BarrierPoint: Sampled Simulation of Multi-Threaded Applications

Trevor E. Carlson[1] Wim Heirman[2] Kenzo Van Craeynest[1] Lieven Eeckhout[1]

[1]Ghent University, Belgium
[2]Intel, ExaScience Lab

***Abstract*—Sampling is a well-known technique to speed up architectural simulation of long-running workloads while maintaining accurate performance predictions. A number of sampling techniques have recently been developed that extend well-known single-threaded techniques to allow sampled simulation of multi-threaded applications. Unfortunately, prior work is limited to non-synchronizing applications (e.g., server throughput workloads); requires the functional simulation of the entire application using a detailed cache hierarchy which limits the overall simulation speedup potential; leads to different units of work across different processor architectures which complicates performance analysis; or, requires massive machine resources to achieve reasonable simulation speedups.**

In this work, we propose BarrierPoint, a sampling methodology to accelerate simulation by leveraging globally synchronizing barriers in multi-threaded applications. BarrierPoint collects microarchitecture-independent code and data signatures to determine the most representative inter-barrier regions, called *barrierpoints*. BarrierPoint estimates total application execution time (and other performance metrics of interest) through detailed simulation of these barrierpoints only, leading to substantial simulation speedups. Barrierpoints can be simulated in parallel, use fewer simulation resources, and define fixed units of work to be used in performance comparisons across processor architectures. Our evaluation of BarrierPoint using NPB and Parsec benchmarks reports average simulation speedups of 24.7× (and up to 866.6×) with an average simulation error of 0.9% and 2.9% at most. On average, BarrierPoint reduces the number of simulation machine resources needed by 78×.

I. Introduction

Simulation is the de facto experimental methodology in architecture research and development. Virtually all research or development projects involve some form of architectural simulation. Architectural simulation, unfortunately, is extremely time-consuming. Depending on the level of detail at which a target architecture is being simulated, simulation incurs slowdowns compared to real hardware execution by at least four orders of magnitude, and in case cycle-accurate simulation is employed, up to six orders of magnitude. This simulation speed gap is widening as the target architectures employ more and more cores, while relying on single-threaded simulators.

Sampling is a widely used technique to dramatically reduce simulation time by simulating a select number of sampling units in detail and extrapolating the results for the entire workload execution. Sampled simulation is a mature technology for single-threaded workloads running on individual cores, and different approaches have been proposed for determining representative sampling units: random [8], periodic [26], and through program analysis [20]. Sampled simulation for multi-threaded workloads on the other hand is much less mature and substantially more complicated.

The fundamental problem in sampled simulation for multi-threaded workloads is to make sure all threads are aligned (i.e., all threads are at the same point in their execution) at the beginning of each sampling unit as if we were to simulate the entire execution up until the sampling unit. This is non-trivial because of how slight timing variations during the execution may affect per-thread progress either through synchronization behavior (e.g., locking) or resource sharing (e.g., shared caches, off-chip bandwidth, interconnection network, etc.). Moreover, making sure the same thread alignment occurs across microarchitectures is even more problematic.

Some classes of multi-threaded workloads do not pose this fundamental problem. For example, commercial server throughput workloads can be accurately sampled by randomly selecting sampling units [25]. This principle more generally applies to multi-threaded workloads in which the individual threads do not synchronize [10]. However, synchronizing multi-threaded applications are more challenging to sample for the reason mentioned above. Recent work proposed time-based sampling for synchronizing multi-threaded workloads [2], [7], which simulates X units of time in detail every Y units of time, and estimates per-thread progress in-between sampling units. The fundamental limitation of time-based sampling is twofold. It requires functional simulation of the entire program execution to determine the sampling units, which limits the amount of speedup that can be achieved through sampling. In addition, it leads to different sampling units being selected across different simulated processor architectures, which complicates performance analysis.

Barrier-synchronized multi-threaded applications are an important subset of synchronizing parallel workloads, especially in the high-performance scientific computing and data-parallel workload domains, for which this fundamental problem in sampling can be overcome by selecting sampling units using barrier semantics. Barriers denote points of global synchronization in the applications at which all threads are naturally 'aligned', i.e., all threads re-start the execution at the same time once the barrier is reached by all threads. Bryan et al. [5] leverage this observation to speed up simulation of barrier-synchronized applications by simulating multiple inter-barrier regions in parallel on a cluster of simulation machines. This approach requires massive simulation resources to achieve substantial simulation latency reductions. Additionally,

because the number of simulations to be performed inevitably outstrips the supply of available machines, maximizing overall simulation throughput becomes the overall goal. The only way to make faster progress is to improve the total simulation throughput across the entire parameter and benchmark space. Bryan et al. does not solve this critical issue: it only reduces latency of an isolated simulation run, but it does not reduce the number of resources required nor overall simulation throughput when many simulations need to be run.

In this work, we propose BarrierPoint, a methodology for sampled simulation of barrier-synchronized multi-threaded machines that simulates a select number of representative inter-barrier regions, called *barrierpoints*, and predicts total application execution time (and other metrics of interest) from these barrierpoints. BarrierPoint computes code and memory access signatures for all inter-barrier regions, clusters regions based on these signatures, and then selects a single representative region, called a barrierpoint, per cluster. Barrierpoints are selected in a microarchitecture-independent way, and can therefore be used across processor architectures. BarrierPoint overcomes several major limitations in prior work. (i) It does not require functional simulation of the entire application as in time-based sampling; barrierpoints can be simulated in parallel. (ii) It leads to well-defined and fixed units of work — unlike time-based sampling — which facilitate comparisons across microarchitectures. (iii) It requires far less (one to three orders of magnitude fewer) simulation resources while achieving similar simulation speedups compared to the approach by Bryan et al. [5].

Specifically, we make the following contributions in this paper:

- We propose a methodology for selecting representative, microarchitecture-independent inter-barrier regions in barrier-synchronized multi-threaded applications for sampled simulation. Barrierpoints enable microarchitectures and processor architectures (including different core counts) to be compared through sampled simulation using well-defined and fixed units of work.
- We propose a method to extrapolate and estimate total application execution time, and other performance metrics of interest, from this select set of barrierpoints.
- We explore different methods for characterizing inter-barrier regions and find that signatures that incorporate both code and data behavior are most accurate at identifying representative barrierpoints.
- We evaluate the BarrierPoint methodology using a set of NPB and Parsec benchmarks on 8 and 32-core machines, and report average speedups of 24.7× (maximum speedup of 866.6×) while maintaining an average error of 0.6% and maximum error of 2.8%. BarrierPoint reduces the number of simulation machine resources needed by 78× on average, compared to simulating all inter-barrier regions.
- We propose an easy-to-implement, fast and accurate cache warmup technique for multi-threaded sampling that incurs a combined sampling and warmup error of 0.9% on average and 2.9% at most.

