The First Annual Workshop on Modeling, Benchmarking, and Simulation (MoBS-1)

Held in conjunction with the
32nd Annual International Symposium on Computer Architecture
(ISCA-32)

Saturday June 4, 2005

Madison, Wisconsin

Message from the Organizers

We are pleased to introduce to you the first edition of the Workshop on Modeling, Benchmarking and Simulation, or MoBS for short.

In response to the call for papers, we received 22 paper submissions. All submissions were reviewed by the program committee members and all papers received three reviews. After all of the reviews were finalized, the program committee discussed which papers to accept and reject through e-mail. Given the quality of the papers that were submitted, it was a difficult to decide on the final program. But in the end, the program committee decided to accept 10 papers. Our idea was to accept as many papers as possible in order to give the ability to researchers to present and get feedback on their on-going research. We hope that by doing so we will stimulate the interaction and discussion during the workshop.

In addition to these 10 regular papers, we also have a mini-tutorial on SimPoint 3.0. Since SimPoint is becoming a commonly-used tool, the program committee accepted this mini-tutorial to be part of the workshop. The mini-tutorial will highlight how SimPoint works, how to properly use SimPoint, and what are the most recent features in version 3.0.

This first edition of the MoBS workshop is held in conjunction with the 32th Annual International Symposium on Computer Architecture (ISCA-32). We therefore would like to thank the ISCA chairs for accepting this workshop as part of the ISCA program. This is greatly appreciated and we hope we will be able to organize this workshop again next year with ISCA.

We would also like to thank the program committee for their hard work and all of the authors for their excellent submissions.

We hope you will enjoy the workshop.

Joshua J. Yi and Lieven Eeckhout

Organizers and Chairs

Lieven Eeckhout
Joshua Yi

Program Committee

David I. August
Pradip Bose
Brad Calder
Lizy Kurian John
David J. Lilja
Peter Magnusson
Jim Smith

Reviewers

David I. August
Pradip Bose
Brad Calder
Jonathan Chang
Lieven Eeckhout
Daniel Gracia Perez
Greg Hamerly
Chris J. Hescott
Shiwen Hu
Lizy Kurian John
Sreekumar V. Kodakara
Byeong Kil Lee
David J. Lilja
Peter Magnusson
Drew C. Ness
Tyler Olsen
David A. Penry
Erez Perelman
Ram Rangan
George A. Reis
Resit Sendag
Jim Smith
Manish Vachharajani
Neil Vachharajani
Joshua Yi

Introduction 11:00 to 11:10

Workshop Organizers: Lieven Eeckhout and Joshua J. Yi

Session 1: Simulators and Modeling 11:10 to 12:25

A Mathematical Model for Accurately Balancing Co-Phase Effects in Simulated Multithreaded Systems

Joshua L. Kihm, *University of Colorado*
Tipp Moseley, *University of Colorado*
Daniel A. Connors, *University of Colorado*

FAST: A Functionally Accurate Simulation Tool Set for the Cyclops64 Cellular Architecture

Juan del Cuvillo, *University of Delaware*
Weirong Zhu, *University of Delaware*
Ziang Hu, *University of Delaware*
Guang R. Gao, *University of Delaware*

Rapid Development of a Flexible Validated Processor Model

David A. Penry, *Princeton University*
Manish Vachharajani, *University of Colorado*
David I. August, *Princeton University*

Session 2: Potpourri 2:00 to 3:15

TraceVis: An Execution Trace Visualization Tool

James Roberts, *NVIDIA*
Craig Zilles, *University of Illinois at Urbana-Champaign*

Evaluating Power Management Strategies for High Performance Memory (DRAM): A Case Study for Achieving Effective Analysis by Combining Simulation Platforms and Real-Hardware Performance Monitoring

Karthick Rajamani, *IBM Austin Research Lab*
Juan Rubio, *IBM Austin Research Lab*
Wael El-Essawy, *IBM Austin Research Lab*
Kartik Sudeep, *IBM Austin Research Lab*
Ram Rajamony, *IBM Austin Research Lab*

A Methodology for Stochastic Fault Injection in VLSI Designs

Christian J. Hescott, *University of Minnesota – Twin Cities*
Drew C. Ness, *University of Minnesota – Twin Cities*
David J. Lilja, *University of Minnesota – Twin Cities*

Session 3: Sampling and Benchmarking 4:00 to 5:40

IDDCA: A New Clustering Approach for Sampling
Daniel Gracia Pérez, *INRIA*
Hugues Berry, *INRIA*
Olivier Temam, *INRIA*

Sampling and Stability in TCP/IP Workloads
Lisa R. Hsu, *University of Michigan*
Ali G. Saidi, *University of Michigan*
Nathan L. Binkert, *University of Michigan*
Steven K. Reinhardt, *University of Michigan*

Using A Multiscale Approach to Characterize Workload Dynamics
Tao Li, *University of Florida*

The Case for Automatic Synthesis of Miniature Benchmarks
Robert H. Bell Jr., *IBM and University of Texas at Austin*
Lizy K. John, *University of Texas at Austin*

SimPoint Mini-Tutorial 5:50 to 6:35

Presenters: Brad Calder (UC San Diego), Greg Hamerly (Baylor)

SimPoint 3.0: Faster and More Flexible Program Analysis
Greg Hamerly, *Baylor*
Erez Perelman, *UC San Diego*
Jeremy Lau, *UC San Diego*
Brad Calder, *UC San Diego*

Closing Remarks 6:35 to 6:40

Workshop Organizers: Lieven Eeckhout and Joshua J. Yi

A Mathematical Model for Accurately Balancing Co-Phase Effects in Simulated Multithreaded Systems

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

Abstract

As various types of multithreaded architectures can increase resource utilization and throughput of modern processors, there exists a very large design space which must be explored to effectively characterize these systems. Since these systems execute instructions from multiple concurrent application threads, understanding the interactions between co-scheduled threads is critical to evaluating potential designs. Unfortunately, understanding the interaction of co-scheduled threads requires complex and computationally intensive simulation. Although methods for accelerating architecture simulation are effective for applications demonstrating phase behavior over time in single threaded architectures, multithreaded systems exhibit even more widely variant behavior as different phases of each thread interact. Overall, there are many simulation technology issues related to phase behavior that must be investigated to effectively explore future multithreaded architectures. This paper explores how differences in multithreaded phase interactions and their relative frequencies impact overall performance evaluation. Further, a mathematical model is presented that balances the effects of inter-thread phase interactions in simulation to more accurately reflect the likely behaviors encountered in a real system.

1. Introduction

Various types of multithreading, including Simultaneous Multithreading (SMT) [4], Fine-Grained Multithreading [1], Coarse-Grained Multithreading [17], and Chip Multiprocessors (CMP) [14], have been proposed and implemented. The common theme among all multithreading techniques is that instructions from multiple threads (typically from independent programs) are executed within the same small time interval. Co-execution masks long latency events such as complex ALU functions, branch mispredictions, and accesses to lower levels of the memory hierarchy by executing instructions from other threads. The common mechanism of these systems is that some processor resources are shared between threads. This sharing ranges from CMP which can share low-level caches to SMT where essentially every microarchitecture resource is shared. The downside of any type of multithreading is that the collective set of requests can overwhelm the capacity of certain shared resources, causing interference between the threads. This interference heavily influences the performance of multithreaded systems [13], complicating the evaluation of multithreaded architectures.

The design exploration space for future multithreaded architectures is vast and it is difficult to quickly identify critical and optimal design decisions. Candidate designs are first evaluated by constructing simulators; unfortunately, simulation of real workloads is a major bottleneck in the design process and inaccurate simulation models have been shown to be multiple degrees from the quality of the final design [2]. In general, cycle accurate simulation of a complex, modern processor entails, at a minimum, a several thousand-fold slowdown over hardware. To explore the space of microprocessor design various simulation techniques and models have emerged. A commonly used solution to this problem is to exploit program phase behavior [3, 6, 10, 12, 8] to eliminate unnecessary simulation time by finding key representative phases within applications. Almost all programs can, to some degree, be divided into regions of common behavior called phases. Phases are differentiated based on either code usage [3, 8, 11] or performance data [15]. For the purpose of this paper, a program is divided into equally sized time slices called *periods*. The set of periods with similar behavior is a *phase*. Finally, a set of consecutive periods with the same phase is an execution *interval*. These concepts are illustrated in Figure 1. In a multithreaded system, behavior is dictated in large part by the phases of each of the co-scheduled threads. The

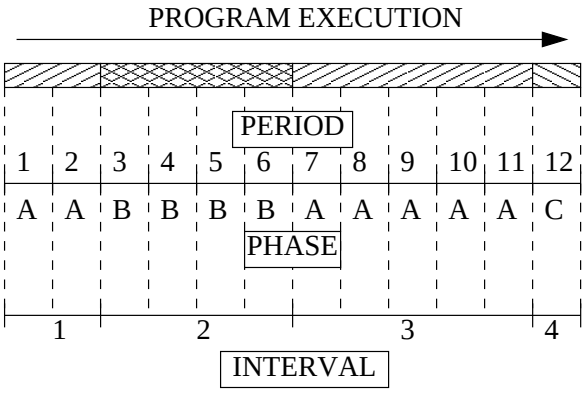


Figure 1. Program execution (represented in the first line)is broken into time slices called *periods*(the second line). Many periods demonstrate similar behavior called a *phase* (labeled on the third line). A set of consecutive periods in the same phase are called a *interval* (the last line)

resulting combination of phases from each thread in a multithreaded environment is termed a *co-phase* [16].

Traditionally, multithreaded systems are simulated by starting each thread together and allowing them to run together until one completes. The fundamental problem with this approach is that the combination of phases that interact between programs will vary in a real system depending on operating system scheduling. For multithreaded simulation, there are many similar instances in which real operating conditions and simulated operating constraints diverge, all of which can impair design decisions.

For instance, it is very unlikely that two programs of more than trivial length will remain co-scheduled for the duration of their execution. Other processes, including those from the OS, will likely use some amount of the processor's time. When any one of the target applications is not resident on the processor, the other threads, which are still resident, will advance and possibly change phase. Even if there is no phase change, the time until the next phase switch will be reduced. These effects lead to two major discrepancies to the single simulation approach. First, many co-phases may not be observed in one simulated run that may well occur in a hardware. These co-phases may exhibit unique and important behaviors that are simply neglected because they are not observed. Second, the balance between co-phases will be heavily skewed toward the simulated run which may emphasize a co-phases which, on average, are very uncommon and under-represent those which are important.

Although program phase has been recognized as a major factor in multithreaded execution, the combination of phases that interact between programs will vary in a real system has and not been fully explored. With this in mind, this paper addresses several missing elements of the characterization of the interaction between threads. This paper introduces a model which determines how different co-phase interactions should be weighted in a way that is representative of the amount it would occur in an OS scheduling environment.

The remainder of this paper is organized as follows. Section 2 supplies motivation for this research including data on the effects of start time offset in a common SMT processor, the Intel Pentium-4 with Hyper-threading. Section 3 walks through a simple, illustrative example with two threads from the *Spec2000* suite. Section 4 explains our multithreading model for two threads and and explains extensions to more than two threads, and Section 5 concludes and outlines future work.

2. Motivation

Recognizing program phase has important ramifications for simulation. Essentially, identification of repeating and representative phases makes it possible to simulate very small portions of a program's execution and then predict overall performance from those samples with high accuracy [11]. This offers advantages over random or statistical sampling where brief periods of execution are used to extrapolate overall behavior [18] in that only unique behaviors need to be simulated, requiring less simulation time [19]. This idea can be extended to simulation of multithreaded systems. The complication is that each combination of phases, or co-phase, can have unique interference. The number of co-phases is the product of the number of phases from each threads. As a result the total number of co-phases will grow exponentially with the number of threads. Faced with this, it is tempting (and probably necessary in many cases) to limit the number of co-phases that are simulated. However, it is important that as many co-phases as possible are tested. Just as program behavior can vary greatly between phases, the behavior in a multithreaded system will vary greatly between co-phases. Therefore, overall observed system behavior will be dependent on the combination of co-phases that are experienced.

