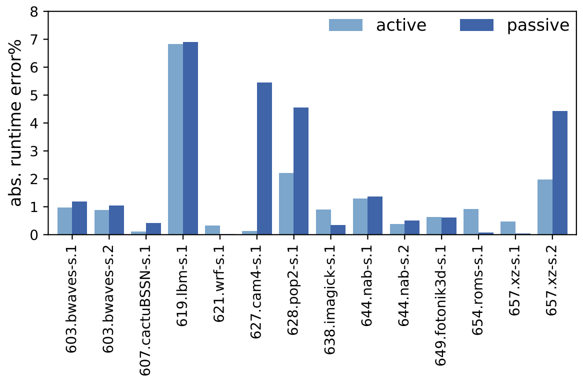
The looppoints identified are representative of the application across microarchitectural configurations. Our up-front analysis is solely based on architecture-level details, not microarchitectural settings or simulation details. Figure 5b shows the error in predicting the runtime of the same applications while simulated for an inorder core instead of the out-of-order Gainestown-like core, while keeping all other simulation parameters the default as in Table I. The graph clearly shows that looppoints can be portable across microarchitectures.

2) Varying the number of threads: We show that LoopPoint supports varying the number of application threads. Figure 6 shows the error rates while predicting the runtime of the NPB benchmarks. The applications are evaluated using 8 threads and 16 threads. Note that the applications using a different number of threads need to be profiled separately, as discussed in Section III. We observe that the average absolute error obtained is 2.87% for 8-threaded applications while for the 16-threaded applications it is as low as 1.78%.

3) Comparison of other metrics: Figure 7 shows the performance prediction of several metrics while simulated on an unconstrained environment for applications using `active` and `passive` wait policies. LoopPoint can determine microarchitectural metrics like the number of cycles (Figure 7a), branch miss rate or MPKI (Figure 7b), the miss rates or MPKI of different components in the memory hierarchy (Figure 7c), etc. In Figure 7b and Figure 7c, we show the absolute differences



(a) Gainestown core



(b) Inorder core

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

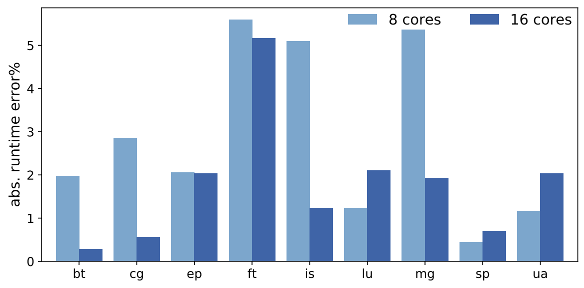


Fig. 6: The runtime prediction results of the NPB benchmarks that use 8 and 16 threads. The applications use a passive wait policy and class C inputs. The y-axis represents the error percentage in predicting the runtime of each of the applications on the x-axis.

in the metrics predicted, rather than the percentage error in prediction, because those metrics have small absolute values and a small difference can result in a high percentage error. Previous research [12], [41] has presented differences in a similar manner.

B. Speedup

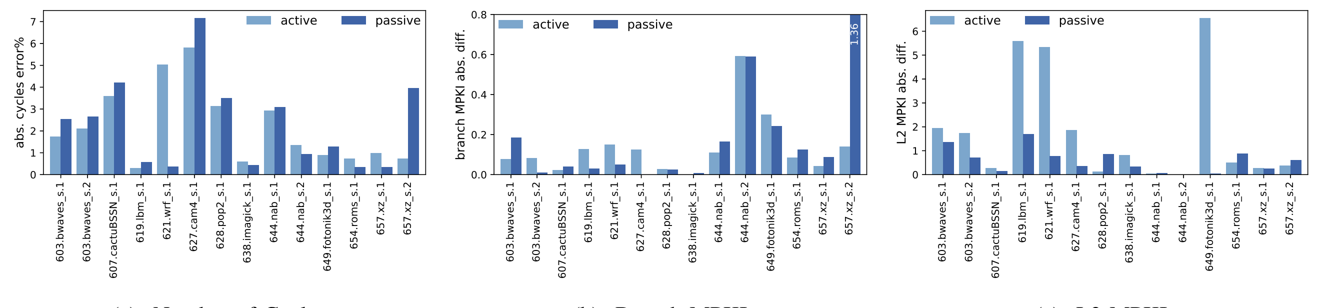
We consider speedup in two different ways: theoretical speedup and actual speedup. Theoretical speedup is the reduction in the number of instructions (ignoring the instructions that contribute to spinloops) to be simulated in detail when using LoopPoint methodology. We also define the actual speedup as the reduction in the simulated runtime using LoopPoint with respect to the simulated runtime of the whole application.

