PCantorSim: Accelerating Parallel Architecture Simulation through Fractal-Based Sampling

CHUNTAO JIANG, Huazhong University of Science and Technology
ZHIBIN YU, Shenzhen Institute of Advanced Technology, CAS
HAI JIN, Huazhong University of Science and Technology
CHENGZHONG XU, Shenzhen Institute of Advanced Technology/Wayne State University
LIEVEN EECKHOUT, WIM HEIRMAN, and TREVOR E. CARLSON, Ghent University
XIAOFEI LIAO, Huazhong University of Science and Technology

Computer architects rely heavily on microarchitecture simulation to evaluate design alternatives. Unfortunately, cycle-accurate simulation is extremely slow, being at least 4 to 6 orders of magnitude slower than real hardware. This longstanding problem is further exacerbated in the multi-/many-core era, because single-threaded simulation performance has not improved much, while the design space has expanded substantially. Parallel simulation is a promising approach, yet does not completely solve the simulation challenge. Furthermore, existing sampling techniques, which are widely used for single-threaded applications, do not readily apply to multithreaded applications as thread interaction and synchronization must now be taken into account. This work presents *PCantorSim*, a novel Cantor set (a classic fractal)–based sampling scheme to accelerate parallel simulation of multithreaded applications. Through the use of the proposed methodology, only less than 5% of an application's execution time is simulated in detail. We have implemented our approach in *Sniper* (a parallel multicore simulator) and evaluated it by running the PARSEC benchmarks on a simulated 8-core system. The results show that *PCantorSim* increases simulation speed over detailed parallel simulation by a factor of 20×, on average, with an average absolute execution time prediction error of 5.3%.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Design Studies, Measurement Techniques, Modeling Techniques; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms: Measurement, Algorithms, Performance

Additional Key Words and Phrases: Microarchitecture simulation, parallel simulation, sampled simulation, fractal, Cantor set, performance evaluation

ACM Reference Format:
Jiang, C., Yu, Z., Jin, H., Xu, C., Eeckhout, L., Heirman, W., Carlson, T. E., and Liao, X. 2013. PCantorSim: Accelerating parallel architecture simulation through fractal-based sampling. ACM Trans. Architec. Code Optim. 10, 4, Article 49 (December 2013), 24 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555305

This work is supported by the National High Technology Research and Development Program of China (863 Program) under Grant No. 2012AA010905, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC Grant Agreement No. 259295.
Author's addresses: C. Jiang, H. Jin, and X. Liao, Services Computing Technology and System Laboratory/Cluster and Grid Computing Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China; Z. Yu (corresponding author), C. Xu, Research Center for Cloud Computing, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences; L. Eeckhout, W. Heirman and T. E. Carlson, Department of Electronics and Information Systems, Ghent University, Belgium; email: {yuzhibinh, chuntjiang}@gmail.com.

DOI: http://dx.doi.org/10.1145/2555289.2555305

1. INTRODUCTION

Architects rely heavily on microarchitecture simulation to explore the design space. Unfortunately, single-threaded detailed simulation is at least 4 to 6 orders of magnitude slower compared to real hardware, leading to weeks or months spent simulating even a single industry-standard benchmark (e.g., SPEC CPU2006). In the multi-/many-core era, this problem is further exacerbated because single-threaded performance, and hence sequential simulation speed, does not improve much, while having to simulate many more cores along with various shared resources such as shared caches, on-chip interconnection network, memory controllers, and so on; furthermore, the design space is dramatically enlarged requiring many more simulations to be conducted.

Parallel simulation is a promising approach to accelerate multi-/many-core simulation, but it is challenging to construct parallel simulators with good trade-offs in speed, accuracy, and ease-of-use. *Graphite* [Miller et al. 2010] and *Sniper* [Carlson et al. 2011] are two recently proposed parallel multi-/many-core simulators, and while both simulators report great simulation speeds—for example, *Sniper* achieves a simulation speed of 1 to 2 Million Instructions Per Second (MIPS) on an 8-core host—the rapid progress toward many-core architectures, with hundreds and up to thousands of cores, requires additional innovation and progress to further accelerate architectural simulation [Sanchez and Kozyrakis 2013].

Sampling is a popular and proven technique to accelerate single-threaded application simulation. However, it cannot be readily applied to parallel simulation of multithreaded applications. It is well known that the execution path of multithreaded applications running on multicore processors may vary substantially due to timing variations (nondeterminism) in conjunction with operating system scheduling and synchronization (e.g., lock) effects [Alameldeen and Wood 2003]. This indicates that the number of dynamically executed instructions of the same multithreaded workload may vary significantly from run to run on the same machine. As a result, the prerequisite of Instructions Per Cycle (IPC) as a valid performance metric—the number of instructions of a workload is a constant across different runs or microarchitectures—is broken [Alameldeen and Wood 2006]. However, most previous sampling techniques sample some *instruction* intervals to simulate in detail, calculate the IPC for each sampled interval, and extrapolate the IPCs from these intervals to the overall IPC of a workload. With the inappropriateness of IPC for multithreaded workloads, existing sampling techniques that work well for single-threaded applications naturally become invalid.

To address this issue, Carlson et al. [2013] and Ardestani and Renau [2013] introduce the concept of time-based sampling, where sample selection is guided by *execution time*[1] rather than *instruction count*. However, there still are some limitations in these methodologies. In particular, the periodic sampling approach proposed in Carlson et al. [2013] needs a fairly complicated preprocessing step to determine valid sampling parameters. The goal of this preprocessing step is to identify an application's periodicities (phase behavior or time-varying execution behavior) and determine a sampling regime that aligns well with these periodicities so as to avoid aliasing problems (e.g., a low-IPC region is used to predict the performance of a high-IPC region, or vice versa). Moreover, some applications do not have regular phase behavior, whereas others have

[1]Instead of IPC, *execution time* is the ultimate metric to compare different configurations' relative speed [Alameldeen and Wood 2006]. Throughout this article, we also use the term execution time or running time to refer to the total running time of the simulated workload, while simulation time is used to refer to the time used to simulate an application on a microarchitecture simulator. For example, the simulation time of NPB-*lu* running on an 8-core simulated system with the *class A* input is 49.15 hours (using *Sniper*) and its predicted execution time is 11.27 seconds.

time-varying execution behavior at different time scales, which either excludes them from being amenable to periodic sampling using the previously proposed methods, or incurs high sampling errors.

In this article, we propose a novel sampling approach by leveraging the fractal nature of program execution behavior to accelerate parallel simulation of general multithreaded applications. This methodology employs the generation procedure of the Cantor set—a classical fractal—to determine which parts of an application are simulated in detail. As in the periodic sampling methodology, we perform sample selection on the basis of running time (i.e., execution time) rather than instruction count to maintain compatibility with parallel multithreaded workloads, and we simulate thread interaction through synchronization and shared memory in detail even while fast-forwarding. We have integrated the Cantor set–based sampling scheme into *Sniper*, a parallel multicore simulator, and named it *PCantorSim*. We evaluate *PCantorSim* by running the PARSEC benchmarks and predict their *execution time* on an 8-core simulated processor target. The results show that *PCantorSim* can significantly speed up parallel simulation with high accuracy in a relatively easy manner.

More specifically, we make the following contributions.

—We propose a novel fractal-based sampling scheme to accelerate parallel simulation of multithreaded workloads. We also verify that there exists fractal behavior during the execution of multithreaded applications, and that using fractals to guide sample selection can avoid the aliasing problem present in periodic sampling.
—We have integrated our sampling scheme *PCantorSim* into *Sniper*, a parallel multicore simulator.
—We evaluate *PCantorSim* by running the PARSEC benchmarks on an 8-core simulated system to predict their *execution time*. The results show that our approach can speed up parallel simulation by a factor of 20×, on average, and up to 32×, with an average execution time prediction error of only 5.3% when compared to detailed simulation.

The remainder of this article is organized as follows. Section 2 describes the background. Section 3 presents the fractal-based sampling approach. Section 4 describes the experimental set-up, and Section 5 presents the results and analysis. The related work and conclusions are given in Sections 6 and 7, respectively.

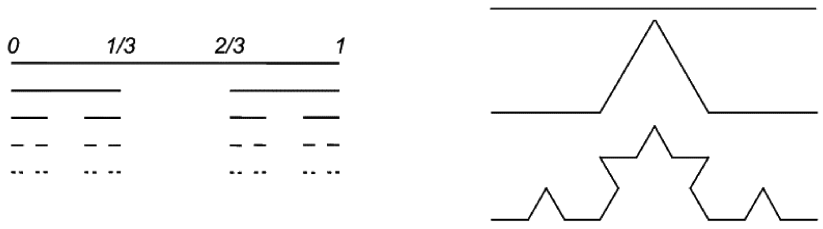
2. BACKGROUND

In this section, we first introduce the different simulation modes used in sampled simulation. Subsequently, we describe fractals and the Cantor set. Finally, we analyze program execution behavior from a fractal perspective.

2.1 Simulation Modes

Sampling is a well-known acceleration technique used in microarchitecture simulation. It reduces simulation time by selecting a small yet representative subset of an application's instruction stream to be simulated in detail. The remainders of the instruction stream are simulated in other, faster simulation modes.

In sampled simulation, the original dynamic instruction stream of a workload is broken up into three kinds of nonoverlapping chunks. Each kind of chunk is simulated in one of three simulation modes: (1) fast-forwarding, (2) warm-up, and (3) detailed simulation mode. The fast-forwarding mode guarantees that the execution of a workload is functionally correct, while no microarchitectural events are recorded during simulation. (Fast-forwarding can be replaced by checkpointing, or by taking a snapshot of the architecture state, i.e., registers and memory.) In detailed simulation, the events pertaining to all microarchitectural structures such as reorder buffers, functional units,



(a) The Cantor Set. (b) The Koch Curve.

Fig. 1. Two classical fractals.

reservation stations, caches, TLBs, prefetchers and branch predictors are simulated in detail. Since the fast-forwarding mode does not maintain the state of microarchitectural structures, these structures such as caches are empty when detailed simulation starts. This is referred to as the so-called "cold-start" problem in microarchitecture simulation. To eliminate the "cold-start" problem, warm-up mode is employed between fast-forwarding and detailed simulation chunks. Unlike detailed simulation, warm-up only considers events related to a few well-selected (usually large) structures such as caches (and TLBs) and branch predictors. This simulation mode is also known as functional warming [Wunderlich et al. 2003].