To test the effects of co-phase mixes, several experiments were performed on a real system with support for multithreading and run-time performance monitoring. Five of the benchmarks from the *Spec2000* suite were run on an Intel Pentium-4 Northwood processor with Hyper-threading (a version of SMT) [5]. The benchmarks were chosen based on their long run-times, which allows a longer start time offsets to be tested. Additionally, the benchmarks were chosen to mix memory intensive with computation and control limited benchmarks and integer and floating point benchmarks. Each possible pairing of the benchmarks was run

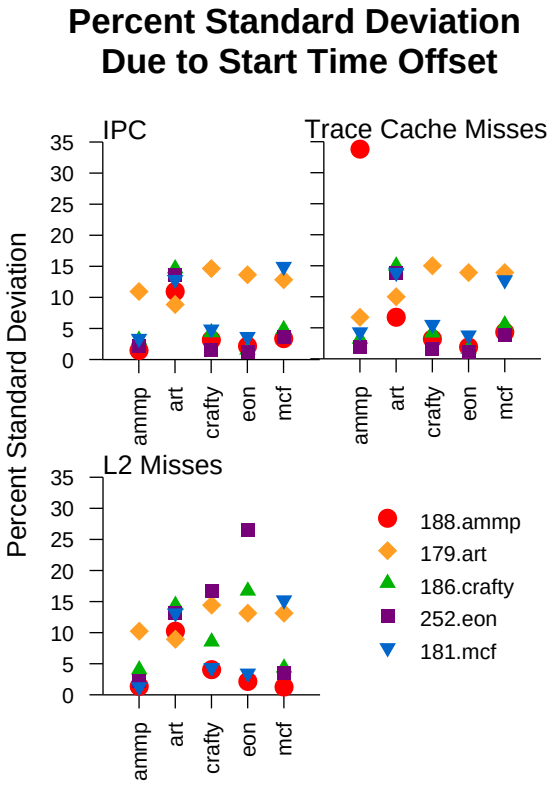


Figure 2. Standard deviation in multithreaded IPC, trace cache misses, and Level-2 cache misses due to start time offset for various *Spec2000* benchmark pairings.

with offsets in start time from negative to positive one hundred seconds, at ten second second intervals (-100, -90, -80, ..., 80, 90, 100), for a total of 21 tests for each pairing. The offset was used to produce a different combination of co-phases in each test. Using the performance counters on the Pentium-4, the instructions per cycle (IPC) and the rates of cache misses in the trace cache and the unified, level-2 cache were determined for the periods in which the threads were co-scheduled. The percent standard deviation between tests for each metric is presented in Figure 2. Each benchmark is represented by a column of the graph and a symbol. The first benchmark in the pairing is represented by the column of the graph, and the second by the symbol. Many pairings demonstrate deviation above ten percent in each of the categories, meaning that any single run of the pairings at a given offset would likely be significantly different from a run at another offset. The benchmark *179.art* is a particularly striking example of this behavior with high variance in all categories for all pairings. Generally, this means that when *179.art* is run with any of the selected applications, a selected time of offsetting the two applications can result in potential differences of 10% IPC, trace cache misses, and L2 misses when compared to a different offset. Overall, these results have significant implication on the techniques used to model and simulate multithreaded architectures.

In order to do divide the programs into phases, various performance counters on the Pentium-4 were used to sample performance at each scheduling interval (approximately 100ms) in a modified Linux 2.6.5 environment. Several runs are made on each benchmark and different performance counters from separate runs are combined into a single vector. The vectors from each sample are clustered using a k-means algorithm. Although more precise methods have been developed for phase analysis in hardware ([8]), using performance counters is sufficient for our purposes as only coarse-grained behaviors are needed for this work.

In addition to which co-phases occur, the time spent in each co-phase is also a product of the offset, which is illustrated in Figure 3. Using some of the data from Figure 2, these graphs illustrate the co-phase behavior of the benchmarks when they are paired with *252.eon*. Each row of graphs represents a different benchmark paired with *eon* and each column represents a different offset between the start times of the benchmarks. In each graph, the x-axis represents the phase of *eon* and the y-axis represents the phase of the other benchmark. The color or shade of each point on the graph represents the number of cycles spent in each co-phase with brighter colors indicating more cycles. The columns of graphs illustrate this behavior for offsets of negative one hundred, negative fifty, zero, fifty, and one hundred seconds. The variation between the graphs illustrates that co-phase behavior varies with offset. The difference due to offset is more quantitatively illustrated in Figure 4. The difference between two consecutive offsets is found by determining the percentage of execution that that occurs in each co-phase, then summing the difference between runs in these percentages across all co-phases (the number is divided by two to give a percentage). The graph demonstrates that for the benchmarks tested, a change in offset of fifty seconds results in at least a 10% difference in co-phase mix, and on average over a 20% difference.

In [16], it is recognized that when threads are run together, only a small number of the possible co-phases actually occur, especially for multithreaded systems with many threads. However, in any real world system, it is very unlikely that two threads will be started at the same time or even that the start time offset will be consistent between runs. Additionally, because other programs, including the OS, are competing for processor time, it is very unlikely that threads will be co-scheduled for the entire duration of their execution. This will also affect the co-phase mix. This means that characterizing a system based on starting the

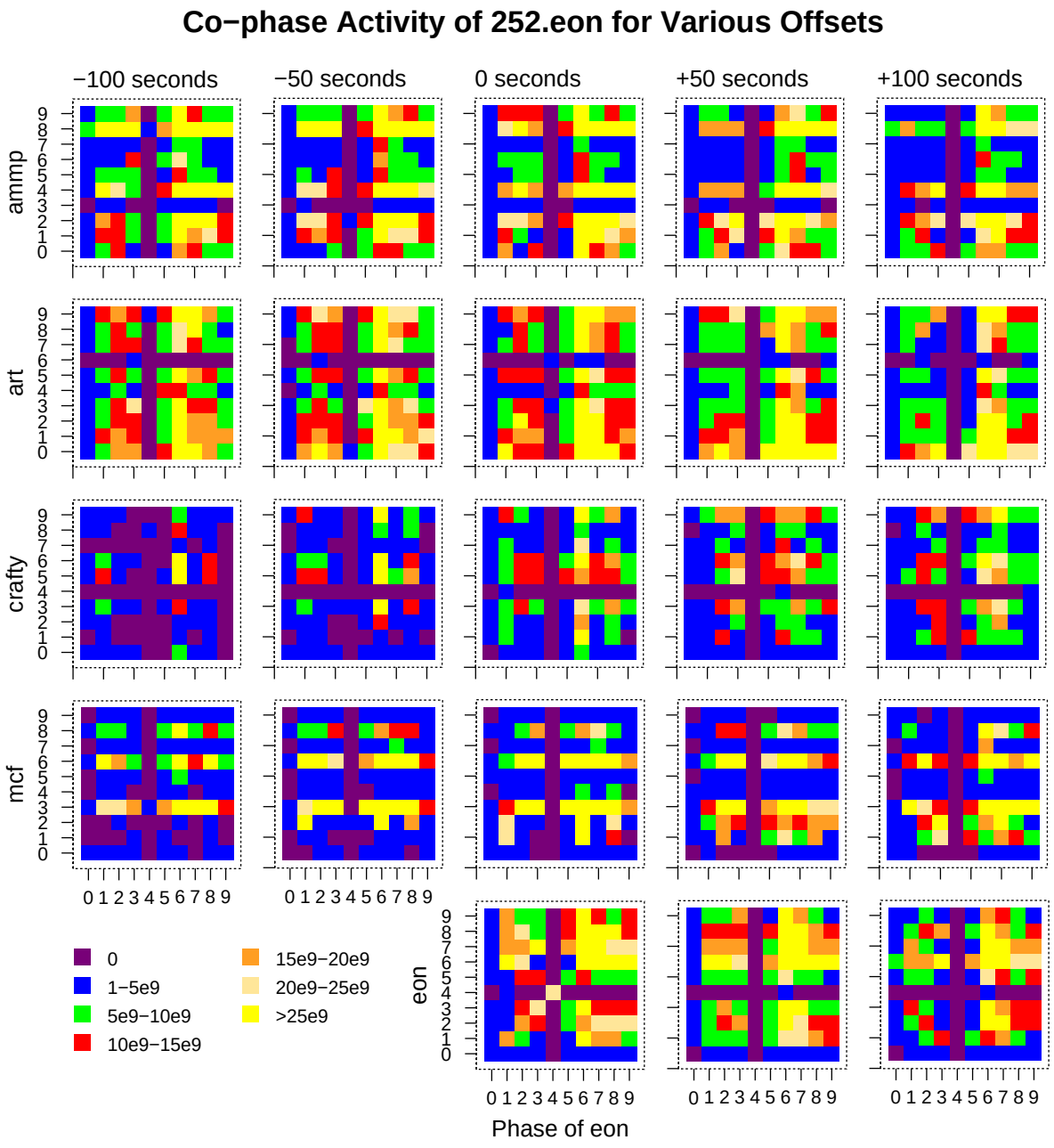


Figure 3. Number of cycles spent in each co-phase of *252.eon* when paired with other *Spec2000* benchmarks and for various offsets. Each row of graphs indicates the benchmark which is paired with *eon* and each column of graphs indicates a different offset. For each graph, the horizontal axis indicates the phase of eon and the vertical axis the phase of the paired benchmark.

threads simultaneously, or with any given single offset, will lead to an incomplete and inaccurate picture. Further, large benchmark suites are such as *SpecCPU* are meant to cover a large number of unique behaviors. By neglecting certain co-phase behaviors, many unique, and possibly very important behaviors may be missed. This paper addresses how to correctly weigh each of co-phases to get the best picture of overall behavior. Further, this gives some initial in-

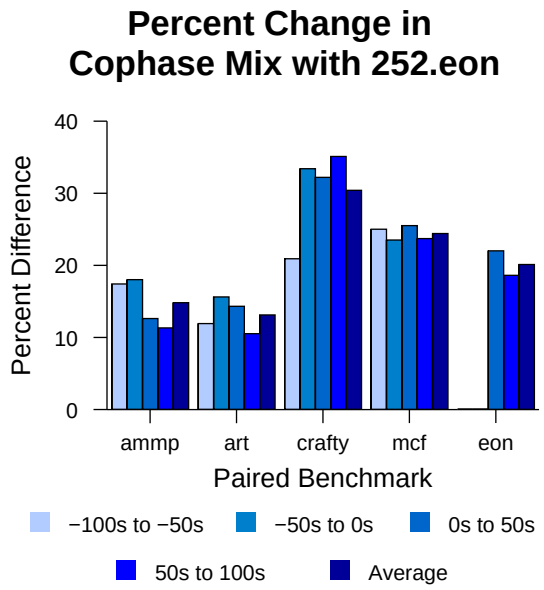


Figure 4. The percent change in co-phase mix due to different start offsets between *252.eon* and four other *Spec2000* benchmarks.

179.art

Co−Phase IPC

Crafty Phase \ Art Phase	X	A	B
X	0	.039	.034
A	0	.028	.015
B	0	.034	.013

Interval Series

#	Phase	Length (ops.)
1	A	35.2e6
2	B	43.9e6

186.crafty

Co−Phase IPC

Crafty Phase \ Art Phase	X	A	B
X	0	0	0
A	.196	.085	.047
B	.192	.100	.039

Interval Series

#	Phase	Length (ops.)
1	A	142.2e6
2	B	438.2e6

Figure 5. Phase behavior of *179.art* and *186.crafty*. The numbers int the upper tables indicate the IPC of each thread individually for each co-phase combination. The lower tables are the interval series of each thread.

sights into the best manner in which to simulate a multi-threaded system that gives the most accurate picture of execution and minimizes total evaluation time.

3. Illustrative Example

In this section, a simple example program is used to demonstrate the methodology. The example consists of the first two intervals of the *Spec2000* benchmarks *179.art* and *186.crafty*. Execution of each thread is terminated after the second interval in order to keep the example simple. The co-phase and single-threaded information is shown in Figure 5. The upper tables contain the IPC data for each thread in each co-phase, with the table row indicating the phase of *crafty* and the column the phase of *art*. The lower tables in the figure contain the length, in operations, of the intervals of each thread. The data was obtained by using the performance counters on a Pentium-4 processor with Hyper-Threading. The reason for the low IPC numbers is the CISC nature of the x86 ISA.

Since the necessary co-phase performance data is available, it is possible to determine the amount of time that is spent in each co-phase for a given offset. The first step is to determine the performance of the threads before the other thread has started. The offset between the start times of the threads is the amount of time the first thread will spend in single-threaded execution. After the time spent in single-threaded execution is determined, it is possible to determine the time spent in the remainder of the co-phase intervals, in the order [*1.1*], [*1.2*], [*1.x*], [*2.1*], [*2.2*], [*2.x*], [*x.1*], [*x.2*] [1]. The amount of time spent in an interval is determined by the number of operations that need to be performed and the IPC of each thread in that co-phase. When any thread completes all of its operations in its current interval, it will change phase an therefore the co-phase will also change. The number of operations to be completed in a co-phase interval is dependent on how many operations have already been completed in previous intervals, and is therefore a function of start-time offset. The functions for the co-phase intervals of *art* and *crafty* versus offset are shown in Figure 6. In this graph, the x-axis is the offset of the start times of the threads. A positive offset indicates that *179.art* started first and a negative offset indicates *186.crafty* started first. The y-axis indicates the amount of time spent in each co-phase interval. Note that there is no offset such that both intervals [*1.2*] and [*2.1*] are encountered. Since the co-phases are unique to these intervals, no single offset could be used

1 The numbering used here for co-phase intervals is one number per thread, separated by periods. The first number indicates the current execution interval number of the first thread (thread *X*), the second number the current execution interval number of the second thread (thread *Y*), and so on. For the purpose of this discussion, only two threads are considered. Models of more threads are discussed in Section 4.2. An interval number of *0* indicates that a thread has not yet started, and threads marked with an *x* have already completed.

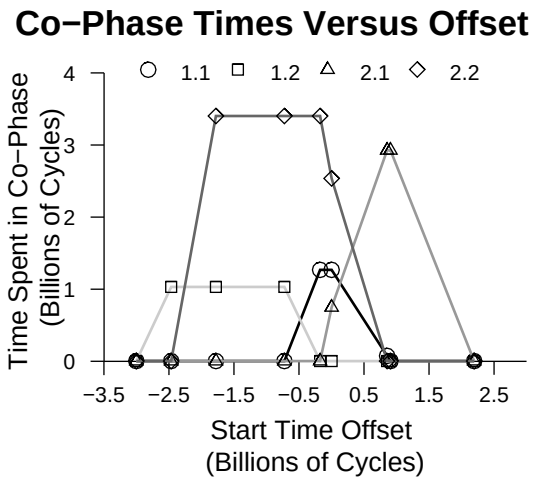


Figure 6. Time spent in the first two co-phases for *179.art* and *186.crafty*.

to characterize both in the same run.

