DAPs: Dynamic Adjustment and Partial Sampling for Multithreaded/Multicore Simulation

Chien-Chih Chen[1], Yin-Chi Peng[1], Cheng-Fen Chen[1], Wei-Shan Wu[1], Qinghao Min[2], Pen-Chung Yew[3], Weihua Zhang[2], Tien-Fu Chen[1]

[1]Dept. of CS, National Chiao Tung University, Taiwan, R.O.C.
{ccchen99, pengyc, chengfen, wswu, tfchen}@cs.nctu.edu.tw

[2]Parallel Processing Institute, Fudan University
{minqh, zhangweihua}@fudan.edu.cn

[3]Department of Computer Science, University of Minnesota, USA
yew@cs.umn.edu

ABSTRACT

Faced with increasingly large multicore chip designs, architects need fast and accurate simulations for their exploration of design spaces within a limited simulation time budget. In multithreaded applications, threads cannot run simultaneously. Sampling is commonly used to reduce simulation time, but conventional sampling barely detects the instantaneous program variations of synchronization events and the inconsistency between phases of each core. This work proposes a dynamic adjustment and partial sampling technique (DAPs), consisting of aggressive sampling, lazy sampling, and regular sampling, to overcome thread interference in multithreaded applications. Moreover, DAPs partially selects sampling cores to reduce the overhead of sampling inconsistent phases.

Categories and Subject Descriptors

B.2.2 **[Performance Analysis and Design Aids]**: *Simulation*

General Terms

Design, Measurement, Performance

Keywords

Dynamic adjustment and partial sampling simulation, Multithreaded/Multicore simulation.

1. INTRODUCTION

Faced with increasing gate counts and on-chip cache sizes in contemporary processors, architectural simulation has become important for exploring the design space of future system architectures and novel research ideas. This has raised the new challenge of simulation time; for example, as the workload's input size must grow to fill the cache, so too does the simulation time and it may take hours or even days to simulate a complete workload.

A modern architectural simulator is generally composed of two models: a *functional model* (FM) and a *timing model* (TM). The FM emulates the behavior of a target system and is only concerned with functional correctness, whereas the TM models the operation latency of micro-architectures that depend on the micro-operation generated by the FM. Along with accurate measurement, simulation speed has become a critical problem because of the ever-increasing complexity of systems. In fact, in an initial cycle of design space explorations, developers do not need a high-accuracy TM because of the very long simulation time, and, in light of this, many researchers have proposed a variety of methodologies to mitigate the simulation speed problem. Sampling is a well-known effective technique that can speed up the simulation and accurately predict performance, the key idea of sampling being that the simulator only runs the TM on certain selected sections in the benchmark's execution stream and uses the FM on the other sections, known as "fast-forwarding".

For single-threaded sampling, the SMARTS methodology [6] applies the statistical sampling length to the simulation. It uses the fixed instructions count to determine the sample lengths, whereas in SimFlex [7] a fixed-cycle-based sampling with more stable results is proposed. Another well-known sampling methodology is SimPoint [8] [9]. As with SMARTS, SimPoint executes sampling based on Basic Block Vectors (BBV). SimPoint needs an off-line full profile of the workload and the pre-processing must be redone if the workload's inputs change.

These studies have all sped up the simulation by using static sampling. They assume that an application consists of many repeatable phases; however, they do not take the feedback from the TM into account and so the sampling simulation may distort the result. Dynamic sampling [2] overcomes this drawback by monitoring internal statistics in fast-forwarded time and then determining when to change from the functional to the timing model. Argollo *et al.* [1] have developed COTSon for single-threaded applications based on dynamic sampling.

Although the above works [1], [2], [6]-[9] are all able to speed up the simulation, for multithreaded applications they may lead to significant estimation errors because they do not take into account the interactions between threads in fast-forwarded time. In addition, researchers have found that IPC is not a good metric for multithreaded applications. To obtain accurate simulation results, ESESC [12] have presented a Time-Based Sampling framework that can simulate any application type. Carlson *et al.* [11] observed that threads affect each other's behavior and that conventional sampling does not apply to these multithreaded workloads. Therefore, their methodology tracked the synchronization event during fast-forwarding, and used application phase behavior to guide reliable sampling parameter selection.