The simulation speed of the three modes is dramatically different. According to the earlier definition of simulation modes, detailed simulation is the slowest, warm-up is faster, and fast-forwarding is the fastest. For example, in *Sniper*, the simulation speeds are on the order of 1,000MIPS, 10MIPS and 1MIPS for fast-forwarding, warm-up, and detailed simulation, respectively. By simulating the nonsampled parts of an application in the much faster warm-up and/or fast-forwarding modes, we can significantly speed up microarchitecture simulation compared to fully detailed simulation while retaining high accuracy.

2.2 Fractals and the Cantor Set

Our sampling scheme is based on the Cantor set—a classical fractal. The term "fractal" was invented by Benoit Mandelbrot to identify fragmented and irregular shapes [Mandelbrot 1983]. It is defined as follows:

Definition 1. Fractals are shapes made of parts similar to the whole.

In other words, a fractal is an object or quantity that displays self-similarity at all scales, where self-similarity means they are "the same from near and from far" [Gouyet and Mandelbrot 1996]. The object does not necessarily exhibit exactly the same structure at all scales, but the same "type" of structure should appear. For example, Figure 1 illustrates two famous fractals: the Cantor set and the Koch curve. They are nearly the same at different scales.

The Cantor set is a famous set, and an exact fractal, which was first constructed by Georg Cantor in 1883 [Peitgen et al. 2004]. It can be best characterized by describing its generation. It is typically created by repeatedly deleting the open middle thirds of a set of line segments. As illustrated in Figure 1(a), starting with the interval [0, 1], the first step removes the open middle third (1/3, 2/3), leaving two line segments: [0, 1/3] ∪ [2/3, 1]. Next, the open middle thirds of these remaining segments are removed, leaving four line segments. This process is continued ad infinitum. At every stage of the process, the result is self-similar to the previous stage, i.e., identical upon rescaling. After the n*th* step, there will be 2^n segments of length $1/3^n$. In this work, we employ a

(slightly) modified Cantor set generation rule in our sampling scheme, as described in Section 3.

2.3 Fractal Behavior in Programs

Fractals have and are being successfully used in the research of program behavior (e.g., cache behavior prediction). Voldman et al. [1983] studied the clustering of cache misses. Based on the sound conjecture that cache misses occur in hierarchical bursts in the same way, the authors suggested that the interaction of software and caches can be modeled as fractals. This line of research is continued by Thiébaut [1989], in which the same theoretical framework is applied to the prediction of the miss ratio of programs in fully associative caches. He et al. [2012] proposed a fractal model, *FractalMRC*, and confirmed that it can accurately capture the shared cache behavior of applications concurrently running on commercial multicore processors.

In our work, we use another program performance metric—IPC—as an example to show the existence of fractals in multithreaded program behavior and demonstrate that IPC can be accurately predicted by using fractal techniques. More specifically, we observe that the IPC variations of many multithreaded applications exhibit fractal or self-similarity behavior as a function of time. We then employ the classical fractal Cantor set and its generation rule to select simulation samples.

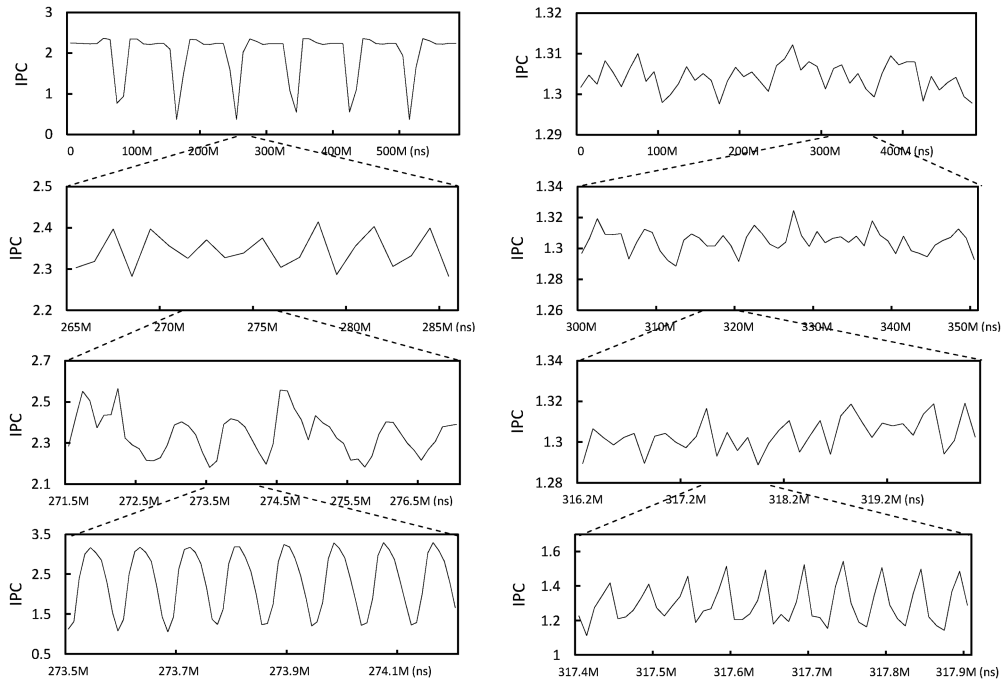
3. CANTOR SAMPLING FOR PARALLEL SIMULATION

In this section, we first verify that fractal behavior does exist in multithreaded applications when we observe IPC variations over their execution time. Subsequently, we describe how we leverage this fractal behavior to accelerate parallel simulation of multithreaded applications through fractal-based sampling. We then present the parameters for our sampling methodology and how we determine them. Finally, we describe how we address the challenges in sampled parallel simulation.

3.1 Fractals in Multithreaded Application Behavior

Most single-threaded and multithreaded applications exhibit time-varying execution behavior [Sherwood et al. 2001; Lau et al. 2004; Perelman et al. 2006; Carlson et al. 2013]. Previous work in workload characterization and simulation techniques focused on detecting phase behavior and has reported that time-varying execution behavior exists across different time scales. We observe that there exists the same "type" of execution behavior appearing at different time scales. To illustrate this phenomenon, we plot the IPC variation at different time scales for one of the eight threads from two applications in Figure 2. Figure 2(a) shows the *ft* application from the NPB benchmark suite with the *class A* input set. Figure 2(b) shows the *swaptions* application from the PARSEC benchmark suite with the *simlarge* input data set. The top subfigures in Figure 2(a) and Figure 2(b) show the IPC traces for the full application at a granularity of 10M ns. The remaining three subfigures (from top to bottom) in Figures 2(a) and 2(b) show the IPC variations at smaller time scales—1M ns, 100K ns and 10K ns (each one zoomed in 10×). As shown, the IPC variations at the four different time scales are similar to each other. Although they are not exactly the same, the same "type" of execution behavior appears. In other words, the shape of the IPC variations for the smaller time scale is similar to that of the larger one. This so-called "self-similarity" is the key property of fractals and is also the evidence of fractals existing in program behavior of multithreaded applications. To confirm this observation, we performed the same experiment for other applications and observed IPC variation at other time scales (e.g., 8M ns, 500K ns, etc.), and we concluded that the same phenomenon appears.

To further verify the existence of fractal execution behavior in a more rigorous way, we present statistical analysis to substantiate our self-similarity hypothesis by



(a) NPB-*ft*. (b) PARSEC-*swaptions*.

Fig. 2. IPC variations over execution time for two multithreaded applications. (a) NPB-*ft* benchmark (thread 1 out of 8, *class A* input data set). (b) PARSEC-*swaptions* benchmark (thread 1 out of 8, *simlarge* input data set). For both (a) and (b), the top subfigure shows the IPC traces for the full application at a granularity of 10M ns and the remaining three subfigures (from top to bottom) show the IPC variations at smaller time scales—1M ns, 100K ns and 10K ns (each subfigure zoomed in 10×). Self-similarity, the key property of fractal behavior, can be seen across the four time scales.

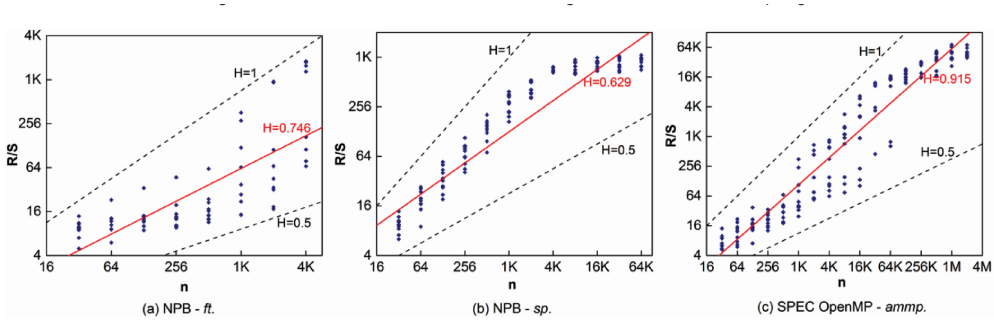


Fig. 3. Find H by performing the Rescaled Range Method. The data sizes of IPC series for the three applications are 59,561, 1,026,725 and 3,669,940, respectively. We plot the $(R/S)_n$ values calculated for multiple subsets of the data (10 subsets for each n, possibly overlapping). Dashed lines showing the slopes corresponding to $H = 1/2$ and $H = 1$ are given for reference.

employing the classical *Rescaled Range Method* [Hurst 1951; Feitelson 2013]. The Rescaled Range Method calculates the Hurst parameter H, which determines whether a series is self-similar. If H is in the range $1/2 < H < 1$, it indicates that the series is self-similar [Hurst 1951; Feitelson 2013]. Figure 3 shows the values of H for time-based

IPC series of three applications: NPB-*ft*, NPB-*sp*, and SPEC OpenMP-*ammp*. The selected three applications have significantly different lengths of execution time. We calculate the IPC values per time unit of 10μs. Thus, the data sizes of IPC series for the three applications are 59,561, 1,026,725 and 3,669,940, respectively. As can be seen, by performing the *Rescaled Range Method*, the values of H for the three representative applications are 0.746, 0.629, and 0.915, respectively, which are all in the range of 1/2 and 1. The Hurst parameters for all other benchmarks also lie between 1/2 and 1 (not shown here due to the space constraints). These results illustrate that the IPC variations over execution time for multithreaded parallel applications are self-similar.

