Sampled Simulation of Multi-Threaded Applications

Trevor E. Carlson[1,2] Wim Heirman[1,2] Lieven Eeckhout[1]

[1]Department of Electronics and Information Systems, Ghent University, Belgium
[2]Intel ExaScience Lab, Belgium

***Abstract*—Sampling is a well-known workload reduction technique that allows one to speed up architectural simulation while accurately predicting performance. Previous sampling methods have been shown to accurately predict single-threaded application runtime based on its overall IPC. However, these previous approaches are unsuitable for general multi-threaded applications, for which IPC is not a good proxy for runtime. Additionally, we find that issues such as application periodicity and inter-thread synchronization play a significant role in determining how best to sample these applications.**

The proposed multi-threaded application sampling methodology is able to derive an effective sampling strategy for candidate applications using architecture-independent metrics. Using this methodology, large input sets can now be simulated which would otherwise be infeasible, allowing for more accurate conclusions to be made than from studies using scaled-down input sets. Through the use of the proposed methodology, we can simulate less than 10% of the total application runtime in detail. On the SPEComp, NPB and PARSEC benchmarks, running on an 8-core simulated system, we achieve an average absolute error of 3.5%.

I. Introduction

There has been extensive research done with respect to application workload reduction. The reduction of workloads to their relevant components allows for large, realistic input sets to be used both for future architecture exploration as well as application evaluation. Through sampling, architects can perform detailed simulation on a small percentage of the application, covering its most relevant portions [8], [20], [26]. Compared to simulating a complete application with hundreds of billions of instructions, being able to simulate just a few hundred million instructions in a detailed fashion, while keeping accuracy high, allows architects to explore new architectures with different core and un-core components in much shorter time.

But with increasing core counts in today's microprocessors, from 10 in the latest Intel Xeon microprocessors, 16 in the AMD 6262 series processors, and more than 50 in the Intel MIC architecture, multi-threaded applications are being seen as a major way to gain performance in the many-core era. In addition to multi-threaded workloads, there is a trend toward larger on-chip caches that require even longer runtimes to stress the cache in a meaningful way.

Recent sampling algorithms assume that a number of restrictions are placed on the applications to study. For single-threaded applications, the most important limitation to using sampling to predict application runtime is the assumption that the sampled IPC can act as a proxy for an entire application's runtime [20], [26]. In recent multi-threaded sampling techniques [10], [25], each thread needs to be treated as independent, where they do not explicitly affect the progress of other threads' execution. In other words, per-thread behaviors must be uncorrelated. But for multi-threaded applications that use synchronization primitives, such as locks or barriers, IPC is no longer sufficient and the threads can now directly affect the progress of one another. Threads in these applications can be idle or spinning, unable to make any additional forward progress until one or more other threads reach a synchronization point. During this time, other threads will continue to advance, introducing a gap that represents this thread's idle time. Because of these gaps, it is not possible to solely use IPC and instruction counts to approximate runtime for general multi-threaded applications [1].

In addition to issues involving a large class of multi-threaded applications, program phase behavior is also an important aspect for application sampling. In existing work, the SMARTS methodology [26] is able to accurately predict application IPC by simulating a large number of very small intervals, whereas others [7], [20] use phase behavior to guide sample selection. Without an understanding of an application's phase behavior during sample creation, the result could end up containing periods that alias the original application. Predicting runtime behavior with a sample that aliases the original application has the potential to provide inaccurate results.

To overcome these issues, we propose a multi-threaded application sampling methodology with the following key features: (i) it performs detailed application synchronization during fast-forwarding while keeping track of per-thread performance, and (ii) it uses application phase behavior to select appropriate sampling parameters. We demonstrate that accurately estimating per-thread performance and simulating thread interactions during fast-forwarding is required to maintain high runtime accuracy in tightly synchronized multi-threaded applications. Additionally, through the use of fast pre-simulation application analysis, we take into account application periodicity to allow for accurate multi-threaded application sampling. To the best of our knowledge, this proposal is the first to offer a methodology for performing sampled simulation of multi-threaded applications while maintaining high predicted runtime accuracy.

In this work we detail the following contributions:

- We provide a methodology for sampling multi-threaded workloads that provides up to a 5.8× simulation time reduction with an average absolute error of 3.5% while simulating less than 10% of the application in detail.
- We show that both computing per-thread IPC as well as handling inter-thread interactions even during fast-forwarding increases accuracy significantly.
- We demonstrate that application phase behavior needs to be understood and properly taken into account when

sampling multi-threaded applications where threads can not be assumed to run independently. To accomplish this, we propose a microarchitecture-independent methodology to determine application phases and how to select the appropriate options for the required speed and accuracy trade-offs.
- We show that large, realistic input sets are needed to make correct design decisions, which in its turn requires accurate sampling for making the simulation of these inputs feasible.

II. Fast-Forwarding Parallel Applications

A. Requirements for Accurate Parallel Fast-Forwarding

Existing techniques for sampled simulation of single-threaded applications, or those treating each thread of a multi-threaded program independently, use purely functional simulation to fast-forward through non-sampled regions [20], [26], or use checkpoints to avoid simulating them at all [23].

These techniques do not directly apply to multi-threaded applications where synchronization or explicit interactions occur. In parallel applications, threads interact through shared memory and synchronization events, influencing the timing of neighboring threads. The use of tracing, or other forms of checkpointing of microarchitectural state, such as used in PinPlay [17] or Flex Points [25], further constrains the absolute ordering of threads in an application. This in turn limits the ability of an architect to view new thread orderings — and their resulting load (im)balance or ability of overlapping communication and computation — that would otherwise occur in an execution on different micro-architectural configurations. Therefore, functional and timing simulation cannot be completely separated during fast-forwarding, but instead, one must take care to preserve the timing of synchronization events. Additionally, the different threads of parallel applications can make progress at different speeds — either because they run different code, or they exhibit data-dependent behavior where distinct memory access patterns cause threads to experience different cache miss rates and thus have unequal performance. Considering these effects on a per-thread basis is therefore necessary.

B. Accurate Multi-Threaded Fast-Forwarding

In our proposed technique, we employ functional simulation of the complete benchmark to capture a sufficient level of accuracy for multi-threaded applications. Sampling is done by periodically simulating detailed performance models during intervals of a predetermined length (the *detailed interval* length D), separated by periods of non-detailed simulation (*fast-forward intervals* of length F). In contrast to single-threaded simulation, we keep track of simulated time, and maintain inter-thread dependencies through shared memory and synchronization events, even while fast-forwarding. We also base sample selection on time, not instruction count, as the latter — due to differences in performance and idle periods among threads — is not comparable across cores.

Figure 1 illustrates our fast-forwarding mechanism. Intervals of a fixed length of simulated time are simulated in detail.