Nevertheless, the works to date have ignored thread variations in a multithreaded application and variations in performance resulting from inter-core communication and shared resource

DAC '14, June 01 - 05 2014, San Francisco, CA, USA

http://dx.doi.org/10.1145/2593069.2593116

competition. They usually use a fixed sampling length (cycles) in the timing model for different programs. Additionally, some multithreaded applications have many synchronization primitives that lead to a divergence between rates of progress of the threads, and using fixed sampling lengths in the application can distort the runtime simulation result. As such, not taking the interference of threads into account can lead to high simulation errors.

Previous works for multithreaded applications or multicore architectures generally employ simultaneous sampling for all cores. This means that there is a global control to switch between the fast-forwarding and the timing period. However, the synchronization event of each core may not actually occur at the same time. To overcome these issues, we propose a Dynamic Adjustment and Partial sampling (DAPs) mechanism that dynamically adjusts not only the sampling length but also sampling frequency according to runtime multithreaded application behavior. According to the sampling length and frequency, we classify sampling into three categories: *regular*, *aggressive*, and *lazy sampling*. Regular sampling is periodic sampling and aggressive sampling has a higher sampling frequency, while lazy sampling has both a lower sampling length and a lower sampling frequency.

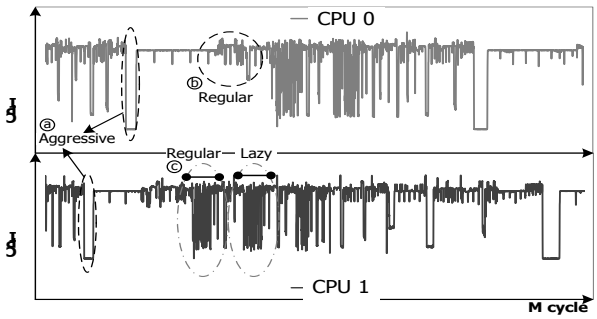

Figure 1. Multithreaded Application periodicity.

Figure 1 demonstrates the progress of a multithreaded application with respect to cycle time. Figure 1(a) indicates the phase of the program that faces a rapid change in the IPC because synchronization events occur in the phase. DAPs will detect the phase and change to aggressive sampling for more accurate result. There are various phases in a program during the execution. As Figure 1(b) shows, DAPs detects a new phase, and then chooses regular sampling for detailed simulation. In Figure 1(c), it shows an opportunity for speeding up the simulation. By a footprints table of DAPs' locality-phase detection, it can determine whether a phase has been executed in the past. DAPs will use lazy sampling for the phase if it has been executed before. Although lazy sampling introduces result that is more inaccurate, it decreases the simulation time by fewer sampling lengths and frequencies. Moreover, by the past IPC of the repeated phase, DAPs can decrease the error rate caused by lazy sampling. Figure 1 shows the overlap of two threads. Synchronization events of these threads do not happen simultaneously. In previous works, if a core changes to detailed simulation, the others also do detailed simulation. DAPs will spend much time on simultaneously detailed simulation. Therefore, we propose a partial selection mechanism on DAPs in order to mitigate the drawback. In other words, DAPs allows one core to execute simulation with aggressive sampling and the others to execute fast-forwarded simulation at the same time.

Furthermore, there is a focus on synchronization events. If one thread experiences a synchronization event, then the program phase is deemed unstable and the DAPs will change from regular to aggressive sampling for higher accuracy. Although aggressive sampling can result in high accuracy when synchronization events occur, it can also slow down the simulation speed. In order to eliminate this problem, we introduce *partial sampling*, which gives control of sampling of each core to its own sampling switcher instead of them all being globally controlled.

This paper is the first to present a dynamic adjustment and partial sampling methodology that dynamically chooses an appropriate sampling length and frequency for multithreaded programs in a runtime simulation. The challenge in doing so is to detect similar phases by locality-phase detections and synchronization events.

The remainder of this paper is organized as follows: first, we review related works; next, we present the DAPs framework and sampling mechanism; Section 4 presents the experimental setup, benchmarks, and sampling parameters; Section 5 shows the results of the DAPs framework; and Section 6 concludes.

2. BACKGROUND and RELATED WORK

In this section, we briefly describe the target simulator on which we based the implementation of our sampling methodologies. We also give a brief introduction to relevant previous research on sampling techniques.

