# Statistical Simulation of Multithreaded Architectures

Joshua L. Kihm and Daniel A. Connors
University of Colorado at Boulder
Department of Electrical and Computer Engineering
UCB 425, Boulder, CO, 80309
{kihm, dconnors}@colorado.edu

## Abstract

*Detailed, cycle-accurate processor simulation is an integral component of the design and study of computer architectures. However, as the detail of simulation and processor design size increase, simulation times grow exponentially, and it becomes increasingly necessary to find fast, efficient simulation techniques that still ensure accurate results. At the same time, multithreaded multi-core designs are increasingly common, and require increased experimental design evaluation for a number of reasons including higher system complexity, interaction of multiple co-scheduled application threads, and workload selection. Although several effective simulation techniques exist for single-threaded architectures, techniques have not been effectively applied to the simulation of multithreaded and multi-core architecture models. Moreover, multithreaded processor simulation introduces unique challenges in all simulation stages. This work introduces systematic extensions of commonly-used statistical simulation techniques to multithreaded systems. The contributions of this work include: tailoring simulation fast-forwarding for individual threads, the effects of cache warming on application threads, and an analysis of the primary issues of efficient multithreaded simulation.*

## 1 Introduction

Detailed simulation is a vital component of the design process of modern processors and exploration of new architectural concepts. In general, integrated circuit manufacturing processes have been delivering ever larger numbers of faster transistors to microprocessor designers, increasing the space for architecture techniques. In turn, designers have relied on simulation to explore using these transistors to improve performance by building more complex structures to exploit single stream parallelism. Each new generation of processors has deeper pipelines, larger instruction windows, and larger cache memories. This increase in design complexity results in a corresponding increase in simulation complexity and a reduction in speed.

Modern, high-level, architectural simulators typically run in the tens to hundreds of kilohertz range, making them four to five orders of magnitude slower than corresponding hardware. This makes full simulation of all but the most trivial programs prohibitively long. In order reduce simulation time, several techniques are used, many of which are compared in [15]. Two recent techniques that have been shown to be effective are SMARTS ([14]) and SimPoint ([9]). SimPoint exploits the repetitive nature of most programs to simulate small segments of code and extrapolates this behavior across samples know to have similar code signatures. SimPoints is extended to multithreaded systems in [13] and further explored in [6]. SMARTS works by taking a large number of small samples over the entirety of the program. The goal of this work is to extend the concepts of SMARTS to multithreaded and multicore architectures.

To meet performance demands, processor designers are turning to multithreading and chip-multiprocessor designs. In a few short years, almost every server and desktop CPU manufactured will include multithreading and chip multiprocessor (CMP) ([8]) support. These techniques allow an easy methodology to utilize resources and circumvent limits on single-thread performance often times with minimal increase in design complexity. However, multiple threads which run simultaneously on chip compete for shared processor resources. This contention is highly influential on the performance of each thread and therefore the whole system. The main issue such designs present is that proper study of workload variations in inter-thread interference and contention leads to an extensive increase in both the design space and the amount of program activity that must be simulated. The increase in complexity is due primarily to choosing which resources should be shared between threads and understanding the interactions of threads in these resources.

Resources sharing ranges from schemes such as Simultaneous Multithreading (SMT) ([11]) where essentially all processor resources are shared to CMP where very few re-

sources, such as lower level caches and buses, are shared. Emerging processors are often designed as a compromise between these extremes or some combination of them. Many versions of Intel Pentium-4 processor include an SMT like design called Hyperthreading where certain processor resources are segregated between processes ([3]). The recent IBM POWER5 processor is a CMP design consisting of two, two-way SMT-cores ([5]). In addition to SMT, Coarse-Grained (Temporal) Multithreading ([1]) in which only one thread is fetched at a time, but light weight context switches occur on long latency events, is also a popular multithreading technique as in the IBM POWER4 architecture ([2]). It has even been shown that SMT and CGMT can be combined ([12]), adding further size to the multithreading design space. The final aspect of the increase in difficulty of characterization of multithreaded systems is the interaction between the programs themselves. The number of benchmark combinations increases exponentially with the number of threads on the core. Additionally, since each program exhibits unique phase behaviors, the total number of unique behaviors of a multithreaded system is the number of *co-phase* combinations ([13]), which also grows exponentially with the number of threads.