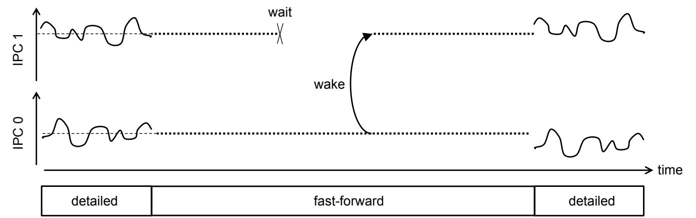


Fig. 1. Proposed mechanism of fast-forwarding during multi-threaded sampled simulation.

For each thread, we record the number of instructions executed, and the time this thread did *not* sleep waiting for synchronization events (locks, barriers, etc.) or spin-locks. This allows each thread's execution speed, in instructions per cycle (IPC) to be calculated for the non-idle periods. This non-idle IPC value summarizes the hardware's performance for the section of code executed during the detailed period. Although it is possible to automatically detect and account for spin-locks [15], we chose to use the OpenMP passive wait policy in this work.

While fast-forwarding, the non-idle IPC, along with the current instruction count is used to keep track of each thread's elapsed time. Most importantly, synchronization events are simulated as normal; i.e., when a thread goes to sleep, functional execution is halted for that thread, and once the thread wakes up it is provided with the current time and functional simulation continues.

In this work, we keep functional cache simulation enabled at all times, and focus on the sampling methodology itself. The use of more efficient warmup techniques, such as Barr et al.'s memory timestamp record [4], would allow for additional speedups and could be a potential direction for future work.

C. Comparison of Fast-Forwarding Techniques

The key aspects of our proposed fast-forwarding mechanism are to show how one can best preserve inter-thread synchronization and its effect on simulated time, and how accurate knowledge of per-thread IPC variations through time improves accuracy. In this section, we will evaluate the importance of each of these aspects.

Prior work [2], [19] ignored synchronization events during fast-forwarding periods (we call this *no-sync* fast-forwarding). In addition to using synchronization during fast-forwarding, we evaluated a number of alternatives for determining the IPC to use during each fast-forwarding interval. The approaches evaluated either account for time very simply (*one-ipc*), or require up-front knowledge about an application's performance on the architecture before the sampled run commences (*oracle-global* and *oracle-perthread*).

The *one-ipc* mechanism fast-forwards each thread at a fixed IPC of one, so — except when threads are idle — each thread is fast-forwarded by the same number of instructions. The *oracle-global* and *oracle-perthread* fast-forward mechanisms use IPC information from a fully-detailed simulation, rather than from the previous detailed interval. This allows for a comparison with a theoretical situation where the IPC error caused by sampling is removed, but through-time IPC variations are not taken into account. The *oracle-global* mechanism

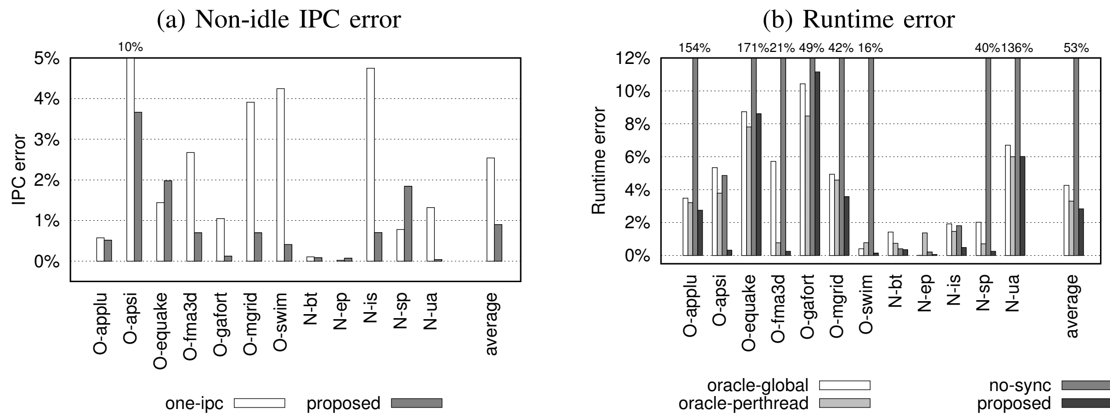


Fig. 2. Accuracy of sampled IPC (left graph) and estimated runtime (right graph) for simulations using different fast-forwarding mechanisms.

uses a single fast-forward IPC (the harmonic mean of the IPC for the complete application over all threads), whereas *oracle-perthread* uses the per-thread average. Additionally, both *oracle-global* and *oracle-perthread* use non-idle periods to determine fast-forwarding IPC in the same way that the proposed method does. Finally, the *no-sync* fast-forwarding method does not model the timing of synchronization events during fast-forwarding. Instead, it uses the fast-forward IPC as measured during the preceding detailed interval, similarly to the proposed method, but now the fast-forward IPC lumps together idle and non-idle periods.

In Figure 2, we contrast different fast-forwarding mechanisms on a simulated 8-core, shared memory machine. (See Section IV for additional micro-architectural details.) The non-idle IPC, the IPC that occurs when the core is not blocked, waiting for other threads, of the *one-ipc* mechanism and our proposed approach is shown in Figure 2(a). Here we see that for most cases, and on average, the proposed fast-forwarding method is more accurate at predicting IPC than the *one-ipc* model. Graph (b) of Figure 2, shows how the prediction of total application runtime is affected by different modeling components. Our proposed fast-forwarding technique predicts a simulated runtime with under a 3% average absolute error compared to an average absolute error of 53% for *no-sync*. This shows that taking synchronization into account during fast-forwarding is essential for high accuracy. In addition, the proposed technique's average absolute runtime prediction error is slightly better than both oracle mechanisms, showing that through-time IPC variations are important as well.

III. Sample Selection in Parallel Applications

In addition to being able to fast-forward a multi-threaded application while accurately keeping track of the threads' relative progress, there is a critical concern about appropriate sample selection. We make the case that an understanding of application periodicity is crucial for effective sample selection. First, we will show how application periodicity leads to sampling errors, and how this problem applies to our multi-threaded sampling methodology. We then show how one can determine application periodicity in microarchitecture-independent ways, and go on to use this information to build a methodology that constructs reliable sampling parameters based on these application characteristics.

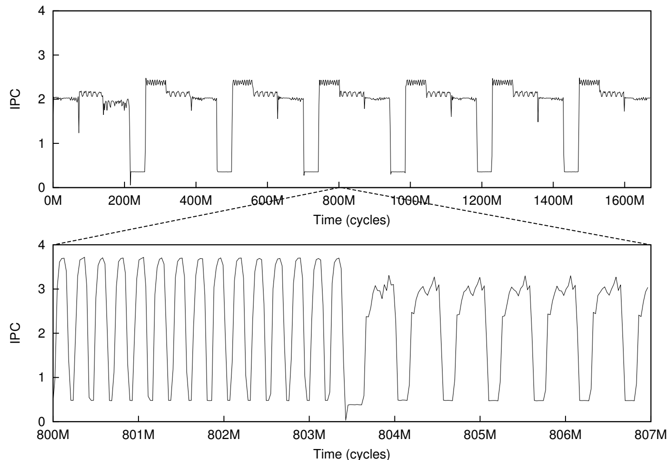


Fig. 3. IPC trace of `N-ft` (thread 0 out of 8, class A input set): full run (top) and zoomed in (bottom). Several periodicities are visible.