2.1 Transformer

Transformer [3] is a loosely coupled functional-driven full-system simulation for multicore designs. The FM of Transformer is QEMU [4] , which is a generic and open-source full-system machine emulator and virtual machine. It uses a Dynamic Binary Translation (DBT) for translating the target instructions into the host instructions. Moreover, QEMU supports many different ISAs without modification by means of a Tiny Code Generator (TCG). It translates target machine code into common intermediate code (IR) and uses TCG operation to convert the Intermediate Representation (IR) into host code. Transformer uses QEMU's DBT to obtain execution information such as instructions and data access information and then passes these to the TM via shared buffers. The TM can then carry out detailed pipeline and cache simulation in accordance with the received execution information.

The above-mentioned TM in Transformer is GEMS [5]. Differing from Transformer, the original GEMS is a timing-first simulation that needs to wait for the TM to trigger the FM, Simics. Transformer uses QEMU to replace Simics and changes the architecture to functional-driven simulation. Guaranteeing cycle accuracy, Transformer adds a divergence detection mechanism to handle branch "mis-prediction", interrupt or exception handling, shared data access order, and shared page access order. According to the correct path of QEMU, Transformer can detect program divergence. In order to ensure shared data access order, the Memory Access Table (MAT) records the shared data access order in the FM, and the TM checks whether the order is correct.

2.2 Sampling Methodologies

The goal of a sampled simulation is to simulate a number of sampling units instead of the full program stream. For a multithreaded application, the simulation must take into account the interference between and variation within each thread. For this reason, previous works used Time-Based Sampling (TBS) rather than Instruction-Based Sampling (IBS).

ESESC [12] is one sampling methodology that uses time-based sampling. Because there is no timing information in the FM, ESESC uses IPC prediction to decide when to change the FM to the TM. ESESC evaluates three different prediction methods. The first one is the *Naive* method. It assumes that IPC equals one in

the FM. The second is the *Last* method. The Last method uses the last estimated IPC in the detailed stage to predict the next IPC. The third is the *Weighted Moving Average*, which uses a weighted average of the last three to five samples to predict the next IPC.

Carlson *et al.* [11] proposed a multithreaded-application sampling methodology based on Sniper [10]. They showed that not only per-thread IPC but also inter-thread interactions can efficiently increase the simulation accuracy. For this reason, the methodology also simulates synchronization events in the fast-forwarded period. Moreover, they found optimized periodicity in multithreaded programs. It is impossible for various applications to have the same length of periodicity, and even in the same application the length of periodicity changes. Thus, they used the concept of BBV to determine the best periodicity of the program and derive the optimal sampling parameters. The disadvantage of this methodology is that it needs an off-line profile before running an application, and it is also unsuitable for programs with non-periodic behavior.

3. THE DAPs FRAMEWORK

This section presents a framework for the Dynamic Adjustment and Partial sampling (DAPs) simulation. DAPs applies dynamically adjusting sampling methodologies according to the detected properties of the phases to enhance simulation-based performance estimations. We design three sampling mechanisms to reduce the overhead of the timing simulation by skipping the limited intervals.

3.1 Overview

Figure 2 shows the simulation framework based on Transformer [3] and the design challenges in 2(c). DAPs can dynamically change the sampling length and sampling frequency in runtime according to the features of synchronization events in multithreaded applications and instruction cache locality. In DAPs, there are two main additional components to decide, control, and perform sampling methodologies. One is an event detector acquiring specified features such as synchronization instructions, repeated phases, and sampling core selection to trigger suitable sampling methodologies, and the other is a sampling controller, which is used to select the sampling policies and pass the instruction flow to the timing model.

In order to execute a suitable sampling methodology during the various program phases, we propose three components to determine the sampling methodology used. The first component is a synchronization monitor, which captures FM runtime information. If one thread executes a LOCK event in the FM, the synchronization monitor will be triggered and will notify the centralized sampling controller to change its sampling methodology. The second component is locality-phase detection. This considers the relationship between program phase and cache locality. We add an instruction cache to the FM, and, as a result, locality-phase detection can receive information about cache locality. Locality-phase detection can distinguish whether a phase has been executed by virtue of the information received from the instruction cache. The last component is a partial selection. Because we propose multiple sampling methodologies to fit the program phases, this may induce overheads in the TM if we need to sample all cores simultaneously. Partial selection can solve this problem.