A similar representation of the data can be found in Figure 7. The x-axis is again the offset between the thread start times. However, in this figure the y-axis represents the total execution time. In essence, instead of superimposing the graphs as in Figure 6, the data is now cumulative. There are several interesting aspects of this graph. First, the execution profile, the set of co-phase intervals encountered for a given offset, can be obtained from this graph by simply finding the offset on the x-axis and then following the graph vertically. The graph regions encountered along that vertical line represent the time spent in each interval. The next interesting feature is the top of the graph. The top edge of the shaded regions indicates the total run times versus start time offset. The total run time in this example ranges between $4.78 * 10^9$ cycles and $6.36 * 10^9$ cycles or a range of **34.5%**. The top edge at the left and right extremes of the graph represent the threads being run sequentially, and will always be the same height. For most start time offsets, multithreading is detrimental in that the total run time is higher than running the threads sequentially. This is mainly due to the heavy amount of interference caused by phase **B** of *179.art*. This phase has very poor behavior when paired with either phase of *186.crafty*. The only time that total run time is reduced is when this phase is run predominantly by itself. Although this graph is a good illustration of behavior for a simple example, it is difficult to produce a similar graph for an entire run of a benchmark pairing. Benchmarks can have thousands of intervals that lead to millions of co-phase intervals and therefore graph regions.

Calculating the Average Behavior The final step is to find the areas of each of the co-phase interval region. The area of a region is proportional to the average amount of time that will be spent in that co-phase interval across all possible offsets. The average amount of time spent, or weight, of a region is simply the area divided by the length of sequential single-threaded execution of the threads, which is the range of possible offsets. The computation is fairly simple since each region is simple a series of line segments. The area under the line $mx + b$ between x_1 and x_2 is $\frac{m}{2}(x_2^2 - x_1^2) - b(x_2 - x_1)$. In this example each co-phase has only one interval so there is no summing across intervals to get the total for co-phases. The calculated areas and weights are shown in Table 1. The table also shows the standard deviation in the amount of time spent in each co-phase across the offsets. Although the standard deviation in total run time is only **7.4%**, the deviation in time spent in each co-phase is above 100% in many cases, further demonstrating that any single run will give an incomplete picture.

To characterize average behavior, we simply multiply the weight or area of a co-phase region times the execution characteristics of that co-phase, then divide by the total area of all co-phase regions. The sum of the products across all co-phase regions is the average performance for all offsets. For this example, the average IPC of the multithreaded areas is **0.0713**. Coincidentally, the IPC obtained from a single run with zero offset is a nearly identical **0.0704**, an error of only **1.3%**. However, this is just a fortunate coincidence. In most cases any single run will be quite different from the average. Even in this case, the run time for zero offset is $6.312 * 10^9$ cycles, which is a **7.4%** error from the average of $5.87 * 10^9$ cycles. Additionally, the zero offset run never encounters the co-phase [*1.2*], which is the only co-phase that demonstrates good parallelism in this example.

Co-phase	Area	Average Run Time (Weight)	Standard Deviation
1.1	$1.15 * 10^{18}$	$2.21 * 10^8$	$3.95 * 10^8$(179%)
1.2	$2.35 * 10^{18}$	$4.52 * 10^8$	$4.73 * 10^8$(105%)
2.1	$3.67 * 10^{18}$	$7.06 * 10^8$	$9.59 * 10^8$(136%)
2.2	$8.23 * 10^{18}$	$1.58 * 10^9$	$1.49 * 10^8$(94%)
total	$3.05 * 10^{19}$	$5.87 * 10^9$	$4.76 * 10^8$(8.1%)

Table 1. Co-phase interval region areas and average weights for *179.art* and *186.crafty*

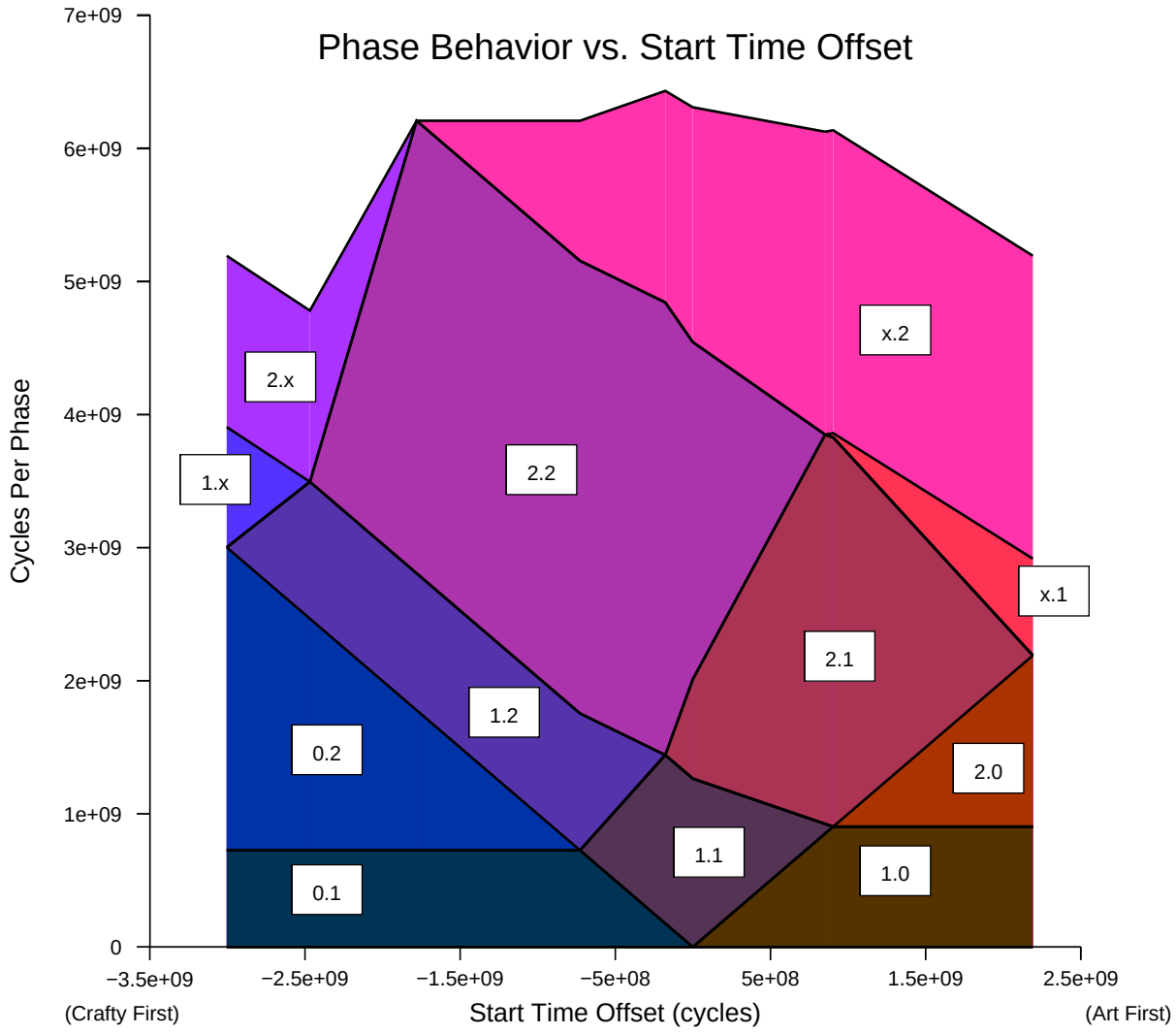


Figure 7. Phase interference behavior of *179.art* and *186.crafty* versus start time offset.

4. Methodology

4.1. Mathematical Model

With the large amount of variation in behavior due to offset, the question is how best to model the overall behavior of the system. Ideally, this also involves minimizing simulation time. The model proposed in this section requires only two inputs. The first is the profile of phase intervals and their lengths for each individual thread and the second is the IPC of each thread in each co-phase, including in each of its single-thread phases. From this data, the expected series of co-phase intervals and their lengths can be derived for any start time offset. More importantly, it is simple to determine the average behavior across all possible offsets as a weighted sum of the behaviors of all of the co-phases.

Once the necessary data is generated, the time spent in each co-phase interval can be determined from the equation in Figure 8. Essentially, the time in an interval will be the shortest possible for one of the threads to complete its current interval. This is only a function of the number of instructions in the interval, the thread's performance in the phase, and the number of instructions that have already been completed from that interval. The number of instructions already completed is the most complicated factor as it is dependent on the performance and the time spent in previous co-phase intervals. For a given interval, this dependency stretches back to all co-phase intervals since the beginning

$$t_{ij} = \min\left(\frac{I_{Xi} - \sum_{k=0}^{j-1} t_{ik} P_{Xik}}{P_{Xij}}, \frac{I_{Yj} - \sum_{k=0}^{i-1} t_{kj} P_{Ykj}}{P_{Yij}}\right)$$

t_{ij} = The number of cycles spent in the *i*th stage of thread *X* and the *j*th stage of thread *Y*.
P_{Xij} = The average number of instructions of thread *X* that are executed per cycle while in the stage combination *ij*.
I_{Xi} = the number of instructions in the *i*th interval of thread *X*.
i,j > 0.

Figure 8. Time spent in co-phase interval i.j.

$$t_{i0} = \begin{cases} 0 & \text{if } t_0 < \sum_{k=1}^{i-1} \frac{I_{k0}}{P_{Xk0}} \\ t_o - \sum_{k=1}^{i-1} \frac{I_{k0}}{P_{Xk0}} & \sum_{1}^{i-1} \frac{I_{k0}}{P_{Xk0}} < t_0 < \sum_{1}^{i} \frac{I_{k0}}{P_{Xk0}} \\ \frac{I_{i0}}{P_{Xi0}} & t_0 > \sum_{k=1}^{i} \frac{I_{k0}}{P_{Xk0}} \end{cases}$$

t_0 = The offset in start times between the threads
t_{i0} = The number of cycles spent in the *i*th stage of thread *X* and in single-threaded mode.
P_{Xi0} = The average number of instructions of thread *X* that are executed per cycle running single threaded in mode in phase *i*.
I_{Xi} = the number of instructions in the *i*th interval of thread *X*.

Figure 9. The time spent in co-phase interval *0i*. Symmetric equations exist for co-phase intervals *0j*

of each thread's current interval. Each of these co-phase intervals in turn is dependent on earlier co-phase intervals, all the way back to the beginning of execution, when only one thread runs because of start time offset. Therefore, the time spent in any co-phase interval is a function of start time offset. In the equation in Figure 8, the portion of the function where the thread X term controls the *min* function, the next co-phase interval will be the next interval of X. Obviously, the same can be said for thread Y. The two terms of the function will only intersect once because although the slope of the line changes many times, the term for one thread will be monotonically increasing versus offset and the other will be monotonically decreasing. This makes sense as the greater the offset is, the further the thread will be in its execution. The intersection of the two point marks the boundary between which co-phase interval will occur next or the point where both phases will end simultaneously.

The minimum function makes it very difficult to create closed form equations for any intervals past the first few intervals. Such functions exist, since any finite execution will have a finite number of co-phase intervals, but they can become prohibitively large very quickly. The number of possible predecessor intervals is proportional to the number of intervals each thread has completed. Since each of these have their own predecessors, the total number of terms in a given equation becomes very large very quickly. Instead, it is much easier to solve the equations numerically. That is, for a given system, find the function of the offset time that determines the amount of time each interval will occur by adding the functions of its predecessors, starting from the intervals that have no predecessors. The only co-phase intervals that have no predecessors in a two thread system are interval [*0.1*] and [*1.0*]. Co-phase interval [*0.1*] can end in one of three ways: either thread X will start and the co-phase interval will become [*1.1*], the interval of thread Y will complete and the interval becomes [*0.2*], or thread X will start exactly as thread Y finishes its interval and the co-phase interval becomes [*1.2*]. The amount of time spent in interval [*0.1*] is equal to the start time offset up to the maximum of the time it takes interval 1 of thread Y to complete running alone. The equation for the amount of time spent in co-phase interval [*i.0*] is shown in Figure 9. With this data from interval [*0.1*], the function for [*0.2*] can be derived. This is repeated for all intervals of thread Y. The process is then repeated for thread X. Once the functions for co-phase intervals [*0.1*] and [*1.0*] are determined, it is possible to derive the function for co-phase interval [*1.1*] using the equation in Figure 8. All of the execution time function of each co-phase interval functions can be derived numerically once all of the functions of all of their predecessors are derived. The simplest order to derive the function is using a nested *for* loop. That is, derive all of the functions for co-phases [*0.j*] for j starting at 1 to the last phase of thread Y, then for [*1.j*] for j starting at 0 and then for each interval of thread X. After all of the interval functions have been derived, the average behavior can be calculated as explained in section 3.

A few key assumptions are made for this model. The first assumption is that co-phase behavior follows the phase behavior of individual threads. This means that the length of the execution intervals will remain constant, in terms of total operations executed between single and multithreaded modes. The reason that this may change is that a phase is the set of periods with similar but not identical behavior. Interference in the multithreaded environment may affect different periods differently, effectively splitting up a phase or making periods that are in different phases in single threaded execution behave in similar ways when multithreaded. This is an area of ongoing research. A second related assumption is that co-phase behavior is consistent in that the behavior in a given co-phase is the same between intervals and within each interval. This assumption must be made anytime phase analysis is used. The final assumption is that no accounting is made for threads switch-

ing out during execution in the model. The assumption is not that this switching will not occur, but that the net effect of such switches will be zero. A change in thread being switched out and then switched back in can be thought of as the simulation stopping when the thread is switched out, then restarting at the same execution time with a new offset. In effect, the thread that is not switched out is fast forwarded relative to the thread which was switched out. The assumption is that either thread is equally likely to be switched out for an equal amount of time. Over the totality of all tests, the effects of switch outs cancel each other out. The overhead of the context switch and of the thread warming up when it is switched back in are neglected because this will be small compared to the overall execution time. The direct and indirect costs of context switches are examined in [7].