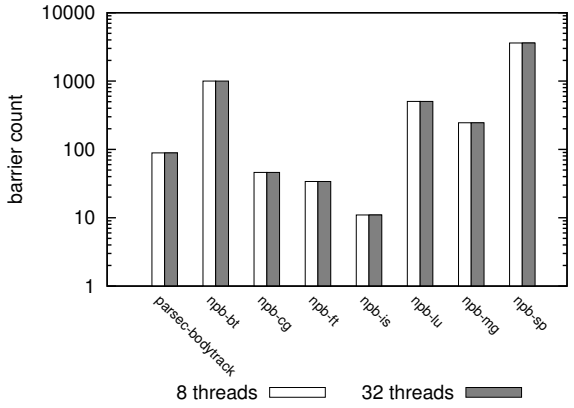

Fig. 1. Total number of dynamically executed barriers.

II. Motivation and Key Idea

Multi-threaded applications have been developed as a way to take advantage of the growing numbers of cores available in current machines. Prior work [2], [7] has shown that, as in single-threaded workloads, redundancy exists in the behavior of multi-threaded applications which allows detailed simulation of only part of the workload to be extrapolated to accurately predict execution time of the complete application. However, the only available techniques for accurate sampled simulation of synchronizing multi-threaded applications required functional simulation of the complete application, which limits the simulation speedup that can be obtained through sampling. In contrast, implementations of the popular SimPoint [20] and SMARTS [26] methodologies for sampled simulation of single-threaded applications are able to load the application's architected state at the start of each sampling unit from a checkpoint [22], [24]. This makes simulation of each sampling unit completely independent and potentially parallelizable, and negates the need for functional simulation of the complete application.

In synchronizing multi-threaded applications, however, it is not known a priori at what rate the execution of each individual thread will progress. Therefore, a checkpoint of architected state taken at a random time during execution of the application does not necessarily represent a valid situation that would occur when executing the same application on a different microarchitecture. Global synchronization barriers, on the other hand, represent points in each thread's instruction stream that denote a common point in time, and are therefore *safe* points at which a checkpoint can be taken. Figure 1 counts the number of barriers encountered during the execution of a number of applications in the NPB and Parsec benchmark suites. As is common for many data-parallel workloads, which are typically written in a structured manner using fork-join parallelism or other paradigms that lead to bulk-synchronous behavior, the absolute number of barriers is large, up to several 1000s in this case. Moreover, the number of barriers present remains constant even when changing the number of threads. This suggests that sections of code in between global barriers can be used as independent units of work.

By simulating an application separated into a number of

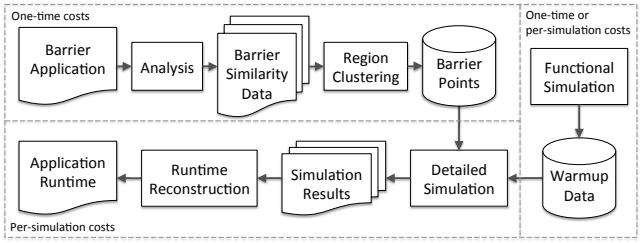

Fig. 2. The BarrierPoint methodology flow diagram.

inter-barrier regions in parallel, it is now possible to speed up simulation of an entire program. Assuming that a large enough amount of simulation resources are available, one can reduce the latency of executing a single application [5].

Taking this one step further, if we are able to classify — and therefore merge — similar inter-barrier regions into a single *representative* region, we could further reduce the number of computing resources required to obtain these speedups. We will call these representative regions *barrierpoints*, akin to *simpoints*, their single-threaded, instruction-delineated counterparts following the SimPoint [20] methodology. The final step that is required is to provide the warmup data for each barrierpoint. The end result is a complete simulation methodology that takes advantage of the ability to parallelize multi-threaded applications at a barrier-by-barrier granularity. With BarrierPoint, we have developed a methodology that allows for existing applications, without modification, to be simulated in parallel, using fewer simulation resources. Using global instruction count as a proxy for the amount of work in a simulation, we demonstrate a minimum, harmonic mean, and maximum speedup of 10.0×, 24.7× and 866.6×, respectively.

III. BarrierPoint Methodology

The BarrierPoint methodology (See Figure 2) provides a practical flow from a barrier-synchronized application to an estimation of performance metrics such as execution time or cache miss rates with a significant reduction in simulation time. The steps in the methodology include a one-time step where characteristics that represent inter-barrier regions of the application are collected, and clustering of the representative regions into barrierpoints for simulation occurs. These barrierpoints, together with the warmup data for each simulation is used to simulate each barrierpoint in parallel. The final step is to reconstruct the performance metrics based on the representative barrierpoint simulation results. In the following sections, we will provide a detailed overview of each step in the methodology.

A. Barrier Region Similarity Metrics

Through the use of barriers, we have the ability to compare the instructions executed inside of inter-barrier regions for similarity. There are a number of different ways to classify the similarity between regions.

1) Basic Block Vectors: Traditionally, Basic Block Vectors (BBVs) [20] from fixed-instruction-length regions have been used as a way to easily classify or cluster regions of single-threaded applications. A basic block vector is a vector with an entry for the dynamic instruction count for each basic block in an application. During program execution, the number of instructions executed from each basic block is counted. After the completion of this region, the basic block vector is saved and a new vector is started for the following region. These BBVs form its fingerprint, or summary of the instructions that have executed during that region. Prior work has shown that BBVs relate strongly to region performance [14]. Therefore, by comparing the BBVs across inter-barrier regions, we can match the different regions to determine performance similarities for each application region.

2) LRU stack distance: LRU stack distance histograms [16] are another metric that can be used to classify program behavior. The LRU stack distance is the number of unique address accesses that occur between two accesses to the same address. A histogram of these distance numbers represents a memory history footprint of a particular region of an application, and have been previously used to automatically detect phases in an application [19], as changes in the LRU stack distance profile is a way to evaluate the changing data reuse patterns of an application. LRU stack distances can therefore also be used for region classification. The intuition behind using LRU stack distances is that dynamic instruction regions, even though they are executing the same code and have the same BBV footprint, could experience different cache access latencies because of micro-architectural state. One example of this is the well-known cold-start effect, where the first few iterations of an application exhibit different progress than later, but BBV-similar, phases. Our goal, therefore, is to improve the accuracy of BBVs by combining them with LRU stack distance information when performing BarrierPoint clustering. To collect LRU stack distance information for the BarrierPoint methodology, we keep track of the reuse distances for each region of the application as it runs. The LRU stack distance data is stored in a power-of-two histogram, where we keep track of the frequency of each of the LRU stack distance accesses for each inter-barrier region. We will further refer to this histogram as the LRU stack distance vector (LDV).

