Hybrid Techniques for Fast Multicore Simulation*

Manu Shantharam, Padma Raghavan, and Mahmut Kandemir

Department of Computer Science & Engineering,
Pennsylvania State University, University Park, PA 16802, USA
{shanthar,raghavan,kandemir}@cse.psu.edu

Abstract. One of the challenges in the design of multicore architectures concerns the fast evaluation of hardware design-tradeoffs using simulation techniques. Simulation tools for multicore architectures tend to have long execution times that grow linearly with the number of cores simulated. In this paper, we present two hybrid techniques for fast and accurate multicore simulation. Our first method, the Monte Carlo Co-Simulation (MCCS) scheme, considers application phases, and within each phase, interleaves a Monte Carlo modeling scheme with a traditional simulator, such as Simics. Our second method, the Curve Fitting Based Simulation (CFBS) scheme, is tailored to evaluate the behavior of applications with multiple iterations, such as scientific applications that have consistent cycles per instruction (CPI) behavior within a subroutine over different iterations. In our CFBS method, we represent the CPI profile of a subroutine as a signature using curve fitting and represent the entire application execution as a set of signatures to predict performance metrics. Our results indicate that MCCS can reduce simulation time by as much as a factor of 2.37, with a speedup of 1.77 on average compared to Simics. We also observe that CFBS can reduce simulation time by as much as a factor of 13.6, with a speedup of 6.24 on average. The observed average relative errors in CPI compared to Simics are 32% for MCCS and significantly lower, at 2%, for CFBS.

1 Introduction

Software simulation models have been used to evaluate the performance of computer hardware for over two decades due to low prototyping costs [3, 5, 7, 10]. With a recent shift in processor design to multicores [1, 2, 17, 20, 21], as a result of technological advances and packaging limits, existing software simulation models for single core processors [3, 16] have been expanded to simulate multicore environments [5, 9, 10, 13, 19]. However, the current generation multicore simulators are slow [11] and the performance degrades as the number of simulated cores increases. This performance degradation is especially visible when large multithreaded benchmarks are used. The execution of these codes on modern multicore simulators is so slow that researchers report experimental results based on partial runs that are not representative of the execution of the entire benchmarks [11, 14]. Hence, there is a need for alternate software based simulation techniques for fast and accurate multicore architecture exploration.

* This work was funded in part through grants 0720749, 0444345, 0811687, 0720645, and 0702519 from the National Science Foundation.

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 122–134, 2009.

In this paper, we propose two hybrid techniques for fast and accurate multicore simulation. Our first method, called the MCCS (Monte Carlo Co-Simulation), divides application execution into phases with each phase representing a subroutine, or a set of subroutines. Each phase is partially run using Simics [10] and partially using a Monte Carlo predictive model. Our proposed MCCS scheme reduces the simulation time by a factor of 1.77 on average, with average relative error in CPI (cycles per instruction) of 32% compared to Simics. Our second method, CFBS (Curve Fitting Based Simulation), is based on the observation that most scientific applications have multiple iterations and their behavior is repetitive over various iterations. Thus, we represent the CPI profile of a subroutine as a *signature* using curve fitting and represent the entire application execution as a set of signatures to predict performance metrics. Our results indicate that the CFBS scheme reduces the simulation time by a factor of 6.24 on average, with average relative error in CPI of 2% compared to Simics.

We would like to observe that our proposed methods, MCCS and CFBS, are different qualitatively from the previously proposed techniques like Simflex [13] and Simpoint [4] that help reduce the actual simulation time. Section 7 gives a detailed comparison between our methods and these techniques.

The reminder of this paper is organized as follows. In Section 2, we discuss our experimental setup and the tools we use. In Sections 3 and 4, we present the proposed MCCS and CFBS schemes respectively. In Section 5, we present experimental results and in Section 6, we report on a sensitivity study. In Section 7, we describe related work, and in Section 8, we conclude this paper with a summary of our major results.

2 Experimental Setup

We use our schemes, MCCS and CFBS, in conjunction with a full system simulator. For our experiments, Simics [10], a full system simulator, is configured as a multicore processor to obtain the necessary inputs for our models. For our base system, we configure Simics to simulate a four-core UltraSPARC-III architecture [6] with a 64 KB private level 1 (L1) cache per core, and a 4MB shared level 2 (L2) cache. Table 2 lists the various parameters used in our base configuration and their values. Additionally, in our base configuration, we use a window size of 10 million instructions for the CPI calculation.

Table 1. Benchmark description. NAS: FT, BT, CG; SPLASH2: Barnes, Ocean, Raytrace.