Serial speedup is the speedup achieved when all the representatives are simulated back-to-back. It is the overall reduction in work given the serial execution of both the full, and reduced workload. Parallel speedup assumes sufficient parallel resources, and evaluates the speedup given the execution of all regions in parallel.

In Figures 8 and 9, we see both the serial and parallel speedups for these applications. We obtain a maximum speedup of 801× for the applications with `train` inputs and 31,253× for the applications with `ref` inputs. The average serial speedup for applications using `train` inputs and `ref` inputs are respectively 9× and 244× whereas the average parallel speedup for the applications are 303× and 11,587× respectively for `train` and `ref` inputs. This implies that a significant reduction of simulation resources is now possible using the LoopPoint methodology, where simulations that would take months to complete can now be finished in hours.

In Figure 9, we compare the theoretical simulation speedup using LoopPoint and BarrierPoint for the benchmarks using `ref` inputs. Note that we do not plot the actual speedup values using the `ref` inputs. We first validate our methodology with `train` inputs, and by extension, we analyze and simulate `ref` input representatives to estimate the performance of the larger application with confidence. Unfortunately, it is not possible to validate the error rates for applications with `ref` inputs because the full runs take too long to simulate (a few months to years as shown in Figure 1).

We observe that LoopPoint consistently achieves good speedup whereas BarrerPoint lags behind for a number of applications. LoopPoint is able to reduce the application into representative regions that can finish simulation in a reasonable time. Additionally, with the BarrierPoint methodology, there is no guarantee on the size of a representative region. For example, the 8-threaded `638.imagick_s.1` benchmark has a very large inter-barrier region (93.06 B instructions) that is comparable to the size of the entire application (93.35 B instructions), defeating the purpose of sampling. However, there are a few applications for which BarrierPoint outperforms LoopPoint. Those applications have a large number of barriers and the inter-barrier regions are typically smaller than the LoopPoint



(a) Number of Cycles (b) Branch MPKI (c) L2 MPKI

Fig. 7: The prediction errors of various metrics for SPEC CPU2017 benchmarks using LoopPoint. The benchmarks use active and passive wait policies with train inputs and 8 threads, and are simulated in realistic unconstrained mode.

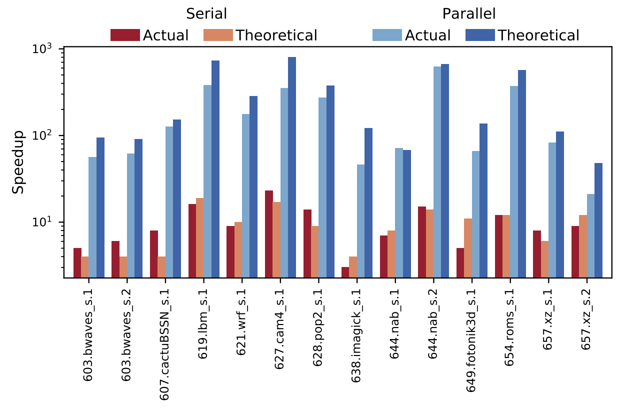


Fig. 8: A comparison of theoretical and actual speedups achieved by LoopPoint. The workload used is SPEC CPU2017 applications (active wait policy) using train inputs.