As Figure 2 shows, we have provided a centralized sampling controller to manage the sampling parameters and control the timing simulation, executing in either detailed mode or fast-forward mode. In our multiple sampling methodologies mechanism, the sampling controller manages three sampling parameters: sampling length (SL), sampling frequency (SF), and sampling core (SC). SL determines how long the period of the detailed mode is. As mentioned, every program has its own phase and even a single program can have many phases. As such, using the same phase length for different programs is unsuitable, especially for multithreaded applications. In addition, performance metrics captured during the detailed period can represent a whole phase that contains fast-forwarding, warm-up, and detailed period itself. This is important for deciding the length of a detailed period in runtime. The second parameter is SF. SF determines how often the detailed period occurs. SF with high frequency means that the detailed period occurs often; it therefore needs to shorten the fast-forwarding period to shrink the sampling length. On the contrary, a low SF has a low frequency of occurrence of detailed periods. In general, a higher frequency of detailed periods will achieve higher accuracy, but the overhead is also increased. The centralized sampling controller must determine the best balance for accuracy and performance by dynamically adjusting SF. The last parameter is SC. This indicates which core needs to perform the sampling. Because every core has its own phase behavior, the centralized sampling controller needs to control each shared buffer per core. We adopt different sampling methodologies for each core. If all the simulated cores in the TM need to enter the detailed period simultaneously when one of them does, it will lead to an increase in simulation time. Hence, SC is an important consideration. The TM can perform partial cores during the detailed period while the other cores execute in the fast-forward period.

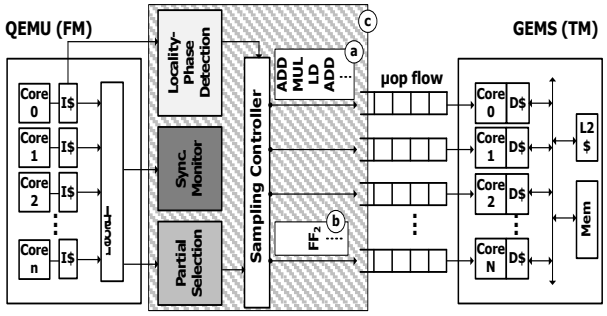

Figure 2. Simulation framework.

Figure 2(a) shows that core 0 runs during the detailed period. In a step-by-step fashion, it simulates all the instructions in the TM. In the fast-forward period, a pseudo instruction is implemented to identify how many instructions can be skipped by the fast-forwarding period, as shown in Figure 2(b).

3.2 Aggressive Sampling Methodology

DAPs applies a new sampling mechanism for multithreaded applications, thus it needs to capture a phenomenon that has high correlation with runtime program behavior for multithreaded applications. Synchronization primitives protect the correction of shared data. Synchronization primitives can be divided into three types: locks, barriers, and conditions. When a synchronization event occurs in a thread, its behavior may be affected. For example, if thread 0 meets a lock event and it does not obtain the lock key because the lock key has been acquired by another, then the thread may be assigned to busy-waiting or sleep by the OS scheduler. At this point, the IPC of thread 0 will be very low or even zero. Hence, when the thread meets the synchronization primitive, the original phase behavior will be changed. DAPs aims to capture this phenomenon during the detailed period. This phenomenon can cause unstable and extremely low performance metrics.

3.2.1 Synchronization Monitor

In order to capture the synchronization phenomenon, we propose a synchronization monitor in DAPs and the synchronization monitor obtains the phenomenon from a tracer added to the FM.

As shown in Figure 3, we add a tracer to the frontend of the TCG, which converts the guest instruction to IR. If the tracer determines that this guest code has a synchronization primitive, it will add a helper function for obtaining trace information. In many ISAs—X86 ISA, for example—LOCK is an instruction prefix. It applies to some atomic instructions that it executes read-modify-write on memory, such as INC and XCHG. LOCK ensures that the core has exclusive ownership of the shared data for the duration of the operation, and provides certain additional ordering guarantees. Consequently, when the tracer detects that the LOCK instructions are being carried out, the tracer will notify the synchronization monitor and the monitor can then trigger the sampling controller to change the sampling methodology to aggressive sampling.

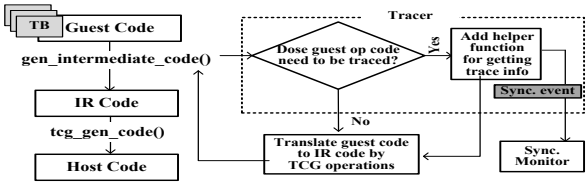

Figure 3. The mechanism of the synchronization monitor.