FT	Computational kernel of 3-D FFT-based spectral method. Performs three 1-D FFT, one for each dimension.
BT	Simulates CFD application that uses an implicit algorithm to solve 3-D compressible Navier-Stokes equations.
CG	Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix.
Barnes	Simulates the interaction of a system of bodies in 3-D over a number of time -steps using Barnes-Hut N-body method.
Ocean	Simulates large-scale ocean movements based on eddy and boundary currents.
Raytrace	Renders a 3-D scene onto a 2-D image plane using optimized ray tracing.

We use a subset of NAS OMP [18] and SPLASH2 [12] benchmark suites with varying workload characteristics. In our experiments, we use three NAS parallel OMP benchmarks: FT, BT and CG, each with problem size 'A' and three SPLASH2 benchmarks: Barnes, Raytrace and Ocean. A brief description of these benchmarks is included in Table 1. The current benchmark selection is based on the ease of identifying the phases in the code manually.

3 Monte Carlo Co-Simulation (MCCS)

In this section, we discuss MCCS which uses a Monte Carlo predictive model in conjunction with Simics to predict performance metrics of scientific applications. In Section 3.1, we describe our Monte Carlo predictive model. In Section 3.2, we introduce the concept of *windowed CPI* that is useful in understanding the behavior of our target applications and finally, in Section 3.3, we describe the simulation flow of our MCCS scheme.

3.1 Monte Carlo Predictive Model

We model processor cores and the memory subsystem, and consider four basic types of instructions - load, store, floating point and "other". The "other" instruction type includes all instructions that are not loads, stores or floats. We further classify a *load* instruction as *load L1 hit, load L2 hit* or *load memory bound*. Thus, any application execution can be represented as a mix of our six instruction types (floating point, store, load L1 hit, load L2 hit, load memory bound, and "other"). The processor is modeled as a unit that issues these six types of instructions based on their probability of occurrence. The input parameters to our model are the total number of instructions to be issued and the histograms representing the instruction mix. The output of our model is a CPI distribution. Figure 1 shows a pictorial representation of our model for a two-core processor, where p1, p2, p3 are the probabilities that a load instruction is satisfied in L1 cache, in L2 cache, or in memory, respectively, and p0 is the probability that the instruction is a

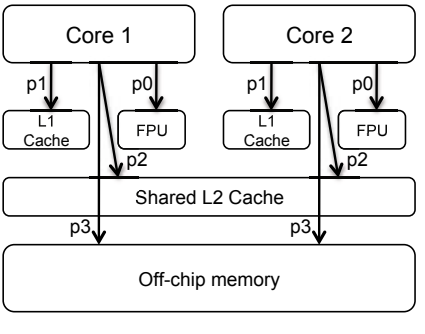

Fig. 1. Monte Carlo predictive model for a two-core processor

Table 2. Parameters used in our base configuration

L1 cache	private, 64KB 2 way assoc
L2 cache	shared, 4MB 16 way assoc
L1 cache latency	3 cycles
L2 cache latency	10 cycles
Off-chip memory latency	260 cycles

floating point instruction[1]. We use Simics to obtain the application instruction mix in terms of our six instruction types, and to calculate the probability of their occurrence. Note that Figure 1 is more of an instruction flow diagram based on certain probabilities and it does not represent the actual hardware.

3.2 Understanding Application Performance Using Windowed CPI

Performance is generally measured in terms of average CPI or average IPC of an application. However, these metrics do not provide a detailed insight into application's performance behavior over time, i.e., these metrics do not reveal if the application performance is uniform or variable over time. In order to gain insight into time-dependent performance behavior of applications, we divide its execution time into windows, where a window comprises a fixed number of instructions, for example, 10 million instructions. The CPI for a window is called as *windowed CPI* and unless stated otherwise, CPI in this paper refers to *windowed CPI*.

3.3 MCCS Simulation Flow

In this section, we describe the simulation flow of our MCCS method. We observe that the CPI behavior of an application varies significantly across subroutines and there are intervals of execution within a subroutine, wherein the CPI behavior is uniform. Based on these observations, we divide the application's execution into phases, each phase representing a subroutine or a set a subroutines. A set of subroutines is represented as a phase if the execution time of individual subroutines is very short. At present, we are using an offline phase detection method wherein we manually identify the beginning and the end of subroutines (phases) using Simics *breakpoints*.