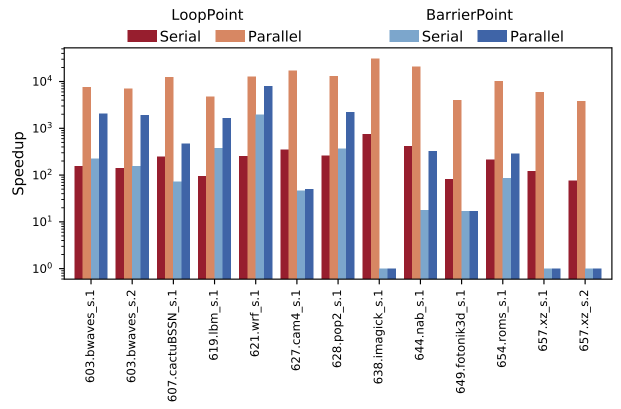


Fig. 9: LoopPoint and BarrierPoint theoretical speedup for SPEC CPU2017 applications (passive wait policy) using ref inputs.

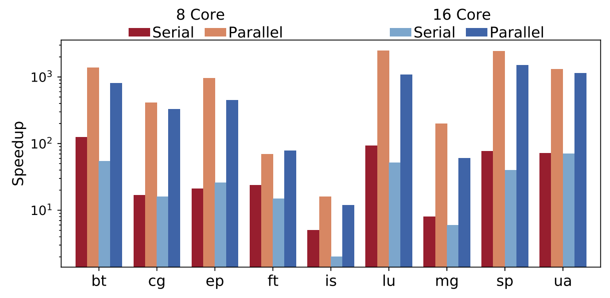


Fig. 10: A comparison of actual speedups achieved by LoopPoint when the applications use 8 and 16 cores. Speedups listed for the NPB suite using the C input set and a passive wait policy.

regions. BarrierPoint is unsuitable to evaluate both of the `657.xz_s` applications as they do not contain barriers at all. Overall, a hybrid approach can be chosen to speed up smaller applications, but LoopPoint provides the first methodology to allow for generic sampling of applications that results both in a high simulation speedup and accuracy.

We also show the speedup achieved using NPB applications in Figure 10. LoopPoint achieves good speedups while the applications are evaluated for 8 threads as well as 16 threads. The maximum parallel speedup achieved while evaluating the 8-threaded applications is 2,503× with an average of 1,031×, whereas for the 16-threaded applications, the maximum speedup achieved is 1,498× and an average of 606×. Do note that NPB applications are less complex and more repetitive in nature than SPEC CPU2017 applications. Therefore, the error rates are lower and the speedups achieved are larger when compared to the `train` inputs of the SPEC CPU2017 suite.

VI. Related Work

Before architects build new hardware designs, it is extremely useful to predict the hardware design's power, performance and area (cost). Existing circuit-design tools are able to simulate complex, modern applications on large, multi-core systems, but at a cost of significant simulation time that can be intractable (requiring months to years of simulation time for the SPEC CPU2017 benchmarks).

While there have been many attempts to solve this problem, previous works were unable to provide a combination of three things for multi-threaded workloads: (1) choosing accurate representatives without detailed simulation, (2) demonstrating simulation speedup based on application representatives, not on overall application runtime and (3) allowing the simulation of hardware designs that might not yet have analytical models. Our proposal addresses all these concerns through the determination of application parallelism, clustering and the extrapolating the results based on this information.

Single-threaded Sampling Methodologies. One of the first works to utilize the structure within the code to create a representative sample application was the Simpoint methodology [1]. The authors show that applications can be broken down into N smaller chunks of size 100M instructions, where N is the number of clusters to which the whole program can be clustered. This helps to determine where to simulate for a large single-threaded application. SMARTS [2] is another methodology which uses alternating fast-forward and detailed simulation phases for a large number of samples. They used a large number of intervals with regions having very small numbers of instructions. Large structures like caches are warmed in the fast-forward mode. The methodology could estimate the average IPC with high confidence. LiveSim [42] is another such methodology that enables interactive simulation making use of in-memory checkpoints. A prior work on software phase markers [19] uses loops to determine simulation regions, but is limited in that they only provide support for single-threaded applications using phase markers denoting phase changes.