Given the existence of fractal behavior during multithreaded application execution, we try to exploit this phenomenon to select representative samples in our parallel sampling methodology. More specifically, we break up the entire execution time of an application into many small time intervals based on a fractal rule (Cantor Set Generation Rule, CSGR), and select only a very small yet representative fraction according to the fractal property. In the end, the selected samples are clustered and the distance between the sample clusters varies throughout the execution. This is an important property of fractals [Peitgen et al. 2004], which makes fractals amenable to phase behavior analysis. As mentioned before, most applications exhibit time-varying execution behavior. However, the length of the various execution phases is typically dramatically different. This makes periodic sampling difficult to align to phase boundaries [Carlson et al. 2013]. In contrast, because of the clustering of samples, fractal-based sampling can potentially better fit an application's phase behavior, and avoid the aliasing problem present in periodic sampling.

3.2 Parallel Sampling Based on the Rule of Cantor Set Generation

We employ the CSGR to select samples in our sampled parallel simulation of multithreaded applications. Before going into details, we need to determine what the samples are. In sampled microarchitecture simulation, the samples could be instruction intervals or execution time intervals. Most prior sampling methods for single-threaded applications select intervals in terms of dynamically executed instructions. Namely, they determine the size of intervals simulated in detailed or fast-forwarding mode by the number of instructions. However, both Ardestani and Renau [2013] and Carlson et al. [2013] independently proved that such methods are no longer reliable for general multithreaded applications because the number of dynamically executed instructions may vary significantly across different runs or microarchitectures for the same workload. We therefore define our sampling parameters based on time instead.

In our methodology, the total execution time of an application is assumed as a straight line segment. Based on the trisection CSGR, we map the remaining and the removed line segments as the time intervals simulated in detailed and fast-forwarding mode, respectively.

However, it is unsuitable to strictly apply the trisection CSGR to accelerate parallel simulation for two reasons. First, we cannot obtain the exact total execution time of an application before the simulation is completed. Yet, the length of the straight line segment must be known at the very beginning to perform the trisection CSGR. Second, as the middle part of an application is more important than the initial and terminal ones in microarchitecture simulation, it should be simulated in detailed mode. But if we apply the first step of the trisection CSGR strictly, the middle third of an application will be simulated in fast-forwarding mode.

We, therefore, modify the trisection CSGR in our study. Since we do not know the entire execution time of an application upfront, we apply the CSGR in a time

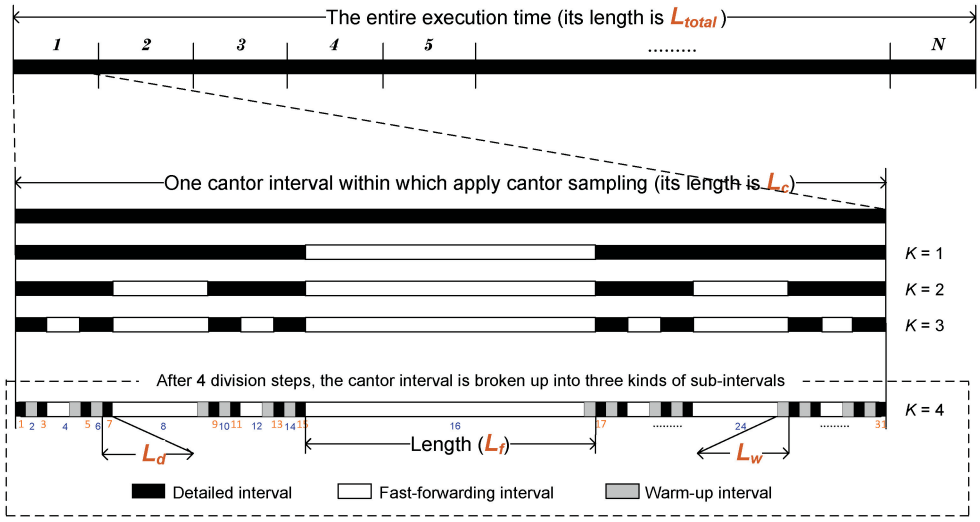


Fig. 4. Cantor sampling methodology. The total execution time of an application is assumed as a straight line segment and is divided into N time intervals (cantor intervals). In each cantor interval, we apply the trisection CSGR. After K division steps performed by the CSGR, the cantor interval is broken up into many subintervals, classified as either detailed or fast-forwarding intervals. Additionally, we enable functional warm-up in front of each detailed interval. Therefore, the warm-up interval is part of fast-forwarding interval. As illustrated, we set "$K = 4$" in the example. After 4 division steps, the black, white and grey bars are detailed, fast-forwarding and warm-up intervals, respectively. We encode the detailed and fast-forwarding intervals using different color numbers. These numbers are used to illustrate our IPC prediction method in Section 3.4.2.

interval with fixed length (e.g., 100ms). We define this time interval as the *cantor interval*. As a result, the entire execution time of an application can be divided into N cantor intervals. In each cantor interval, we apply the CSGR strictly, as shown in Figure 4.

After several division steps performed by the CSGR, each cantor interval is broken up into many subintervals that can be classified into two kinds: detailed intervals that are simulated in detailed mode, versus fast-forwarding intervals that are simulated in fast-forwarding mode. Additionally, we also enable functional warm-up (i.e., warm-up interval) prior to each detailed interval to eliminate the "cold start" issue during sampled simulation [Wunderlich et al. 2003], see also Figure 4.

3.3 Cantor Sampling Parameters

We now introduce the sampling parameters in our Cantor set–based sampling scheme and show how they can be determined. As mentioned in Section 3.2, there are four kinds of time intervals: cantor, detailed, fast-forwarding, and warm-ups. We define their lengths as L_c, L_d, L_f, and L_w, respectively. In fact, the total execution time of an application is finally broken up into three kinds of time intervals: detailed, fast-forwarding, and warm-up. We can thus perform sampling as soon as we know the lengths of each of them—L_d, L_f, and L_w.

We determine the length of the detailed interval (L_d) empirically, through hundreds of experiments. In a first set of trace-driven experiments, instruction and cycle counts are recorded every 10μs (see Section 4.1 for details). This implies that the smallest time interval that we can calculate its IPC for is 10μs. Thus, we start to explore L_d from 10μs. We have tried 41 different values for L_d in a very large range from 10μs to 100ms. Figure 8 illustrates the IPC error variations along with different values of L_d. The IPC

```
1  # F_Interval_list is the list storing the lengths of fast-forwarding intervals.
2  # K is the number of division steps performed by CSGR.
3  # L_d is the length of detailed interval.
4
5  Initialize F_Interval_List = null.
6  Repeat for i = 0, 1, 2, ... , K-1:
7     a) Compute (3^i * L_d) and put the value in the tail of the list.
8        F_interval_list . append (3^i * L_d).
9     b) Double the list, i.e., duplicate the list and joint the original and duplicate one.
10       F_interval_list = F_interval_list * 2.
11    c) Pop the last element in the list.
12       F_interval_list . pop (the last element).
13 End algorithm.
```

(a) The algorithm.

```
Initialization (null)
i = 0
a)  1
b)  1,1
c)  1
i = 1
a)  1,3
b)  1,3,1,3
c)  1,3,1
i = 2
a)  1,3,1,9
b)  1,3,1,9,1,3,1,9
c)  1,3,1,9,1,3,1
End
```

(b) The example.

Fig. 5. (a) The Algorithm that generates L_f values using the input parameters L_d and K. (b) The example to illustrate the algorithm. We use "$L_d = 1$" and "$K = 3$" in the example to show the value of the list after each step. The final generated L_f list is [1,3,1,9,1,3,1].

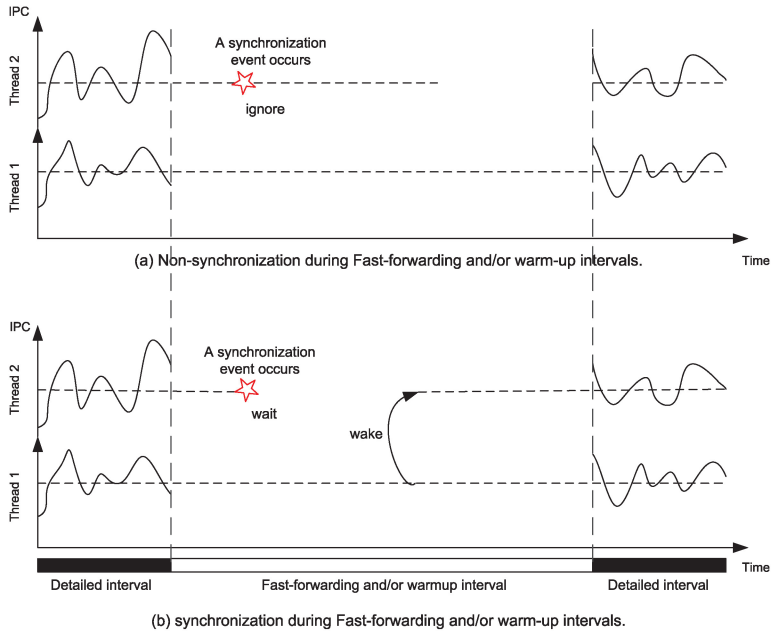
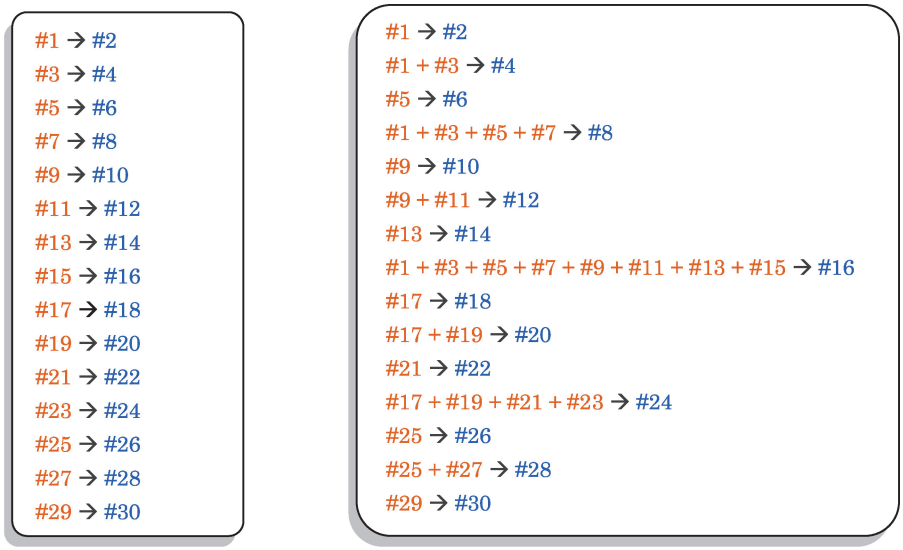


Fig. 6. The differences between nonsynchronization and synchronization during fast-forwarding and/or warm-up intervals.