We integrate our Monte Carlo predictive model with Simics to take advantage of these phases, and call this integrated method the MCCS. Each application phase is simulated with Simics and Monte Carlo model in an interleaved manner. For each phase, simulation starts with the Simics timing model. After executing a fixed number of instructions, we switch to timing model of our Monte Carlo predictive technique. During the Simics run, we construct histograms representing the probabilities of occurrence of our instruction types for every *window* within a phase. The histograms capture the phase-wise behavior of the application during the Simics execution. We use these histograms in our Monte Carlo predictive model to predict the latencies of our instruction types. While in the Monte Carlo predictive mode, Simics is executed in background without its timing model ("Simics ff"). Because of "Simics ff" mode, Simics does not capture any latencies caused by memory subsystem. We account for these latencies in our Monte Carlo predictive model. We use Simics in "Simics ff" mode to ensure correctness of benchmark execution.

In our experiments, we interleave the executions of Simics and our model twice within a subroutine so that we capture the heterogeneous behavior, i.e., the intervals with uniform CPI behavior within a subroutine. A simple execution flow diagram for this model is shown in Figure 2 where there is a single instance of interleaving of our

[1] Note that since we target scientific applications, we consider only floating-point data. Our strategy can easily be extended to other data types as well.

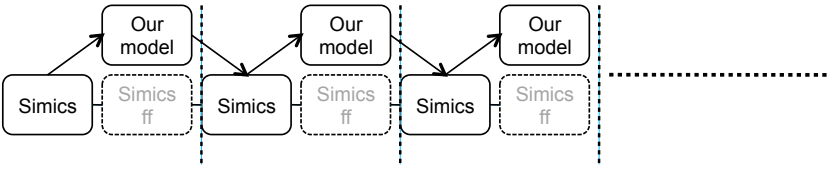

Fig. 2. MCCS simulation flow. The vertical dashed lines represents phase boundaries, "Simics ff" represents Simics's fast forward execution mode.

model with Simics. The expected accuracy and speed of MCCS is dependent on the number of *interleavings* within a phase. The more the interleavings, the better would be the accuracy and slower would be the simulation. We have fixed the number of interleavings to two per phase so that we could tradeoff between speed and accuracy.

4 Curve Fitting Based Simulation (CFBS)

In this section, we introduce our second hybrid simulation technique, referred to as the CFBS. This technique is tailored to evaluate the behavior of scientific applications on multicores. We observe that scientific applications have multiple iterations of the same code sequences. For example, consider Figure 3 which gives the pseudo code for the NAS FT benchmark. Notice that *fft* subroutine is called repeatedly over different iterations and *fft* subroutine in turn calls subroutines *cffts1, cffts2, cffts3*.

```
do iter = 1, niter
  call evolve(u0, u1, …)
  call fft(-1, u1, u1)
  call checksum(iter, u1, …)
end do
```

```
subroutine fft(dir, x1, x2)
   .
   .
call cffts3(-1, dims(1), dims(2), dims(3), …)
call cffts2(-1, dims(1), dims(2), dims(3), …)
call cffts1(-1, dims(1), dims(2), dims(3), …)
   .
   .
 end
```

Fig. 3. Left: Code segment of FT benchmark showing fft subroutine being called multiple times within a loop. **Right:** Code segment of FT benchmark showing fft subroutine.

A detailed analysis of application behavior shows that the CPI behavior of our target applications is repetitive over their entire execution. Consider as an example Figure 4, which shows the CPI behavior of all iterations of the *cffts3* subroutine of NAS FT benchmark. Observe that the CPI behavior of *cffts3* subroutine over different iterations exhibit similar CPI behavior. This repetitive behavior is due to the same memory access pattern and similar cache behavior of this subroutine over different iterations. We use this property of subroutines to reduce the simulation time of scientific applications.

In the CFBS scheme, we first run Simics for one iteration of the application. During this run, we compute the windowed CPIs for each subroutine call and use a *curve fitting method* to fit these data points (windowed CPIs) per subroutine. There are different