3) Signature Vectors: To make an abstraction of the exact metric that is used, we define the Signature Vector (SV) as a representation of a region's phase behavior. Functionally, the Signature Vector consists of indices representing application characteristics, and the value of a vector element is the quantity or weight of that characteristic. In Section VI, we explore the clustering obtained by SVs consisting of BBV information only, LRU stack distance vectors (LDVs) only, and SVs that contain a combination of both. When both types of metrics are used, the BBV and LDV are normalized individually and then concatenated into a single, longer vector.

In addition, we evaluate the use of a weighing function to the LRU stack distance counters. Conceptually, longer stack distances will correspond to memory accesses that hit further away in the memory hierarchy and therefore have a larger impact on application performance. When comparing two

regions with the aim of clustering them by having similar performance, we therefore want to give more importance to long-latency accesses. We will show results for unweighted distances, and variants where those elements corresponding to a distance of $2^n \ldots 2^{(n+1)} - 1$ are weighted by $2^{n/v}$, for different values of v.

4) Multi-threaded SVs: Both BBVs and LDVs are initially collected for each inter-barrier region on a per-thread basis. To combine these per-thread metrics into a single SV, two options are possible. One option is to sum the vectors, aggregating data from different threads into the same vector element. A second option is to concatenate vectors for each thread into a longer vector. This second option separates the behavior for each of the threads.

When threads behave in a homogeneous manner, both options will be equivalent. But, if threads have different behavior, a concatenated vector will expose these differences to the clustering phase which allows the relevant regions to be separated into different clusters. We therefore choose to use concatenation for combining SVs of different threads.

B. Region Clustering

To automatically determine the similarity among regions, we employ clustering. The first step is to normalize the SVs to allow the clustering phase to ignore each region's length, and to cluster regions based on their intrinsic characteristics. To reduce the computational demands of the clustering process, the dimensionality of the SVs is then reduced through random linear projection into 15 dimensions. In addition to the normalized SV, the region clustering process will use the region lengths, in aggregate number of instructions across cores, to weigh different regions such that more emphasis can be placed on those regions that represent a large fraction of total execution time. We then perform weighted k-means clustering on the randomly projected SVs to determine a reduced set of representative regions. When within a cluster multiple regions of similar characteristics but different length occur, weighing by instruction count will favor longer-running regions both in determining the cluster center, and in choosing the representative region.

Signature vector normalization, random projection and k-means clustering are compatible with the implementation of SimPoint [13] for variable-length regions. We are therefore able to leverage the existing SimPoint infrastructure when implementing BarrierPoint.

C. Detailed Region Execution

After we have defined the representative regions of the application, detailed simulation of each barrierpoint will provide the necessary information required to rebuild the application's execution time. Before detailed simulation can be started, both architected and microarchitectural state must be properly initialized. As the region starts with a barrier, it is possible to store the program's architected state in a checkpoint in a consistent manner across threads. Alternatively, functional simulation or direct execution can be used to fast-forward an application up to the barrier that marks the start of the region.

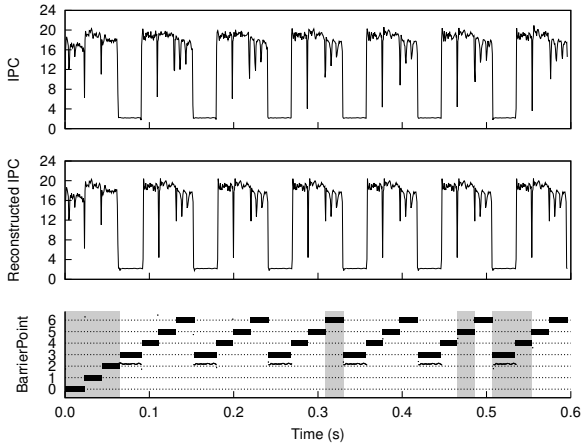

Fig. 3. Aggregate application IPC (above), reconstructed IPC (middle) and the selected barrierpoints (below) for `npb-ft` using the *class A* input set on 32-cores. (Non-significant barrierpoints are omitted for clarity.)

With respect to microarchitectural state, care must be taken to ensure caches are warmed sufficiently before the start of the detailed simulation. Several warmup options are possible and are compatible with the BarrierPoint methodology, see Section IV for an overview of existing techniques, and the method used in our evaluation of BarrierPoint. Since barrierpoints are usually long (in the order of millions of instructions), detailed core or branch predictor warmup is not normally required.

D. Whole-Program Runtime Reconstruction

As variable-length barrierpoints are used as the basis for the identification of common regions, we need to add an additional step to properly map these representatives to each region to compute overall application execution time. This step is not required for the original SimPoint methodology as they are using fixed-length representative regions to determine application CPI, which in turn, can be used as a proxy for single-threaded application execution time. In contrast, the BarrierPoint methodology may cluster inter-barrier regions of differing lengths. Yet, as the clustering phase guaranteed that these regions have the same behavior, we can assume that their performance characteristics (when expressed as per-instruction metrics, e.g., cycles per instruction (CPI), cache misses per 1000 instructions (MPKI), etc.) are constant. We can therefore rebuild the original application by concatenating scaled (by relative global instruction count) versions of each barrier's representative region. We call the sum of scaling factors from each barrierpoint its *multiplier*. Put another way, the sum of the instruction counts from all of the inter-barrier regions that are represented by the barrierpoint is equal to the instruction count of the barrierpoint times the multiplier. This can be written as $\sum_{i=1}^{m} insncount_i = insncount_j \cdot mult_j$, where m is the number of regions that are represented by the jth barrierpoint. To calculate a metric of interest, we sum, over each barrierpoint, the metric weighted by the multiplier, $metric_{app} = \sum_{j=1}^{n} metric_j \cdot mult_j$, where $metric_j$ and $mult_j$ are the metric and multiplier, respectively, for the jth barrierpoint out of a total of n barrierpoints.

Component	Parameters
Processor	1 and 4 sockets, 8 cores per socket
Core	2.66 GHz, 4-way issue, 128-entry ROB
Branch predictor	Pentium M (Dothan) [21], 8 cycles penalty
L1-I	32 KB, 4 way, 4 cycle access time
L1-D	32 KB, 8 way, 4 cycle access time
L2 cache	256 KB per core, 8 way, 8 cycle
L3 cache	8 MB per 8 cores, 16 way, 30 cycle
Main memory	65 ns access time, 8 GB/s per socket

TABLE I
SIMULATED SYSTEM CHARACTERISTICS.