(a) The *Last* prediction method. (b) The *L-WMA* prediction method.

Fig. 7. IPC prediction methods during fast-forwarding. The numbers are indexes for different kinds of intervals in Figure 4: The even numbers represent detailed intervals, and the odd ones represent fast-forwarding intervals. (a) The *Last* method does the prediction using the IPC of the last detailed interval. (b) The *L-WMA* method uses the *Last* method for shorter fast-forwarding intervals and the simplified *WMA* method for longer fast-forwarding intervals.

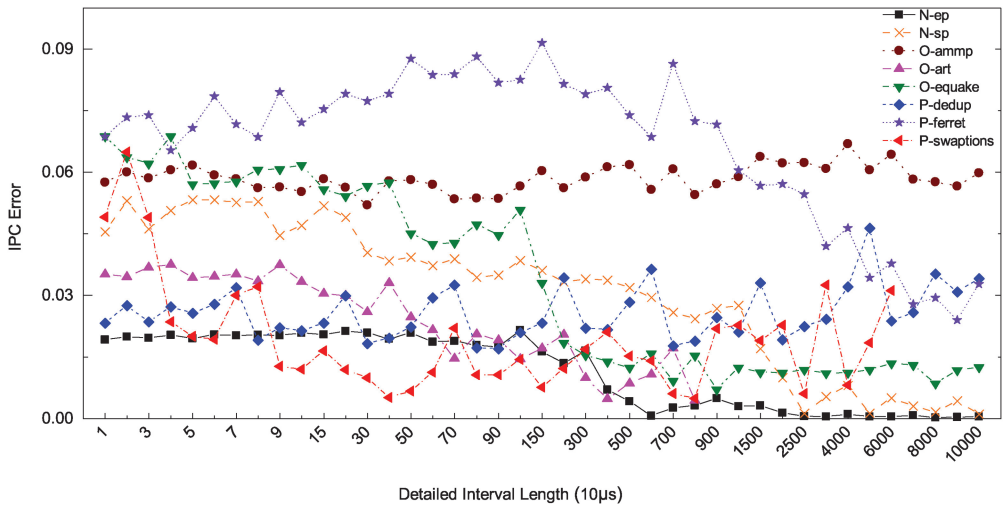


Fig. 8. IPC prediction errors (calculated according to Equation (7)) for eight applications using our cantor sampling method with up to 41 different values of L_d.

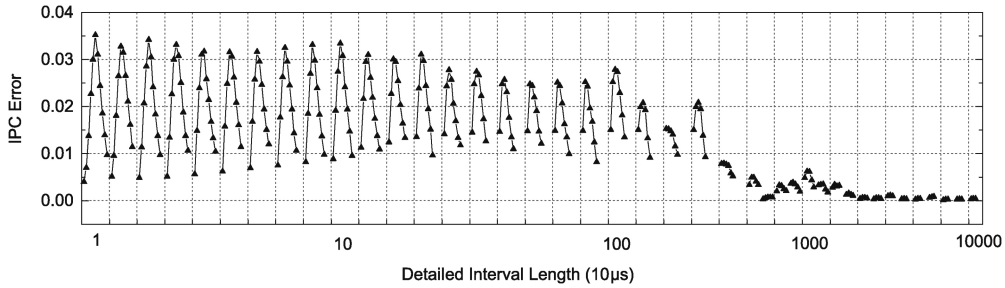


Fig. 9. We plot the IPC errors of different K for the same L_d in one column for the application N-*ep*. Thus, there are 41 columns corresponding to 41 different values of L_d and in each column there are several IPC errors corresponding to different K (from 1 to K_{max}) for the same L_d.

error does not change significantly when we vary the L_d for the same multithreaded program. In addition, smaller values of L_d indicate a smaller fraction of execution time simulated in detail and thus a higher simulation speed-up. We, therefore, choose $L_d = 10\mu$s.

We then determine the length of the fast-forwarding interval (L_f) based on L_d. L_f has a regular relationship with L_d through the strict application of the CSGR within a cantor interval. To model this relationship, we define the parameter K, which is the number of division steps performed by the CSGR. Figure 5(a) shows the algorithm that generates a list of L_f values using the input parameters L_d and K. To illustrate the algorithm more clearly, we use $L_d = 1$ and $K = 3$ as an example to show the value of the L_f list after each step in Figure 5(b).

Now the problem is how to determine the parameter K. As we apply the CSGR in each cantor interval, there is a regular relationship among L_c, L_d, and K, which can be described by Equation (1). In turn, we can determine K using Equation (2), where K is an integer.

$$L_c = L_d \times 3^K \tag{1}$$

$$K = \left\lfloor \frac{\ln(L_c/L_d)}{\ln 3} \right\rfloor \tag{2}$$

Therefore, to determine K, we first need to determine L_c. Obviously, L_c should be less than the total execution time of the application (defined as L_{total}). Although we do not know L_{total} at the start of a simulation, we can get an estimated value of it through instrumentation tools or hardware performance counters [Bhadauria et al. 2009]. To express the relationship between L_c and L_{total}, we introduce the parameter N, which is defined as the number of cantor intervals in the entire execution time. The relationship can be described by Equation (3).

$$L_c = L_{total}/N \tag{3}$$

We have explored N in a large range (from 1 to 100,000) for several different multithreaded applications to find its proper value. The results show that the IPC error does not change much when we vary N for the same multithreaded program (Figure 10). However, we would like to suggest a range for N—10 to 100—for two reasons: (1) Very small N (e.g., 1 or 2) indicates that there are only one or two cantor intervals in the entire execution time. In this case, after the first step of the trisection CSGR, large representative parts of interest of an application's execution would be simulated in fast-forwarding mode and the overall prediction error would be high. Therefore, N should be sufficiently large ($>$10). (2) Very large N (e.g., 10,000) implies

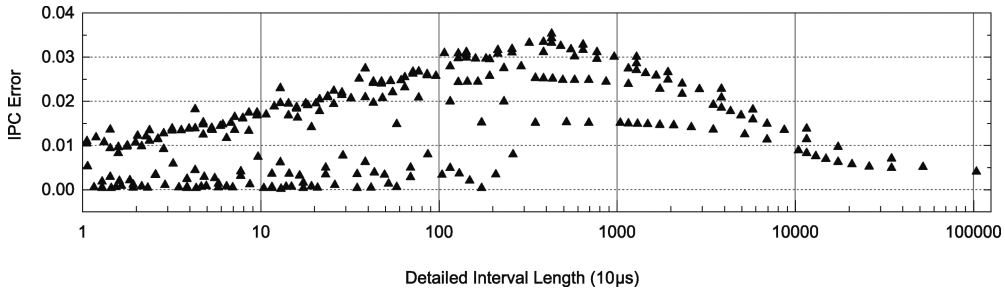


Fig. 10. The IPC error variations along with different values of N for the N-*ep* application.

Table I. Descriptions and Examples for the Parameters in the Fractal-Based Sampling

Parameters	Descriptions	Examples (see Figure 4)
L_{total}	Length of the entire execution time of the application	The entire execution time is assumed as a straight line segment with the length L_{total}.
L_c	Length of each cantor interval	There are N cantor intervals in L_{total}, each of them has the length L_c.
L_d	Length of a detailed interval	After K division steps, detailed intervals all have the same length L_d.
L_f	Length of a fast-forwarding interval	After K division steps, fast-forwarding intervals have different lengths L_f.
L_w	Length of a warm-up interval	*PCantorSim* enables functional warm-up in front of each detailed interval, its length equals L_w.
K	Number of division steps performed by the CSGR	Figure 4 shows the details of "$K = 4$".
N	Number of cantor intervals in L_{total}	There are N cantor intervals in L_{total}.

many cantor intervals in the entire execution time. According to Equations (2) and (3), both L_c and K would then be very small. This can reduce the simulation speed because K determines the fraction of an application simulated in detail. Equation (4) describes the fraction. Therefore, N should be smaller than 100.

$$fraction = \left(\frac{2}{3}\right)^K \tag{4}$$

We determine the length of the warm-up interval (L_w) based on our warm-up strategy. Carlson et al. [2013] did not explore reduced warm-up and enabled functional warming throughout simulation. In our experiments, we find that the prediction error of execution time does not decrease significantly, when more than 10μs (the length of L_d) is taken to warm up caches and branch predictors. This indicates that longer warm-up intervals do not contribute much to error reduction, yet they decrease simulation speed. Warm-up intervals that are as short as possible are therefore preferred. We choose L_w to be the same length as that of a detailed interval (L_d). More details about determining L_w are provided in Section 5.2.3.

In summary, we have determined the values of L_d, L_f and L_w through experimentation or equations. Table I shows the descriptions and examples for all the necessary parameters required to find suitable values of L_d, L_f, and L_w in our Cantor set–based sampling methodology. We will further discuss how we derive suitable parameters at the beginning of Section 5.2.