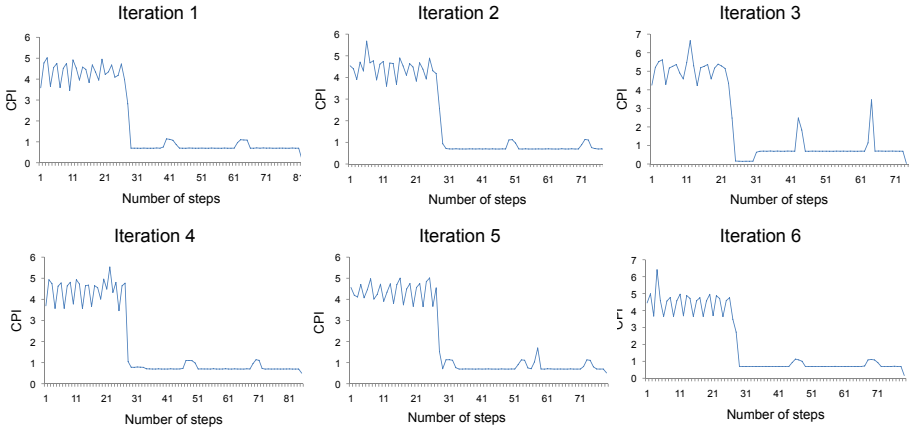

Fig. 4. CPI behavior of cffts3 function for all six iterations. We see that all iterations exhibit a similar behavior.

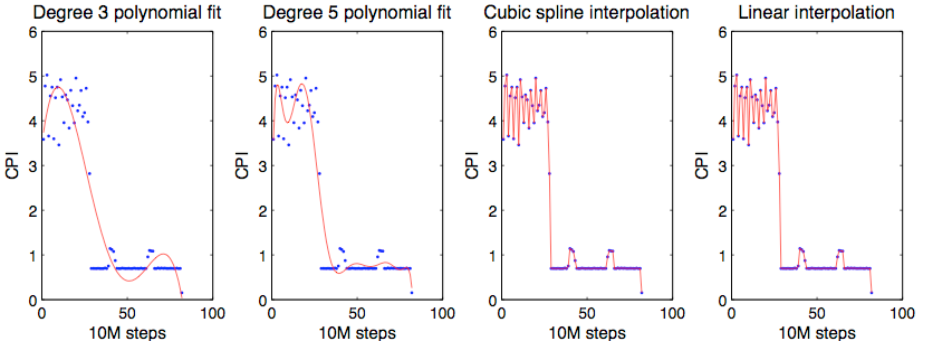

Fig. 5. Curve fitting techniques

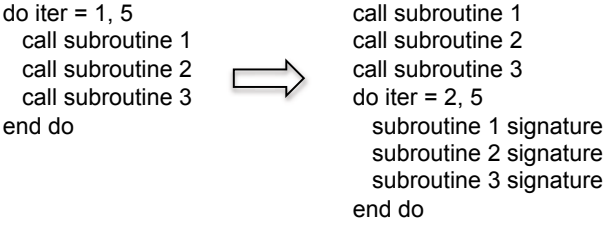

Fig. 6. Left: Actual code representation. **Right:** Representation in CFBS model.

ways in which we can fit a curve to the given data [15]. Figure 5 illustrates four curve fitting techniques, a curve fit with polynomial of degree 3 ($0.0004X^3 - 0.0212X^2 + 0.3057X + 3.4483$), a curve fit with polynomial of degree 5 ($0.0002X^5 - 0.006X^4 + 0.0883X^3 - 0.6928X^2 + 2.4728X + 1.7$), a cubic spline interpolation, and a linear

interpolation, applied to data points of FT application. We observe *linear spline interpolation* and *cubic spline interpolation* fit our data better than the polynomial curve fitting techniques. Since the linear spline interpolation technique is simpler than cubic spline interpolation technique, we use linear interpolation for all our experiments. The fitted curve represents the CPI profile of that subroutine, we call this as a *subroutine signature*. For the subsequent iterations of the application, we use the subroutine signature of each subroutine to represent its execution. As a result, we are able to reduce the simulation time of the application by reducing the number of iterations to be simulated using Simics while accurately predicting the CPI behavior. Figure 6 illustrates a simple example of how the subroutine calls are represented in the original application and in our CFBS model.

5 Experimental Evaluation

In this section, we present our experimental results. Consider Figure 7, which shows the CPI behavior of FT benchmark on a four-core processor configuration. The left plot shows the CPI metric as reported by Simics, the middle plot shows the CPI metric as reported by MCCS, and the right plot shows the CPI metric as reported by CFBS. We observe that the CPI behavior of CFBS is much closer to Simics than MCCS. In Figure 13, we show the average relative error in CPI for MCCS and CFBS models compared to Simics. For FT benchmark, these errors are 90% and 2% respectively. In this case, the inaccuracy in MCCS can be attributed to lesser number of instructions simulated using Simics before switching to our Monte Carlo predictive model. Figure 8 shows results for Simics, MCCS and CFBS methods for BT benchmark. The average relative error in CPI for MCCS model is around 42%. The other observations and results for BT are similar to that of FT. Figures 9, 10, 11 and 12 show the CPI behavior reported by these models for CG, Barnes, Ocean and Raytrace benchmarks respectively. We can see from Figure 13 that apart from FT and BT, MCCS model's average relative error compared to Simics is around 20%. Overall, we see that the MCCS and CFBS schemes are 68% and 97.4% accurate on average. The higher accuracy of CFBS is due to the repetitive behavior of our target applications.