3.3 Lazy Sampling Methodology

DAPs increases detailed period times because of the addition of aggressive sampling. Even though it avoids incorrect performance estimation, it leads to additional simulation time overhead. Lau *et al.* [14] takes advantage of a program with similar behavior at different execution times to enhance the simulation speed; however, DAPs does not use the idea of the basic block. Finding a detailed and precise phase is unnecessary for DAPs and it simply needs a coarse-grained phase detection to classify and make changes to lazy sampling. In the architecture design, caches work based on the locality of program behavior. For that reason, we have added instruction caches to the FM and proposed runtime locality-phase detection in DAPs, as shown in Figure 2.

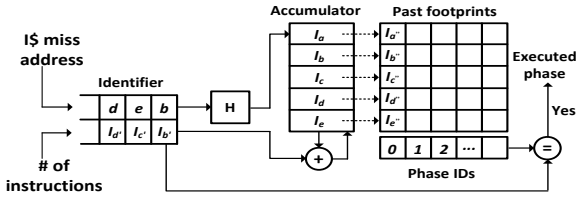

Figure 4. The mechanism of locality-phase detection.

Locality-phase detection uses the features of caches and the locality of program behavior to detect phase. Every core has different rates of progress in multithreaded applications. DAPs adds instruction caches to each core for detecting instruction cache miss streams. As shown in Figure 4, the cache miss address will pass through a hash function and receive an index to map to an accumulator table. The accumulator table records the number of the instruction when each cache miss occurs until a certain interval elapses. Each cache miss stream can generate a vector with miss instruction counts from the accumulator table. After the interval elapses, the locality-phase detection compares the current vector with the vectors that have been executed in a past footprints table. If there is the same vector in the past footprints table, this means that the miss stream has been executed; however, using the strict criterion, a fully matched stream is hard to identify. Because there is variability in the FM, even the number of instructions in the same basic block may be different when it is repeated. To solve this problem, the locality-phase detection uses the concept of Manhattan distance. If the two vectors are similar, then the Manhattan distance of these two vectors will be small. On the contrary, if the two vectors are diverse, then the Manhattan distance will be large. DAPs defines a threshold to decide what size Manhattan distance suffices to regard two vectors as the same. Not only the threshold but also the interval size can affect this decision. The value of the threshold interval will be discussed in Chapter 4.

Fang *et al.* [13] proposed a Multi-Level Phase Analysis (MLPA) that can identify a coarse-grained interval just by the fine-grained intervals at the beginning of its execution. Nevertheless, DAPs does not define the unambiguous coarse-grained interval length. If the vectors have uninterrupted identical vectors in the past footprints table, then locality-phase detection can define these continuous vectors to be a phase. The locality-phase detection will then notify the centralized sampling controller because the same phase has been executed.

3.4 Partial Selection Mechanism

Previous works [10], [12] have carried out detailed periods to obtain simultaneous performance metrics for all cores. DAPs triggers aggressive sampling when one of the cores performs a synchronization event. However, it leads to an increase in the number of detailed periods, because the synchronization events of each core may not occur at the same time. The partial selection mechanism lets each core has its own sampling switcher to reduce the overhead. Figure 5 shows that previous works perform more detailed phases than CPU0 or CPU1 in DAPs because of simultaneously detailed simulation.

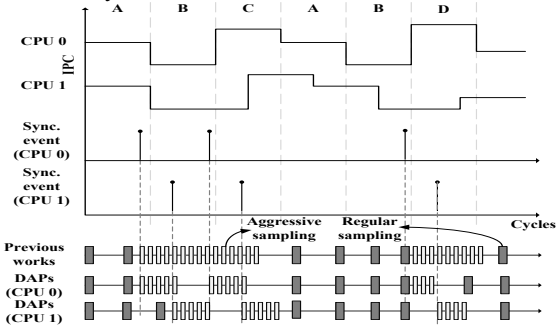

Figure 5. Example of partial selection.