A. The Effect of Periodicity on Sampling

Many applications exhibit inherent periodicity or phase behavior [18], [20]. Figure 3 plots the IPC variation through time of one of the eight threads for the class A input set of the `N-ft` benchmark from the NAS Parallel Benchmarks (NPB) suite. At the macro scale, seven iterations can be seen of a single main period, which has a length of approximately 220M clock cycles. In Figure 3 (bottom), the periodicities can be observed at a different scale, with two phases, each with their own iteration length (at 230k and 550k cycles).

Previous work has shown that these inherent application periods can be used to guide sampling. For instance, Casas et al. [7] sample hardware performance counters for exactly an integer number of periods. In this situation, it is feasible to measure the length of one period exactly, since the application runs on actual hardware at native speed. In simulation, however, the application's performance on a given architecture, and hence its periodicity, is a priori unknown.

Casas et al. also note that if the sampling period does not exactly match the size of the periodicity of an application (or

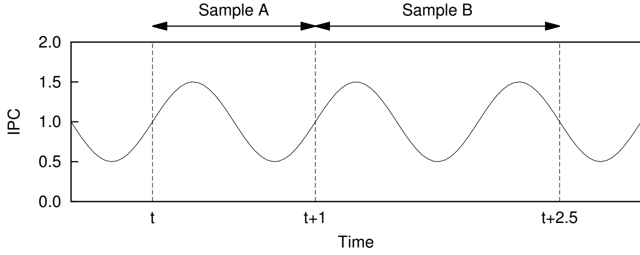


Fig. 4. Sampling with intervals of exactly one period yields a correct IPC average; when application period and detailed length do not match, sampling errors occur.

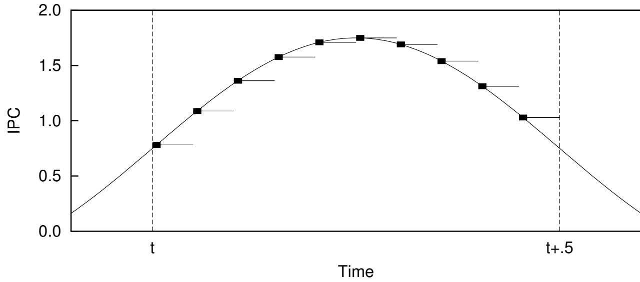


Fig. 5. When sampling inside of an application's period, a sufficient number of intervals need to be collected to ensure that fast-forwarding IPC accurately tracks actual IPC.

an integer multiple), aliasing can occur which can significantly increase the error and variability. In our framework, the IPC sample from the collection of detailed intervals needs to have a high degree of accuracy, as the detailed application performance is used to determine the progress each thread makes during fast-forwarding. Consider for instance the example IPC trace of Figure 4. Interval A contains exactly one period; its IPC is therefore equal to the global average. However, interval B, which has a length close to but not exactly equal to the periodicity, has a measured IPC that can be incorrect.

A related problem occurs when sampling intervals are taken *inside* one (much larger) application period, see Figure 5. Here, solid squares represent detailed regions, their average IPC is projected forward during fast-forward phases (horizontal lines). Taking a small number of intervals within each iteration can yield inaccurate results, as the instantaneous IPC will change too much in-between intervals. We therefore want to maximize the number of intervals taken inside one period, so the shape of its IPC curve can be accurately described.

In other sampling methodologies, this type of aliasing is not an issue since the IPC of many small intervals inside a sample is averaged, which because of the central limit theorem yields an accurate estimate of the IPC of the whole application. In our run-time prediction methodology, however, we rely on the detailed regions to be an accurate representation of that region and extrapolate it during fast-forwarding, in order to compute the total program run-time. In contrast to the IPC of single-threaded applications, where positive and negative errors can cancel out during averaging, run-time of parallel applications with inter-thread synchronization behaves differently: positive errors (overestimation of run-time) are

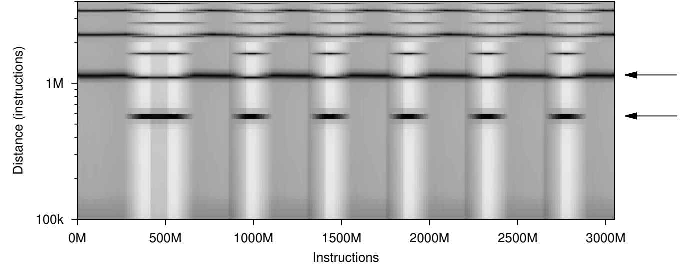


Fig. 6. BBV autocorrelation for `N-ft` (thread 0 out of 8, class A input set). Strong correlations (dark bands), pointing to periodic behavior, are visible at 550k and 1.14M instructions and their harmonics.

often propagated when other threads wait on the slow thread; whereas negative errors (underestimation) are masked for non-critical threads. Therefore, for parallel applications with a substantial amount of synchronization, total run-time is not simply the mean or sum of run-times of the detailed periods, and the central limit theorem does not directly apply to it.

To avoid these aliasing problems, we determine the sampling parameters on a per-application basis, given the application's periodicities. By taking application periodicity into account, one can avoid introducing sampling errors that are caused by the aliasing of the application's periodicities with the detailed sampling period. The next step in our proposed methodology determines the periodicities which allows one to generate the necessary sampling parameters to avoid aliasing.

B. Determining Application Periodicity

While there are multiple methods to determine application periodicity, we chose to look at those that are micro-architecture independent to allow for up-front calculation of application periodicities regardless of the simulated architecture or the simulation infrastructure used. We use signal analysis techniques in a similar fashion compared to prior work [9], [12] to allow us to capture micro-architecture independent application characteristics.

Our primary method to determine application periodicity is through the collection of basic-block vectors (BBVs) as outlined in [20]. We then perform a windowed auto-correlation on these BBVs. We have created a parallel Pin [16] tool that both generates BBVs and performs the auto-correlation step with minimal application slowdown (around 10× vs. native execution).

An auto-correlation $A(d)$ of the time series of BBVs $B(t)$ is the comparison of a vector of BBVs (a call history) with a version of itself that is at a given offset in time d:

$$A(d) = \sum_t \|B(t) - B(t+d)\|$$

By comparing the vector with itself, the sum $A(0)$ is zero which denotes a perfect match. As the offset d between the two vectors is increased, one is able to measure the similarity of the BBVs seen with those at a later point in time. By detecting the points of highest correlation between the two shifted vectors, we can obtain the periodicities of the application. Typically, an auto-correlation is done with the entire vector against itself;

by using a windowed auto-correlation in which the summation runs over a localized window around t, rather than over the length of application, changes in periodicity throughout the application's run can be made visible.