Parameter	Value
-dim (number of projected dimensions)	15
-maxK (maximum number of clusters)	20
-fixedLength (clusters are not normalized)	off
-coveragePct (percent coverage)	1 (100%)

TABLE II
SIMPOINT PARAMETERS. DEFAULT VALUES USED FOR THOSE OPTIONS NOT SPECIFIED.

An example is provided in Figure 3 for `npb-ft`. The top graph plots aggregate IPC over time for the original, unsampled application. The bottom graph shows the different phases, and the result of the clustering step. The representatives for each region are marked in dark gray. The middle graph shows IPC as rebuilt by concatenating each region's representative. Aside from small differences, the representative is almost identical to the original.

IV. Micro-architectural State Reconstruction

There are a large number of micro-architectural warmup options available today [3], [9], [11], [22], [24]. Nevertheless, selecting a warmup strategy that is flexible, non-intrusive to the simulator and has a high speedup can be difficult. We propose a middle-ground for multi-threaded simulation cache warmup that maintains its speed and flexibility, while maintaining accuracy and being non-intrusive.

The two main micro-architectural warmup strategies are checkpointing and functional cache replay. Checkpointing tends to be the fastest warmup strategy as one only needs to load the amount of data that represents the state of the machine. But it tends to either be the least flexible, as naive checkpoints require a state snapshot for each micro-architecture and application combination, or require coherency-specific knowledge for accurately rebuilding the cache state. A more flexible but slower method is to run a functional simulation while updating microarchitecture state (e.g., issue memory requests to the cache hierarchy using a simple core timing model). This has the disadvantage that the execution time overhead is proportional to the number of instructions during the warmup period.

Instead of either of these extremes, we extended prior work [9], [24] that saves and then replays only the most recent unique address access requests. We first dynamically instrument an application and run them at near-native speeds (using a Pintool with only a 20× to 30× slowdown compared to native execution) to capture the most recently used data on a per-core basis. Each core keeps track of its most-recently used cache lines with a total capacity equal to the size of the shared LLC. Next, each thread replays their most recent access data in execution order to restore the state of the caches. The result is a significant reduction in simulated warmup replay time, as the size of the replayed cache state is on the order of the total LLC cache size and not based on the number of dynamically executed instructions up to this point.

V. Experimental Setup

In this work, we are using a modified version of the Sniper multi-core simulator [6], version 5.0, updated with a cycle-level core model. Each processor socket that we model is an octo-core processor with a three-level cache hierarchy, in which the L1 and L2 caches are private per core, and the last-level L3 cache is shared among all cores. Each core is 4-wide superscalar running at 2.66 GHz. We simulate both single-socket and four-socket shared-memory machines; we assume an MSI directory cache coherency protocol. See Table I for the main characteristics of the simulated machines.

The benchmark suites used in this paper are the NAS Parallel Benchmarks (NPB) version 3.3 with OpenMP parallelization (*class A* inputs) [12], and the PARSEC 2.1 benchmark suite (*simlarge* inputs) [4]. Of the 10 NAS Parallel Benchmarks, three were not used. We are unable to run the `npb-dc` (data cube) benchmark in our simulator because it produces a lot of output data that needs to be written to a hard drive; Sniper is a user-level simulator and does not model HDDs, for which reason it cannot accurately run the `npb-dc` benchmark. The `npb-ua` (unstructured adaptive mesh) benchmark generates a very large number of barriers which makes it difficult to analyze. Our current BarrierPoint implementation cannot handle that many inter-barrier regions. There is no fundamental reason to believe that BarrierPoint cannot be applied to this benchmark, however, it might need an extension to filter or combine regions before processing by the BarrierPoint methodology; we leave this for future work. Finally, `npb-ep` is an embarrassingly parallel benchmark, that only contains a single region between barriers. This type of workload does not apply to the BarrierPoint methodology. From Parsec, we use the `parsec-bodytrack` benchmark, as it is one of the three barrier-synchronized benchmarks that uses the OpenMP infrastructure. The other two benchmarks use pthread-based barrier synchronization. The current BarrierPoint implementation works with OpenMP-parallelized applications only, however, this is not a fundamental limitation to the methodology, and it should therefore be applicable for pthread barrier-synchronized applications as well.

Only the parallel Region of Interest (ROI) of each application is included in our timing measurements. We do not consider the serial fractions of these benchmarks; sampled simulation of sequential code running on individual cores has been studied extensively in prior work and is considered mature.

Other methodological settings are as follows. We use the passive OpenMP wait policy for thread synchronization, which specifies that waiting threads do not consume CPU resources.

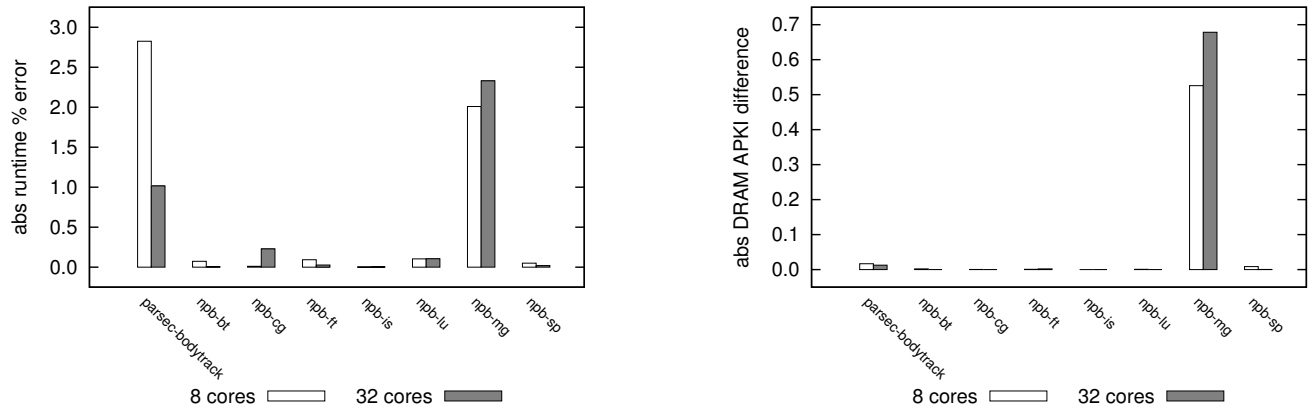

Fig. 4. Percent absolute error for predicting application execution time (left) and absolute DRAM APKI difference (right) across all benchmarks using the BarrierPoint methodology, assuming perfect warmup.