3.4 Fast-Forward Parallel Simulation during Fast-Forwarding Intervals

Sampling techniques achieve high simulation speed by simulating the vast majority of an application's dynamic instruction stream (about 90% to 99.9%) in fast-forwarding mode. Previous work shows that the mechanism used to fast-forward the long fast-forwarding intervals influences simulation speed and accuracy significantly [Carlson et al. 2013]. In single-threaded sampling techniques, purely functional simulation can be used to fast-forward through nonsampled intervals [Sherwood et al. 2002; Wunderlich et al. 2003], while checkpointing strategies can be used to avoid simulating them at all [Biesbrouck et al. 2006; Wenisch et al. 2006].

However, these techniques cannot be readily applied to sampled parallel simulation of multithreaded applications for two reasons. First, the synchronization mechanisms in multithreaded applications influence the progress of coexecuting threads. Therefore, functional simulation during fast-forwarding intervals is not sufficient to maintain the consistency in progress among threads. Second, time-based parallel sampling techniques define the fast-forwarding intervals with respect to time. Therefore, one needs to know how far each thread has progressed in a fast-forwarding interval and from where (which instruction) the next detailed interval should start. We thus need to predict the IPC of fast-forwarding intervals to pass through them properly.

To address these concerns, we perform synchronization in fast-forwarding intervals and propose a novel IPC prediction method named *L-WMA*. We will introduce them in detail in the following discussion.

3.4.1 Synchronization during Fast-Forwarding Intervals. In multithreaded applications, synchronization mechanisms regulate communication and/or interaction among threads. Therefore, if synchronization were disregarded during fast-forwarding intervals, sampled simulation would distort the interaction between threads. In addition, the progress of each thread could also start diverging compared to an unsampled execution. As illustrated in Figure 6(a), if we ignore the synchronization events, threads would make different progress from what they should do. This is particularly important when we perform the warm-up strategy in the fast-forwarding intervals. The wrong overlap and interference of threads could distort the ordering of memory references leading to an incorrect cache state after warm-up. As a result, the sampled simulation might come to an incorrect result, which might lead to wrong conclusions.

Previous work has proved the importance of the synchronization during fast-forwarding intervals [Ardestani and Renau 2013; Carlson et al. 2013]. In our sampling scheme, we use the existing synchronization mechanism proposed in Carlson et al. [2013]. We maintain the inter-thread dependencies through shared memory and synchronization events (pthread_mutex, futex system calls, etc.) during fast-forwarding intervals. More specifically, threads waiting on the synchronization events do not execute instructions until they are woken up, that is, the waiting threads do not advance time but inherit the time of the thread that later wakes them up. After all threads have advanced in this way to the end of the fast-forwarding interval, a new detailed interval starts (see Figure 6(b)).

3.4.2 IPC Prediction for Fast-Forwarding Intervals. As we mentioned earlier, IPC prediction of fast-forwarding intervals is very important in time-based sampled simulation. Since we know the length in time of each fast-forwarding interval, an accurate IPC prediction can help us identify the instruction from where the simulator should switch to detailed simulation.

Therefore, the main problem is how to predict the IPC of each fast-forwarding interval accurately. To this end, several different methods have been proposed. For example, *IPC of 1* assumes the constant IPC value 1 for all threads and all fast-forwarding intervals.

However, Carlson et al. [2013] and Ardestani and Renau [2013] show that this method does not predict IPCs accurately. The *Last* method does the prediction using the IPC of last detailed interval, and was used by Carlson et al. [2013]. As illustrated in Figure 7(a), the numbers are indexes for different kinds of intervals in Figure 4: The even numbers represent detailed intervals, whereas the odd ones represent fast-forwarding intervals. *Last* method predicts IPC of the fast-forwarding *#2, #4, #6,* and so on, using the detailed interval *#1, #3, #5,* and so on, respectively. The *Weighted Moving Average (WMA)* method does the prediction using a weighted average of last h (3 or 5) samples. The most recent sample has the highest weight, and the last h^{th} sample has the lowest. Because of the inherently larger noise presented in their relatively short detailed intervals, Ardestani and Renau [2013] use the *WMA*.

In our sampling approach, we present a novel prediction method called *L-WMA*, which combines *Last* and simplified *WMA* method (all the chosen samples have the same weight). We can see that lengths of the fast-forwarding intervals in our sampling are very different. Therefore, we use different prediction methods for fast-forwarding intervals with different lengths—taking into consideration both locality and periodicity behavior of applications. Specifically, for the fast-forwarding intervals that have the same length as a detailed interval, we use the *Last* method to predict their IPC. This is because these fast-forwarding intervals are very short, and there is a high probability that they have the same IPC with that of the last detailed intervals due to the principle of locality. For example, as Figure 7(b) shows, for the fast-forwarding intervals with the index *#2*, *#6*, *#10*, *#14*, and so on, we predict their IPCs using the detailed intervals with the index *#1*, *#5*, *#9*, *#13*, and so on. For the remaining longer fast-forwarding intervals, we predict their IPCs using the simplified *WMA* method. More specifically, we do the prediction using the average IPC of the last h samples (detailed intervals), where h is the number of detailed intervals contained in the most recent period of time with a length equal to that of the fast-forwarding interval being predicted. For example, for the fast-forwarding interval with the index *#4*, we predict its IPC using the average IPC of two detailed intervals with the indexes *#1* and *#3*. All index numbers mentioned here for different time intervals are shown in Figure 4, and Figure 7(b) describes the details of the *L-WMA* method.

4. EXPERIMENTAL SET-UP

In order to comprehensively evaluate our sampling scheme for parallel simulation, we have performed both trace-driven and execution-driven experiments. Although the trace-driven evaluation method lacks flexibility, it is much faster than the execution-driven method, while providing more initial insight and enabling us to explore the impact of the sampling parameters quickly. Nonetheless, our main goal is to develop an execution-driven framework to perform sampled parallel simulation for multithreaded applications. We, therefore, integrated the proposed sampling scheme into the parallel simulator *Sniper*—we refer to this implementation as *PCantorSim*. The detailed experimental set-ups for the two evaluation methods are as follows.

4.1 Trace-Driven Experimental Set-Up

We obtained our multithreaded application trace files from the *Sniper* project.[2] These trace files were collected by running benchmarks on *Sniper* modeling an 8-core system (the detailed configurations are provided in Table II). The performance metrics are gathered at an interval of 10μs, including the number of instructions and cycles.

Benchmarks corresponding to these trace files are from different multithreaded benchmark suites, the SPEC OpenMP (medium) suite (*train* inputs) [Aslot et al. 2001],

[2]http://snipersim.org/.

Table II. Architectural Parameters

Item	Specification
Processor	2 sockets, 4 cores per socket
Core	2.66GHz, 4-way issue, 128-entry ROB
Branch predictor	Pentium M [Uzelac and Milenkovic 2009], 17 cycles penalty
L1-Data (private)	32KB * 8, 4-way, 4-cycle access time
L1-Instruction (private)	32KB * 8, 8-way, 4-cycle access time
L2 (private)	256KB * 8, 8-way, 8-cycle access time
L3 (shared by four cores)	8MB *2, 16-way, 30-cycle access time
Main memory	8GB/s per socket, 65ns access time

Table III. Applications and Inputs of the Trace Files Used in Our Trace-Driven Experiments

Benchmark Suite	Application	Input size
NPB	N-ep, N-sp, N-ua, N-ft	class A
SPEC OpenMP	O-ammp, O-equake, O-art	train
PARSEC 2.1	P-dedup, P-swaptions, P-ferret	simlarge
SPLASH-2	S-cholesky	default

the NAS Parallel Benchmarks version 3 with OpenMP parallelization (*class A* inputs) [Jin et al. 1999], the PARSEC 2.1 benchmark suite (*simlarge* inputs) [Bienia et al. 2008], and the SPLASH-2 benchmark suite [Woo et al. 1995]. We refer to the benchmarks from these suites using the O-*, N-*, P-* and S-* notation, respectively. Because of space limitations in the article, we report 11 randomly chosen applications from these benchmark suites to evaluate our approach. Table III shows the details of the selected applications. The reason for picking benchmarks from different benchmark suites is to study and understand how widely applicable Cantor-based sampling is.

4.2 Execution-Driven Experimental Set-Up

We have integrated our newly proposed Cantor-based sampling approach into *Sniper*, which is a fast parallel simulation infrastructure. There are three simulation modes in *Sniper*: detailed, cache-only and fast-forward mode. The dynamic instrumentation framework Pin [Luk et al. 2005] is used as the functional simulation front-end to send detailed instruction information to the various *Sniper* simulation modes. We can switch the simulator between the different modes during the simulation by determining which analysis routines are enabled.

We use the existing detailed mode in *Sniper* during the detailed intervals in our sampling. In order to enable synchronization during fast-forwarding intervals, we use the synchronization mechanisms proposed in Carlson et al. [2013]. A modified sampling indicator is added to *Sniper* to switch the simulator between the different simulation modes based on the Cantor sampling method.

We configure the simulator to model an 8-core out-of-order processor. Table II shows the detailed architectural parameters. The applications used in the execution-driven experiments are from the PARSEC 2.1 parallel benchmark suite (*simlarge* inputs). We limit the execution-driven simulation evaluation to a single benchmark suite to limit the overall simulation time, which is the motivation of this work in the first place. We failed to run some of the benchmarks because of the limitation of the simulator. In our measurements, only the parallel Region of Interest (ROI) of each application is included. The fast-forward mode is used to skip over the initialization and cleanup phases.

Table IV. Selected Values of the Sampling Parameter L_d in the Trace-Driven Experiments

Sampling methodology	Detailed interval length (10μs)
Cantor sampling	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 2500, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000.
Periodic sampling	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 2500, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000.

5. EVALUATION

In this section, we present the evaluation results of both the trace-driven and execution-driven experiments. In the trace-driven evaluation, we verify the robustness of the proposed fractal-based sampling algorithm by performing a sweep of sampling parameters. We then compare our approach against the previous periodic sampling in terms of the IPC prediction accuracy, the fraction of detailed simulation and the robustness of the sampling algorithm. In the execution-driven evaluation, we first show the details how we derive suitable sampling parameters, and then provide the accuracy and speed-up of our approach with respect to that of fully detailed simulation. We also make a comparison with previous multithreaded sampling methodologies in terms of accuracy and simulation speed. Finally, we discuss our warm-up strategy and evaluate it through experimentation.