This work presents a methodology for simulating multithreaded and multicore systems using a methods based upon SMARTS ([14]) in which a large number very short periods of detailed simulation over the course of the entire program with longer periods of fast, functional simulation in between. This allows a significant speed up in simulation without sacrificing accuracy. Additionally, statistical analysis can be used to determine the bounds of error due to sampling. Several issues, unique to multithreaded simulation are addressed. First, the work in Section 2. The issues due to multithreading involved in individual simulation phases of functional simulation (Section 3), detailed simulation (Section 4), functional warm-up (Section 5) are each explored. Finally, Section 6 concludes.

## 2 Motivation

The emergence of multithreaded and multicore processors is one of the most prevalent trends in modern computer architecture. The ability to run multiple threads simultaneously circumvents many of the bottlenecks on instruction level parallelism (ILP) and allows overall system throughput to increase. However, one of the most important factors in the performance multithreaded systems is how individual threads interact and compete for shared resources. Since each combination of threads exhibits unique performance as the constituent threads interact, the number of experiments needed to accurately characterize a multithreaded system will grow with the number of combinations of benchmarks. The number of combinations will in turn grow exponentially with the number of threads on the system. For example, the *Spec2000* CPU benchmark suite ([10]) contains 26 benchmark programs. Ignoring the fact that several of the benchmarks have multiple input sets, a single threaded system needs to run 26 tests to characterize SPEC performance. There are 351 combinations of two benchmarks (including combinations of two instances of the same benchmark) which means a two-thread system would have to run 351 tests to achieve the same level of characterization. If the system were designed in an asymmetric manner, such that which thread were placed in each context was important, the number of tests grows to 676.

Since each thread demonstrates variable behavior during execution, what is really needed to is to evaluate the potential combinations of program phases encountered in real-world systems. In fact, it may only be necessary to study phases that exhibit important behaviors or corner cases in characterizing a system. If each SPEC benchmark were divided into ten phases, 33,930 combinations would be possible on a symmetric, two-thread system and 67,600 would be possible on a asymmetric system. In order to simulate 33,930 co-phase combinations for ten-million cycles each on a 100 kilohertz simulator on a single design would take almost 40 CPU days of testing. An equivalent test on a four-thread system would take approximately 590 CPU years. These numbers grow exponentially as the number of threads increases which is illustrated in Table 1. The large number of possible tests makes all full simulation of all but the most trivial tests prohibitively long and promotes the need for efficient simulation techniques even more critical in multithreaded systems than in their single-threaded counterparts, especially as the number of threads on chip increases.

| Phases per Benchmark | Symmetry | Number of Threads | | | |
|---|---|---|---|---|---|
| | | 1 | 2 | 4 | 8 |
| 1 | Yes | 26 | 351 | 1.5e4 | 6.3e10 |
| | No | 26 | 676 | 4.6e5 | 2.1e11 |
| 10 | Yes | 260 | 3.4e4 | 1.8e8 | 4.6e14 |
| | No | 260 | 6.8e4 | 4.6e9 | 2.1e19 |
| 20 | Yes | 520 | 1.3e5 | 3.0e9 | 1.3e17 |
| | No | 520 | 2.7e5 | 7.3e10 | 5.3e21 |

**Table 1. Number of tests needed to characterize multithreaded systems for *Spec2000* CPU.**

## 3 Fast-Forwarding Simulation

### 3.1 Proportional Fast-Forwarding

Because detailed simulation is prohibitively slow, many simulators contain a second, faster operation mode called