4.2. Extension to More Than Two Threads

The situation becomes considerably more complex when more threads are used. The first complication is that the number of co-phases and co-phase intervals grows exponentially with the number of threads. Dealing with this problem is a matter of predicting which co-phases will contribute the most to the final results and have interesting interference. This is the subject of ongoing research.

Next, representing the data becomes more difficult. The number of possible offset variables between any two threads is the factorial of the number of threads. This problem is solved by defining the start time of one thread as a zero point and defining all other start times relative to this one. The relative start times of any two threads can be easily determined from their offset from the reference.

It would seem that the choice of which thread is chosen as the reference would affect the projection and thus area and weights between the regions. Fortunately the areas are constant no matter which representation is chosen. The matrix representation of the transform between representations is shown in Figure 10. That is, in order to move a point from the representation where all offsets are defined in terms of thread **1** to one where the offsets are all defined in terms of thread **2**, the point is represented in column vector form and multiplied by this matrix. It can be shown that $R_n^{n+1} = I_n$. In other words, applying the transform in an *n*-thread system *n+1* times yields the original representation. A formal proof is beyond the scope of this paper, but the basic idea is that since any change in area produced by the transform would cause a change in the representation once it was repeatedly applied, it follows that each application of the single transform does not change the areas. More information on transforms can be found in [9].

Fortunately, the equation in Figure 8 is still valid if terms are added for the additional threads. However, the equation in Figure 9 becomes somewhat more complicated with more threads but the basic idea is the same. The region which represents a given startup co-phase interval will start at the point where that co-phase is reached, based on previous startup intervals. The region ends where another thread starts or the interval of that thread ends.

$$R_n = \begin{bmatrix} -1_{(n-1)x1} & I_{n-1} \\ -1 & 0_{1x(n-1)} \end{bmatrix}$$

Figure 10. The matrix for transforming representations between reference threads (I_n is the n-dimensional identity matrix).

5. Conclusion

5.1. Future Work

The accuracy of multithreaded simulation is vitally important to the development of systems which are becoming increasingly ubiquitous across the industry. However, also important is the speed of these simulations. Because of inter-thread interactions, the number of unique behaviors of a multithreaded system is the product of the number unique behaviors in each thread. This further exacerbates the problem of long simulation times. Although this paper has shown that accurate simulation of multithreaded systems requires that the entire co-phase space be covered, it is impractical in all but the most trivial cases to simulate all co-phases for their duration. The next step in our research involves developing methods to efficiently cover that space. This entails applying modern sampling techniques such as SMARTS [18] and SimPoint [11] to multithreaded environments to reduce simulation times. Along these lines, we are also investigating techniques to increase parallelism in simulation to improve overall simulation latency. Additionally, we are developing methods to reduce the effective size of the co-phase space without sacrificing accuracy. We are investigating techniques to predict multithreaded behavior from single threaded behavior and determining which co-phases will have the most important and interesting behavior to reduce the number of co-phases which must be simulated.

5.2. Summary

Multithreaded architectures, in various forms, are an increasingly popular technique for circumventing the bottlenecks of modern processors. The interference between

threads, however, has a dramatic impact on overall performance. Interference is dictated by how the individual threads interact in their various phases. Since phase interaction is determined by the relative position of the threads, this further complicates the characterization of multithreaded systems. The experimental results presented in this paper demonstrate that this offset has a substantial effect on overall performance. The model presented balances the importance of individual co-phases and allows more accurate modeling of real world performance of multithreaded architecture, where effects such as OS scheduling cause random offsets between threads. This increase in the correlation between simulated and achieved performance is vital to the effective and efficient design of future multithreaded architectures.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfi eld, and B. Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, 1990.

[2] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 266–277. ACM Press, 2001.

[3] A. S. Dhodapkar and J. E.Smith. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 217–227, 2003.

[4] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–26, / 1997.

[5] Intel Corporation. *Intel Pentium 4 Processor with 512-KB L2 Cache on 0.13 Micron Process and Intel Pentium 4 Processor Extreme Edition Supporting Hyper-Threading Technology Datasheet*. Santa Clara, CA, 2004.

[6] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Proceedings of the Third IEEE Annual Workshop on Workload Characterization*, pages 102–110, 2000.

[7] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 75–84. ACM Press, 1991.

[8] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *Proceedings of the 37th Annual Symposium on Microarchitecture (Micro-37 2004)*, 2004.

[9] L. Sadun. *Applied Linear Algebra: The Decoupling Principle*. Pearson Education, Prentice Hall, Upper Saddle River, NJ, 2001.

[10] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to fi nd periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2001.

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[12] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 17th annual international conference on Supercomputing*. ACM Press, 2003.

[13] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[14] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 414–425, 1995.

[15] R. Steven E and S. K. Reinhardt. The impact of resource partitioning on smt processors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2003.

[16] 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*, 2004.

[17] W. Weber and A. Gupta. Exporing the benefi ts of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Interational Symposium on Computer Architecture (ISCA)*, pages 273–280, June 1989.

[18] R. E. Wunderlich, T. F. Wenisch, B. Falsafi , and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30nd Annual International Symposium on Computer Architecture (ISCA)*, pages 84–95, 2003.

[19] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *The 11th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2005.

FAST: A Functionally Accurate Simulation Toolset for the Cyclops64 Cellular Architecture

Juan del Cuvillo Weirong Zhu Ziang Hu Guang R. Gao
Department of Electrical and Computer Engineering
University of Delaware
Newark, Delaware 19716, U.S.A
{jcuvillo,weirong,hu,ggao}@capsl.udel.edu

Abstract

This paper reports our experience and lessons learned in the design, implementation and experimentation of an instruction-set level simulator for the IBM Cyclops-64 (or C64 for short) architecture. This simulation tool, named Functionally Accurate Simulation Toolset (FAST), is designed for the purpose of architecture design verification as well as early system and application software development and testing. FAST has been in use by the C64 architecture team, system software developers and application scientists. We report some preliminary results and illustrate, through case studies, how the FAST toolchain performs in terms of its design objectives as well as where it should be improved in the future.

1. Introduction

It is increasingly clear that the huge number of transistors that can be put on a chip (now is reaching 1 billion and continues to grow) can no longer be effectively utilized by traditional microprocessor technology that only integrates a single processor on a chip. A new generation of technology is emerging by integrating a large number of tightly-coupled simple processor cores on a chip empowered by parallel system software technology that will coordinate these processors toward a scalable solution.

This paper reports our experience and lessons learned in the design, implementation and experimentation of an instruction-set level simulator for the IBM Cyclops-64 architecture that integrates on a single chip up to 150 processing cores, an equal number of SRAM memory banks and 75 floating point units. This simulation tool, named Functionally Accurate Simulator Toolset (FAST), is designed for the following goals (1) architecture design verification; (2) early system software development and testing; (3) early application software development and testing. For our purposes, a cycle accurate (rather than function accurate) simulator would be too slow for a system consisting of one or more fully-populated C64 chips. Currently, FAST efficiently handles C64 systems consisting of either a single processing core, a C64 chip fully populated or a system built out of several nodes connected with a 3D mesh.

We present several important aspects of the FAST simulator and highlight the tradeoffs faced during its design and implementation. Some design decisions are made based on the unique features of the C64 architecture. For instance, C64 employs no data caches. Instead, on-chip memories are organized in two levels — global interleaved memory banks that are uniformly addressable, and scratch memories that are local to individual processing cores.

FAST has been in use by the C64 architecture team, system software developers and application scientists. We report some preliminary results and illustrate, through case studies, how FAST performs in terms of its design objectives as well as where it should be improved in the future.

2. Cyclops64 chip architecture

The Cyclops-64 (C64) is the latest version of the Cyclops cellular architecture designed to serve as a dedicated petaflop compute engine for running high performance applications [10]. A C64 supercomputer is attached — through a number of Gigabit Ethernet links — to a host system. The host system provides a familiar computing environment to application software developers and end users.

A C64 is built out of tens of thousands of C64 processing nodes arranged in a 3D-mesh network. Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic. A C64 chip employs a multiprocessor-on-a-chip design with a large number of hardware thread units, half as many floating point units, embedded memory, an interface to the off-chip DDR SDRAM memory and bidirectional inter-chip routing ports,

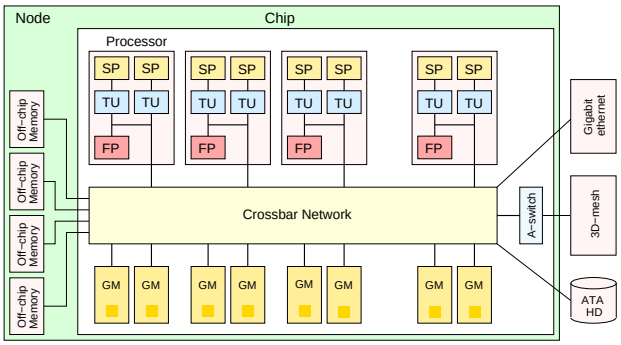


Figure 1: Cyclops-64 node

see Figure 1. A C64 chip has 75 processors, each with two thread units, a floating-point unit and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. The C64 chip has no data cache. Instead a portion of each SRAM bank can be configured as scratchpad memory (SP). The remaining sections of SRAM together form the global memory (GM) that is uniformly addressable from all thread units. On-chip resources are connected to a 96-port crossbar network, which sustains all the intra-chip traffic communication and provides access to the routing ports that connect each C64 chip to its nearest neighbors in the 3D-mesh network. The intra-chip network also facilitates access to special devices such as the Gigabit Ethernet port and the serial ATA disk drive attached to each C64 node.

The C64 architecture represents a major departure from mainstream microprocessor design in several aspects. The C64 chip integrates processing logic, embedded memory and communication hardware in the same piece of silicon. However, it provides no resource virtualization mechanisms. For instance, execution is non preemptive and there is no hardware virtual memory manager. The former means a single application can run at a given time on a set of C64 nodes. Additionally, the OS will not interrupt the user program running on the thread units unless the user explicitly specifies preemption or an exception occurs. The latter means the three-level memory hierarchy of the C64 chip is visible by the programmer. From the processing core standpoint, a thread unit is a simple 64-bit, single issue, in-order RISC processor with a small instruction set architecture (60 instruction groups) operating at a moderate clock rate (500MHz). Nonetheless, it incorporates efficient support for thread level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely; and when asleep it can be woken up by another thread through a hardware interrupt. Additionally, the integration of processing logic and memory is further leveraged with a rich set of hardware supported in-memory atomic instructions. Unlike similar instructions on common off-the-shelf microprocessors, atomic instructions in the C64 only block the memory bank where they operate upon while the remaining banks proceed servicing other requests. This functionality provides a higher memory bandwidth.

Table 1: Simulation parameters

Component	# of units	Params./unit
Threads	150	single in-order issue, 500MHz
FPUs	75	floating point/MAC, divide/square root
I-cache	15	32KB
SRAM (on-chip)	150	32KB
DRAM (off-chip)	4	256MB
Crossbar	1	96 ports, 4GB/s port
A-switch	1	6 ports, 4GB/s port

3. FAST design and implementation

FAST is an execution-driven, binary-compatible simulator of a multichip multithreaded C64 system. It accurately reproduces the functional behavior and count of hardware components such thread units, on-chip and off-chip memory banks, and the 3D-mesh network, see Table 1. The actual number of simulated chips is limited by practical reasons, since the memory corresponding to all the chips need to be allocated in the host machine.

FAST has been developed following a modular approach, such that additional features could be easily incorporated into the existing design. To help the architecture team with the verification of the C64 chip design, the simulator executes instructions (3.1), models the architecture exceptions (3.2), reproduces the C64 memory map (3.3) and produces histograms of the instruction mix as well as detailed traces of all instructions executed (3.4). For the purpose of early system and application software design and evaluation, in addition FAST accounts for memory and interconnect contention (3.5), supports intra-chip communication through the A-switch device (3.6) and incorporates debugging facilities (3.7). Finally, an overview of the simulator internals is provided (3.8).

3.1. Instruction execution

FAST simulates the four-stage pipeline employed in the C64 architecture, see Figure 2.

At the first stage of the pipeline, an instruction (see Table 2) is fetched from the program instruction buffer (PIB) and decoded. FAST may account for the access to the PIB and subsequent delay if the instruction has to be read from the instruction cache or memory, if a miss should occur. Whenever the branch prediction is incorrect, execution in a thread unit stalls for three cycles while the pipeline is

Table 2: Instruction set summary

Core Integer and Branch	**Floating Point**
Load, Store	Add, Subtract
Load, Store Multiple	Multiply, Divide
Add, Subtract [Immediate]	Multiply and Add
Multiply, Divide	Conversions
Compare [Immediate]	Square Root
Trap on Condition [Immediate]	
Logic [Immediate]	
Shift [Immediate]	
Shift left 16 then OR immediate	
Insert, Extract	
Move if Condition	
Branch on Condition	
Branch and Link	
Exotic	**Control**
Bit Gather (permute bits)	All Stop
Count Leading Zeros	I-Cache Invalidate
Count Population	Move From/To SPR
Load then Op	Return from Interrupt
Multiply and Accumulate	Sleep
	Supervisor Call

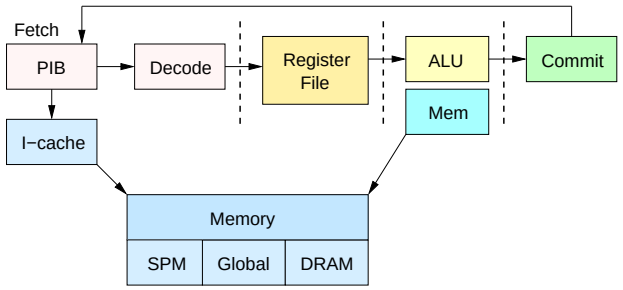


Figure 2: Four-stage instruction pipeline

flushed. However, FAST does not reflect the operation of the branch predictor and regards all conditional branches as correctly predicted.