An example of the output can be seen in Figure 6. Application runtime, measured in instructions, is on the horizontal axis; the vertical axis represents the offset d. Light colors denote a low similarity between the BBVs at a given point in the program with those a distance d away, whereas dark colors denote strong correlation — which implies similar execution behavior [14]. In this case, we can see that the `N-ft` benchmark, running the class A input set with 8 threads, has one periodicity at 550k instructions that occurs for a part of the application runtime, and another which occurs at 1.14M instructions and exists during the entire application execution. These periodicities correspond directly to the 230k and 550k cycle periodicities, respectively, in Figure 3.

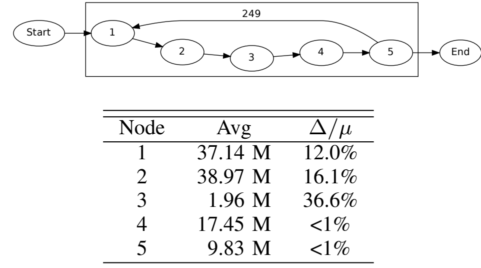
C. Detecting Large Application Variability over Long Periods

The proposed sampling methodology for multi-threaded workloads makes the assumption that we can detect, and therefore avoid aliasing of the periodicities in an application. Some applications, however, have irregular behavior which can be difficult to determine using the BBV autocorrelation technique alone. This behavior affects the quality of the detailed sample, potentially causing aliasing which can lead to large errors. We therefore augment the BBV-based analysis with a second technique which allows detection of irregular behavior early; this analysis will indicate which applications are not amenable to be sampled reliably.

This technique works by detecting loops in the application's call graph, and comparing the instruction counts of each iteration. For example, by using the OpenMP runtime library, we can monitor the high-level application periodicity. The OpenMP functions are usually called as the result of `#pragma omp` directives in the source code and are therefore closely related to the high-level structure of the application. Using different sets of marker functions, this technique can be applied to most parallel programming models.

The call structure of each thread is derived using a separate Pin tool, which at near-native execution time records a call graph of the application limited to the set of marker functions. This call graph is annotated with both a per-thread and per-loop instruction count. We then use Tarjan's algorithm [21] to determine nested loops inside of the graph. This makes the high-level structure of the application apparent; comparing instruction counts for different iterations provides insight into the application's level of irregularity. For regular applications, the instruction counts thus obtained can be used to verify or augment those obtained through BBV-based analysis.

From this analysis, irregular behavior that can potentially alias the detailed sampling of applications can be detected early. In the case of `N-lu`, the instruction count variabilities as observed in Figure 7 are such (>10%) that we can a priori say that sampled simulation will not provide accurate runtime predictions; we have experimentally verified that errors for this application are indeed as high as 20 to 30%, while the maximum error for applications that show regular behavior is around 8%, with an average absolute error just below 3%. The same holds for `O-ammp` which shows high variability both in the BBV and OMP analysis methods. We therefore leave generalizing the proposed sampling method to irregular applications for future work.



Node	Avg	Δ/μ
1	37.14 M	12.0%
2	38.97 M	16.1%
3	1.96 M	36.6%
4	17.45 M	<1%
5	9.83 M	<1%

Fig. 7. Loop structure (top) with application call points as edges for the `N-lu` application (thread 3 of 8, class A input set). Node instruction counts (below) with relative spread Δ/μ, defined as $\frac{i_{max}-i_{min}}{i_{count}}$.

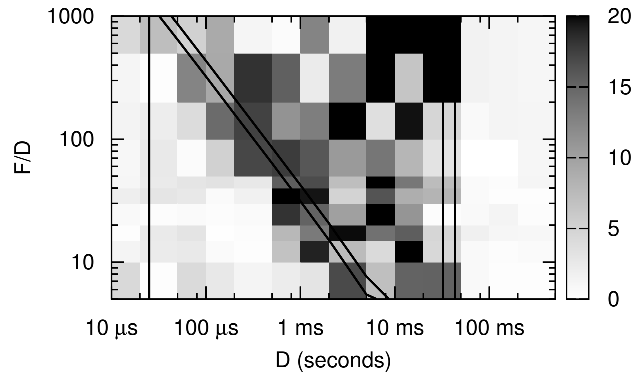


Fig. 8. Sampling error versus application periodicity for `N-bt`, class A input set with 8 threads. Also shown are the periodicities of the application (solid lines).

D. Deriving Optimal Sampling Parameters

As noted in Section III-A, using a detailed interval length that is close to an application's periodicity can lead to large sampling errors. For short periods, we will therefore want to sample using a detailed interval length D that is significantly larger than the periodicity P. Similarly, for long application periods we want to take a sufficient number of intervals to accurately describe the IPC changes during this period. Note that in this case, the period at which intervals are taken is of size $D+F$, so the number of intervals taken within a period P equals $P/(D+F)$.

Figure 8 shows the results of a complete set of runs across all sampling parameters for the `N-bt` benchmark, showing the runtime error compared to a full-detailed simulation at each (D,F) combination. The graph also shows, for the three different periodicities that occur in this application, the iso-lines where $D=P$ (vertical lines) and $D+F=P$ (diagonal lines). On the right side of the graph, for $D > 50$ ms, the detailed region length is longer than all of the application's

TABLE I
SIMULATED SYSTEM CHARACTERISTICS.

Component	Parameters
Processor	2 sockets, 4 cores per socket
Core	2.66 GHz, 4-way issue, 128-entry ROB
Branch predictor	Pentium M [22], 17 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 4 cores, 16 way, 30 cycle
Main memory	65 ns access time, 8 GB/s per socket

periodicities, here sampling works well as the error is low (<3%). Moving from the middle towards the lower left corner also increases accuracy, as this moves $D+F$ farther away from the two larger periodicities and thus increases the number of intervals taken inside each period. On the other hand, sampling parameters close to $D = P$ or $D + F = P$ yield much higher errors, up to 15% even for conservative amounts of fast-forwarding. Additionally, the area in between the $P = D$ and $P = D + F$ region also does poorly. Here, an interval is measured that represents part of the application's main period, but this interval is subsequently used in fast-forwarding across multiple iterations of this period. This can introduce large errors when the IPC of the collected interval is not representative of the IPC of the whole period. It therefore makes sense that we would not want to collect intervals using parameters in this region.

Converting Instruction Periodicities to Time: Although both methods for determining instruction periodicity described in the previous section yield a (micro-architecture independent) result expressed in instructions, our method of multi-threaded sampling requires its parameters to be expressed in time. Since we will only use these periodicities to define *forbidden zones* for these parameters, we do not need the periodicity's exact length in cycles. Instead, we assume the benchmark will be executed at an IPC of between 0.5 and 2.0, which are typical long-term average IPC values for the benchmarks used. Each periodicity P thus becomes a range of $[0.5P \ldots 2.0P]$. Note that the four-way issue out-of-order processor core we model regularly achieves an IPC close to four; this is usually only for short periods whereas the long-term averages can be much lower, see also Figure 3.