*functional simulation*. This increased speed is accomplished by disabling features of the simulation, such as timing information and stats collection, which are not necessary to maintain the correctness of the simulated program. In a single-thread simulation, this is a straight forward problem, a given number of instructions can be skipped and detailed simulation is restarted, optionally after a warm up period [4]. However, in a multithreaded simulation the problem is more complex. Since in almost all cases the simulated threads will be running at different rates, simply skipping a number of instructions from each thread will not be representative of actual execution. The relative position of the threads determines the co-phase behavior, and hence the performance of the system. By *proportionally fast-forwarding* the threads, relative position of the threads will approximate the position of the threads if detailed warming had been sustained. Proportional fast-forwarding simply means that the performance of each thread during a detailed warming period is extrapolated over the course of fast forwarding period. Each thread is fast-forwarded in proportion to its IPC over the last detailed simulation period. The assumption that the smaller, detailed simulation period will be representative of the larger program is central to all fast-forwarding schemes. The extension of the assumption for multithreaded systems is that the performance of each thread will remain constant over the functional simulation period.

To test this theory, all pairings of nine of the *Spec2000* benchmarks were tested in an 2-way SMT simulator. Each paring was run for one billion total retired instructions. Detailed simulation was maintained throughout the execution and program progress was captured every one thousand total completed instructions. This provides very fine-grained progress data of each of the threads. The data was then broken up into periods of one million instructions. Each of these was then broken up into shorter periods of one thousand, ten thousand, and one hundred thousand instructions. The standard deviation of the IPC within each of the samples within the one million instruction periods was measured and averaged across all periods and each benchmark in each pairing. The results are shown in Figure 1. Because the benchmark *181.mcf* has a very low IPC, the percent standard deviation is very high, even tough the absolute standard deviation is small. Because of this, the data is shown with and without *181.mcf* included. The data shows that the IPC of each thread can vary significantly within a period. Because of this, longer detailed simulation periods are necessary in multithreaded systems, which is discussed in the next section.

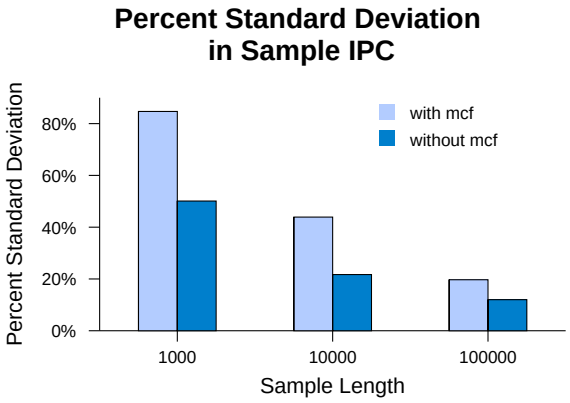


**Figure 1. Percent percent standard deviation for different sample sizes within a 1 million operation period.**

### 3.2 Experimental Results on Multithreaded Hardware

In order to further explore the effects of sampling, several pairings of *SPEC* benchmarks were run the Pentium4 Northwood (P4) processor with Hyperthreading. The performance counters on the P4 were sampled at each scheduling interval under a Linux operating system, or approximately every 2.5 billion clock cycles. Each paring was run several times with different offsets in start times in order to more fully characterize the performance of the threads in as many co-phases as possible. With this data it was possible to test the effects of different sampling periods would have on the characterization of a system. More importantly, it served as a fast proof of concept that sampling can be used to characterize a multithreaded system as long as the samples were kept reasonably short. An additional advantage of using hardware is that it eliminates simulation inaccuracies so that the effect of sampling can be tested in isolation. The methodology for this test was very simple. The IPC of each of threads in the Hyperthreading system are calculated based on the sample performance counter data. The operation count of each thread was incremented based on the IPC data and the sample rate. From this new point, the IPC was calculated based on the performance data closest to the relative positions of the threads and the process was repeated. This was done until one of the threads finished it's execution. This is illustrated in Figure 2. The x-axis in the figure represents the number of instructions completed in the benchmark *188.ammp* and the y-axis the instructions completed in *252.eon*. The lines indicate the relative progress for various sampling rates. For sampling rates up to ten

million instructions, the sampling has little effect on the relative progress of the threads. However, if the sampling rate is increased to once per one hundred million instructions, the path of the simulation becomes very different.