In the second pipeline stage, the instruction input operands are read from the register file. For all the C64 instructions, except the floating multiply and add (FMA), one or two register operands are read in one cycle. FMA instructions have three input operands, hence an extra cycle may be required to read the third operand since the register file has two read ports.

In the third stage the instruction is executed. RISC-like instructions such as integer, floating-point, branch and memory operations are modeled based on execution times expressed by x/d pairs, where x is the execution time in the ALU, and d represents the delay before the result of the instruction becomes available. Instruction timing reported in Table 3 is based on information provided by the C64 chip designer team. For instance, integer division is said to take one cycle in the ALU but a subsequent instruction will not be able to use the result until 33 cycles later. During this delay, execution of independent instructions can proceed normally. However, if the result of an instruction is to be used by another instruction before it is available, the pipeline will stall. It is the compiler and programmer responsibility to cover these delays as much as possible, with the appropriate instruction scheduling.

Table 3: Instruction timing

Instruction type	x	d
Branches	2	0
Count population	1	1
Integer multiplication	1	5
Integer divide, remainder	1	33
Floating add, mult. and conv.	1	5
Floating mult. and add	1	10
Floating divide double	1	30
Floating square root double	1	56
Floating mult. and accumulate	1	5
Memory operation (local SRAM)	1	2
Memory operation (global SRAM)	1	20
Memory operation (off-chip DRAM)	1	36
All other operations	1	0

The result is finally committed in the fourth stage if no exception is generated. Otherwise, a context switch causes execution to continue from the address specified by the interrupt vector. When the results are to be put away, conflicts may occur, since the register file has two write ports. However, these events are not expected to happen frequently and FAST does not account for them.

In terms of instruction execution, FAST allows thread units to fetch, decode and execute instructions independently, following the sequence of events dictated by each thread's instruction stream. However, care need to be taken for some special instructions. The sleep instruction, the wakeup signal, the inter-thread interrupt, etc., all imply a synchronization between threads. For instance, a thread unit, while asleep, does not execute any instruction. During this time the simulator will not update its clock counter. When a wakeup signal is received, the clock counter is set to that of the remote thread that executed a store in the wakeup memory area (plus some delay). To handle these synchronizations, threads shall commit instructions once the simulated chip clock reaches the time point at which the instruction is executed by the thread. In other words, although instructions are executed asynchronously they are committed in a synchronized fashion.

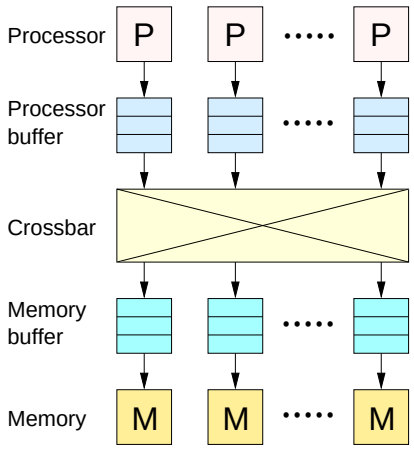


Figure 3: Interconnection to the on-chip crossbar

3.2. Exception handling

Exceptions are thread-specific events. Some are caused by instructions and trigger what we call synchronous interrupts that cannot be disabled. For instance, an attempt to execute an instruction with an invalid opcode generates an illegal interrupt. Others, known as asynchronous, are caused by events such as a timer alarm and can be disabled. While disabled, only the first exception of each type generated by a sequence of events is held pending; subsequent ones are lost. Throughout the instruction's execution, multiple exceptions of both classes may occur. FAST checks for exceptions at the end of the execution stage. Before the results are put away, if one or more enabled exception exists, FAST generates an interrupt according to the priority order specified by the architecture.

3.3. Segmented memory space

The C64 chip hardware supports a shared address space model: all on-chip SRAM and off-chip DRAM banks are addressable from all thread units/processors within a chip. That is, all threads see a single shared address space.

Architecturally, each thread unit has an associated 32KB SRAM bank. Each memory bank can be partitioned (configured) into two sections: one called "global" (or "interleaved") section, the other "local" (or "scratchpad") section. All such global sections together form the (on-chip) global memory in an interleaved fashion that is free of holes and uniformly addressable from all thread units. Although scratchpad memory, global memory and off-chip DRAM memory are addressable from any thread within the chip, the access is not uniform. Besides having different latencies, these three memories have a separate address space, resulting in a three level hierarchy. Furthermore, there is no virtual memory manager in the C64 architecture, hence this memory hierarchy is directly exposed to the programmer.

The FAST simulator accurately reproduces the C64 memory map by implementing the above mentioned non-uniform shared address space. It also includes the address upper limit special purpose registers (AULx) that define the highest existing location in scratchpad memory, global memory and DRAM memory, respectively. Nonetheless, all memory-specific parameters such as the number of banks, size of each bank, latency, and bandwidth are easily configurable. Additionally, it considers three protection boundary special purpose registers (PBx). These registers define regions in scratchpad, interleaved and DRAM memory that can only be written in supervisor state, which effectively provide a basic mechanism to protect the kernel against malign user code.

3.4. Execution trace and instruction statistics

Given the appropriate command line option, the toolset generates the execution trace of a program. There are two mechanisms to select the instructions that are to be stored in the trace. The user can either specify the time interval (in clock cycles) for which the program execution is to be traced, or enclose the instructions to be output to the trace within TraceOn/TraceOff macros. These macros access unarchitected special purpose registers, i.e. SPRs that control the simulator's functionality but are not present in the C64 chip design. The output consisting of a text file per active thread on the C64 system, contains detailed information such as clock cycle, instruction executed, source and target register values, address of the memory location touched by the instruction, if applicable, and specific information regarding events that could have delayed the execution of the instruction (contention in the crossbar network, operand not available yet, etc).

FAST may also collect instruction statistics over an execution interval and produce histograms of the instruction mix. Similarly to the procedure available for tracing, the user can specify an interval in clock cycles or use StatsOn/StatsOff macros to start/stop collecting statistics, respectively. A combined report for each node as well as individual reports for all active threads are generated.

3.5. Memory and interconnect contention

One of the latest additions to the FAST simulator is a module that accounts for the contention in the crossbar network and in the memory system.

Figure 3 illustrates the data path between processors and memory banks on a C64 chip. Every memory instruction executed on a processor results in a network packet delivered by the crossbar network to the appropriate memory bank (global SRAM or off-chip DRAM). For load operations, the

memory replies with another packet containing the data retrieved from memory.

FAST models the following sources of contention: (1) Packets issued by threads on the same processor are queued on a 7-slot FIFO (processor buffer) until they are retrieved by the crossbar. If a thread issues a memory operation when the FIFO is full, the pipeline will stall until space is available; (2) The crossbar retrieves packets from the input ports and delivers packets to the output ports, one per cycle. If at the same cycle, two packets are to be delivered to the same output port, the crossbar blocks one of them arbitrarily; (3) Between the crossbar and each memory bank there is another 7-slot FIFO (memory buffer) where packets are held until processed by the memory. Whenever this buffer becomes full, the crossbar stops delivering packets to this destination. At the same time, it stops retrieving packet from any input that tries to send packets to the blocked output port; (4) Memory latencies are also taken into account. SRAM memory banks can perform a load or store operation every cycle, i.e., 4GB/s per bank. Whereas DRAM memory can sustain a much lower bandwidth. DRAM memory consists of four banks and each bank is subdivided into four subbanks. Subbanks can service requests simultaneously, one every 32 cycles. While a memory subbank is in service, an incoming request is held pending in the memory buffer. Therefore, the DRAM bandwidth is 2GB/s for single loads and stores. For multiple transfers, using load multiple (LDM) and store multiple (STM) instructions, the DRAM bandwidth is 16GB/s instead.

3.6. A-switch device

In FAST, there are two optional modes for simulating the A-switch: message accurate and packet accurate simulation. The former is faster but less accurate, since it copies the whole message directly to the destination node. The latter models all of the hardware mechanisms involved in transferring packets double word by double word through the 3D-mesh network. However, this model is still under testing. Mainly, because it does not account for the interaction between the A-switch and the crossbar network. In other words, reading from and writing to memory while sending or receiving messages do not generate the corresponding packets in the crossbar. Therefore, performance estimations obtained with FAST for multichip simulations should be regarded as less accurate.

3.7. Debugger

FAST integrates a user-friendly assembly-level debugger. In debugging mode, there are commands to set a breakpoint, continue with the execution after a breakpoint, single-step the execution, inspect and modify the values of registers or memory, etc. Although useful, this method is tedious. To eliminate the hazard of mapping statements in the source code to assembly instructions and vice versa, a source level debugger is a necessary tool. The GNU debugger, GDB, has been partly ported to the C64 architecture.

3.8. Simulator internals

The simulated C64 system starts running when one of the three main simulator functions is called. To maximize performance, each function specifically handles a C64 system consisting of a single processing core, a C64 chip fully populated, or a system built out of several nodes. Therefore, the decision is simply based on the system configuration.

In multinode simulations, the main function starts with a loop that iterates over all the active threads on all the nodes. Each thread unit attempts to execute an instruction. For a new instruction, calls to routines that take care of instruction fetch, instruction decode, read the input operands from the register file, and instruction execution, are invoked. If the thread unit is asleep, stalled waiting for an operand of due to a resource hazard, or waiting to commit the previous instruction, it does nothing but return.

Back in the main function, the chip clock is moved forward, just enough to allow one thread unit, at least, to commit the current instruction. Once the clock is updated, the crossbar and memory banks proceed to flush packets and memory operations that are to be performed by this time.

Then a second loop iterates over all the threads, regardless of their status. First, thread units check whether an exception occurred, and if it did, the corresponding interrupt is serviced with the appropriate context switch. If no interrupt was triggered, they try to commit the last instruction. At this stage, threads compare the chip clock with their own internal clock. When the execution on the chip reaches the time step at which a thread can commit an instruction, the results are put away. Otherwise, the thread waits.

Finally, after the status of the A-switch is updated, execution returns to the beginning of the main loop. The process is repeated until thread units on every node execute the ALLSTOP instruction in supervisor state.

To simplify the communication among components of the simulator, the representation of the simulated C64 system is kept in a single multilevel data structure. At the chip level, it contains information regarding thread units, floating point units, on-chip SRAM and off-chip DRAM memories, I-caches, crossbar model, and A-switch. At the thread level, it accounts for general, special purpose and accumulator registers, in addition to timing information as to when the value stored in a general purpose register will be available, the last decoded instruction, program counter, exception flags, thread status, and a third-level data structure with statistics counters.

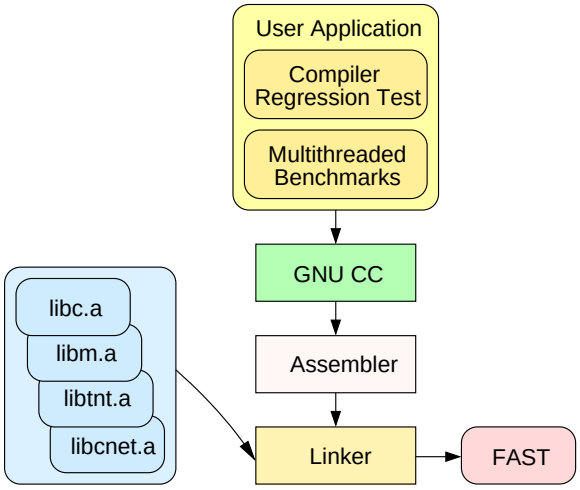


Figure 4: Cyclops-64 software toolchain

4. Experience

The goal of FAST is three-fold: FAST is designed for the purpose of architect design verification (section 4.1). As part of the C64 toolchain, FAST provides the basic platform for early system software development and testing (section 4.2). FAST has been in use by other users for application development and testing. Although not cycle accurate, the timing information provided by the simulator has proven to be useful for performance estimation and application tuning as well (section 4.3).

4.1. Design verification

For the purpose of architecture design verification, the execution trace generated by FAST is compared to the output of the VHDL simulator that reproduces the C64 at a gate level. Initial verification of the C64 design was carried out following this procedure with a set of short programs intended to test the C64 instruction set architecture [11].

The Cyclops-E is another cellular architecture design, target to the embedded market. The first hardware implementation of a single-chip Embedded Cyclops system was accomplished with DIMES, an FPGA-based multiprocessor emulation system [21]. Concurrently with the development of DIMES an earlier version of the FAST simulator, known as CeDIMES, was also implemented [9]. Since this simulation tool is also binary-compatible, once the hardware emulation system was brought up, design verification started immediately. A test suite consisting of more than 200 programs specifically designed to test the Cyclops-E ISA were run on the actual hardware platform and the results were compared to those produced by the simulator. The initial testing revealed a few bugs in the chip design, which were fixed by the chip architect.