IV. EXPERIMENTAL SETUP

A. Simulation Configuration

For the results shown in this paper, we used the Sniper multi-core simulation infrastructure [6]. We configured it to model a multi-core out-of-order processor resembling the Intel Nehalem processor, see Table I for its main characteristics. The benchmark suites used in this paper are the SPEC OpenMP (medium) suite (*train* inputs) [3], the NAS Parallel Benchmarks version 3 with OpenMP parallelization (*class A* inputs) [13], and the PARSEC 2.1 benchmark suite (*simlarge* inputs) [5]. We refer to the benchmarks from these suites using the `O-*`, `N-*` and `P-*` notations, respectively. Only the parallel Region of Interest (ROI) of each application is included in our measurements; fast-forwarding (with functional modeling of caches and branch predictors enabled) was used to skip over the (sequential) initialization and cleanup phases. The passive OpenMP wait policy was used for thread synchronization. All benchmarks were compiled with GCC 4.3 for x86_64 with SSE2 extensions enabled.

B. Implementing Sampled Simulation in Sniper

We implemented multi-threaded sampling as detailed in Section II in Sniper, building on its existing detailed and cache-only simulation modes. Sniper uses the Pin dynamic instrumentation framework [16] as its functional simulation front-end. Pin is instructed to add analysis routines, which send detailed instruction information to Sniper's timing models. By changing which analysis routines are enabled, one can efficiently switch into a functional simulation-only mode which runs at near-native execution speed (by adding no analysis routines), or simulate just caches and branch predictors for functional warming (by instrumenting only memory operations and branch instructions). The latter mode is used in this paper during the fast-forward phases between detailed intervals.

A sampling director, which we added to Sniper, takes the length of the detailed and fast-forward intervals as input. Once the region of interest begins, it starts out in detailed mode and runs the simulation for the required amount of simulated time to complete one detailed interval. Note again that all intervals are expressed in absolute time (seconds), which is required to keep track of simulation modes consistently across threads when they execute instructions at different speeds, or even run at different clock frequencies. When the detailed interval completes, the simulation director computes the non-idle IPC over the preceding interval for each thread based on the instructions it executed and the time it did *not* sleep waiting for synchronization events.

The simulator is then switched into functional warming mode. Only a small amount of instrumentation is needed here (one analysis function per basic block) to be able to keep track of instruction counts; these are used to increment each thread's time (using its current fast-forward IPC). Synchronization events (`pthread_mutex`, futex system calls, etc.) still happen as before, i.e., threads waiting on them do not execute instructions and do not advance time, but inherit the time of the thread which later wakes them up. Once all (non-sleeping) threads have advanced in this way to the end of the fast-forward interval, a new detailed interval starts.

During fast-forwarding, as in detailed mode, barrier synchronization is used periodically to ensure threads make forward progress at roughly the same pace. This is especially important when keeping cache simulation enabled, to make sure the ordering of memory references — and their resulting performance impact due to for instance associativity conflicts among threads sharing a last-level cache — are simulated accurately.

At the end of the simulation, since every thread has kept track of time, the time of the last thread to finish will be equal to the application's total runtime; no further computation or extrapolation on the results is needed. In addition, per-thread idle times can be kept and the application's synchronization overhead or load imbalance can be derived without extra effort.

C. Selecting Sampling Parameters

Sample selection during simulation is done periodically using fixed parameters for the detailed (D) and fast-forward interval lengths (F), both are expressed in seconds. These parameters are determined up-front based on application periodicity obtained from BBV and call structure information.

The methodology described in Section III defines forbidden ranges (D and $D + F$ close to one of the application's periods or its end). We start by converting all periods P_i that were found, and the application length L (instruction count for the longest thread), into a range of cycle counts in which we expect these values to lie using an expected IPC range of $[0.5 \dots 2.0]$. Multiplying this range with the clock frequency of the simulated processor yields a lower $\underline{\bullet}$ and an upper bound $\overline{\bullet}$ expressed in seconds for each of these application characteristics. We then enumerate all possible D and F combinations, and remove those which do not satisfy the following conditions:

$$\begin{aligned} &D > \alpha \cdot \overline{P_i} \quad \vee \quad (D+F)\cdot\beta < \underline{P_i}, \quad \forall P_i \\ &(D+F)\cdot\gamma < \underline{L} \end{aligned} \tag{1}$$

The constants α, β and γ used were $\alpha = 2$ (for *outside* sampling, where the detailed interval is larger than one period: at least two iterations per detailed interval), $\beta = 25$ (for *inside* sampling, where multiple intervals lie inside one application period: at least 25 intervals per iteration) and $\gamma = 10$ (at least 10 intervals in total).

We rank all of the remaining options, maximizing their distance from the closest periodicity or the end of the application, since a maximum distance yields the lowest potential for error as discussed in Section III-D. We then select two points: *predicted fastest* which is the one with the highest ratio of F/D, and *predicted most-accurate* which is the point with the largest minimum distance and a fast-forward interval of at least $F \geq 5 \cdot D$.

Since the potentials for error of *inside* and *outside* sampling are not easily comparable, we select a set of parameters for each of them. For results in Section VI, where only one parameter set is used, we prefer *outside* sampling for those benchmarks where a valid point is available, and use *inside* sampling otherwise. *Outside* sampling is preferred because it increases an individual sample's accuracy by averaging the IPC over a number of application periods.

It can be the case that appropriate sampling parameters cannot be found for a particular combination of benchmark, input size and core count. In these cases, the resulting configuration options would provide either minimal speedup or too high of an expected error. We consider these configurations not able to be sampled with the proposed methodology.

V. Results

In this section, we will evaluate the proposed sampling methodology with respect to fully-detailed (non-sampled) runs. We first review the entire sampling space for a single application. Next, we review the accuracy and performance trade-offs for all applications that have valid sampling parameters and compare the parameter sets selected by the *predicted fastest* and *predicted most-accurate* methods.

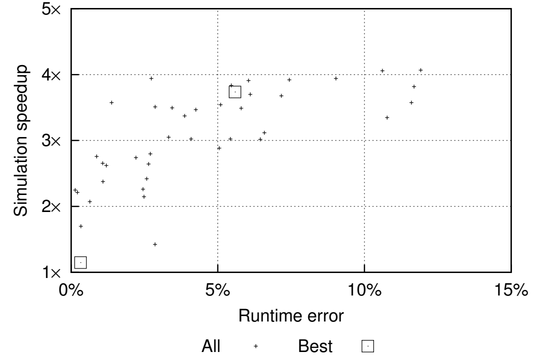


Fig. 9. Simulation speedup versus accuracy for all valid sampling parameters, with those selected by the methodology marked. `O-apsi`, train input set, 8 threads.

A. Sampling Parameter Space

Figure 9 compares the simulated runtime error and simulation speedup, both compared to a fully-detailed simulation of the `O-apsi` application, for a wide range of sampling parameters. The `O-apsi` benchmark was chosen here because it has a large number of sampling opportunities available. In this graph, the methodology selected two points as the *predicted fastest* and *predicted most-accurate* options. The fastest option, as defined above, resulted in a 3.74× speedup with a 5.59% error. The most accurate result was chosen to be conservative and achieved a 1.15× speedup with an error of 0.32%. For this application, our selection comes close to predicting Pareto-optimal results.