5.1 Trace-Driven Evaluation

In order to evaluate our fractal-based sampling thoroughly, we did thousands of experiments using trace-driven simulation. Table IV shows the L_d values used in the experiments. We choose L_d to cover a very large range, from 10μs to 100ms for Cantor-based sampling, and from 10μs to 1s for periodic sampling [Carlson et al. 2013].

For each L_d in Cantor sampling, we sweep the parameter K from 1 to K_{max}, where K_{max} can be calculated by Equation (5).

$$K_{\max} = \left\lfloor \frac{\ln(L_{total}/L_d)}{\ln 3} \right\rfloor \tag{5}$$

(Note that L_{total} is known in the trace-driven experiments.)

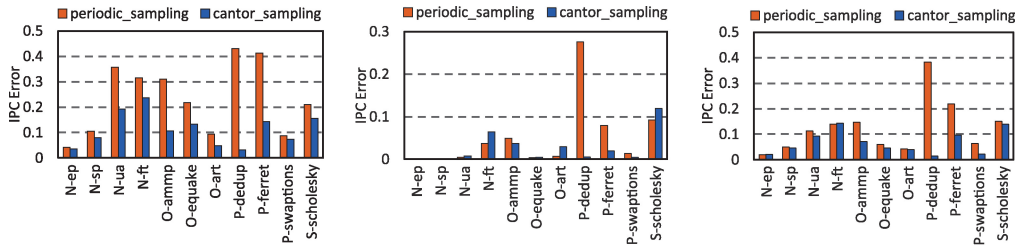
For each L_d in periodic sampling, we sweep F/D in the suggested range from 5 to 10 [Carlson et al. 2013]. F/D is a parameter used in periodic sampling, which determines the length of fast-forwarding intervals based on that of detailed intervals using Equation (6).

$$L_f = L_d \times (F/D) \tag{6}$$

To compare the IPC prediction accuracy between the two sampling methods, we use Equation (7) to compute the IPC error.

$$Error = \frac{\sum_1^n (|IPC(d_n) - IPC(f_n)| \times L(f_n))}{L_{total}} \tag{7}$$

In Equation (7), n is the number of detailed and/or fast-forwarding intervals in the entire execution time. The number of detailed intervals is equal to that of fast-forwarding ones for both sampling methods. $IPC(d_n)$ and $IPC(f_n)$ is the predicted and real IPC for the n^{th} fast-forwarding interval, respectively. $IPC(d_n)$ is computed using the IPCs of one or several detailed intervals according to different IPC prediction



(a) Maximum error. (b) Minimum error. (c) Average error.

Fig. 11. The comparisons of maximum, minimum and average error between cantor sampling and periodic sampling.

strategies (see details in Section 3.4.2). $L(f_n)$ is the length of the n^{th} fast-forwarding interval. We believe the error calculated by Equation (7) reflects the IPC prediction accuracy and provides a reliable way to compare different sampling methods.

5.1.1 Sampling Algorithm Robustness. Figure 8 shows the average IPC prediction error (calculated according to Equation (7)) of our fractal-based sampling with 41 different values for L_d (listed in Table IV). The results show that IPC errors of these applications are very small and almost the same or fluctuate in a tiny range at different time scales (i.e., different values of L_d). For example, in Figure 8, the IPC errors of O-*ammp* are almost the same (0.06) and the IPC errors of P-*dedup* fluctuate in a tiny range (from 0.02 to 0.04) when we take different values of L_d. This indicates that our fractal-based sampling is effective at different time scales, which also confirms our observation in Section 3.1. That is, many multithreaded applications not only exhibit phase behavior, but also exhibit fractal behavior.

Although Figure 8 shows the average IPC error for each L_d, it is still not sufficient to draw the conclusion that the IPC errors stay almost the same at different time scales. This is because there are several *Ks* (from 1 to K_{max}) for each L_d, and the errors for the same L_d with different *Ks* could fluctuate wildly but maintain the same average value. To reveal the truth, we take N-*ep* as the example and plot its IPC prediction errors of different *K*s for the same L_d in one column (Figure 9). Thus, there are 41 columns corresponding to 41 different values of L_d and in each column there are several IPC errors corresponding to different *Ks* (from 1 to K_{max}). The results show that the IPC errors are also small and fluctuate only in a tiny range (from 0 to 0.035). We did the same experiments for all the 11 applications (not shown here because of space constraints) and reached the same conclusion.

Figure 10 shows the IPC error variations with different values of N. We find that accuracy does not change much as we vary N.

In summary, these experiments demonstrate that our method for predicting IPC during fast-forwarding in fractal-based sampling is effective and accurate. In addition, these results once more confirm that multithreaded applications exhibit fractal execution behavior.

5.1.2 Comparison to Periodic Sampling. We now compare our fractal-based sampling with the periodic sampling proposed by Carlson et al. [2013] along the following three criteria: (1) the accuracy of IPC prediction for fast-forwarding intervals, (2) the fraction of detailed simulation, and (3) the robustness of the sampling algorithm. We measure the accuracy using IPC errors calculated by Equation (7). Figure 11 shows the comparisons of maximum, minimum and average errors between fractal-based versus periodic sampling. In general, fractal-based sampling outperforms periodic sampling by a small margin when it comes to the accuracy of IPC prediction of fast-forwarding intervals.

Table V. The Prediction Error and Simulation Speed-Up for Different L_d, 100ns, 1μs, 10μs, 20μs, Respectively

	L_d (100ns)		L_d (1μs)		L_d (10μs)		L_d (20μs)	
Application	Error (%)	Speed-up (×)	Error (%)	Speed-up (×)	Error (%)	Speed-up (×)	Error (%)	Speed-up (×)
blackscholes	30.2	12.6	21.5	20.4	13.4	8.9	8.1	9.3
fluidanimate	25.1	16.7	10.3	27.4	2.7	18.9	3.0	13.1
streamcluster	−2.5	16.8	−3.8	29.9	−0.6	18.1	−0.3	12.7
dedup	25.2	24.8	17.6	43.6	3.8	32.3	0.7	23.1
swaptions	−4.4	16.0	−3.6	25.6	−2.4	18.9	−2.1	14.4
raytrace	9.2	15.7	4.6	23.2	2.9	13.8	2.6	9.7

For the fraction of detailed simulation, in Carlson et al. [2013], the parameter F/D was set to 5 or 10. This means the length of a fast-forwarding interval is five or ten times as big as that of a detailed interval. Hence, the fraction of detailed simulation is 16.67% or 9.09%, respectively. In our sampling scheme, we take the parameter L_d as 10μs, the value of K_{max} is more than 10 for most of the applications. According to Equation (4), the fraction of detailed simulation is only 1.74% or less, which is much less than that for periodic sampling. In other words, Cantor-based sampling has greater potential to improve simulation speed compared to periodic sampling.

Furthermore, we have shown the robustness of fractal-based sampling in Section 5.1.1, as it applies to all benchmarks evaluated in this study. For periodic sampling on the other hand, Carlson et al. [2013] demonstrated that some applications (N-*lu* and O-*ammp,* etc.) are not suitable for periodic sampling because of a lack of regular phase behavior, or it was found to be impossible to find a sampling period that matches the application's periodicities.

5.2 Execution-Driven Evaluation

As discussed in Section 3.3 and evaluated in Section 5.1.1, the fractal-based sampling algorithm is more robust than periodic sampling and is less sensitive to the selection of sampling parameters. For example, Figures 8, 9, and 10 illustrate the IPC error variations along with changes of the parameters L_d, K, and N, respectively. The results show that the parameter changes do not influence the prediction accuracy significantly. Although we have discussed how we determine the sampling parameters in Section 3.3, we still would like to provide more details about how to derive suitable sampling parameters.

The most important parameter in our sampling scheme is the length of a detailed interval L_d. Because we cannot divide the execution time of an application infinitely as suggested by the Cantor set theory, we must determine when to stop the division. In Section 3.3, we suggest to set "$L_d = 10\mu$s" for the reason that the smallest time interval in our trace-driven experiments is 10μs. However, there is no limit in execution-driven experiments. The L_d could be 5μs, 1μs, 100ns or smaller. To find its proper value for the execution-driven mode, we have explored several different values of L_d. As illustrated in Table V, we tried 100ns, 1μs, 10μs, and 20μs for L_d, and we chose six applications randomly. We did not explore many values for L_d as we did on trace files because it is time consuming in real simulation.