In Table 3, we list the simulation times, in minutes, for our benchmarks using Simics, MCCS, and CFBS. Note that under MCCS, values within the parenthesis are the simulation times of pure Simics and our Monte Carlo predictive model, respectively.

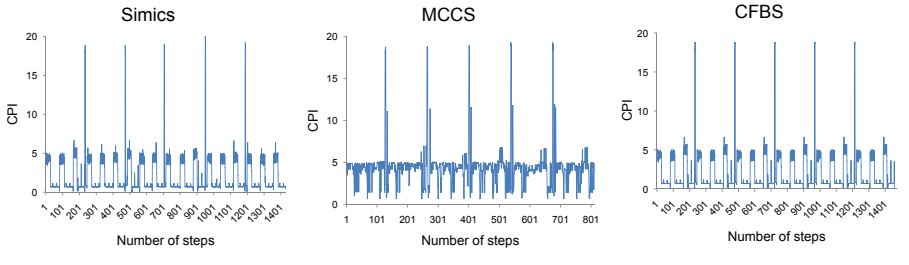

Fig. 7. CPI behavior of FT when run on a four-core processor

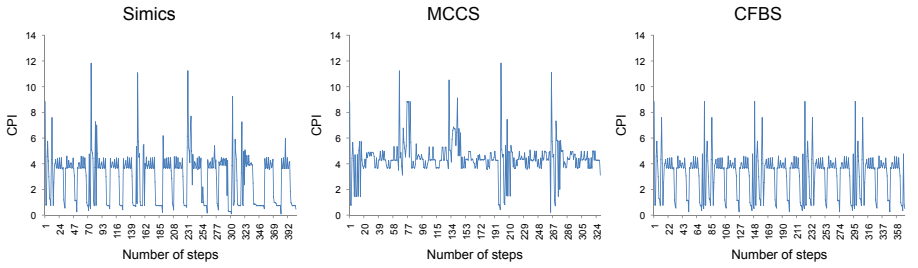

Fig. 8. CPI behavior of BT when run on a four-core processor

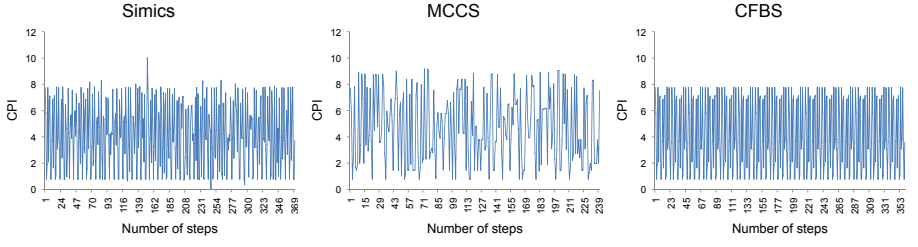

Fig. 9. CPI behavior of CG when run on a four-core processor

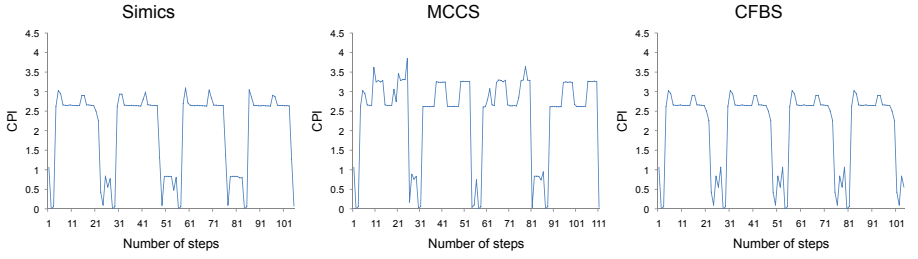

Fig. 10. CPI behavior of Barnes when run on a four-core processor

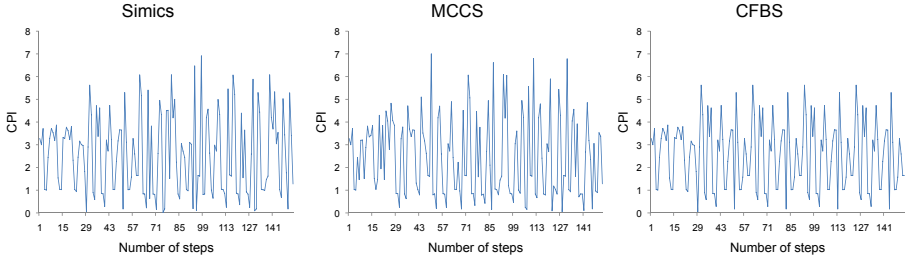

Fig. 11. CPI behavior of Ocean when run on a four-core processor

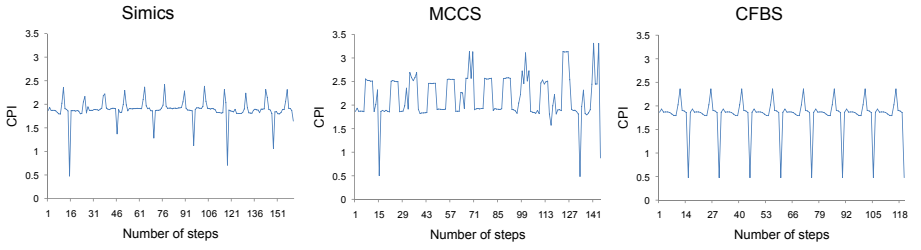

Fig. 12. CPI behavior of Raytrace when run on a four-core processor

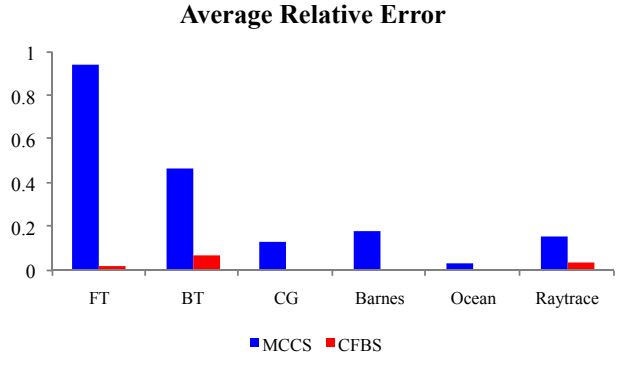

Fig. 13. Average relative error of benchmarks with respect to Simics

We see from Table 3 that CFBS takes the least time to simulate these benchmarks. We also observe that simulation time of MCCS is dominated by the execution within Simics. For CFBS, the simulation time is calculated as the time required to run one iteration of the application using Simics, as the rest of the simulation is done using curve fitting. In Table 4, we report the speedup of our techniques over Simics. We observe that CFBS model reduces the simulation time by as much as 13.6 times and 6.42 times on average while MCCS model reduces the simulation time by as much as 2.37 times and 1.77 times on average.

Table 3. Time taken by simulators in minutes

Benchmark	Simics	MCCS	CFBS
FT	1,245	524 (470 + 54)	190
BT	530	320 (280 + 40)	110
CG	410	176 (160 + 16)	30
Barnes	105	74 (70 + 4)	25
Ocean	113	100 (98 + 2)	55
Raytrace	170	104 (80 + 24)	25

Table 4. Speedup with respect to Simics

Benchmark	MCCS	CFBS
FT	2.37	6.55
BT	1.65	4.8
CG	2.32	13.6
Barnes	1.41	4.2