B. Predicting Optimal Sampling Parameters

In Figure 10, we detail the results when selecting the best options as predicted by the methodology for both *inside* and *outside* sampling. Note that not all of the benchmarks have valid sampling options either because of their internal periodicities or a short application runtime. The average absolute error for applications with valid sampling periodicities using the *predicted most-accurate* method is just 3.5% with an average speedup of 2.9×. The maximum speedup achieved is 5.8× faster than full-detailed simulation for an 8-core architecture. The *predicted fastest* method achieves a maximum speedup of 8.4× with an average speedup across applications with valid sampling parameters of 3.8× and an average absolute error of 5.1%. With many applications seeing a 10× F/D fast-forwarding ratio, we are therefore simulating just 9.1% of the application in detail (one detailed period followed by 10 fast-forwarding periods). While other single-threaded sampling techniques can achieve a much larger speedup, the speedup in our methodology is limited by two factors: the complexity of the multi-core memory hierarchy models, which is enabled both during fast-forwarding and detailed intervals, and the relatively high speed of our core model. See Section V-E for a detailed discussion on speedup potential.

Table II lists application periodicities found for each benchmark, and the parameters that were used when sampling them for the *predicted most-accurate* case. Two benchmarks, `O-ammp` and `N-lu` were excluded a priori according to the

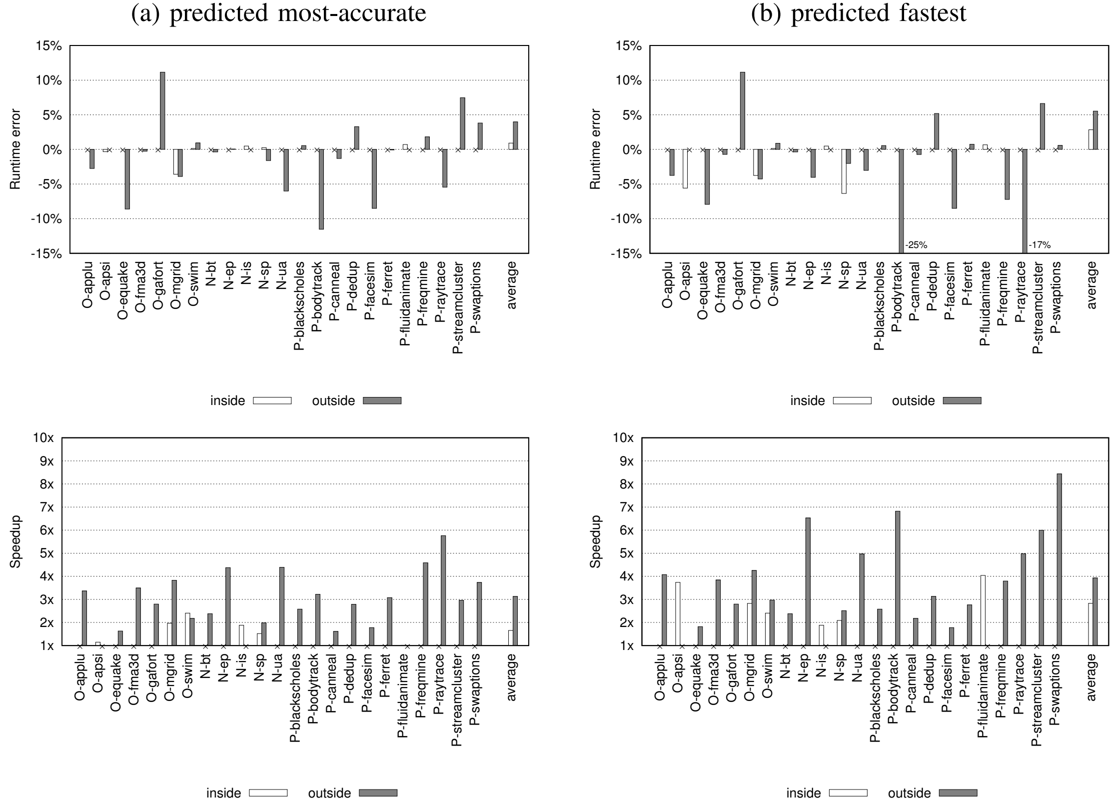


Fig. 10. Overview of sampling accuracy and speedup using the *predicted most-accurate* and *predicted fastest* parameter sets, for both *inside* and *outside* sampling when available.

TABLE II
OVERVIEW OF ALL BENCHMARKS, THEIR PERIODICITIES, THE CHOSEN SAMPLING PARAMETERS AND THEIR SPEED AND ACCURACY.

Application	periodicities (ins)	length (ins)	in/outside	D	F/D	error	speedup	sampled simulation time
O-ammp	*non-periodic behavior*							
O-applu	603k, 104M	38.1B	outside	500 ms	5×	-2.75%	3.37×	21.96 h
O-apsi	676M	48.8B	inside	10 μs	5×	-0.32%	1.15×	69.89 h
O-art	1.40M	110M	*no valid range*					
O-equake	3.66M, 9.00M	17.3B	outside	200 ms	5×	-8.61%	1.63×	19.88 h
O-fma3d	354k, 99.8M	36.2B	outside	500 ms	5×	-0.26%	3.49×	35.27 h
O-gafort	17.0k, 34.3M	10.2B	outside	100 ms	5×	11.15%	2.80×	4.85 h
O-galgel	3.36M, 5.60M, 548M	64.4B	*no valid range*					
O-mgrid	60.5M	61.8B	outside	500 ms	10×	-3.90%	3.83×	41.28 h
O-swim	26.4M	21.8B	outside	200 ms	5×	0.94%	2.18×	72.19 h
N-bt	140k, 180M, 241M	52.7B	outside	500 ms	5×	-0.36%	2.38×	26.81 h
N-cg	2.20M, 56.8M	860M	*no valid range*					
N-ep	420k, 14.2M	7.04B	outside	100 ms	5×	0.06%	4.37×	2.13 h
N-ft	550k, 1.14M, 449M	3.11B	*no valid range*					
N-is	25.0M	333M	inside	10 μs	5×	0.49%	1.88×	0.33 h
N-lu	*non-periodic behavior*							
N-mg	95.0k, 146M, 292M	1.26B	*no valid range*					
N-sp	60.4M	27.0B	outside	200 ms	10×	-1.61%	1.98×	27.91 h
N-ua	1.89M	30.0B	outside	200 ms	10×	-6.02%	4.39×	11.13 h
P-blackscholes	4.08M, 4.56M, 5.60M, 6.36M	712M	outside	10 ms	5×	0.56%	2.58×	0.30 h
P-bodytrack	138k	2.74B	outside	20 ms	10×	-11.52%	3.22×	0.70 h
P-canneal	200k	250M	outside	2 ms	10×	-1.31%	1.61×	0.38 h
P-dedup	—	17.5B	outside	200 ms	5×	3.29%	2.79×	4.10 h
P-facesim	6.36M, 19.2M, 32.0M	3.44B	outside	50 ms	5×	-8.52%	1.78×	2.75 h
P-ferret	13.2M	12.7B	outside	100 ms	10×	-0.07%	3.08×	2.66 h
P-fluidanimate	584M	3.03B	inside	10 μs	5×	0.71%	1.02×	2.40 h
P-freqmine	—	6.01B	outside	50 ms	10×	1.83%	4.59×	1.52 h
P-raytrace	50.0k	1.22B	outside	10 ms	10×	-5.45%	5.76×	0.23 h
P-streamcluster	—	2.93B	outside	20 ms	10×	7.47%	2.96×	1.00 h
P-swaptions	200k	2.47B	outside	20 ms	10×	3.82%	3.73×	0.93 h