4.2. System software development

4.2.1. Toolchain FAST is part of the software toolchain available for application development on the C64 platform, see Figure 4. Programs written in C or Fortran are compiled using a porting of the GCC-3.2.3 suite. The assembler and linker, which are based on binutils-2.11.2, along with the necessary libraries, produce a 64-bit ELF executable that can then be loaded into FAST and executed. The C standard and math libraries are based in newlib-1.10.0. In addition, we wrote the TNT runtime system and the CNET communication libraries to manage hardware resources such as the thread units and the A-switch, respectively.

4.2.2. Thread library We reported our work in the design of a thread model for C64 that maps directly to the architecture assisted by a native thread runtime library, called TNT (or TiNy Threads) [8]. In the development, debugging and evaluation of the TNT library, FAST's capability to accurately simulate a large number of hardware threads with practical time has proven to be useful.

4.2.3. Spin lock For a thread library, it is important that all components are efficiently implemented. In multithreaded environments, especially for architectures like C64 with 150 threads on a chip, spin lock, as an indirect synchronization mechanism is known to be a key factor for scalability. For this reason, we conducted a study on spin lock algorithms on the C64 architecture. We implemented eight programs based on well known spin lock algorithms: three based on test-and-set, one on tickets, two on array queue, and two on list queue [15]. All programs consist of a short critical section (a single variable update) enclosed within calls to procedures that acquire and release a lock following the corresponding algorithm. The entire process (lock, critical section, unlock) is repeated a thousand times as part of a loop body. We run the programs on FAST with memory contention enabled and measure the execution time as well as the overhead due to contention in the crossbar. As expected, the results show list-based queuing locks are the most efficient algorithms, see Figure 5. On C64, contrarily to most shared memory multiprocessors, array-based queuing lock methods do not perform well, because there is no data cache. In other words, accesses to the array queue are as expensive as any memory operation seen in test-and-set based implementations. Indeed, test-and-set based algorithms with linear and exponential backoff perform better. Not surprisingly, list queue locks generate the least amount of memory traffic on the crossbar, since threads spin locally on their own scratchpad memory, see Figure 6. That means they would interfere least with the normal execution of a program if it had additional memory accesses. As a result of this experience, the implementation of mutexes in the TNT library is based on list queue locks.

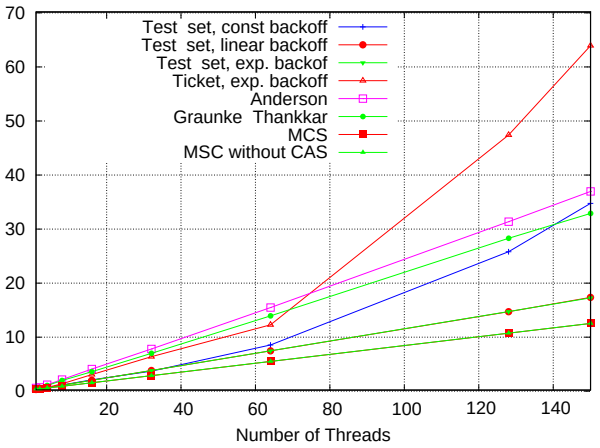


Figure 5: Execution time of spinlock programs (in ms)

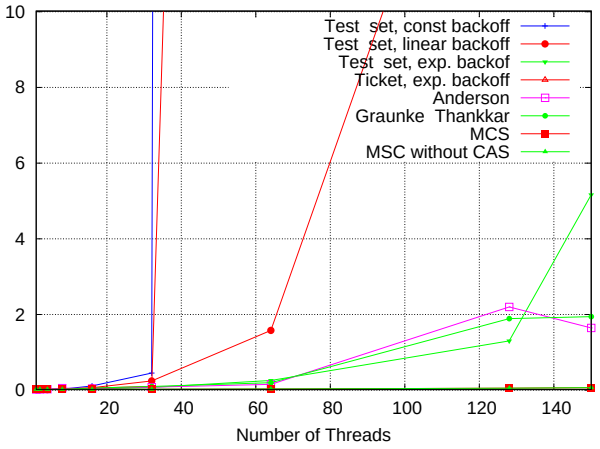


Figure 6: Execution delay of the spinlock programs (in ms)

4.2.4. Communication library With the A-switch module, FAST can be used to simulate C64 multichip system configurations. With this feature, we developed a communication library implemented as several layers, each accessible through its own interface. At the lowest level, the packet transfer layer accesses the A-switch directly, hiding all hardware details from layers above. A second layer, built on top of the packet transfer layer, provides user-level remote memory read and write, inter-chip synchronization primitives and remote procedure call mechanisms. Finally, we are in the process of porting the SHMEM library [17] to the C64 architecture based on the two previous layers.

4.3. Application development and evaluation

To demonstrate FAST is functionally accurate, stable and hence, useful for software development and performance estimation, we write several benchmarks programs to confirm that the trends predicted by the simulator match to what the C64 architecture is capable of.

4.3.1. GUPS Table Toy, which is also called Random Access benchmark, is an important benchmark included in the *HPC Challenge Benchmark Suite* [1]. It uses a metric known as GUPS (Giga Updates Per Second) to evaluate the random access capabilities of the memory system. In the context of this experience, we use Table Toy to verify that FAST reflects the C64 memory system accurately.

The kernel operations of Table Toy can be summarized as follows:

```
1    tmp1 = stable[j];     (load)
2    tmp2 = table[i];      (load)
3    val = tmp2 xor tmp1; (xor)
4    table[i] = val;       (store)
```

The i, j are the pseudo random locations chosen for $table$ and $stable$. Ideally, thread units should access different locations of the $table$ to avoid conflicts. The $table$ can be placed either in the on-chip SRAM or the off-chip DRAM, and is accessed by all thread units. The substitution table ($stable$) is allocated in the thread's scratchpad memory. The key point is that the last three operations (load, xor, and store) must be atomic in a multithreaded execution.

Figure 7 shows the GUPS obtained on a C64 node with up to 150 thread running in parallel. By taking advantage of C64's *xor_m* in-memory atomic instruction (xor to memory), we guarantee the atomicity needed while all data dependences are removed from the kernel loop. Therefore, the number of memory updates is actually the number of *xor_m* instructions issued. If the updates are performed in SRAM, the curve scales well as the number of threads increases, due to the large on-chip memory bandwidth. On the other hand, the off-chip DRAM memory bandwidth is limited. Consequently, the DRAM curve flattens when the number of threads exceeds 16. In both cases, the maximum achievable memory bandwidth is not reached. It appears that the pseudo random numbers generated in Table Toy result in several threads accessing a memory bank at the same time. Hence, the bandwidth limitation is not due to the crossbar network but to conflicts accessing the memory banks.

To prove our hypothesis, we write three separate microbenchmarks with a deterministic access pattern to the memory banks. In our first microbenchmark, New Toy 1, each thread issues 3 store operations every 8 cycles. In addition, each thread targets one SRAM bank only. Therefore, a processor issues 6 store operations every 8 cycles to the on-chip SRAM memory. This represents 75% of the peak throughput of the crossbar which is indeed achieved because there are no conflicts as the memory bank addressed by each thread unit is different, see Figure 8. The other microbenchmarks test the off-chip DRAM memory subsystem in different ways. In New Toy 2, each thread targets one of the 16 DRAM subbanks based on the thread identifier. Therefore, threads 0 and 16 access subbank 0, threads 1 and 17 access subbank 1, and so forth. A DRAM subbank can

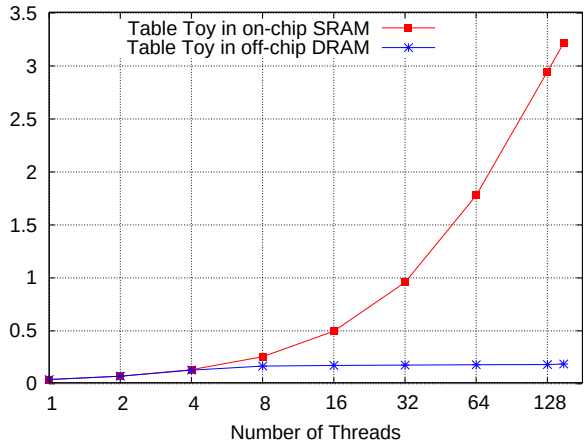


Figure 7: GUPS on a C64 node (Table Toy)

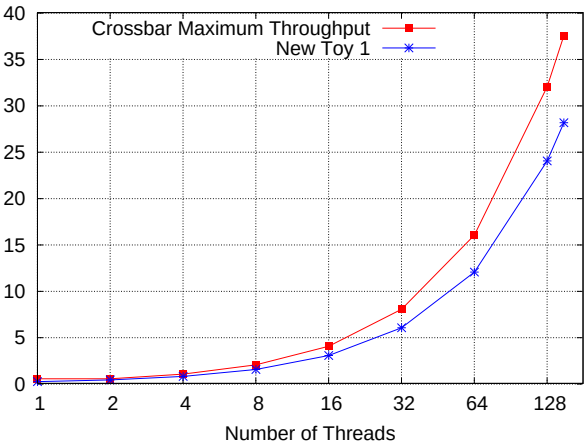


Figure 8: GUPS on a C64 node (New Toy 1)

only service one request every 32 cycles, this is 15 MUPS, but all 16 subbanks can service requests in parallel. Figure 9 confirms the expected result. As long as no more than 16 threads are active, the DRAM throughput increases linearly, at a rate of 15 MUPS per thread, up to 250 MUPS. In New Toy 3, every thread executes 16 consecutive stores every 22 cycles and each store targets one of the DRAM subbanks. That means a thread can issue operations to memory faster than the memory can handle. Figure 9 shows that for a small number of active threads, contention can be tolerated, and the crossbar and DRAM memory system deliver the peak throughput, 250 MUPS. Finally, as contention increases, performance drops.

4.3.2. Matrix-matrix-multiply As an example of what an application developer can expect to learn using the FAST toolset, we hereby report a tuning experience using the matrix-matrix-multiply program for a problem size of 1024×1024. Throughout this exercise we use the simulator's accurate time counter, the histograms file with the instruction mix and the execution trace to determine the cause of delays and/or bottlenecks that may prevent the program from achieving higher performance.

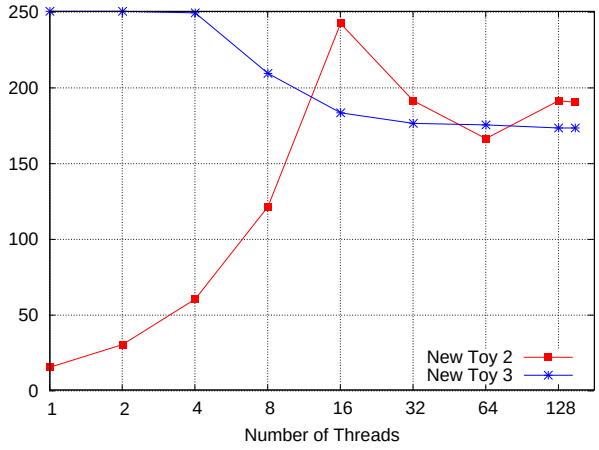


Figure 9: MUPS on a C64 node (New Toy 2 and 3)

Our baseline is a straightforward sequential code with the matrices stored in DRAM. The program that is compiled with -O3, achieves 16.7 MFLOPS. From the trace file, we found that the main reasons for the low performance are the poor data re-usage and the long latency to access DRAM. In order to improve the performance, we unroll the two outermost loops 4 times each and manually prefetch data and re-schedule the instructions with the hints from the trace files generated by FAST. In the resulting code, data is fed to the floating point unit in a pipelined manner such that all load latencies are hidden. This implementation achieves 216.1 MFLOPS, a speedup of 13 compared to the baseline version. We also parallelized our tuned MxM program to make use of multiple thread units. As shown in Figure 10, the curve of the parallel version scales almost linearly up to 32 threads and then flats out because of the bandwidth limitation. Afterwards, it even drops when memory contention becomes too high. We believe higher performance can be achieved by employing other techniques. However this is not the purpose of this experiment.

4.3.3. Multi-chip benchmarks To verify the correctness (not the accuracy) of FAST's multichip simulation and the communication libraries, we developed an assorted set of multichip multithreaded benchmarks. It includes implementations of matrix-matrix-multiply, 1D Laplace solver, heat conduction and Sobel edge detection.

5. Related work

To analyze and understand the impact of various architectural parameters and components as well as study the application performance and get detailed statistics, both academia and industry developed a number of simulators. Simulation frameworks for microarchi-

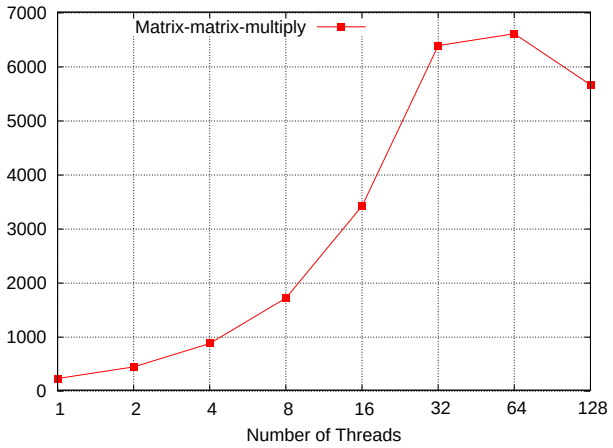


Figure 10: MFLOPS for the matrix-matrix-multiply with prefetching on a C64 node

tecture research and design exploration, such as SimpleScalar [7, 3], Microlib [18], Liberty [23], RSIM [12] and Turantdot [16], concentrate on accurately modeling the architecture design and normally they are cycle accurate. FAST is a functional simulator since cycle accurate simulation would be too slow for a system consisting of one or more C64 chips. There are also full system simulators capable of running commercial workloads on an unmodified operating system, such as SIMOS [20], Simics [13] and PearPC [4]. A C64 compute engine is attached to a host system. The host system provides a familiar computing environment to application developers. FAST only simulates the compute engine running a custom microkernel, whereas conventional OS services are provided by the native OS running on the host system.