Multi-threaded Sampling Methodologies. Ekman et al. [43] propose a methodology to reduce the number of simulation points using a matched-pair comparison method to estimate the full application performance. SimFlex [9] extends SMARTS methodology to support multiprocessor applications with an increased sample length. SMARTS and SimFlex use random sampling and therefore the samples are not necessarily representative. Perelman et al. in [44] extend the Simpoint methodology to use for phase analysis of multi-threaded workloads.

The instruction-count-based sampling mechanisms for single-threaded applications cannot be used directly for synchronizing multi-threaded applications. To sample multi-threaded applications, [10] and [11] were proposed which use time as a sampling unit by fast-forwarding between the detailed simulation intervals. The execution time during the fast-forwarding phase is extrapolated. But these techniques need to functionally simulate the entire application, which limits the speedup of simulation. Another methodology which reduces the simulation time of MT applications significantly is [12], a microarchitecture-independent, Simpoint-like approach which operates on OpenMP barrier synchronized applications by identifying inter-barrier regions as the unit of work. This technique takes into account the fact that all the threads are synchronized after a barrier, but is less effective when inter-barrier regions are very large or non-existent.

TaskPoint [13] is another work which proposes a sampling methodology which is applicable to a subset of multi-threaded applications. The technique is applicable to task-based programs and considers task instances as sampling units. The intervals between detailed simulation phases are fast-forwarded.

Analytical Modeling. There has recently been some progress on the development of a completely analytical model for single-threaded [45] [46] and multi-threaded [47] workloads. One major drawback of analytical models is the inability to estimate the performance of next-generation hardware designs. New processor, cache, and memory techniques without analytical models will not be able to use these methodologies. The general evaluation of future hardware designs can therefore require the use of execution-driven analysis.

Constrained simulation. Multi-threaded checkpoints were used [38] for constrained simulation. Their goal was to estimate the relative performance analysis of regions-of-interest across multiple micro-architectures. They describe a mechanism for speedup computation in the presence of artificial stalls added by the constrained replay of checkpoints during simulation. There could be cases where the speedup computation is inconclusive. We support unconstrained simulation as well as constrained simulation and also provide an absolute performance extrapolation methodology. For relative, cross-micro-architectural performance analysis, unconstrained simulation is desirable as it need not have to deal with artificial stalls.

Handling busy-waiting. The problem of busy-waiting is mentioned in [16] although in the context of multi-process programs using Message Passing Interface (MPI). The work focuses on simulating a specific single-threaded process from multiple processes in an MPI program and uses the selective logging feature of PinPlay to exclude the busy waiting code from consideration, both in the profiling and simulation phases.

Statistical workload generation. There are different works that study the time-varying runtime behavior of standard benchmarks. Wu et al. [48] study the phase behavior of SPEC CPU2017 workloads. Moreover, the work identifies the single-threaded simulation points using SimPoint methodology and correlates them with the phase behavior. Nair et al. [49] study the phase behavior of SPEC CPU2006 and SPEC CPU2000 using SimPoint methodology. The work identifies similar CPI prediction results using SimPoint for the applications in both suites and concludes that these applications have similar phase behavior.

VII. Conclusion

The need to understand larger, more complex multi-core processors continues to increase. This becomes even more critical as the multi-core processors (and the serial code) tend to be the bottleneck in highly parallel applications. General-purpose applications are found on embedded devices, mobile phones, and back-end data center servers. While each platform has its requirements and demands, the need for an accurate understanding of the applications at hand is clear.

Simulation solutions alone are insufficient because of the significant slowdown (10,000$\times$ or more [50]) seen when simulating applications with industrial-quality simulators. Simulation solutions today require alternatives like sampling to reduce the workloads to realistic simulation times. But, current sampling solutions either target single-threaded workloads or are only applicable to specific workload types.

In this work, we present a generic multi-threaded sampling methodology, one that considers the inherent parallelism of the application and allows for automatic reduction of workloads to sizes that are on the order of the representatives of the workloads themselves. We demonstrate how our classification methodology automatically partitions the workload into representatives and allows us to predict the performance of the workloads at hand with high accuracy.