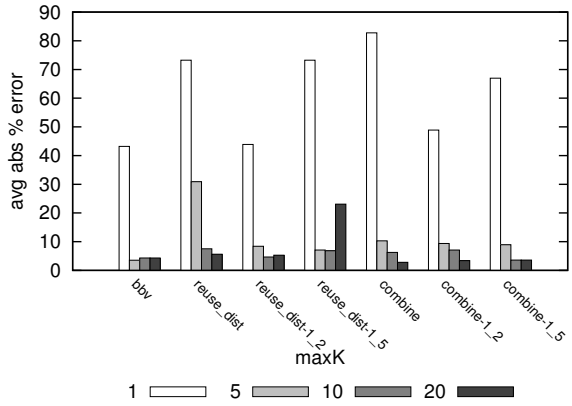

Fig. 5. Average absolute error for application execution time prediction for different maxK and clustering methods. This data is averaged across 8 and 32-core runs assuming perfect warmup. The LDV vectors are weighted equally ($1/v = 1/1$) or according to the value indicated.

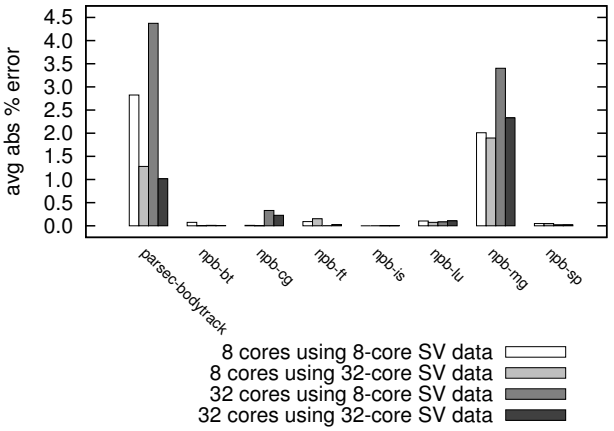

Fig. 6. Barrierpoint selection cross-validation. Results for 8 and 32-core runs are interchangeable, and therefore we can use the same regions as representative regions.

All benchmarks were compiled with GCC 4.3 for x86-64 with SSE and SSE2 extensions enabled. BBV and LRU stack distance profiles of inter-barrier regions are collected using a custom Pintool [15]. We use the SimPoint clustering software version 3.2 for identifying representative inter-barrier regions. Non-default parameters used for SimPoint are listed in Table II.

VI. Results

A. Barrierpoint selection

We start by evaluating the barrierpoints selection first. We do this by assuming perfect warmup in order to isolate the error due to barrierpoint selection. (We will evaluate the error due to warmup in Section VI-B.) This is done by running the simulation of the entire benchmark once and by recording performance metrics on an inter-barrier region granularity. We determine the barrierpoints as described before using microarchitecture-independent signatures, and pick the performance metrics for the barrierpoints from the full benchmark simulations to reconstruct total application execution time. Because all of the inter-barrier regions have a perfectly warmed up microarchitecture state, this approach thus evaluates barrierpoints selection in isolation. Comparing the estimated application execution time against the measured execution time for the full benchmark execution yields a metric for the accuracy of the barrierpoints selection.

Figure 4 quantifies the error of the BarrierPoint methodology assuming perfect warmup for (left graph) estimating total application execution time, and (right graph) LLC miss rate, or the number of memory accesses per thousand instructions (APKI). Accuracy is high for both metrics with an average absolute error of 0.6% (max error of 2.8%) for predicting application execution time, and an average absolute error of 0.1% (max error of 0.6%) for predicting the number of memory accesses per kilo instructions. Without barrierpoint scaling (where inter-barrier code regions are similar, but instruction counts differ; see Section III-D for details), the average error increases significantly from 0.6% to 19.4%. Overall, these results demonstrate the high accuracy of the barrierpoints selection strategy as well as the reconstruction mechanism for estimating total application execution time and performance

Application	input size	num cores	total num barriers	significant barrierpoints	num insignificant barrierpoints, combined multiplier, and total weight	barrierpoint number and multiplier				
npb-bt	A	8	1001	11	3 / 12.0 / 8.9e-04	3 (1.0)	144 (190.0)	172 (100.0)	407 (46.0)	448 (98.0)
						463 (48.0)	521 (200.0)	537 (54.0)	595 (10.0)	980 (189.0)
						993 (53.0)				
		32	1001	11	4 / 13.0 / 9.2e-04	195 (10.0)	196 (100.0)	199 (189.0)	220 (189.0)	327 (46.0)
						408 (100.0)	552 (65.0)	733 (100.0)	771 (100.0)	787 (54.0)
						802 (35.0)				
npb-cg	A	8	46	3	7 / 29.1 / 4.4e-04	0 (1.0)	15 (12.0)	21 (2.0)		
		32	46	3	5 / 28.3 / 5.0e-04	3 (4.0)	6 (7.0)	30 (4.0)		
npb-ft	A	8	34	9	3 / 7.0 / 1.5e-05	0 (1.0)	1 (1.0)	2 (1.0)	3 (1.0)	11 (3.0)
						15 (6.0)	19 (6.0)	26 (3.0)	28 (5.0)	
		32	34	9	3 / 7.0 / 1.7e-05	0 (1.0)	1 (1.0)	2 (1.0)	3 (1.0)	16 (3.0)
						25 (6.0)	28 (5.0)	29 (6.0)	31 (3.0)	
npb-is	A	8	11	10	1 / 1.0 / 5.9e-07	0 (1.0)	1 (1.0)	2 (1.0)	3 (1.0)	4 (1.0)
						5 (1.0)	6 (1.0)	7 (1.0)	8 (1.0)	9 (1.0)
		32	11	10	1 / 1.0 / 7.0e-07	0 (1.0)	1 (1.0)	2 (1.0)	3 (1.0)	4 (1.0)
						5 (1.0)	6 (1.0)	7 (1.0)	8 (1.0)	9 (1.0)
npb-lu	A	8	503	7	1 / 2.0 / 8.8e-05	60 (10.0)	98 (68.0)	122 (53.0)	195 (250.0)	222 (16.0)
						282 (56.0)	332 (47.0)			
		32	503	2	0 / 0.0 / 0.0e+00	11 (250.1)	276 (250.0)			
npb-mg	A	8	245	8	0 / 0.0 / 0.0e+00	2 (2.0)	52 (4.6)	57 (9.0)	116 (4.6)	123 (4.6)
						179 (4.6)	182 (17.6)	230 (4.7)		
		32	245	10	5 / 112.2 / 9.8e-04	2 (2.0)	50 (4.8)	51 (4.6)	63 (4.0)	112 (4.6)
						116 (4.0)	117 (9.0)	179 (4.6)	185 (4.6)	238 (13.0)
npb-sp	A	8	3601	16	2 / 2.0 / 3.5e-04	0 (1.0)	159 (400.0)	238 (137.0)	607 (91.0)	1111 (56.0)
						1478 (399.0)	1813 (51.0)	1948 (65.0)	1970 (20.0)	1976 (399.9)
						2169 (399.0)	2465 (379.9)	2595 (399.9)	2746 (399.9)	3004 (202.0)
						3319 (198.0)				
		32	3601	12	0 / 0.0 / 0.0e+00	502 (400.0)	784 (94.0)	850 (400.0)	955 (200.0)	964 (106.0)
						2069 (400.0)	2157 (21.0)	2643 (379.0)	3360 (400.0)	3456 (400.0)
						3497 (400.0)	3584 (400.0)			
parsec-bodytrack	large	8	89	13	1 / 1.0 / 5.4e-04	12 (1.0)	21 (3.0)	25 (7.0)	29 (16.0)	39 (16.0)
						40 (5.0)	60 (2.0)	62 (8.0)	72 (9.0)	74 (12.0)
						77 (4.0)	80 (4.1)	87 (1.0)		
		32	89	7	1 / 1.0 / 5.4e-04	6 (16.0)	30 (12.0)	32 (16.0)	39 (16.0)	65 (4.0)
						77 (4.0)	86 (19.5)			