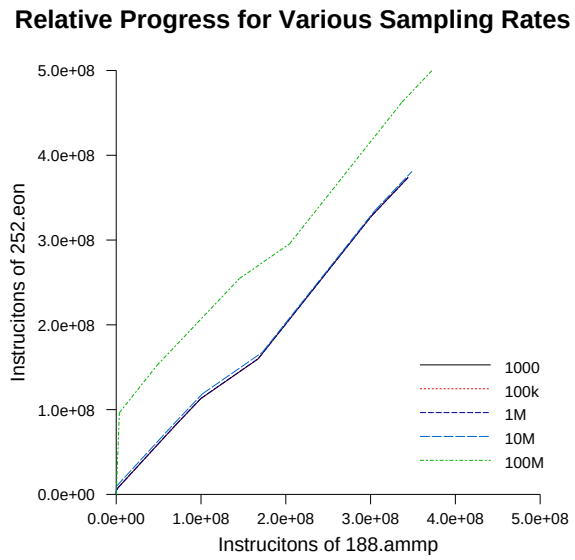


**Figure 2. relative progress of *188.ammp* and *252.eon* for various sampling rates.**

This difference in simulated path will lead to different co-phases being simulated. In right panel of Figure 3 the number of co-phases encountered is plotted for different sampling rates for the 15 benchmark pairings which were tested. Since each thread was divided into ten co-phases, a total of 100 co-phases may have been encountered on a given run. Because some program phases are very short, very rare, or both, it is unlikely all would be encountered on a single run. From the figure, it can be seen that the number of co-phases encountered stays steady for most pairings with sampling rates as high as once per million, or even once per ten million operations. However, any sampling rate above that starts to experience a significant drop in number of co-phases encountered. It is also important to see the difference in co-phase make-up of the run as illustrated in the center panel of Figure 3. The difference in co-phase makeup is defined as the sum across all co-phases of the absolute difference in the percent of execution spent in that co-phase between the two runs (this number is then divided by two in order to yield a percentage). Again in this graph, sampling rates of once per one million instructions and once per ten million instructions show little error for all but a few of the pairings.

Perhaps most important is the end result of the simulation. The left panel of Figure 3 illustrates the error in calculated IPC over the entire simulated run. In this figure, the error in IPC is negligible for a sampling rate of one million instructions and for all of the pairings and for all pairings except *eon-mcf* for once per ten million instructions. These are two of the slowest running benchmarks in terms of IPC so the sample rate is actually the slowest, in terms of cycles between samples and even a small absolute error will be large in terms of percentage. This demonstrates that sampling can be utilized with only minimal loss of accuracy in these idealized circumstances.

## 4 Simulation

The most important step in the simulation process is the actual detailed simulation. Because the results of a small, detailed simulation are extrapolated over the course of longer sampling period, it is vital that those results be as accurate as possible. In a single thread simulation, the length of a detailed simulation is a simple matter of a number of operations or cycles. Since threads run at different rates, the number of instructions is typically chosen. In a multithreaded simulation, it is slightly more complicated to choose the length of a period. Since the performance of each thread is needed to determine the length of the proportional fast forwarding, the IPC of each thread must be carefully determined. Because there is often a large disparity in the speeds of co-scheduled threads, simulation periods last until the slowest thread has completed a minimum number of operations. This can increase the length of the simulation significantly, but is necessary to characterize the system.

To test the effect of detailed simulation length all possible 1 million instruction periods in the the data from Section 3 were found. The IPC of each interval was found. Starting from the beginning of the interval, the cumulative IPC was found up to 25,000 instructions. This IPC was then extrapolated to one million instructions and compared to the actual measured value. This was done to model what a detailed simulation followed by a functional simulation fast-forwarding would do: measure performance for a short period, then extrapolate over the remainder of the sampling period. The average percent error across all intervals of all benchmarks in each pairing is plotted against modeled detailed simulation length in Figure 4. As would be expected, the error decreases with longer samples (the low error in the first few samples is due to the small number of very short detailed simulation intervals available). From this data, it follows that longer periods of detailed simulation will yield better results at the cost of slower simulation time.