3.5 Sampling Methodology Controller

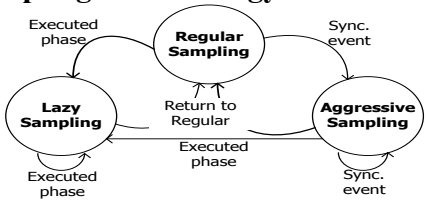

Figure 6. Control flow of sampling methodologies

Depending on the situation, the sampling controller adjusts the sampling parameters according to the appropriate sampling methodology. It uses the last one-fifth of the shortest fast-forwarding length as a window of signal detection. We mark the locality-phase detection for high priority, the synchronization event for medium priority, and the other for low. It detects the first event in the window and bases on the priority to switch the methodology. Figure 6 shows a finite-state machine of the

controller. At the beginning, DAPs performs under the regular sampling methodology with default values for SL, SF, and SC. When the synchronization monitor passes a synchronization event to the controller, regular sampling will change to aggressive sampling with high SF to handle the performance variations. After a number of repetitions of aggressive sampling, it returns to regular sampling if there are no other events. The other statement is lazy sampling. After the sampling controller receives the appropriate signal from the locality-phase detection, the simulator switches to lazy sampling with short SL and low SF. As for SC, the controller will use partial selection to decide which cores need to perform sampling. Figure 6 shows the detailed state transitions.

4. EXPERIMENTS SETUP

In the proposed DAPs experiment, we implement the sampling methodologies in Transformer simulator and run the experiments on Intel® Xeon® Processor E5-2620 with two threads, one is for the FM and another is for the TM. The additional components are in the FM. Because the FM is much faster than TM, the overhead of them will be hidden. Furthermore, we add 3 different execution modes: "FF" only emulates the functional behaviors; "Warm-up" performs detailed timing simulation for warm-up timing models, but statistics are discarded; "Detailed" performs detailed timing simulation to collect statistics. The unit for the sampling modes is cycles (TBS).

Table 1. The experimental environment

Parameter	Value
Num. of cores	1/4/8
Dispatch width	4 micro-operations
Reorder buffer	128 entries
LSQ	64
Branch predictor	YAGS predictor [15]
Cache line size	64 B
Functional units	4 Int-ADD/MUL/ 2 Int-DIV/ 2 LD/ST/ 2 Branch/ 4 FP-ADD/ 2 FP-MUL/DIV/SQRT
I$ **(in FM)**	1KB/2 ways
D$	64KB/2 ways/2 cycles
Unified L2	4MB / 8 ways / shared / 20 cycles
Coherence	MOESI
Mem.	200 cycles
Benchmarks	PARSEC, SPLASH-2

Table 1 shows the target system parameters configured in the TM and the multithreaded benchmarks in our evaluation. This work evaluate with parallel benchmarks, SPLASH-2 and PARSEC. This experiment constructs 1-8 cores with private separate L1 caches and a unified shared L2 cache system as MOESI directory cache-coherency protocol.

In Transformer, the FM is responsible to boot an OS and trigger the TM. We discard the initial parts of the benchmark. After the FM runs into a Region of Interest (ROI), the parallelization part, in the benchmark, the FM triggers the TM to start simulating. At the same time, the sampling mechanism also starts until the finish of ROI and the TM finishes execution, too.

4.1 Selecting Sampling Parameters

In order to characterize DAPs, we first analyze the impact of different parameters for our work. We divide them into two parts, one is the locality-phase detection and another is three parameters in the centralized sampling controller.

4.1.1 Locality-phase Detection

We set the instruction caches in the FM with a small size (1KB) to occur more cache misses for finding more locality-phases. There are two parameters, interval size and threshold, in the locality-phase detection. Without timing information, the locality-phase detection uses the number of instructions as a detection interval. DAPs simulates with regular and lazy methodologies to fix the parameters. Figure 7 illustrates that we can choose 10K instructions as one interval and threshold 100 for the loose locality-phase detection.

Sampling techniques use the metrics in detailed mode to estimate performances. We set the warm-up period length which is two times the length of the detailed period to decide the detailed length. Depending on the IPC coefficient of variation, DAPs can decide the optimum detailed length as shown in Figure 9. The coefficient of variation is a normalized measure of dispersion of a probability distribution or frequency distribution. It can be chosen by low coefficient of variation. So DAPs sets the length of detailed period is 10K cycles and warm-up period is 20K cycles.