Acknowledgments

This work has benefited tremendously from the support, contribution, and suggestions of many individuals. We sincerely thank all of them. In particular, we acknowledge Chuck Yount, Carl Beckmann, Alexey Klimkin, Igor Ermolaev, and Ady Tal at Intel. The work was partially supported by a grant from Intel. We also thank the anonymous reviewers and our colleagues at NUS for their valuable comments.

References

[1] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 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 *International Symposium on Computer Architecture (ISCA)*, Jun. 2003, pp. 84–97.

[3] A. J. KleinOsowski and D. J. Lilja, "Minnespec: A new SPEC benchmark workload for simulation-based computer architecture research," *Computer Architecture Letters (CAL)*, vol. 1, no. 1, pp. 7–7, 2002.

[4] R. H. Bell and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *International Conference on Supercomputing (SC)*, Jun. 2005, pp. 111–120.

[5] 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)*, Jun. 2018, pp. 29–42.

[6] 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)*, Nov. 2011, pp. 52:1–52:12.

[7] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2013, pp. 475–486.

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

[9] 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, 2006.

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

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

[12] 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)*, Mar. 2014, pp. 2–12.

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

[14] T. Grass, T. E. Carlson, A. Rico, G. Ceballos, E. Ayguadé, M. Casas, and M. Moreto, "Sampled simulation of task-based programs," *Transactions on Computers (TC)*, vol. 68, no. 2, pp. 255–269, 2019.

[15] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood, "Evaluating non-deterministic multi-threaded commercial workloads," in *Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Feb. 2002.

[16] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs," in *International Symposium on Code Generation and Optimization (CGO)*, Apr. 2010, pp. 2–11.

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

[18] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[19] J. Lau, E. Perelman, and B. Calder, "Selecting software phase markers with code structure analysis," in *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2006, pp. 135–146.

[20] H. Patil and T. E. Carlson, "Pinballs: portable and shareable user-level checkpoints for reproducible analysis and simulation," in *Workshop on Reproducible Research Methodologies (REPRODUCE)*, Feb. 2014.

[21] H. Patil, A. Isaev, W. Heirman, A. Sabu, A. Hajiabadi, and T. E. Carlson, "ELFies: Executable region checkpoints for performance analysis and simulation," in *International Symposium on Code Generation and Optimization (CGO)*, Feb./Mar. 2021, pp. 126–136.

[22] G. Hamerly, E. Perelman, and B. Calder, "How to use SimPoint to pick simulation points," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 25–30, Mar. 2004.

[23] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, "Motivation for variable length intervals and hierarchical phase behavior," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2005, pp. 135–146.

[24] E. W. Forgy, "Cluster analysis of multivariate data: efficiency versus interpretability of classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.

[25] G. Schwarz, "Estimating the dimension of a model," *The Annals of Statistics*, pp. 461–464, 1978.

[26] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic, "Accelerating multiprocessor simulation with a memory timestamp record," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2005, pp. 66–77.

[27] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe, "Simulation sampling with live-points," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2006, pp. 2–12.

[28] N. Nikoleris, L. Eeckhout, E. Hagersten, and T. E. Carlson, "Directed statistical warming through time traveling," in *International Symposium on Microarchitecture (MICRO)*, Oct. 2019, pp. 1037–1049.

[29] 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 *Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[30] "SPEC CPU®2017 documentation index," http://spec.org/cpu2017/Docs/index.html.

[31] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *International Conference on Performance Engineering (ICPE)*, Apr. 2018, pp. 41–42.

[32] A. Limaye and T. Adegbija, "A workload characterization of the SPEC CPU2017 benchmark suite," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2018, pp. 190–200.

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

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

[35] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," NAS-99-011, NASA Ames Research Center, Tech. Rep., Oct. 1999.

[36] "DCFG generation with PinPlay," https://software.intel.com/content/www/us/en/develop/articles/pintool-dcfg.html.

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

[38] 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)*, Sep. 2008, pp. 173–182.