TABLE III

APPLICATIONS, INPUT SIZES USED, TOTAL NUMBER OF DYNAMICALLY EXECUTED BARRIERS, SIGNIFICANT AND INSIGNIFICANT BARRIERPOINT INFORMATION AND THE SELECTED BARRIERPOINTS AND THEIR MULTIPLIERS. NOTE THAT BECAUSE OF SCALING A BARRIERPOINT CAN REPRESENT A FRACTION OF ANOTHER INTER-BARRIER REGION AND THEREFORE MULTIPLIERS DO NOT NECESSARILY SUM TO THE TOTAL NUMBER OF REGIONS.

metrics from a select number of barrierpoints.

1) Similarity and clustering metrics: We now explore the effect of a number of BarrierPoint parameters on accuracy. Figure 5 quantifies average error rates for predicting application execution time for different similarity methods and clustering parameters. We evaluate the impact of the maximum number of barrierpoints selected (maxK) on overall accuracy. We also consider signature vectors consisting of BBVs only, LDVs only, and combined BBV-LDVs. In addition, we also consider weighted LDVs, as previously described in Section III-A3, where $1/v$ is $1/1$, $1/2$ and $1/5$ as indicated in the figure. There are several interesting observations to be made here. First, a single barrierpoint yields poor accuracy but accuracy generally improves with an increasing number of barrierpoints. This makes intuitive sense as more regions are being simulated in detail and used to predict overall performance. It also illustrates the widely varying execution characteristics of inter-barriers regions in these workloads. Second, combined signature vectors that characterize both code and data memory access behavior yield the highest possible accuracy, especially with larger numbers of barrierpoints. Weighted LDVs improve accuracy only slightly when used in combined signature vectors; hence, we consider unweighted LDVs ($1/v = 1$) in our default setting. The highest accuracy is achieved with combined signature vectors and a maxK of 20, which is the default setting used throughout the paper unless mentioned otherwise.

2) Barrierpoints: A key feature of the BarrierPoint methodology is to provide an easy to use model for sampled simulation. The output of the methodology is a number of select, representative barrierpoints used for detailed simulation along with their multipliers which enables estimating total application execution time. Table III lists the significant barrierpoints for each of the benchmarks used in this study. The summary details for insignificant barrierpoints are defined as barrierpoints with a contribution of less than 0.1%. Across the benchmarks used, the number of selected barrierpoints is quite small, ranging between 2 and 16, and two to three orders of magnitude smaller than the total number of dynamically executed barriers. Note that because of instruction scaling as described in Section III-D, inter-barrier regions can be larger or smaller than similar ones, meaning that the multipliers do not necessarily sum to the total number of regions.

3) Barrierpoints across architectures: In Figure 6 we present the core cross-validation results. Here we can see that results from an 8-core BarrierPoint run produce similar accuracy numbers compared to the results from the 32-core BarrierPoint similarity generation. This demonstrates that for the OpenMP barrier runtime, the unit of work remains the same across core counts.

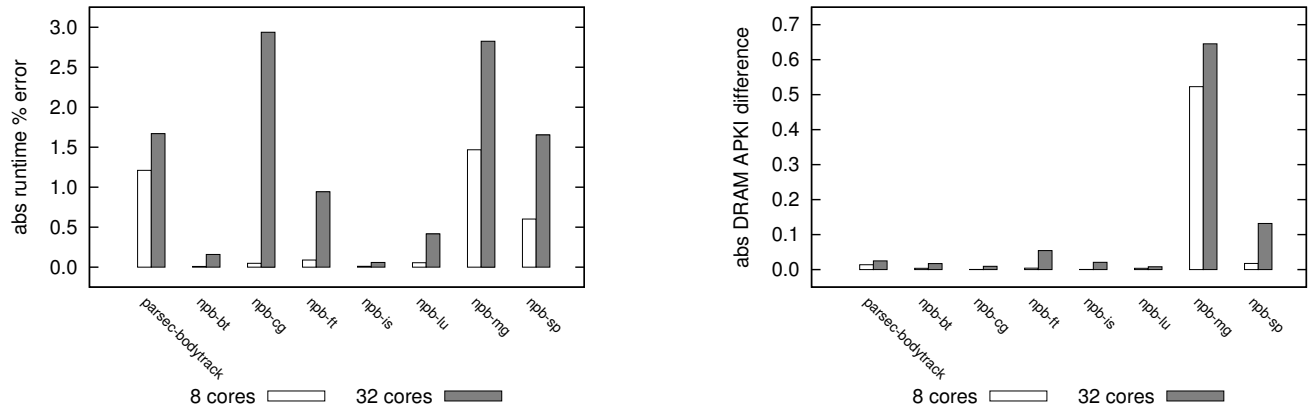

Fig. 7. Percent absolute error for predicting application execution time (left) and absolute DRAM APKI difference (right) across all benchmarks using the BarrierPoint methodology, with unique warmup.

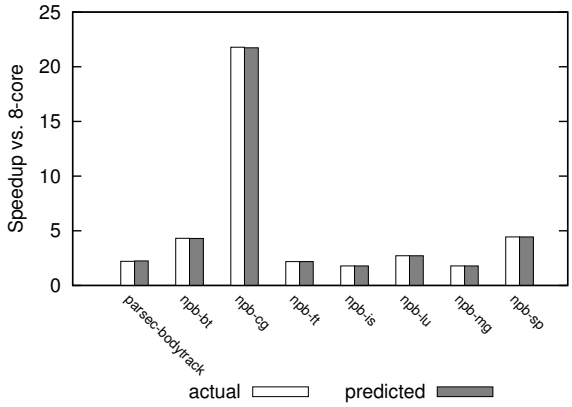

Fig. 8. Relative scaling results: estimating 8-core versus 32-core speedup.