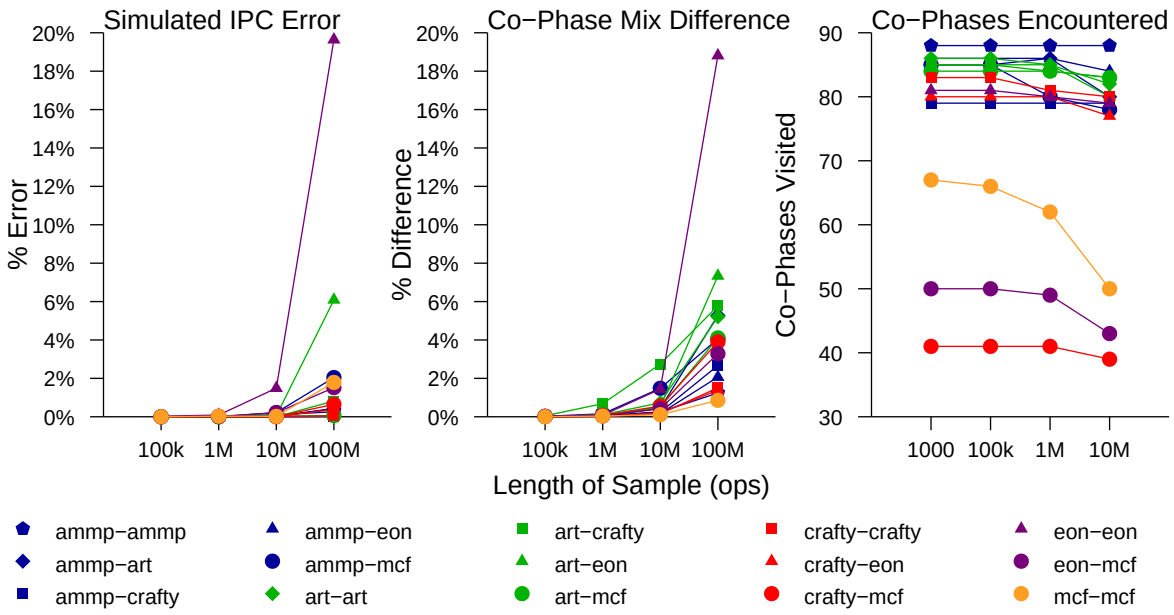


**Figure 3. Effects of ideal sampling on measured IPC, co-phase composition, and co-phase coverage.**

## 5 Functional Warm-up

At the beginning of a detailed simulation interval following a period of functional simulation, the processor state must approximate the state that would have occurred had full detailed simulation occurred throughout the sample period. However, maintaining accurate simulation state is the costly part of detailed simulation so what ever state is maintained during fast-forwarding comes with a time cost. Since fast-forwarding makes up the vast majority of the simulation and therefore the simulation time, anything that slows down fast-forwarding will cause significant slowdown in overall execution. With this in mind, only minimal simulation state should be maintained during fast-forwarding.

The ideal case is if no simulation state beyond the PC and number of instructions skipped is maintained. This requires that some warm-up period is used to create a viable simulation state when detailed simulation resumes. If this is not done, the detailed simulation will be very inaccurate for several reasons. Most importantly is that if the cache hierarchy is not kept warm, extraneous cache misses will occur during detailed simulation that would have been hits had cache state been maintained. The unfortunate factor in this is that cache state has a long lifetime. In large, lower level caches, data can have a lifetime of many thousands of cycles. Another important system with a long lifetime is the dynamic branch predictors. Warming up a single thread simulation is a relatively straight forward process as appropriate cache blocks and branch behaviors are tracked during the warm-up phase. The complication in multithreaded simulations is that these resources take up large amounts of physical hardware and, as a result, are often shared between threads. Since most associative caches have replacement policies based on least recently used (LRU) or pseudo-LRU information, timing of cache accesses between threads is also important, as it determines how much data from each thread is resident in the cache, called the *cache affinity*. Timing information is also vital in shared dynamic branch predictors because branch histories are traced and used to make predictions. Keeping detailed timing information as to when requests are made, however, requires something very close, and hence nearly as slow, as detailed simulation. further complicating this issue is that since the cache and branch predictors are cold at the beginning of the warm-up period, it is impossible to produce accurate detailed timing information. For example, if one thread makes a large number of cache requests, it will be artificially delayed versus the other thread in the system because of cold cache misses.