Ocean	1.13	2.05
Raytrace	1.63	6.8

6 Sensitivity Study

In this section, we conduct a sensitivity study, on the impact of the number of cores and the instruction window size, on the accuracy of the CPI prediction. In our first study, we consider the impact of changing a four-core base processor configuration to a one-core processor configuration and an eight-core processor configuration. In our second study, we increase the instruction window size to 20 million instructions and observe its impact on prediction accuracy.

For our first sensitivity study, we simulate FT and Raytrace benchmarks on one-core and eight-core processor configurations. Consider Figure 14, the plots on the left and the center show the average relative error in CPI prediction for these benchmarks for a one-core and eight-core processor configurations respectively. We observe that CFBS method has a good prediction accuracy for both the benchmarks on one-core and eight-core processor configurations. Notice that the prediction accuracy of MCCS for a one-core processor is much higher than that for a eight-core processor for FT benchmark. We attribute this higher prediction accuracy to a more consistent CPI behavior of FT within its phases for a one-core processor. For FT, the accuracy of MCCS model is much better for one and eight-core processors when compared to a four-core processor. We believe this difference in accuracy is due to the way in which FT behaves on one, four and eight-core processors. The behavior of FT on one and eight-core processors is favorable for MCCS as we are able to easily capture the sub-phase CPI behavior. We note that by having more sub-phases (interleavings), we would be able to get better accuracy for four-core processors as well. The CFBS method has higher accuracy for one and eight-core processors for both FT and Raytrace as these applications have repetitive codes and our CFBS method is able to capture this repetitive behavior.