As we mentioned before, a smaller value for L_d implies a smaller fraction of the application simulated in detail and, therefore, a higher simulation speed-up. However, too small a value for L_d would partition the execution time into too many time intervals, which would require the simulator to change simulation modes between detailed, warm-up and fast-forwarding modes very frequently, ultimately affecting simulation speed. The experimental results also confirm this. In Table V, the speed-up for $L_d =$ 100ns is smaller than that for $L_d = 1\mu$s. Moreover, too small L_d will cause that there

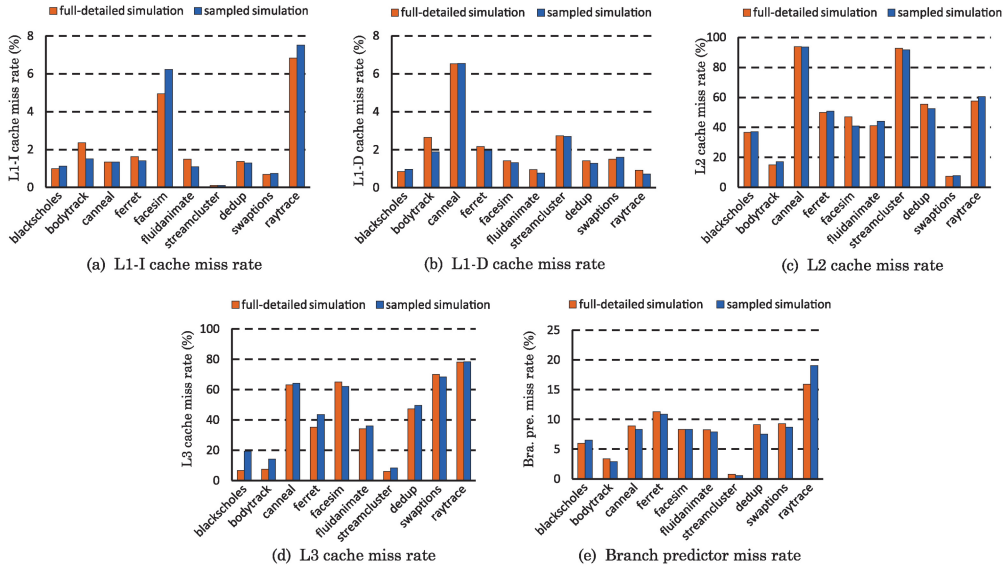


Fig. 12. The accuracy of the proposed fractal-based sampled simulation in several performance metrics compared with fully-detailed simulation, including L1-I cache miss rate, L1-D cache miss rate, L2 cache miss rate, L3 cache miss rate and the branch predictor miss rate.

is very little program information in each detailed interval. For example, there is only approximately 100 to 1,000 instructions in the time interval of 100ns, which leads to an increase in prediction error. As shown in Table V, the error rate for $L_d = 100$ns is relatively high. On the other hand, when $L_d \geq 10\mu$s, the error rate almost remains the same or fluctuates in a tiny range. These results are consistent with what we obtained through the trace-driven experiments.

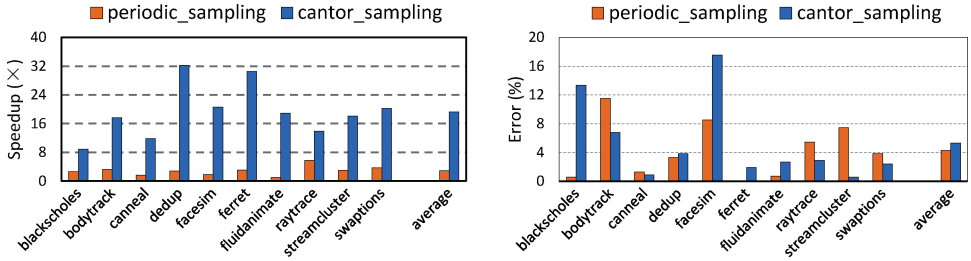
We, therefore, suggest that one chooses $L_d \geq 10\mu$s in the proposed sampling scheme (preferably $L_d = 10\mu$s with the consideration of simulation speedup). The remaining parameters, for example, L_f, L_w, N, and K can be derived following the procedures in Section 3.3. Additionally, we will discuss more about L_w in Section 5.2.3.

5.2.1 Accuracy and Speed of Fractal-Based Sampled Simulation. Figure 12 presents the accuracy of fractal-based sampled simulation compared to detailed (nonsampled) simulation in several aspects, including L1-I cache miss rate, L1-D cache miss rate, L2 cache miss rate, L3 cache miss rate, and the branch predictor miss rate. Fractal-based sampled simulation accurately matches detailed simulation with respect to these performance metrics. Additionally, when compared to detailed simulation, the average error of sampled simulation regarding to the golden performance metric—execution time—is only 5.3%. Meanwhile, the average speed-up over detailed simulation equals 20× with a maximum speed-up at 32×. Table VI presents the overview of all applications, the chosen sampling parameters, simulation speed-up, and accuracy.

The results show that the proposed sampling methodology reduces simulation time substantially, while providing a stable and accurate estimation in many important performance metrics for most of the applications. However, for a couple applications, the error is somewhat higher. For example, the execution time prediction error for *blackscholes* and *facesim* exceeds 10%. The highest error (−17.56% for *facesim*) is caused by our relatively simple warm-up strategy (see details in Section 5.2.3). We experimentally verified that accuracy improves by increasing the length of the warm-up intervals: The error goes down to −6.06% and 5.38% with a warm-up length of

Table VI. Overview of All Applications, the Chosen Sampling Parameters, and Their Speed-Up and Accuracy

Application	L_d (μs)	K	Error	Speed-up	Sampled simulation time
P-blackscholes	10	6	13.37%	8.85×	0.06h
P-bodytrack	10	8	6.80%	17.67×	0.11h
P-canneal	10	7	0.89%	11.84×	0.07h
P-dedup	10	10	3.82%	32.27×	0.37h
P-facesim	10	8	−17.56%	20.59×	0.26h
P-ferret	10	10	−1.92%	30.49×	0.27h
P-fluidanimate	10	8	2.69%	18.88×	0.13h
P-raytrace	10	7	2.88%	13.85×	0.10h
P-streamcluster	10	8	−0.57%	18.12×	0.16h
P-swaptions	10	8	−2.43%	20.23×	0.13h



(a) Simulation speed-up. (b) Execution time prediction error.

Fig. 13. Comparison of cantor sampling against periodic sampling in terms of simulation speed-up and execution time prediction error.

270μs and full warm-up during the entire fast-forwarding intervals, respectively. The second highest error (13.37% for *blackscholes*) is due to the relatively small detailed simulation interval; Table V reports a smaller error with a longer detailed simulation interval (8.1% for $L_d = 20\mu$s).

5.2.2 Comparisons of Fractal-Based Sampling Scheme against Previous Sampling Methodologies. There are essentially two prior works proposing and evaluating sampled simulation of multithreaded applications running on multicore systems. The first work by Carlson et al. [2013] presented periodic sampling based on the regular phase behaviors in multithreaded applications. They also implemented their sampling methodology in the *Sniper* simulator and evaluated it with an 8-core architecture. The average speed-up of periodic sampling for applications in the PARSEC 2.1 benchmark suite is 3× with an average execution time prediction error of 4.05% (see Figure 13). Clearly, Cantor-based sampling achieves higher simulation speed-up (20×, on average) while sacrificing very little accuracy (the average error is 5.3%); see also Figure 13.

The second prior work by Ardestani and Renau [2013] proposed a statistical sampling framework for the simulation of multithreaded applications. They evaluated their sampling methodology using a single-threaded simulation infrastructure SESC [Renau et al. 2005]. For applications in the PARSEC 2.1 benchmark suite with an 8-core architectural configuration, the average execution time error of their approach is around 7%, with an average simulation speed of 9MIPS. In contrast, our fractal-based sampling methodology in *Sniper* achieves an average simulation speed of 20MIPS, with an average error of 5.3%.

5.2.3 Warm-up Strategy. In the periodic sampling methodology, functional warming of caches and branch predictors is enabled all the way through the fast-forwarding intervals. However, previous work shows that it is not necessary to enable warm-up at all

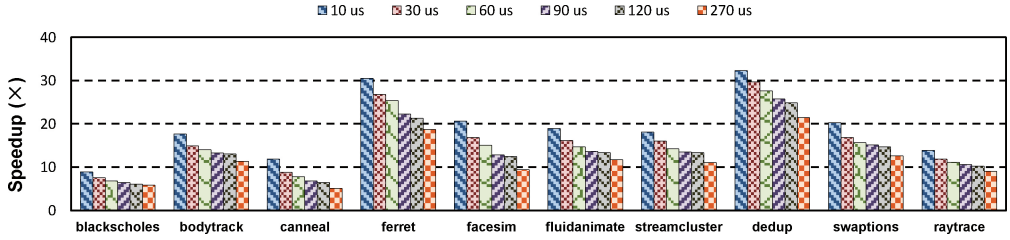


Fig. 14. Simulation speed-up as a function of warm-up length (10, 30, 60, 90, 120, and 270 μs).

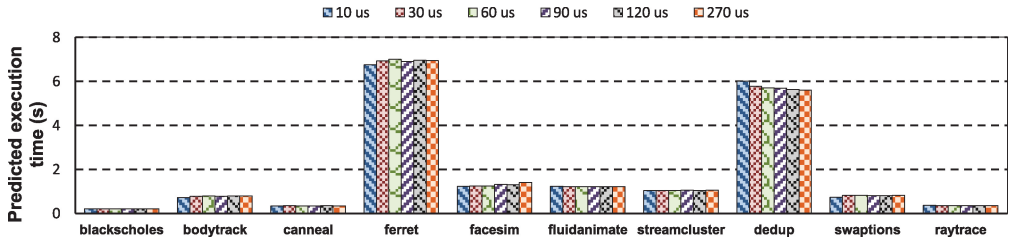


Fig. 15. Predicted execution time as a function of warm-up length (10, 30, 60, 90, 120 and 270 μs).

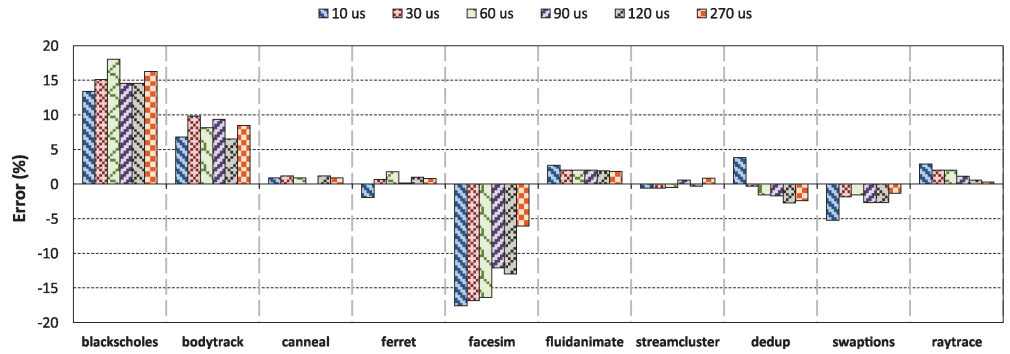


Fig. 16. Execution time prediction error as a function of warmup length (10, 30, 60, 90, 120 and 270μs).

times. An effective warm-up strategy can select shorter warm-up intervals to achieve substantially higher simulation speed with limited effect on accuracy [Eeckhout and Bosschere 2006].