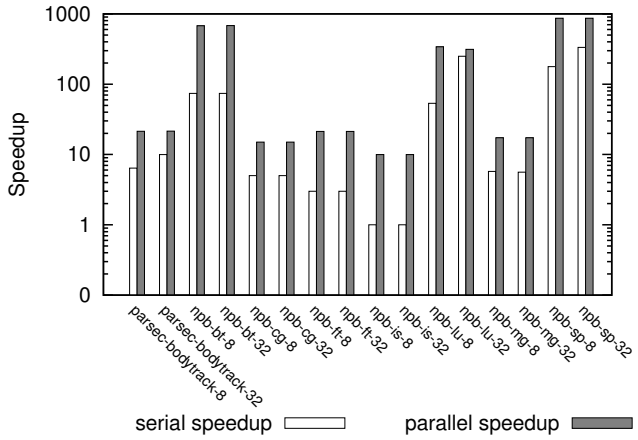

Fig. 9. Achieved speedups for each benchmark with the BarrierPoint methodology. The serial speedup results from back-to-back execution of barrierpoints and represents the reduction of required simulation resources. The parallel speedup results from a parallel simulation of barrierpoints and shows the simulation latency reduction assuming sufficient server resources.

Note that the BarrierPoint methodology requires taking a default thread (or core) count to collect the statistics that serve as input to the analysis. This is why Table III lists different barrierpoints for different core counts. However, barrierpoints can be reliably used across core counts (as long as the number of executed barriers does not depend on thread count). This is quantified in Figure 6 which shows accuracy results for using the barrierpoints determined with 8 threads on a 32-core system, and, vice versa, barrierpoints determined with 32 threads on an 8-core system. The key conclusion from this graph is that barrierpoints can be transferred across processor architectures, and hence form well-defined, fixed units of work that can be reliably used to compare processor architectures.

B. Warmup

In our evaluation up to this point, we have assumed perfect state warmup, which is an idealized situation. Typically when using BarrierPoint, one will need to warm up the microarchitectural state prior to detailed simulation of a barrierpoint. Figure 7 quantifies prediction error for application execution time (left graph) and the number of memory accesses per instruction (right graph). In spite of the simplicity of the proposed warmup technique, we find it to be quite accurate for determining application performance metrics. The execution time prediction error increases only slightly to 0.9% on average and 2.9% at most. The error due to incorrect warming is higher for 32 cores than for 8 cores (compare Figure 7 to Figure 4 which assumed perfect warmup): this is due to the larger total cache capacity (32MB versus 8MB) and performance prediction being more sensitive to accurate warmup for larger caches.

C. Relative accuracy

Up to this point, we have been concerned with estimating performance in a single processor architecture design point. However, architects are often more interested in predicting relative performance between two design points. Figure 8 quantifies the accuracy of BarrierPoint for predicting the relative performance difference between two design points, namely the 8-core versus the 32-core system. BarrierPoint's accuracy is very high for relative performance trends. Three of the benchmarks exhibit superlinear speedups, with `npb-cg`

as the most notable example, the reason being cache effects (32MB LLC in the 32-core system versus 8MB in the 8-core system).

D. Simulation speedup

Figure 9 quantifies the simulation speedups for the BarrierPoint methodology across different core counts for the NPB and Parsec benchmark suites. Speedups are defined as the reduction in aggregate instruction count. The serial speedup is the reduction in required resources, and can be thought of as the speedup when running each barrierpoint back-to-back in serial. The parallel speedup results when all barrierpoints are run in parallel with sufficient machine resources. Using the BarrierPoint methodology, our results show a (harmonic) mean parallel speedup of 24.7× and a maximum parallel speedup of 866.6×. Along with application performance improvements, we see an average reduction of 78× in the number of machine resources required for simulation.

VII. Related Work

A. Single-Threaded Sampling

The SimPoint [20] methodology clusters large intervals, on the order of 100M instructions, using BBVs and machine learning (cluster analysis) to identify representative chunks of an application in a microarchitecture-independent way. The SMARTS [26] methodology and Conte et al. [8] construct a sample consisting of a large number of sampling units of a relatively small number of instructions per sampling unit. These approaches are unable to accurately estimate run-time of synchronized multi-threaded applications.

B. Multi-Threaded Sampling

Ekman et al. [10] propose matched-pair comparison as a way to reduce sample size for multi-threaded workloads, but show that synchronized applications do not see a significant sample size reduction with their technique.

Wenisch et al. [25] propose Flex Points as a way to increase simulator performance for multi-threaded commercial workloads. Van Biesbrouck et al. [23] propose the Co-Phase Matrix as a reduction technique for multi-program workloads. Both of these techniques depend on the fact that each thread is independent. Explicit thread synchronization violates their assumptions.

Perelman et al. [17] extend the SimPoint methodology to parallel workloads via instruction-based sampling. Recent work [2], [7] has shown however that instruction-based sampling is inaccurate for multi-threaded workloads that employ active or idle waiting due to synchronization. Furthermore, application IPC is an inappropriate metric for multi-threaded workloads [1]. BarrierPoint, instead, predicts total application execution time, and uses both code and data memory signatures to identify representative barrierpoints.

Time-based sampling of multi-threaded applications, proposed both by Ardestani et al. [2] and Carlson et al. [7], allows for accurate sampled simulation of synchronizing multi-threaded applications. By extrapolating time during fast-forwarding phases, and taking care to keep thread interactions through synchronization and shared memory intact, execution time can be accurately predicted. These approaches suffer from two major limitations, which we overcome with BarrierPoint. First, it requires functional simulation of the entire program execution and in addition requires the memory hierarchy to be warmed in between sampling units. In contrast, BarrierPoint can leverage checkpointing to allow barrierpoints to be simulated independently and in parallel, and does not require functional simulation and cache warming of the complete benchmark execution. Second, time-based sampling can lead to the creation of different samples across processor architectures which may complicate performance analysis. BarrierPoint overcomes this limitation by proposing well-defined, fixed units of work in a microarchitecture-independent way.

C. Simulation parallelism

Simulation latency can be improved by exploiting parallelism in the units of work that need to be simulated. For example, both SimPoint and SMARTS have proposed checkpointing techniques to simulate each sampling unit independently and in parallel [22], [24]. Bryan et al. [5] demonstrates the potential speedup when executing multiple inter-barrier regions in parallel, provided that massive simulation resources are available. Our work takes this work one step further by reducing the number of inter-barrier regions to be simulated in detail, which dramatically reduces the number of simulation machines needed. This improves overall time-to-discovery while being able to accurately predict total application execution time from the selected of barrierpoints.