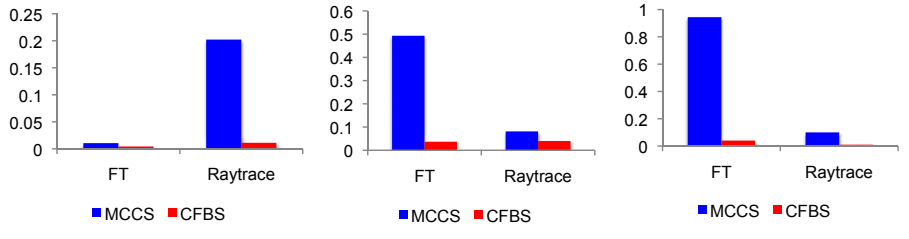

Fig. 14. Average relative error of benchmarks with respect to Simics for a **Left:** one-core processor. **Center:** eight-core processor. **Right:** four-core processor when instruction window size is set to 20 million instructions.

For our second sensitivity study, we change the size of instruction window from 10 million instructions to 20 million instructions. The rightmost bar-chart in Figure 14 shows the average relative error for FT and Raytrace benchmarks for a four-core processor configuration. We observe that there is not much impact on the CPI behavior prediction accuracy of our models by increasing the instruction window size. We can observe this by comparing the right plot in Figure 14 with Figure 13.

7 Related Work

For over a decade, computer architects have used software simulation techniques to evaluate the future computer hardware. SimpleScalar [3] was one of the earliest simulators used to simulate uniprocessor superscalar architectures. Some of the other flavors of uniprocessor simulators used are PTLsim [24], a cycle accurate x86 microprocessor simulator and SIMCA [16], a simulator for multithreaded computer architectures.

Recent advances in multicore architectures have led researchers to use simulators that can model multicore architecture. SESC [5], is a cycle accurate multicore simulator, while Simics [10], is a full system simulator that can be used to simulate multicores. GEMS [9] is a Simics based multiprocessor simulator with focus on accurate performance modeling. Due to the slow and complex nature of these cycle accurate and full system simulators, new statistical and analytical performance modeling techniques have emerged. Three such popular simulation techniques are SMARTS [11], Simflex [13], and Simpoint [4]. SMARTS framework applies statistical sampling to microarchitecture simulation. It samples and simulates a subset of benchmark's instructions to estimate the performance of the entire benchmark. Simflex use Simics to provide functional simulation and applies SMARTS methodology to do the sampling. It leverages Simics's checkpointing capability to store the state of simulation. A large checkpoint library is created by checkpointing at various points during the execution of the entire program in Simics. Such libraries are required for every $<$*simulated configuration, benchmark*$>$ pair. These libraries are sampled to find the checkpoints that needs to be simulated. Although Simflex is statistically rigorous and accurate, it is rigid in terms of simplicity of use and requires a lot of memory to store the checkpoint libraries. We believe that for good quick estimate of performance, our techniques, CFBS and MCCS are more suited as they are easy to implement and have no extra memory requirements. SimPoint uses offline algorithms (clustering techniques) to detect phases in a program. This classification helps to choose simulation points that is representative of the phases, thereby, reducing the overall simulation time. This approach is independent of the architecture on which the program is run. We believe that it would be a challenge to use SimPoint for multicore simulation of applications (like FT) that have different runtime behavior based on the number of cores on which it is run.

Other simulation techniques to speedup the simulation include, HLS [8], a hybrid simulator that uses statistical profile of applications to model instruction and data streams, and MonteSim [23, 22], a predictive Monte Carlo based performance model for in-order microarchitectures. However, these simulators were developed for uniprocessor architecture. Our methods differ from HLS in the way we profile applications, are generic and can be applied to multicores. Our predictive model is similar to [23] in some aspects like the use of Monte Carlo technique, however unlike MonteSim, we can use our methods to model multicore processors.

8 Conclusions and Future Work

We have presented two hybrid models, MCCS and CFBS, to address the challenge of fast evaluation of design-tradeoffs for multicore architectures. Our experimental analysis indicates that MCCS can reduce simulation time by as much as a factor of 2.37, with

a speedup of 1.77 on average compared to Simics. However, its average relative error is rather large at 32%. The results also reveal that CFBS can reduce simulation time by as much as a factor of 13.6, with a speedup of 6.24 on average. Additionally, the observed average relative error in CPI compared to Simics is significantly less at 2%.