In this work, we present a simple and effective warm-up strategy. Based on our experiments, larger warm-up intervals do not necessarily contribute much to error reduction but rather decrease the simulation speed-up significantly. To evaluate it, we change the length of warm-up intervals from 10μs to 270μs, and collect the speed-up, the predicted execution time, and the execution time prediction error. Figures 14, 15, and 16 illustrate these metrics as a function of warm-up length. The predicted execution time and the execution time prediction error remain almost the same when we increase the length of warm-up intervals from 10μs (the L_d we choose in our setting) to 270μs; but the speed-up becomes significantly smaller. Therefore, we enable warm-up before each detailed interval and choose the same length of detailed intervals (10μs).

6. RELATED WORK

Sampling is an effective and widely used technique to accelerate microarchitectural simulation. A number of methodologies have been proposed to speed up the simulation

of single-threaded applications [Conte et al. 1996; Wunderlich et al. 2003; Sherwood et al. 2002; Argollo et al. 2009]. Conte et al. [1996] were the first to apply sampling theory on processor simulation. Wunderlich et al. [2003] (SMARTS) demonstrate that periodic sampling with very small detailed simulation intervals (on the order of 1,000 instructions) leads to accurate performance predictions provided that functional warming is maintained throughout the simulation. Sherwood et al. [2002] propose a phase-based sampling technique that chooses large representative intervals (on the order of 100M instructions) by using Basic Block Vectors (BBVs) to simulate in detail. All these sampling approaches specify their sampling parameters in terms of the number of instructions, which cannot be readily applied to sampled simulation of multithreaded applications, as extensively argued in this article.

Van Biesbrouck et al. [2004] introduce the co-phase matrix as a reduction technique for multiprogram workloads. Since the cophase matrix dimensions grow quadratically with the number of cores, this method does not scale well. Wenisch et al. [2006] propose SimFlex for multicore throughput applications based on statistical sampling. However, these sampling techniques ignore the synchronization among the threads and depend on the assumption that each thread is independent.

The work of Carlson et al. [2013] and Ardestani and Renau [2013] is closely related to our approach. Carlson et al. [2013] propose a periodic sampling scheme for multi-threaded applications based on application periodicity behavior. Prior to simulation, a preprocessing step is needed to determine the length of application periodicities using BBV-based tools. Misalignment between the sampling parameters and the application's periodicities may lead to aliasing problems, potentially compromising accuracy. Or, applications that lack stable periodic phase behavior may not be amenable to sampling through this method. Ardestani and Renau [2013] present Time-Based Sampling (TBS). Determining the sampling parameters is done experimentally. The simulation infrastructure of TBS itself is single-threaded, which significantly limits overall simulation speed. Both sampling methods recognize, for the first time, that TBS is an accurate approach for sampled simulation of multithreaded applications.

Yu et al. [2009, 2010] find similarities between the Cantor set fractals and execution behaviors of single-threaded benchmarks. They show that most single-threaded programs exhibit periodic behavior as a function of time but the length of these periods may be dramatically different. Measuring behavior in terms of IPC or cache miss rates, bursts or clusters of periodic behavior can be observed. They then propose a Cantor set fractal approach to accelerate microarchitecture simulation based on this insight. In contrast to this article, this prior work focused on single-threaded programs running on single-core machines. Additionally, their sampling parameters are specified in terms of dynamically executed instructions, which is invalid in the context of parallel simulation.

7. CONCLUSIONS

This work presents a fractal-based sampling to speed up parallel microarchitecture simulation with multithreaded applications. We find that there exists fractal behavior as well as phase behavior over the execution time of parallel programs. By leveraging this observation, we propose a sampling scheme that reconstructs the execution time of an application by selecting representative time intervals simulated in detail based on a fractal rule—the rule of Cantor set generation. The sampling parameters are specified regarding time and are easy to determine. We also enable synchronization during fast-forwarding intervals and perform a simple yet effective warm-up strategy to improve the runtime prediction accuracy while maintaining high simulation speed.

Cantor-based sampling enables simulating less than 5% of an application's execution time in detail, yielding substantial simulation speed-up at high accuracy. We have

implemented Cantor-based sampling in *Sniper* (a parallel multicore simulator) and evaluated it by running the PARSEC benchmarks on an 8-core simulated system. Our experimental results show that Cantor-based sampling speeds up simulation by a factor of 20×, on average, over detailed simulation, with an average absolute execution time prediction error of 5.3%.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable and constructive comments.

REFERENCES

ALAMELDEEN, A. R. AND WOOD, D. A. 2006. IPC considered harmful for multiprocessor workloads. *IEEE Micro 26*, 8–17.

ALAMELDEEN, A. R. AND WOOD, D. A. 2003. Variability in architectual simulations of multi-threaded workloads. In *Proceedings of 9th Annual International Symposium on High Performance Computer Architecture (HPCA'03)*. IEEE Computer Society, Washington, DC, 7–18.

ARDESTANI, E. K. AND RENAU, J. 2013. ESESC: A fast multicore simulator using time-based sampling. In *Proceedings of 19th Annual International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE Computer Society, 448–459.

ARGOLLO, E., FALCON, A., FARABOSCHI, P., MONCHIERO, M., AND ORTEGA, D. 2009. COTSon: Infrastructure for full system simulation. *ACM SIGOPS Operat. Syst. Rev. 43*, 1, 52–61.

ASLOT, V., DOMEIKA, M., EIGENMANN, R., GAERTNER, G., JONES, W. B., AND PARADY, B. 2001. SPEComp: A new benchmark suite for measuring parallel computer performance. *Shared Memory Parallel Programming,* R. Eigenmann and M. Voss, Eds., 2104, 1–19.

BHADAURIA, M., WEAVER, V. M., AND MCKEE, S. A. 2009. Understanding PARSEC performance on contemporary CMPs. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE Computer Society, 98–107.

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM Press, New York, 72–81.

BIESBROUCK, M. V., CALDER, B., AND EECKHOUT, L. 2006. Efficient sampling startup for simpoint. *IEEE Micro 26*, 4, 32–42.

BIESBROUCK, M. V., SHERWOOD, T., AND CALDER, B. 2004. A co-phase matrix to guide simultaneous multi-threading simulation. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04)*. IEEE Computer Society, 45–56.

CARLSON, T. E., HEIRMAN, W., AND EECKHOUT, L. 2013. Sampled simulation of multi-Threaded applications. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. IEEE Computer Society.

CARLSON, T. E., HEIRMAN, W., AND EECKHOUT, L. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*. ACM, New York.

CONTE, T. M., HIRSCH, M. A., AND MENEZES, K. N. 1996. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the International Conference on Computer Design (ICCD'96)*. IEEE, 468–477.

EECKHOUT, L. AND BOSSCHERE, K. D. 2006. Yet shorter warmup by combining no-state-loss and MRRL for sampled LRU cache simulation. *J. Syst. Software 79*, 645–652.

FEITELSON, D. G. 2013. Workload modeling for computer systems performance evaluation. Version 0.41.

HE, L., YU Z., AND JIN, H. 2012. FractalMRC: Online cache miss rate curve prediction on commodity systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*. IEEE, 1341–1351.

HURST, H. E. 1951. Long term storage capacity of reservoirs. *Tran. Am. Soc. Civil Eng. 116*, 770–808.

JIN, H., FRUMKIN, M., AND YAN, J. 1999. The OpenMP implementation of NAS parallel benchmarks and its performance. Tech. rep., NASA Ames Research Center.

LAU, J., SCHOEMACKERS, S., AND CALDER, B. 2004. Structures for phase classification. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04)*. IEEE Computer Society, 57–67.

Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Lauguage Design and Implementation (PLDI'05)*. ACM, New York, 190–200.

Mandelbrot, B. B. 1983. The fractal geometry of nature. Free man.

Miller, J. E., Kasture, H., Kurian, G., Gruenwad III, C., Bechmann, N., Celio, C., Eastep, J., and Agarwal, A. 2010. Graphite: A distributed parallel simulator for multicores. In *Proceedings of 16th Annual International Symposium on High Performance Computer Architecture (HPCA'10)*. IEEE Computer Society, 1–12.

Peitgen, H., Jürgens, H., and Saupe, D. 2004. Chaos and fractals: New frontiers of science. Springer.

Perelman, E., Polito, M., Bouguet, J.-Y., Sampson, J., Calder, B., and Dulong, C. 2006. Detecting phases in parallel applications on shared memory architectures. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE.

Renau, J., Basilio, F., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., and Montesinos, P. 2005. SESC: Cycle accurate architecutral simulator. Retrieved November 19, 2013 from http://sesc.sourceforge.net/.

Sanchez, D. and Kozyrakis, C. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of 40th Annual International Symposium on Computer Achitecture (ISCA'13)*. IEEE Computer Society.

Sherwood, T., Perelman, E., and Calder, B. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*. ACM, New York, 3–14.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. ACM, New York, 45–57.

Thiebaut, D. 1989. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Trans. Comput. 38*, 7, 1012–1026.

Uzelac, V. and Milenkovic, A. 2009. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. IEEE Computer Society, 207–217.

Voldman, J., Mandelbrot, B., Hoevel, L. W., Knight, J., and Rosenfeld, P. 1983. Fractal nature of software-cache interaction. *IBM Journal of Research and Development 27*, 2, 164–170.

Wenisch, T., Wunderlich, R., Ferdman, M., Ailamaki, A., Falsafi, B., and Hoe, J. 2006. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro 26*, 4, 18–31.

Wenisch, T. F., Wunderlich, R. E., Falsafi, B., and Hoe, J. C. 2006. Simulation sampling with live-points. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'06)*. IEEE Computer Society, 2–12.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of 22th Annual International Symposium on Computer Achitecture (ISCA'95)*. IEEE Computer Society, 24–36.

Wunderlich, R. E., Wenisch, T. F., Falsafi, B., and Hoe, J. C. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of 30th Annual International Symposium on Computer Achitecture (ISCA'03)*. IEEE Computer Society, 84–95.

Yu, Z., Jin, H., Chen, J., and John, L. 2010. CantorSim: Simplifying acceleration of micro-architecture simulations. In *Proceedings of the 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'10)*. IEEE Computer Society, 370–377.

Received June 2013; revised August 2013, September 2013; accepted November 2013