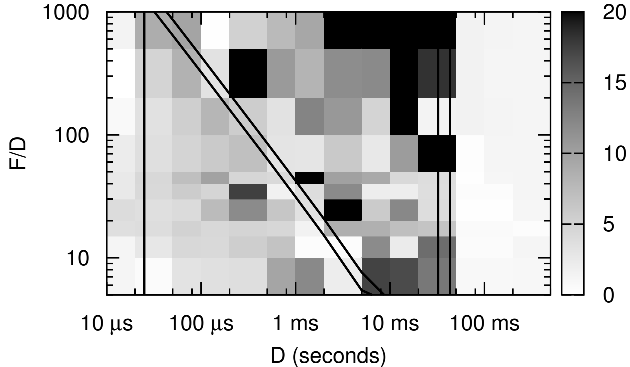


Fig. 11. Sampling error versus application periodicity for `N-bt`, class A input set with 8 threads, and random placement of the detailed interval within each D+F region.

analysis made in Section III-C, while for five more benchmarks their periodicities were such that no valid sampling parameters could be found that satisfied the constraints of Equation 1.

C. Random Sampling

When sampling periodic signals, random sampling is often used to avoid aliasing. The underlying idea is that each sampling interval covers just part of the period, but collectively, the average of all intervals approaches the average of the signal. In our methodology, however, we require each single detailed interval to be representative for the current IPC as this IPC is used during the subsequent fast-forwarding phase. When the application synchronizes, it is the slowest thread which determines progress — application runtime is therefore not determined by the sum of all intervals (allowing high and low estimates to cancel each other out), but has a more complex relationship in which at several points the maximum value of a set of intervals determines progress. The central limit theorem is therefore not applicable, as discussed in Section III-A.

Figure 11 revisits the experiment of Figure 8, but implements random sampling. The execution is divided into intervals of size $D + F$, the detailed intervals (again of size D) are placed at a random position within this interval. This randomizes the sampling period, while making sure that no fast-forward interval becomes larger than $2F$ (larger effective F lengths would extrapolate the detailed IPC for too long, causing additional error). Although some regions in the (D, F) design space have become slightly more accurate when compared to periodical sampling in Figure 8, accuracy is still largely dependent on the relation between the sampling parameters and application periodicity. Random sampling can therefore be used as an extra component to increase overall accuracy, but it does not relieve one from knowing about application periodicities and designing the sampling parameters accordingly.

D. Detailed Warmup

In the SMARTS methodology [26], in addition to continuous functional warming of caches and branch predictors, a detailed core warmup step was required to minimize the cold-start bias of the core model. Following their analysis, the maximum life-time of stale state inside the core could be computed as the product of store-buffer depth, memory latency in cycles, and the maximum IPC. For our configuration, this upper bound is 25,600 ($32 \times 200 \times 4$) instructions. In our methodology for sampling multi-threaded applications, however, detailed intervals much longer than SMART's 1,000 instructions are favored. This makes the detailed region (very) long in comparison to the potential cold-start effects, negating the need for a separate detailed warming phase. Simulation results confirmed this: even for scenarios with a 10 µs detailed period, the shortest considered, adding a detailed warmup period of as much as 10 µs (approximately 10k-100k instructions) did not cause a change in run-time predictions beyond the expected run-time variability.

E. Potential for Simulator Speedup

Simulation results presented in this study use Sniper's *interval* core model, which is significantly faster to simulate than detailed core models found in other academic simulators [11]. In addition, its memory model is relatively complex as it supports shared caches in a parallelized simulation platform. This makes the ratio of execution speed in detailed mode versus that of functional warmup rather low, around 5–10$\times$ depending on the application. In SMARTS, this ratio was much higher (around 50–100$\times$), due to its complex (8- and 16-way issue) core models and a simpler, single-core memory hierarchy. This ratio directly affects the potential speedup that can be obtained from sampled simulation: as the fraction of the application simulated in detail is reduced, the simulation speed asymptotically reaches that of functional warming. Any additional gains in simulation speed will have to be made by relaxing the continuous functional warmup requirement, which is an open research problem [4].

VI. Application: Architectural Exploration

In architectural exploration, a simulator needs to have high fidelity with respect to architectural changes, whereas absolute accuracy against any given architecture is less important. Figure 12 shows the results of an experiment where we compare our baseline, 8-core architecture with two 16-core architectures: one is a straightforward doubling of cores and cache sizes (*16-full*), whereas the alternative architecture keeps cache sizes constant but splits each core into two dual-issue out-of-order cores (*16-half*). For those applications in the NPB benchmark suite that had valid sampling parameters, we simulated the class A and B input sets on all architectures using both full-detailed and sampled simulation and plot the simulated speedup achieved over the baseline architecture.

Comparing the full-detailed (top) with the sampled (bottom) graphs, it is clear that our sampling methodology has a good fidelity with respect to architectural changes, as it preserves the performance differences between the architectures for all benchmarks and input sets shown.

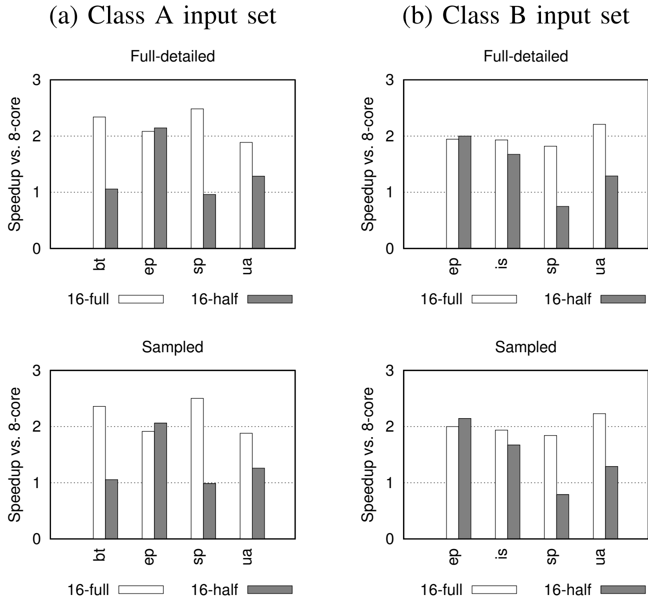


Fig. 12. Results of the architectural exploration study: speedup over the baseline architecture for all benchmarks, A (top) and B (bottom) input sets, comparing full-detailed (left) with sampled (right) simulation.