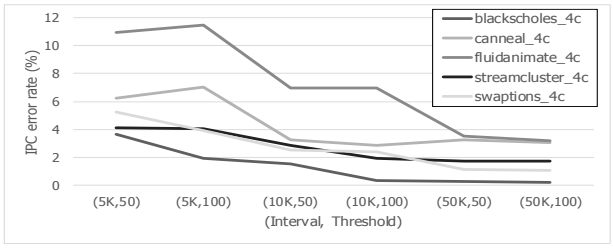

Figure 7. Analysis of error rate with locality-phase detection

4.1.2 FF, Warm-up and Detailed Length

The execution lengths of the three execution modes determine what the sampling frequency of our sampling methodologies. DAPs uses warm-up mode to supply the TM with data of branch predictors, pipeline, and caches, etc. Figure 8 shows the comparison about two kinds of warm-up lengths. It shows that the divergence of these two kinds of warm-up lengths for 4 cores and 8 cores is bigger when the detailed length becomes longer. Accordingly, DAPs uses the two times of the detailed length as the warm-up length.

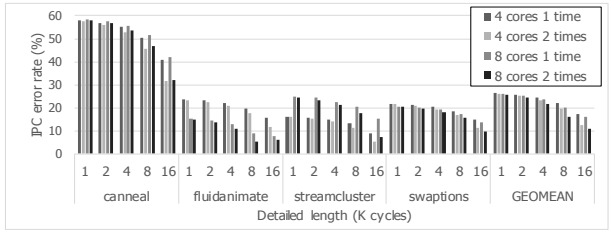

Figure 8. Comparison with warm-up interval length

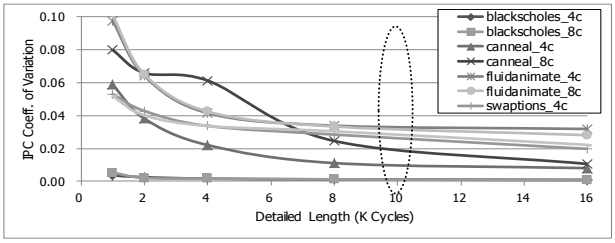

Figure 9. IPC coefficient of variation over detailed length

Sampling frequency allows us to control the level of sampling methodologies. Dynamically adjusting the fast-forwarding length achieves the diverse sampling frequencies. Figure 10 shows that the IPC error rates are tending to be stable and enduring when the sampling frequency is higher than 1/200. Swaptions becomes worse at high frequencies, but it contains few synchronization primitives. Therefore, we adopt the sampling frequencies, from 1/200 to 1/25, for our sampling methodologies.

In conclusion, all of the sampling parameters are shown in Table 2. In order to reduce the sampling overhead, we use the half detailed length for the lazy sampling because it simulates an executed program phase. The repeated program phase simulates with the regular sampling for the first time; therefore, we correct the sampling IPC with last sampled IPC.

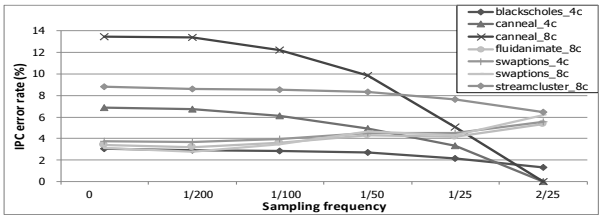

Figure 10. Error rate over sampling frequency.

Table 2. Sampling parameters.

Length(Cycles)	Aggressive sampling	Regular sampling	Lazy sampling
Fast-forwarding	220K	470K	985K
Warm-up	20K	20K	10K
Detailed (SL)	10K	10K	5K

5. RESULTS

All of our results are compared against full-timing run and the TBS only with regular sampling (ESESC without memory warm-up). There are accuracy and speedup results for comparison.

Figure 11 shows the error rate of IPC (normalized by full-timing run) for comparing different configurations (1, 4, and 8 cores). We observe the average error rates of each are 1.8%, 2.3% and 5% respectively. DAPs provides the synchronization monitor to find the synchronization primitive. In fluidanimate and streamcluster, the results show that DAPs reduces 7% error rates compared to TBS because these benchmarks contain many synchronization instructions in runtime. Blackscholes running on 8 cores has a performance drop in full-timing run. It causes that both DAPs and TBS show the high error rates.

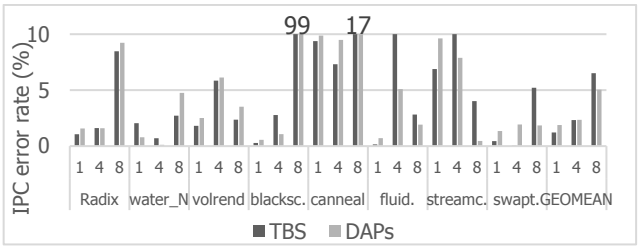

Figure 11. IPC error rate per benchmark.