D. Warmup

The cold-start problem is a well-known problem in sampled simulation. No-State-Loss (NSL) [9] and Live-points [24] propose the playback of unique addresses in the memory hierarchy to warm up cache state prior to each sampling unit, and apply it to single-threaded workloads running on single-core systems. MRRL [11] uses the distribution of reuse distances to determine how far to go back prior to the sampling unit to warm up caches. We extend these methods to support multi-threaded workloads and multi-core cache hierarchies by replaying an amount of data equal to the largest last-level cache visible to each core.

Van Biesbrouck et al. [22] propose a warmup method, memory hierarchy state (MHS), that uses a snapshot of the largest cache to be simulated, and reduces the state for the target simulated cache. This technique requires explicit cache state reconstruction in the simulator, is limited to single-core systems, and does not provide a way to reconstruct coherency state.

Barr et al. [3] propose a method for reconstructing cache and coherency state in multi-processor systems. They therefore propose a data structure, called the Memory Timestamp Record (MTR), that records the timestamp of the last access to each cache block before a sampling unit. The MTR allows the simulator to reconstruct coherency state as well as a cache

hierarchy of arbitrary size and associativity, assuming a lower bound on cache line size.

Our warmup methodology offers an alternative to MTR where coherency state and cache hierarchy is reconstructed without detailed knowledge of the cache hierarchy's coherence and multi-level state. The only information needed is the largest total shared LLC capacity that will be simulated in any system configuration.

E. Similarity analysis

Perelman et al.'s [17] similarity analysis extends Sherwood et al.'s work [20] to multi-threaded workloads. Both of these works evaluate agglomerated/combined thread views into fixed-length intervals but disregard its use as they compare threads to one another during an execution run. On the other hand we compare the entire multi-threaded application's behavior between barriers, not each thread to one another. We are the first to:

- Show that combined thread views can be used to compare inter-barrier regions to perform workload reduction.
- Use LRU stack distances (along with BBVs) to more accurately compare regions.
- Enable the use of variable-length multi-threaded execution profiles which can occur between barriers. Perelman et al. [17] assume fixed-length intervals that are not compatible with variable-length barrierpoints.

Sherwood et al. [20] use BBVs to identify regions of similar execution behavior, called phases, in long-running single-threaded applications; Perelman et al. [17] extend this method to multi-threaded workloads. Shen et al. [18] propose LRU stack distances [16] to automatically determine phases of a single-threaded application. In contrast, our methodology uses program semantics, barriers, to delineate phases, and we find the combined usage of BBVs and LRU stack distances to outperform either alone.

Alameldeen et al. [1] suggest that a common unit of work is required, but does not extend this to multi-threaded workloads, nor do they address the reduction of simulation requirements. In contrast, we are the first to provide an automatic workload reduction and performance estimation methodology for barrier-based multi-threaded workloads.

VIII. Conclusion

Sampling is a well-known technique to speed up architectural simulation. Only recently have researchers extended sampled simulation towards multi-threaded workloads. Some prior work assumed non-synchronizing multi-threaded workloads for which random sampling allows for an accurate representation of the overall application. Time-based sampling proposes a solution for synchronizing multi-threaded workloads but the main limiting factor for achieving significant simulation speedups is the requirement for using functional warming to maintain an accurate micro-architectural state in-between sampling units. In addition, time-based sampling leads to different sampling units across different processor architectures, complicating performance analysis. Prior work in speeding up the simulation of barrier-synchronized applications requires massive simulation resources to simulate all inter-barrier regions in parallel.

To address these limitations, we propose BarrierPoint, a methodology for the reconstruction of application execution time using the similarity of multi-threaded benchmarks between barriers. With BarrierPoint, it is now possible to evaluate the performance of each selected inter-barrier region independently, leading to a higher potential for simulation speedup. Our proposed methodology automatically identifies a select number of most representative regions, called barrierpoints, from which it is possible to estimate total application execution time. Using a set of barrier-synchronized parallel benchmarks from the NPB and Parsec benchmark suites, we demonstrate that high simulation speedups that can be achieved (with a harmonic mean of 24.7× and up to 866.6×) while using a limited number of simulation machine resources, and while being within 0.9% on average and at most 2.9% compared to detailed simulation. Overall, we reduce the amount of machine resources needed by an average of 78×.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work is supported by Intel and the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). Additional support is provided by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295. Experiments were run on computing infrastructure at the ExaScience Lab, Leuven, Belgium; the Intel HPC Lab, Swindon, UK; and the VSC Flemish Supercomputer Center.

References

[1] A. R. Alameldeen and D. A. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, pp. 8–17, Jul./Aug. 2006.

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

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

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[5] P. Bryan, J. Poovey, J. Beu, and T. Conte, "Accelerating multi-threaded application simulation through barrier-interval time-parallelism," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, Aug. 2012, pp. 117–126.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011, pp. 52:1–52:12.

[7] ——, "Sampled simulation of multi-threaded applications," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2013, pp. 2–12.

[8] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *Proceedings of the International Conference on Computer Design (ICCD)*, Oct. 1996, pp. 468–477.

[9] T. Conte, M. Hirsch, and W.-M. Hwu, "Combining trace sampling with single pass methods for efficient cache simulation," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 714–720, Jun. 1998.

[10] M. Ekman and P. Stenström, "Enhancing multiprocessor architecture simulation speed using matched-pair comparison," in *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2005, pp. 89–99.

[11] J. W. Haskins, Jr. and K. Skadron, "Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2003, pp. 195–203.

[12] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS Parallel Benchmarks and its performance," NASA Ames Research Center, Tech. Rep., Oct. 1999.

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

[14] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder, "The strong correlation between code signatures and performance," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2005, pp. 236–247.

[15] 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 *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

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

[17] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting phases in parallel applications on shared memory architectures," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2006, p. 88.

[18] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, Oct. 2004, pp. 165–176.

[19] ——, "Predicting locality phases for dynamic memory optimization," *Journal of Parallel and Distributed Computing*, vol. 67, no. 7, pp. 783 – 796, Jul. 2007.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

[21] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 207–217.

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

[23] M. Van Biesbrouck, T. Sherwood, and B. Calder, "A Co-Phase Matrix to guide simultaneous multithreading simulation," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Sep. 2004, pp. 45–56.

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

[25] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe, "SimFlex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, Jul./Aug. 2006.

[26] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2003, pp. 84–95.