[39] “OpenMP 3.1 API C/C++ Syntax Quick Reference Card,” https://www.openmp.org/wp-content/uploads/OpenMP3.1-CCard.pdf.

[40] T. Li, A. R. Lebeck, and D. J. Sorin, “Spin detection hardware for improved management of multithreaded systems,” *Transactions on Parallel and Distributed Systems (TPDS)*, vol. 17, no. 6, pp. 508–521, 2006.

[41] 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)*, Dec. 2004, pp. 81–92.

[42] S. Hassani, G. Southern, and J. Renau, “LiveSim: Going live with microarchitecture simulation,” in *International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 606–617.

[43] M. Ekman and P. Stenstrom, “Enhancing multiprocessor architecture simulation speed using matched-pair comparison,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2005, pp. 89–99.

[44] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, “Detecting phases in parallel applications on shared memory architectures,” in *International Parallel Distributed Processing Symposium (IPDPS)*, Apr. 2006.

[45] S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout, “Analytical processor performance and power modeling using micro-architecture independent characteristics,” *Transactions on Computers (TC)*, vol. 65, no. 12, pp. 3537–3551, 2016.

[46] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout, “Micro-architecture independent analytical processor performance and power modeling,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2015, pp. 32–41.

[47] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout, “RPPM: Rapid performance prediction of multithreaded workloads on multicore processors,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2019, pp. 257–267.

[48] Q. Wu, S. Flolid, S. Song, J. Deng, and L. K. John, “Invited paper for the hot workloads special session hot regions in SPEC CPU2017,” in *International Symposium on Workload Characterization (IISWC)*, Sep./Oct. 2018, pp. 71–77.

[49] A. A. Nair and L. K. John, “Simulation points for SPEC CPU 2006,” in *International Conference on Computer Design (ICCD)*, Oct. 2008, pp. 397–403.

[50] 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 Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[51] A. Sabu, H. Patil, W. Heirman, and T. E. Carlson, “LoopPoint artifacts,” https://doi.org/10.5281/zenodo.5667620.

[52] “LoopPoint source code,” https://github.com/nus-comparch/looppoint.

[53] “Sniper simulator,” https://snipersim.org.

A. Artifact Description Appendix

A. Abstract

In this artifact, we provide the required tools and information needed to replicate the primary experiments demonstrated in this paper. The artifact provides the necessary tools and scripts to run the three parts:

1) profiling the application to collect the necessary data needed for multi-threaded sampling;
2) sampled simulation of the selected regions; and
3) extrapolation of performance results, and plotting the key results

This appendix describes these parts and how to run them to replicate our experiments. We have also included a demo multi-threaded application to test the methodology end-to-end.

B. Artifact check-list (meta-information)

Parameter	**Value**
Program	C++ program and Python/shell scripts
Compilation	C++11 compiler
Benchmarks	Any multi-threaded benchmark suite
Run-time environment	Ubuntu 18.04, Docker
Hardware	x86-based Linux machine
Output	Plain text, tables
Experiments	Run profiling, run simulations, and run extrapolation scripts
Experiment customization	Benchmarks to verify the methodology, number of threads
Disk space requirement	≈50 GB
Workflow preparation time	≈1 day
Experiment completion time	≈1-4 weeks
Publicly available?	Zenodo [51] and GitHub [52]
Code licenses	Academic and Proprietary

C. Description

How to access: We use SPEC CPU2017 benchmark suite to evaluate the proposed methodology, *LoopPoint*. In this artifact, however, we do not include SPEC CPU2017 binaries and provide a demo application to test the end-to-end methodology. The setup can be used to replicate any results that we show in the paper. Pin kit and Sniper will be downloaded while setting up the artifact (see Installation section for instructions). The tool binaries are provided that works with Intel Pin. The artifact is available on Zenodo with DOI 10.5281/zenodo.5667620 [51] and on GitHub [52].