Our results show that CFBS performs consistently better than MCCS in terms of both accuracy and speedup. One reason for this is our target application domain, namely, scientific applications. Since most of these applications have repetitive (iterative) codes, CFBS performs better as it is able to capture the entire phase behavior while MCCS only predicts the entire phase behavior based on partial phase results. Although MCCS performs worse in both speedup and accuracy, it has the potential of being more generic that CFBS model. For example, if the CPI behavior of a subroutine varies across different iterations, we believe that MCCS method would perform better than CFBS method. As part of future work, our initial plan is to investigate non-scientific applications to test the applicability of these methods.

References

1. From a few cores to many: A tera-scale computing research overview. Technical report, Intel
2. Teraflops research chip, `http://techresearch.intel.com/articles/Tera-Scale/1449.htm`
3. Austin, T., Larson, E., Ernst, D.: Simplescalar: An infrastructure for computer system modeling. Computer 35(2), 59–67 (2002)
4. Perelman, E., et al.: Using simpoint for accurate and efficient simulation. SIGMETRICS Perform. Eval. Rev. 31(1), 318–319 (2003)
5. Renau, J., et al.: SESC simulator (January 2005), `http://sesc.sourceforge.net`
6. Lauterbach, et al.: Ultrasparc-iii: a 3rd generation 64 b sparc microprocessor. In: ISSCC 2000. IEEE International on Solid-State Circuits Conference, 2000. Digest of Technical Papers., pp. 410–411 (2000)
7. Rosenblum, M., et al.: Complete computer system simulation: the simos approach. IEEE Parallel and Distributed Technology: Systems and Applications 3(4), 34–43 (Winter 1995)
8. Oskin, M., et al.: Hls: combining statistical and symbolic simulation to guide microprocessor designs. SIGARCH Comput. Archit. News 28(2), 71–82 (2000)
9. Martin, M.M.K., et al.: Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput. Archit. News 33(4), 92–99 (2005)
10. Magnusson, P.S., et al.: Simics: A full system simulation platform. Computer 35(2), 50–58 (2002)
11. Wunderlich, R.E., et al.: Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. SIGARCH Comput. Archit. News 31(2), 84–97 (2003)
12. Woo, S.C., et al.: The splash-2 programs: characterization and methodological considerations. In: ISCA 1995: Proceedings of the 22nd annual international symposium on Computer architecture, pp. 24–36 (1995)
13. Wenisch, T.F., et al.: Simflex: Statistical sampling of computer system simulation. IEEE Micro. 26(4), 18–31 (2006)
14. Sherwood, T., et al.: Automatically characterizing large scale program behavior. In: ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pp. 45–57. ACM, New York (2002)
15. Heath, M.T.: Scientific computing: An introductory survey (2002)

16. Huang, J., Lilja, D.: An efficient strategy for developing a simulator for a novel concurrent multithreaded processor architecture. In: MASCOTS 1998: Proceedings of the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Washington, DC, USA, p. 185. IEEE Computer Society, Los Alamitos (1998)
17. Kahle, J.: The cell processor architecture. In: MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, p. 3 (2005)
18. NASA. Nas benchmark suite, `http://www.nas.nasa.gov/Resources/Software/npb.html`
19. Pai, V.S., Ranganathan, P., Adve, S.V.: Rsim: Rice simulator for ilp multiprocessors. SIGARCH Comput. Archit. News 25(5), 1 (1997)
20. Intel Core Duo processor Frequently Asked Questions, `http://www.intel.com/support/processors/mobile/coreduo/sb/CS-022131.htm`
21. UltraSPARC T1 Niagara Specifications, `http://www.sun.com/processors/UltraSPARC-T1/specs.xml`
22. Srinivasan, R., Cook, J., Lubeck, O.: Ultra-fast cpu performance prediction: Extending the monte carlo approach. In: SBAC-PAD 2006: Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing, Washington, DC, USA, pp. 107–116. IEEE Computer Society, Los Alamitos (2006)
23. Srinivasan, R., Lubeck, O.: Montesim: a monte carlo performance model for in-order microachitectures. SIGARCH Comput. Archit. News 33(5), 75–80 (2005)
24. Yourst, M.T.: Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In: IEEE International Symposium on Performance Analysis of Systems Software, 2007. ISPASS 2007, April 2007, pp. 23–34 (2007)