Recently a new generation of simulators capable of simulating SMT and CMP architectures have been developed: SMTSIM [22], SESC [19], GEMS [14], M5 [5] and Mambo [6]. It would appear the latter simulation frameworks as well as extensions of SimpleScalar, Simics, SimOS and Turantdot are normally used to simulate 2/4/8 way SMT/CMP processors under multiprogramming, thread level speculation, and commercial workloads[1]. FAST is designed to simulate and model a CMP system consisting of several C64 nodes, each with up to 150 processing cores. However, the C64 architecture is designed for the purpose of running massively parallel applications, which deal with the complexity of scientific and engineering multithreading workloads.

Probably, the closest related work to FAST is the Cyclops-32 simulator. These simulators are as similar as the architectures they simulate. However, there are significant differences as well. For instance, FAST detects dependences and conflicts as instructions are executed. Therefore, it directly produces performance estimates. On the other hand, the C32 simulator does not have timing information. Performance estimates are generated by two other performance tools (a cache simulator and a trace analyzer) that post-process the execution trace produced by the simulator [2].

1 Based on papers published in HPCA from 2000 to 2005.

6. Summary

This paper presents FAST, a functionally accurate simulation toolset for the IBM Cyclops-64 architecture that is fast, flexible and efficient. To the best of our knowledge, it is the only simulation tool capable of simulating multichip multithreaded cellular architectures with reasonable accuracy and practical speed. We report some preliminary results and illustrate, through case studies, how the FAST tool chain accomplishes its purpose of architecture design verification as well as early system and application software development and testing.

As future work, we plan to increase the amount of profile information FAST produces, including text and data symbols, and to incorporate integer counters to facilitate the performance analysis of multithreaded programs.

Acknowledgments

We acknowledge support from IBM, in particular, Monty Denneau and Henry Warren. We thank ETI for support of this work. We also acknowledge our government sponsors. Finally, we also thank many CAPSL members for helpful discussions.

References

[1] HPC challenge benchmark. URL http://icl.cs.utk.edu/hpcc.

[2] G. S. Almási, C. Caşcaval, J. G. Castaños, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber, J. E. Moreira, D. Newns, M. Snir, and H. S. Warren, Jr. Demonstrating the scalability of a molecular dynamics application on a petaflops computer. *International Journal of Parallel Programming*, 30(4):317–351, August 2002.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[4] S. Biallas. PearPC - PowerPC architecture emulator, May 2004. URL http://pearpc.sourceforge.net/.

[5] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, pages 36–43, Anaheim, California, February 9, 2003. Held in conjunction with the 9th International Symposium on High-Performance Computer Architecture.

[6] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson,

E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: A full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2004.

[7] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin at Madison, Madison, Wisconsin, June 1997.

[8] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. TiNy Threads: A thread virtual machine for the Cyclops64 cellular architecture. *Fifth Workshop on Massively Parallel Processing*, page 265, Denver, Colorado, April 8, 2005. Held in conjunction with the 19th International Parallel and Distributed Processing Symposium.

[9] J. del Cuvillo, R. Klosiewicz, and Y. Zhang. A software development kit for DIMES. CAPSL Technical Note 10 Revised, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 2005.

[10] M. Denneau. Computing at the speed of life: The BlueGene/Cyclops supercomputer. CITI Distinguished Lecture Series, Rice University, Huston, Texas, September 25, 2005.

[11] M. Denneau. Personal communication, February 2005.

[12] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: Simulating shared-memory multiprocessors with ILP processors. *Computer*, 35(2):40–49, February 2002.

[13] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[14] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Submitted to Computer Architecture News.

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[16] J. Moreno, M. Moudgill, J. Wellman, P. Bose, and L. Trevillyan. Trace-driven performance exploration of a PowerPC 601 OLTP workload on wide superscalar processors. In *First Workshop on Computer Architecture Evaluation using Commercial Workloads*, Las Vegas, Nevada, February 1, 1998. Held in conjunction with the 4th International Symposium on High-Performance Computer Architecture.

[17] K. Parzyszek, J. Nieplocha, and R. A. Kendall. General portable SHMEM library for high performance computing. In *Proceedings of SC2000: High Performance Networking and Computing*, Dallas, Texas, November 4–10, 2000. URL http://www.supercomp.org/sc2000/proceedings/.

[18] D. G. Pérez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 43–54, Portland, Oregon, December 4–8, 2004.

[19] J. Renau. SESC, 2004. URL http://sesc.sourceforge.net.

[20] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[21] H. Sakane, L. Yakay, V. Karna, C. Leung, and G. R.Gao. DIMES: An iterative emulation platform for multiprocessor-system-on-chip designs. In *Proceedings of the IEEE International Conference on Field-Programmble Technology*, pages 244–251, Tokio, Japan, December 15–17, 2003.

[22] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, pages 384–393, San Diego, California, December 10–13, 1996.

[23] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 271–282, Istanbul, Turkey, November18–22, 2002.

Rapid Development of a Flexible Validated Processor Model

David A. Penry David I. August
Department of Computer Science
Princeton University
Princeton, NJ 08544
{dpenry,august}@princeton.edu

Manish Vachharajani
Dept. of Electrical and Computer Engineering
University of Colorado at Boulder
Boulder, CO 80309
manishv@colorado.edu

Abstract

Given the central role of simulation in procesor design and research, an accurate, validated, and easily modified simulation model is extremely desirable. Prior work proposed a modeling methodology with the claim that it allows rapid construction of flexible validated models. In this paper, we present our experience using this methodology to construct a flexible validated model of Intel's Itanium 2 processor, lending support to their claims. Our initial model was constructed by a single researcher in only 11 weeks and predicts processor cycles-per-instruction (CPI) to within 7.9% on average for the entire SPEC CINT2000 benchmark suite. We find that aggregate accuracy for a metric like CPI is not sufficient; aggregate measures like CPI may conceal remaining internal "offsetting errors" which can adversely affect conclusions drawn from the model. We then modified the model to reduce error in specific performance constituents. In $2\frac{1}{2}$ person-weeks, overall constituent error was reduced from 3.1% to 2.1%, while simultaneously reducing average aggregate CPI error to 5.4%, demonstrating that model flexibility allows rapid improvements to accuracy. Flexibility is further shown by making significant changes to the model in under eight person-weeks to explore two novel microarchitectural techniques.

1. Introduction

Simulation is the preferred method of measurement for much of the computer architecture community. For simulation, computer architects would prefer to use *validated* models of realized systems over *non-validated* models for several reasons. First, a real system provides a golden standard which the community can use as a point of reference. Second, the existence of the real system provides proof that the model is implementable and suggests that reasonable variants are implementable as well. Finally, and perhaps most importantly, a validated model provides more confidence in conclusions drawn using the model. Prior to validation, models often fail to represent important interactions between different parts of the system and fail to capture surprising corner case behavior. Previous works such as those by Black and Shen [2], Gibson et al. [6], and Desikan et al. [3] have shown that this effect can be significant and that non-validated models can lead to incorrect conclusions.

Unfortunately, computer architects tend not to use validated models for several practical reasons. First, validated models can be extremely time-consuming to construct because constructing highly detailed models is difficult [2]. Second, even if simulators were easy to construct, not enough information about a design may be publicly available to decide what to construct [7]. Finally, validated models are often so detailed that they may be too time-consuming to modify for initial studies of many different design points [4]. This may impede innovation if it becomes difficult to study design points not immediately adjacent to the reference machine.

Prior work claims that validated models need not be hard to construct and need not be difficult to modify to explore wide areas of the design space. To support this claim, prior work presented a modeling methodology (which we will call the Liberty Modeling Methodology) [17, 16]. This modeling methodology is centered around three modeling principles:

1. structural modeling of the system
2. aggressive reuse of model components
3. iterative refinement of the model

They have also released a modeling framework consisting of a structural modeling language and a simulator-constructing compiler to support these principles called the Liberty Simulation Environment (LSE). However, their claims have not yet been substantiated with a complete validated model.

In this paper, we present our experience building a validated Itanium 2 model using the Liberty methodology and tools. This experience serves as an instance proof that it

is indeed possible to rapidly construct an easily-modifiable *validated* processor model. Using LSE, a lone computer architect was able to construct an initial validated model of Intel's Itanium 2 processor in only 11 weeks including the time to reverse engineer the physical hardware. This model predicted hardware cycles-per-instruction (CPI) to 7.9% with a maximum error of 20% across all SPEC CINT2000 benchmarks.

During this investigation, we discovered that traditional metrics of validated model *quality* are inadequate. This paper shows that models validated against a single aggregate metric, such as CPI, are insufficient for proper design-space exploration since the model may still contain large internal error constituents. With *aggregate validation*, internal error constituents may simply offset each other. We show that such errors exist in our initial model and that these "offsetting errors" can lead to poor design decisions. To correct these errors, we refined our model until it was validated against the hardware for multiple constituent metrics. This refinement took an additional 2.5 person-weeks and resulted in the current *constituent-validated* model that predicted CPI to within 5.4% across all SPEC CINT2000 benchmarks and contained far fewer offsetting errors.

To further assess the model's flexibility, we constructed two novel derivatives — an Itanium 2 with a variable latency tolerance technique and an Itanium 2 CMP processor with an unconventional interconnection mechanism [14].

The remainder of this paper is organized as follows. Section 2 describes the Liberty Modeling Methodology. Section 3 describes the Itanium 2. Section 4 then describes our first experience using the Liberty Modeling Methodology to build a validated model of the Itanium 2 and presents data regarding the aggregate quality of the model. Section 5 describes why, despite low CPI error, an aggregate-validated model may be unsuitable for microarchitecture research. Section 6 describes how we refined our initial model using *constituent validation* to correct this shortcoming and gives results. Section 7 describes experience modifying the model to explore novel ideas. Section 8 concludes.

2. The Modeling Methodology

To build a validated model, one must be able to control the sources of significant error in a system. Black and Shen identify three such sources of error in performance models [2]. These sources of error are:

Specification errors One does not fully understand the system being modeled and so models the wrong system.

Modeling errors Mistakes are made while incorporating understood system behavior into the model. The model does not do what one thinks it does.

Abstraction errors One deliberately decides not to model some behavior accurately, either by leaving out the behavior entirely or by not modeling all of its details.

Prior work describes a modeling methodology, which we will call the Liberty Modeling Methodology, with the claim that it allows rapid construction of validated models [17, 16]. The Liberty Modeling Methodology is centered around three modeling principles, each of which they claim has a role in ensuring a reduced error rate and an improved time-to-model. Here is brief summary of these principles and the role they play:

Structural system modeling A hardware system design is conceived as hierarchically composed concurrently executing blocks. Attempts to map this hierarchical and concurrent system to another composition strategy, such as composition via procedure invocation in C or C++, naturally introduce modeling errors [17]. As a result, the modeling environment should be concurrent and structural. This feature is also key to reducing model specification time because it simplifies the mapping of hardware design to model *and* because it is the key enabler for component reuse [17].

Aggressive component reuse By aggressively reusing model components, the cost of building a component is amortized across many different designs, reducing total exploration time. Reuse also has the side benefit of reducing error rates because basic behaviors are specified once and validated in many different models. This limits the source of modeling errors to incorrect component usage and eliminates the component specification from consideration in most cases.

Iterative model refinement In order to build an accurate model rapidly one should iteratively refine the model. This occurs by constructing a model for each hardware component and insuring that it functions correctly when added to the model. As the modeling effort proceeds, hardware component models are refined and their accuracy validated. Once all hardware components are modeled, the overall accuracy of the model is checked. The model portions responsible for any error are identified and the model is refined to the desired level of accuracy.

Prior work showed that the above principles require explicit tool support in practice, requiring both a structural modeling language and a simulator-generating optimizing compiler to generate an *efficient* simulator [17]. Furthermore, for reuse to be practical, the system must simplify the use of flexible components by inferring parameters and avoiding overly redundant user specifications [16].

Note that the above methodology is focused on reducing modeling errors and model construction times. Elimination of specification and abstraction errors are left to the

architect. In the remainder of this paper, we will describe our experience of managing and controlling abstraction and specification errors as well as our experience of using the Liberty methodology to control modeling errors.

3. Our Target: Itanium 2

The Intel Itanium 2 processor is a member of the Itanium Processor Family (IPF) and implements the IA-64 instruction set architecture (ISA) [10]. Each IA-64 instruction has a complexity similar to that of a single RISC instruction. Three instructions are grouped into a 128-bit bundle, which also contains *stop bits* which indicate which instructions may not be issued together due to data dependencies. Instructions have access to 128 architected general purpose registers and 64 predicate registers and may be independently predicated. The ISA also supports limited compiler controlled renaming for procedure arguments, locals, and return values reminiscent of the rotating register windows in the SPARC architectureas well as rotating registers for use in conjunction with software pipelining.