It is vital for accuracy, to keep track of which blocks from each thread are in the cache. In Figure 5 the distribution of affinity changes over a one million instruction window is shown. Each cache level was broken up into smaller regions of 4 sets each. The cache affinity of each thread was measured at the beginning and at the end of a one mil-

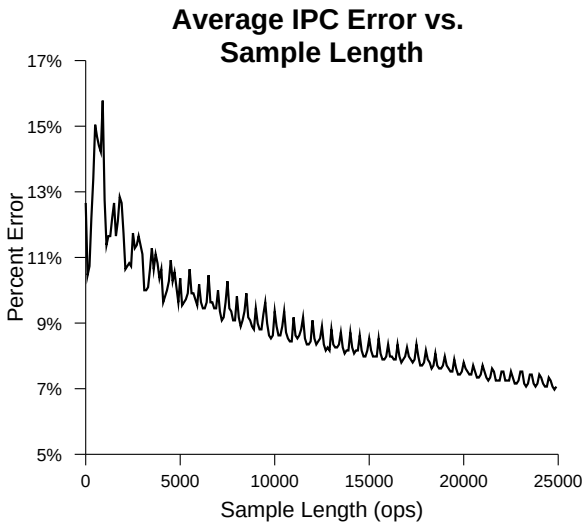


**Figure 4. Percent error in projected versus measured simulated IPC of 1 million operation segments for various length detailed simulation samples in a 2-way SMT system.**

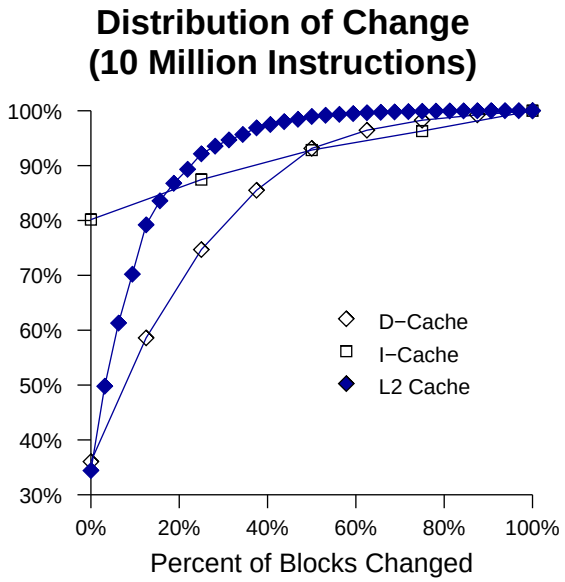


**Figure 5. Cumulative distribution of cache affinity changes over a one-million-instruction SMT window.**

lion instruction window. The graph show the distribution the absolute difference in cache affinity before and after the period. For a significant number of the samples, the cache affinity changed more than 20% especially in the lower level caches meaning that the cache affinity must be modeled in some way during warm-up.

In order to interleave instructions, and approximate timing information with minimal overhead, from multiple threads during warm-up a system called *Monte Carlo warming* was developed. The system randomly interleaves instructions from the threads being simulated based on their IPC from the last detailed simulation period. The setup of the warm-up requires two steps. The first step is the find the total IPC of the system from the last detailed simulation period and the ratio of the IPC of each thread to that total. Next, a random number generator with a even probability distribution over some range is need. Each thread is assigned some subrange of that range, with the size of the subrange proportional to the portion of the total IPC for which that thread is responsible. The actual warm-up occurs in a loop where each iteration a random number is generated, it is determined which thread's subrange it falls in, and a instruction from that thread is executed. If the instruction is neither a memory operation or a branch only functional state is updated, just as it would be in full fast-forwarding. If the instruction is a branch, the branch prediction tables are updated and the simulator PC is updated accordingly. This neglects the effects of speculative instructions after mispredicted branches, but since only cache and branch history state are being tracked, this effect is minimal. If the instruction is a memory instruction, a cache request is made. The cache simulator used in this work has a *quick* mode where no time information is kept. During detailed simulation, the cache state is updated each cycle as cache requests propagate between cache levels. In quick mode, cache requests propagate instantaneously. If the quick cache request is a miss, it is immediately propagated to the next level of the cache and the new block is brought in. Although this obviously sacrifices accuracy of cache, it allows the simulation to progress very quickly.