On the other hand, attempting to speed up simulation by using smaller input sets does not have the same fidelity. For instance, using the A input set, the `N-sp` benchmark achieves a superlinear speedup of around 2.5× when going from the baseline 8-core architecture to the 16-full architecture — whereas the `N-ua` benchmark sees a normal 2× speedup. However, on the larger class B input set, this trend is reversed: here `N-ua` has a (slightly) superlinear speedup whereas `N-sp` achieves less than linear scaling. Clearly, reducing input set size is not an accurate method when doing architecture exploration studies. However, running the larger class B input sets in full-detailed mode takes several weeks, whereas our sampling methodology can bring down this simulation time by a factor of 2.6× while still allowing the correct conclusions to be made.

VII. Related Work

Below we discuss prior work that is most closely related to this work.

A. Single-Threaded Sampling

The SMARTS [26] methodology constructs a sample consisting of a large number of intervals (10,000) of a relatively small number of instructions (1000) per detailed region. They are able to estimate IPC very well because with large numbers of intervals per sample, the average IPC error for the application as a whole decreases — even when each individual interval may suffer from aliasing and is therefore by itself not reliable (the central limit theorem applies here). In our sampling methodology, we require each detailed region to be an accurate representative of its $D+F$ time slice, and extrapolate run-time from it. Because of thread synchronization, we cannot rely on averaging to counter aliasing and the central limit theorem is not applicable, see Section III-A.

SimPoint [20] clusters large intervals, on the order of 100M instructions, using BBVs to represent common chunks of an application in a microarchitecture-independent way. Although SimPoint allows one to accurately predict overall application IPC, it does not allow for the prediction of multi-threaded application run-time, nor does it take application synchronization into account during fast-forwarding.

COTSon [2] uses dynamic sampling to speed up simulation. It uses feedback from the JIT engine of the SimNow simulator to react to changes in the executed code and to switch out of fast-forwarding back into detailed simulation when necessary. Their implementation was used and evaluated on single-threaded applications, and is similar to the SimPoint methodology with respect to running long intervals (100 million instructions) without functional-warming during fast-forward phases.

B. Multi-threaded Sampling

Ekman et al. [10], propose matched-pair comparison as a way to reduce the number of simulation points required to gain an accurate understanding of multi-threaded workloads. Matched-pair comparison relies on the assumption that threads are independent and not synchronized to be able to reduce the sample size. Ekman et al. showed that for their methodology, synchronized applications, such as those in the Splash-2 suite, do not see a significant sample size reduction.

Wenisch et al. [25] propose Flex Points as a way to increase simulator performance for multi-threaded commercial workloads. Van Biesbrouck et al. [24] 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, and therefore over time, a sample will contain a number of thread interleavings that can act as a representation for the overall system execution. Explicit thread synchronization through OS or architected instructions prevents these thread interleavings from occurring, which violates their assumptions. Additional details are discussed in Section II-C.

Perelman et al. [18] cluster multi-threaded applications by looking at each thread in isolation. They predict the IPC and cache hit rates of clustered application phases, but do not evaluate runtime error.

Ryckbosch et al. [19] use interval simulation as a core model in the COTSon [2] simulator and compare their sampled simulation results directly to hardware. COTSon's sampling mechanism throttles functional simulation during fast-forwarding to ensure that relative thread progress, at a ratio corresponding to each thread's IPC, is observed during detailed simulation. The timing of synchronization events is considered to be part of the fast-forward IPC and is not considered independently during fast-forwarding. In Figure 2 we show that not taking into account the detailed interactions between threads during fast-forwarding can lead to significant runtime estimation errors. Our proposed implementation provides for thread-to-thread synchronization and shared cache hierarchy interactions that should be represented to obtain accurate runtime results. Additionally, COTSon does not perform functional-warming of caches, meaning that detailed NUMA behavior, for example,

or other interactions through shared caches will be lost during fast-forwarding.

VIII. Conclusions

Previous sampling work has primarily focused on either single-threaded, IPC-based runtime prediction methods or multi-threaded workloads where per-thread behavior is uncorrelated. Synchronizing multi-threaded applications, where threads affect each other's behavior directly, pose a challenge when it comes to accurately predicting runtime as the traditional sampling methods do not apply to these workloads.

To address these limitations we propose a general-purpose multi-threaded application sampling methodology. We show that taking into account application synchronization during fast-forwarding while determining progress in a per-thread manner significantly improves the prediction of application run-time. In addition to synchronization, application periodicity needs to be taken into account to prevent detailed sampling intervals from aliasing with the application's periodic behavior, affecting both the fast-forwarding IPC and overall runtime prediction. Through the use of micro-architecture independent methods of detecting periodicity we derive sampling parameters up-front to allow for accurate run-time prediction.

Using our sampling method inside Sniper, we are able to achieve a maximum speedup of 5.8× and an average speedup of 2.9× when simulating parallel applications running on 8-core processors while being able to predict application runtime with an average absolute error of 3.5%.

IX. Acknowledgments

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). Experiments were run on computing infrastructure at the ExaScience Lab, Leuven, Belgium; the Intel HPC Lab, Swindon, UK; and the VSC Flemish Supercomputer Center. 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.

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. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, Jan. 2009.

[3] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady, "SPEComp: A new benchmark suite for measuring parallel computer performance," in *OpenMP Shared Memory Parallel Programming*, R. Eigenmann and M. Voss, Eds., Jul. 2001, vol. 2104, pp. 1–10.

[4] 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.

[5] 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.

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

[7] M. Casas, H. Servat, R. M. Badia, and J. Labarta, "Extracting the optimal sampling frequency of applications using spectral analysis," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 3, pp. 237–259, Mar. 2011.

[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] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep./Aug. 2003, pp. 220–231.

[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] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2010, pp. 307–318.

[12] T. Huffmire and T. Sherwood, "Wavelet-based phase classification," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2006, pp. 95–104.

[13] 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.

[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] T. Li, A. Lebeck, and D. Sorin, "Spin detection hardware for improved management of multithreaded systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 6, pp. 508–521, Jun. 2006.

[16] 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 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[17] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Apr. 2010, pp. 2–11.

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

[19] F. Ryckbosch, S. Polfliet, and L. Eeckhout, "Fast, accurate, and validated full-system software simulation of x86 hardware," *Micro, IEEE*, vol. 30, no. 6, pp. 46–56, Nov./Dec. 2010.

[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] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, Jun. 1972.

[22] 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.

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

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

[25] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe, "SimFlex: Statistical sampling of computer system simulation," *Micro, IEEE*, 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.