Hardware dependencies: The artifact is developed such that it runs on an x86-based Linux machine. We strongly recommend running the artifact using the provided Dockerfile. We expect the size of files generated in the profiling stage of LoopPoint to be a few GBs, hence we suggest a minimum free space of 50 GB when profiling SPEC CPU2017 with the train input set. Memory and space requirements are much lower for the included demo application.

Software dependencies:

- GNU Make
- C++11 build toolchain
- Python2, Python3, numpy, tabulate
- Docker

Data sets:

- matrix-omp: An OpenMP based demo application that can be used to test the end-to-end methodology in a reasonable amount of time.

D. Installation

1) Download the artifact from the Zenodo link and navigate to the artifact base directory.

2) Request access to the Sniper git repository [53] to download Sniper. Provide a valid email address to receive the link to the Sniper Simulator (we used version 7.4 in this artifact).
3) Follow the below steps to setup and build the artifact once you have received the path to the Sniper git repository.
   a) Build the docker image
      `$ make build`
   b) Run the docker image
      `$ make`
   c) Build the provided applications
      `$ make apps`
   d) Download and build the required tools once you have the Sniper gitid link
      ```
      $ make tools SNIPER_GIT_REPO="http://
      snipersim.org/<path-to-git-repo>.git"
      ```
4) These steps should automatically download the required versions of Pin kit and Sniper, and apply the required patches. All the other tools that we use are provided along with the artifact.

E. Experiment workflow

In this section, we describe the steps to generate the results shown in the paper. The end-to-end methodology of LoopPoint involves several steps. However, we have automated the process so that the user need not run every single step.

In the first stage, the selected benchmark is executed to record it as a pinball in `whole_program.<input>` directory. This pinball is then profiled by generating BBV data and making use of DCFG information generated by Pin. This is stored in `<basename>.Data` directory. The BBVs are clustered using K-means clustering and the representative region boundaries are identified.

In the following stage, the representative region information is used to launch region simulations, one after the other. Note that the region simulations can be launched in parallel as the runs are independent of each other. One can update these scripts to run several simulations in parallel or to run them on a collection of machines.

When all the region simulations are completed, the runtime of the full application is extrapolated using the runtimes of the representative regions and compared with that of the full application run. The estimated error and speedup numbers are displayed as the final output on the console.

All the profiling and simulation results are stored in the `results` directory. The end-to-end methodology is automated completely for ease of use.

The driver tool to process and evaluate the key results of sampling a multi-threaded application is the `run-looppoint.py` script which controls launching the profiling and simulation runs. The arguments are defined as follows:

- `-p` or `--program`: program to be executed, supplied in the format `<suite>-<application>-<input-num>` Multiple programs can be submitted as comma-separated values
  Default: `demo-matrix-1`
- `-n` or `--ncores`: number of threads
  Default: `8`
- `-i` or `--input-class`: input class
  Default: `test`
- `-w` or `--wait-policy`: OpenMP wait policy
  Options: `passive`, `active`
  Default: `passive`
- `--force`: start a new set of end-to-end run
- `--reuse-profile`: reuse the default profiling data (used along with `--force`)
- `--reuse-fullsim`: reuse the default full application simulation (used along with `--force`)
- `--native`: Run the application natively

Below are some example commands that can be used for the users to readily run the tool after installation.

Usage Examples:

1)
   ```
   ./run-looppoint.py -p demo-matrix-1 -n 8
   --force
   ```
   This example starts a new end-to-end run for the `demo-matrix-1` program using `test` inputs with `8` cores and using the `passive` wait policy.

2)
   ```
   ./run-looppoint.py -p
   demo-matrix-2,demo-matrix-3 -w active
   -i test --force
   ```
   This example starts a new end-to-end run for the `demo-matrix-2` program followed by the `demo-matrix-3` program using the `active` wait policy, `8` threads, and `test` inputs.

F. Evaluation and expected results

To replicate the results shown in this paper, it is necessary to run each of the applications in SPEC CPU2017 benchmark suite. One can integrate any multi-threaded application in a similar fashion (the demo application `matrix-omp`, can be used as an example). Note that launching an end-to-end evaluation can be long-running for large applications as the full application simulation can take a long time.