The next problem is determining how long to warm up the cache. Typically, this is determined experimentally through trial and error as to what warm-up is necessary for accurate results. However, in [7] it was shown that by monitoring the caches during warm-up, the warm-up period can be minimized without sacrificing accuracy. The system works by tracking cache accesses for instances where a cache miss occurs, but the cache block replaced has not been touched since the beginning of the warm-up period. This is called a *cold miss* Since the replaced block could have been holding the data the request was after, this may not have been a if cache state had been maintained. By tracking how many old misses occur, a simulator can deter-

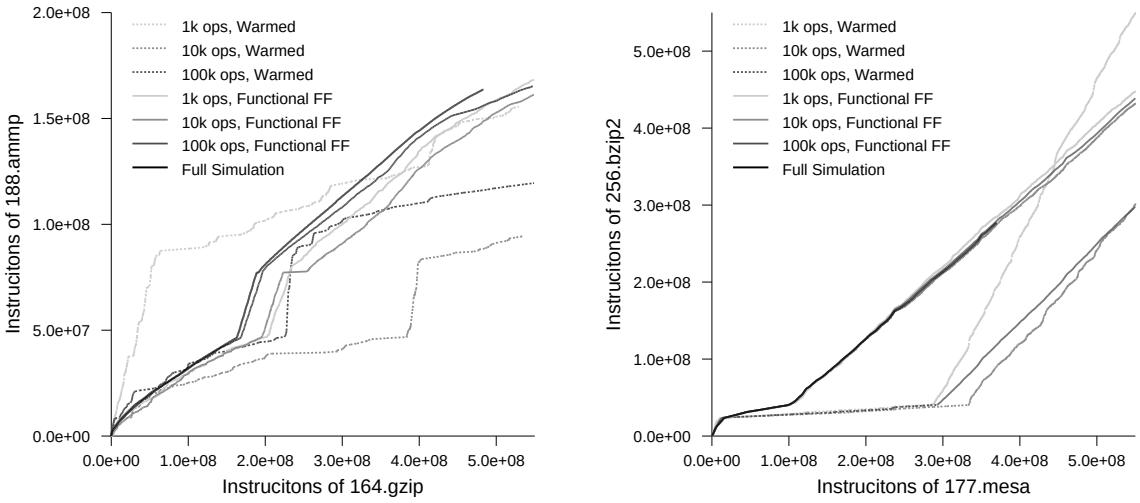


**Figure 6. Relative Progress of two benchmark parings for full detailed simulation and various warm-up schemes.**

mine when the cache system is sufficiently warm. In this work, cold miss history is maintained using a 32-bit vector. Each bit represents one access to the cache. If no cold misses have occurred in the last 32 accesses to any level of the cache hierarchy, the system is considered warm.

An alternative, which is used in SMARTS, is to simply keep the caches and branch predictors warm throughout fast-forwarding. This means that cache and branch predictor state are always up to date, except for the approximations made on timing for the sake of speed. By using functional fast forwarding instead of instead of a full fast-forwarding incurred an overhead of 16% on simulation time. However, simulation accuracy was greatly increased. This is due in part to the fact that our 32-request history is probably not sufficiently long to get a full picture a of warming, and experimenting with longer warm-up periods is part of on-going research.