Comparing the speedup of simulation time, we distribute the simulation time into three sampling methodologies, regular, aggressive, and lazy, as shown in Figure 12. DAPs spends simulation times to control the various sampling parameters due to the multiple sampling methodologies. We mark the lazy sampling for high priority to recover the simulation speed for the aggressive sampling. In fluidanimate, streamcluster, swaptions, DAPs has a gain of simulation speed via lazy sampling. The simulation speed of DAPs is 1.5x~1.6x faster than TBS.

6. CONCLUSION

In this study, we proposed DAPs, a dynamic adjustment and partial sampling simulation framework, to rapidly simulate multithreaded applications and multicore architectures with minimal error rate raise. DAPs is more than 16x and 1.5x speedup compared to Transformer and TBS. DAPs presents aggressive sampling, which is used to ensure the timing accuracy, to detect the instantaneous variations that is caused by the synchronization events. Besides, using a locality-phase detection to identify the phase behavior, controller can apply the lazy sampling. Finally, DAPs uses partial selection to let each core has its own sampling switcher to reduce the overhead.

7. REFERENS

[1] E. Argollo et al., "COTSon: Infrastructure for System-Level Simulation," *Operating Systems Review*, vol. 43, pp. 52-61, 2009.

[2] A. Falcon et al., "Combining Simulation and Virtualization through Dynamic Sampling," in *Proc. ISPASS*, 2007, pp. 72–83.

[3] Z. Fang et al., "Transformer: a functional-driven cycle-accurate multicore simulator," in *Proc. DAC*, 2012, pp. 106–114.

[4] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annual Technical Conference*, 2005, pp. 41–41.

[5] M. M. K. Martin et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 92–99, Nov, 2009.

[6] R. E. Wunderlich, T. F. Wenisch, B. Falsafi et al., "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *Proc. ISCA,* 2003, pp. 84-97.

[7] T. F. Wenisch et al., "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro*, vol.26, pp.18-31, 2006.

[8] T. Sherwood et al., "Automatically characterizing large scale program behavior," in *Proc. ASPLOS*, 2002, pp. 45-57.

[9] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proc. ISCA,* 2003, pp. 336–349.

[10] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. SC*, 2011, pp. 1–12.

[11] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Proc. ISPASS*, 2013, pp. 2–12.

[12] E. K. Ardestani and J. Renau, "ESESC: A fast multicore simulator using Time-Based Sampling," in *Proc. HPCA*, 2013, pp. 448–459.

[13] Z. Fang et al., "Improving dynamic prediction accuracy through multi-level phase analysis," in *Proc. LCTES*, 2012, pp. 89–98.

[14] J. Lau, S. Schoenmackers, and B. Calder, "Transition Phase Classification and Prediction," in *Proc. HPCA*, 2005, pp. 278-289.

[15] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," in *Proc. MICRO*, 1998, pp. 69-77.

[16] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in Proc. AFIPS 1967 Spring Joint Computer Conference, 1967, pp. 483–485.

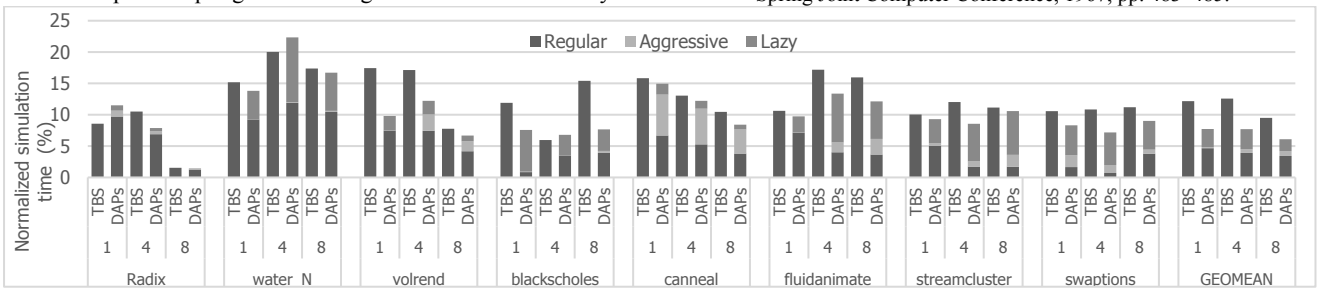

Figure 12. Speedup result per benchmark.