The increase in accuracy from functional fast-forwarding can be seen in Figure 6. The relative progress of two pairings of *Spec2000* benchmarks are shown both using full detailed simulation without skipping (the black line) and with several sampling schemes. Each sampling scheme had a sample period of 1 million operations, with each thread required to complete at least 500 thousand. Detailed simulation intervals of at least 1 thousand (lightest), 10 thousand, and 100 thousand instructions (darkest gray lines) were tested with both functional fast-forwarding (solid lines) and adaptive warming (dotted lines). The intervals are described as "at least" because each thread is required to complete at least half the nominal number of instructions of the sample length. Although more detailed simulation definitely improves accuracy, the functional fast-forwarding makes a dramatic improvement on how well the data sampling tracks the full detailed simulation. This is further demonstrated in Figure 7. For this graph, the ratio of instructions executed in each thread was measured for each run. The average percent error over all of the pairings between the full simulation and the sampled runs is shown. The accuracy advantage in functional fast-forwarding is clear.

## 6 Conclusion

Modern processors are increasingly dependent on simultaneously running multiple threads for maintaining high throughput. The disadvantage of these systems is the exponential growth in the simulation space needed to fully characterize them. Compounding this problem, efficient methods for simulating these systems are in their infancy. This work has presented methodologies for applying statistical simulation techniques along the lines of SMARTS ([14]) to multithreaded and multicore simulation. The techniques of *proportional fast-forwarding* and *Monte Carlo Warm-up* have been introduced as integral parts of efficient and accurate multithreaded simulation.

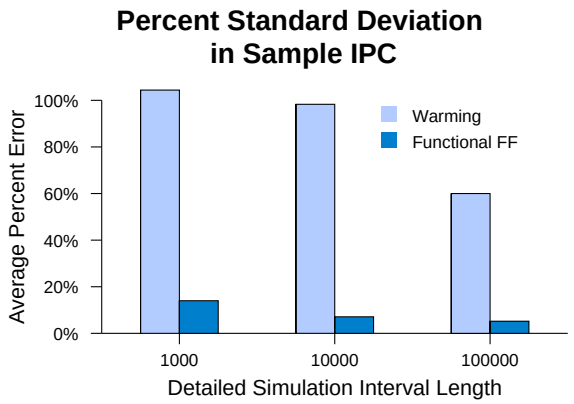


**Figure 7. Average percent error in number of instructions executed from each benchmark across all benchmark pairings for various sampling schemes.**

## References


[1] A. Agarwal, J. Kubiatowicz, D. Kranz, B. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, 1993.

[2] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, 2000.

[3] Intel Corporation. Special issue on intel hyperthreading in pentium-4 processors. *Intel Technology Journal*, 1(1), January 2002.

[4] J. John W. Haskins and K. Skadron. Accelerated warmup for sampled microarchitecture simulation. *ACM Trans. Archit. Code Optim.*, 2(1):78–108, 2005.

[5] R. N. Kalla, B. Sinharoy, and J. M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[6] J. Kihm, T. Moseley, and D. Connors. A mathematical model for balancing co-phase effects in simulated multithreaded systems. In *Proceedings of the 1st Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, 2005.

[7] Y. Luo, L. K. John, and L. Eeckhout. Self-monitored adaptive cache warm-up for microprocessor simulation. In *SBAC-PAD*, pages 10–17. IEEE Computer Society, 2004.

[8] K. Olukotun, B. A. Nayfeh, L. Hammond, K. G. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11, 1996.

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

[10] Standard Performance Evaluation Corporation. The SPEC CPU 2000 benchmark suite, 2000.

[11] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *The Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, pages 392–403, 1995.

[12] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proceedings of The 37th Annual International Symposium on Microarchitecture (MICRO-37 2004), 4-8 December 2004, Portland, OR, USA*, pages 183–194. IEEE Computer Society, 2004.

[13] M. VanBeisbrouk, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004.

[14] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *The Proceedings of the 30th International Symposium on Computer Architecture (ISCA 2003), 9-11 June 2003, San Diego, California, USA*, pages 84–95, 2003.

[15] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *The Proceedings of the 11th International Conference on High-Performance Computer Architecture (HPCA-11 2005), 12-16 February 2005, San Francisco, CA, USA*, pages 266–277. IEEE Computer Society, 2005.