The Itanium 2 has an eight-stage in-order pipeline which can issue up to two bundles per cycle (i.e., up to 6 RISC-like operations per cycle). A diagram of the pipeline appears in Figure 1. Bundles are fetched from the instruction cache in the IPG stage and placed into an instruction buffer in the ROT stage. Fetched instruction bundles are broken into issue groups based upon the stop bits and functional unit structural hazards in the EXP stage. Once an issue group is formed, the group proceeds in lock-step down the pipeline until reaching the DET stage. The REN stage implements a Register Stack Engine which manages register stack frames and inserts implicit register spill and fill instructions. The REG stage detects and stalls on data hazards. The EXE and DET stage and additional stages for floating-point and memory operations execute instructions; all exceptions are known in the DET stage. Branches are also resolved in the DET stage. The WRB stage updates registers.

The data cache unit is quite complex. The L1 data cache is tightly integrated into the main pipeline. The L2 unified cache operates independently from the main pipeline; it is non-blocking and reorders transactions to avoid bank conflicts. A unified L3 cache and system bus controller handles misses from the L2 cache. Data cache sizes are indicated in Figure 1. For comparison, we use a HP workstation zx6000 with 2 900 MHz Intel Itanium 2's running Redhat Advanced Workstation 2.1. This system has 4GB of memory with a minimum latency of 141 processor cycles.

4. Constructing the Initial Model

In this section, we describe how we applied the Liberty Modeling Methodology to construct a validated model of Intel's Itanium 2 processor. Since the Liberty Modeling Methodology is focused on modeling error, we also present extensions to its iterative refinement principle that address specification and abstraction errors.

4.1. The Modeling Process

Using iterative refinement, each pipeline stage or major processor component of the model was developed in three steps: investigating the system behavior, determining the level of abstraction to use, and building a model for the hardware. This process was repeated for each stage of the pipeline moving from the front of the pipeline to the back.

The development activities for each week were (as recorded in the modeler's journal):

Week 1: Read documentation and decided on basic overall model structure. Modeled basic IPG and ROT stages without branch prediction.

Week 2: Investigated branch behavior on short loops. Discovered that branch predictor updates insert pipeline bubbles in a complex fashion. Determined structure of pipeline logic to use branch prediction results.

Week 3: Continued investigating branch behavior and front-end bubble insertion.

Week 4: Finished investigation and modeling of branches and front-end bubbles. Investigated and modeled the EXP stage. Began investigating the REN stage, and discovered that speculation of the bottom of the register stack frame was required.

Week 5: Finished modeling the REN stage without speculation. Implemented a simple scoreboard (REG stage) w/o bypasses, EXE, DET, and WRB stages.

Week 6: Implemented REN-stage speculation. Added logic for corner cases of predicate scoreboarding. Added sampling support. Began debugging major benchmarks.

Week 7: Continued debugging. Added bypass logic. Started investigating the data-cache unit (DCU) structure.

Week 8: Continued to investigate DCU structure.

Week 9: Implemented the DCU L1 data cache, advance load address table (ALAT), and translation look-aside buffers (TLBs).

Week 10: Added very abstract level two data cache (L2), level 3 data cache (L3), memory models. Implemented level 1 instruction cache (L1I), and instruction TLBs (ITLBs).

Week 11: Cleaned up the model and added monitors to match hardware counters. Continued debugging. Added dynamic branch prediction and return address stack.

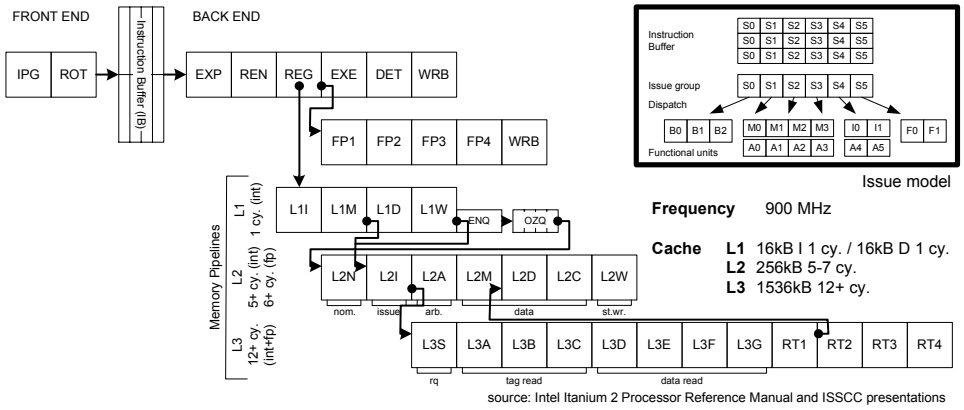


Figure 1. Itanium 2 pipeline

Notice that in each phase of the iterative refinement we strictly repeated three steps: each component was first systematically investigated, modeling decisions were made, and *then* the model portion was constructed. This discipline extends iterative refinement to act as our primary means of controlling specification error in addition to modeling error. A fourth step, evaluation of the overall model, was used throughout the refinement process when appropriate. We now describe each of these steps in more detail.

4.2. Investigation

In total, investigation took 5 of the 11 weeks. The purpose of the investigation step at each phase of refinement was to understand the behavior and structure of the processor to avoid specification errors. Two sources of information proved to be useful: documents and experiments run upon the actual hardware.

The documents used included processor manuals [9, 8], slides from symposium presentations [15, 11], white papers [1], magazines [12], and academic publications [5]. The different kinds of documents served different purposes. Slides, magazines, and white papers provided the basic pipeline structure and the parameters of structures such as caches. Processor manuals described instruction latencies, structural hazards, interesting corner cases, and performance counters. Academic publications clarified structural details of the cache designs and integer bypass paths.

While all documents were helpful, their use was not without difficulty. Documents sometimes lacked clarity or organization, obscuring vital details. For example, some elements of the second-level cache request queue are reserved. This information is provided, not in the L2 section, but rather in the L1 data cache section. Much useful information was also omitted. For example, only one document explicitly stated the write allocation policy of the L3 cache. Even worse, sometimes when a single document provided information, it was wrong. For example, the description of the EXP stage rules for memory instructions in the processor reference manual [9] is incorrect with respect to usage of load and store ports. Finally, documents were sometimes contradictory. For example, the processor reference manual [9] and the microarchitectural optimization manual [8] make contradictory statements about how bank conflicts in the L2 data cache delay accesses.

These difficulties led to the formulation of a general principle: *Quantitatively verify all documents.* Documents are good for making hypotheses about structure or behavior, but they cannot be relied upon.

Experiments were used for two purposes: to test hypotheses and to explore the behavior of the processor. Experiments were generally performed using *micro-benchmarks*, as advocated by Black and Shen [2] and Desikan et al. [3], with Perfmon [13] used to provide measurements of the Itanium 2 hardware performance counters. The typical micro-benchmark consisted of a loop with the code to be tested inside of it. The loop had a trip count high enough to overcome fluctuations in the tools used to measure hardware performance and other transients.

As an example of hypothesis testing, consider the contradiction in the documentation which was described earlier. To test which document was correct, it was necessary to set up a bank conflict between two loads which miss the first-level data cache with a use of the second load following immediately, as in Figure 2(a). The results from this microbenchmark indicated an 11 cycle latency, validating the claims made in the microarchitectural optimization manual.

As an example of behavioral exploration, consider Figure 2(b). By varying the number of issue groups of nops (no-operation instructions) inserted between the first and second issue group, we discovered that the third load could be caused to have a bank conflict with the second load's reissue after its own bank conflict. Surprisingly, the latency of the third load becomes 7, not 11 as would be expected based on the previous experiment.

Detailed investigation and modeling of behavior proved

```
{ // 1st issue group
   ld4.nt1   r20 = [r5]    // forces L1 miss
   ld4.nt1   r21 = [r5] ;; // will conflict
}
{ // 2nd issue group
   add       r2 = r21, r0  // to see latency
}
```

(a) Bank conflict micro-benchmark

```
 { // 1st issue group
    ld4.nt1 r20 = [r5] ld4.nt1 r21 = [r5] ;; }
 { //2nd issue group
    ld4.nt1 r22 = [r5] ;; // extra conflict }
 //insert nop groups here
 { // 3rd issue group add r2 = r22, r0 }
```

(b) Three bank conflicts micro-benchmark

Figure 2. Micro-benchmarks

to be very beneficial for final accuracy, even when the rationale behind the behavior was initially unclear. For example, we found that the Register Stack Engine sometimes issues one spill or fill per cycle but at other times issues two, depending upon the address at which the spill or fill begins. This behavior was very easy to model, even though we did not understand why it was happening. Later, as the data cache unit was being modeled in more detail, we observed that this behavior is precisely that required to avoid bank conflicts in the second-level data cache.

4.3. Abstraction

Along the way, many decisions about the abstraction level of the model needed to be made. Some of the abstractions and approximations were:

- No instruction prefetch engine was implemented.
- The memory hierarchy beyond the L1 caches was extremely abstract: the model probed the caches, calculated a hit/miss latency, and then delayed the instruction by that many cycles.
- A constant value was charged for hardware page table walks.

Note that these abstractions were *not* validated using quantitative measurements. As will be discussed in Section 6, this mistake led to large abstraction errors. Based on our experiences with this non-quantitative strategy, we strongly discourage its use.

Of particular interest is that poorly chosen abstractions can *reduce* flexibility. The Register Stack Engine was originally modeled by causing pipeline stalls proportional to the number of registers to spill or fill, without actually performing memory accesses. This proved to require special case logic in the scoreboard, which in turn made it difficult to change pipeline organization. The time necessary to correct this poor abstraction choice (two weeks) is included in the time needed to change the model in Section 7, though the performance effects of the correction are included in all reported results.

4.4. Modeling

The model was constructed using the Liberty Simulation Environment (LSE) [17], which provides explicit support for structural modeling, aggressive reuse, and iterative refinement. Recall that the investigation took 5 of 11 weeks, meaning that the modeling activity required only 6 weeks in total. The features of the LSE designed to support the three Liberty Modeling Methodology principles were essential to this rapid model development.

The modeling of the EXP stage illustrates how LSE language features were helpful in modeling. The EXP stage takes two bundles of instructions from the Instruction Buffer (IB) as inputs and creates issue groups (i.e., groups of instructions that have no internal data dependencies). The EXP stage routes each instruction in the issue group up to the first stop bit to one of 11 *ports*. Each kind of instruction can be routed to only a subset of the ports; over-subscription is possible, in which case the EXP stage must "split" the issue group.

Figure 3 shows the structure of the EXP stage model. The entire stage is modeled by instantiating, connecting, and parameterizing modules (component templates in LSE) from the standard module library. The parameterization sets a few simple parameters on the components and uses special userpoint parameters [17] to fill in the routing computation and the bundle-to-instruction conversion algorithms. All of the work of actually manipulating signals and routing instruction information is handled by the modules, saving much time and effort.

The model of this stage has no global controller. This is a result of LSE's default flow-control semantics, which provide back-pressure from later stages automatically. LSE's default control semantics, together with a small piece of stop-bit flow-control logic and the structure of the datapath of the EXP stage model, *automatically* prevent instructions after a stop bit from leaving the IB. Because so much behavior was implicit in the modules and LSE, this stage could be modeled in only a few hours. Note however that, had the implicit behavior been incorrect, LSE would have allowed us to override the default control.

The EXP stage also illustrates another desirable outcome of structural modeling in LSE: a separation of mechanism from policy. The modules and their interconnections provide the mechanism, while the customizations (i.e., the parameter values) provide the policy. This makes it easy to modify the policy during design-space exploration; only the

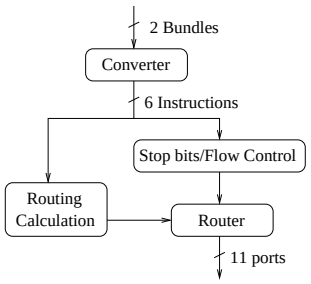


Figure 3. EXP stage model

routing computation, described earlier, need be changed.

In the EXP stage example above, all of the components came from the module library. In the complete model, 82% of the 210 component instances were instantiated from 19 modules in the standard LSE module library. The remaining instances were instantiated from hierarchical modules created through composition and parameterization of component instances. In all, 23,000 lines of composition and parameterization code were written by the user and 293,000 lines of code were generated by LSE from the modules. This indicates a high degree of reuse and is consistent with data previously presented by Vachharajani et al. for non-validated models [16].

4.5. Evaluation

Evaluation was carried out as new components of the processor were added to the model. During the initial phases of refinement, we used simple micro-benchmarks to determine whether modeling errors had been introduced. Benchmark programs were introduced during later phases; their principal purpose was to ensure that the model executed programs properly. After the full model was developed, performance accuracy was evaluated.

The initial model quality is shown in Figure 4, which reports the percentage difference in CPI between the model and the hardware. A positive difference indicates that the model was slower than the hardware, while a negative difference indicates that the model was faster. The input sets are the longest (in instruction count) "train" input (indicated in the name of the benchmark) from each of the SPEC CINT2000 benchmarks. Sampling using the TurboSMARTS framework[18] was used during simulation, with 10,000 to 20,000 samples per benchmark. The error bars indicate 99.7% confidence intervals. Only user-mode instructions are measured and modeled.

Overall error in this initial model was 7.9% with a maximum error of 20%, and it was constructed in 11 weeks. The initial model's accuracy compares favorably with with that reported in the literature [2, 3]. For example, Desikan, et al.'s validated model of the Alpha 21264 achieved an average error of 18.19% on a selection of SPEC CPU2000 benchmarks with a maximum error of 43.0%. This experience supports the Liberty Modeling Methodology claim that validated models can be constructed rapidly.

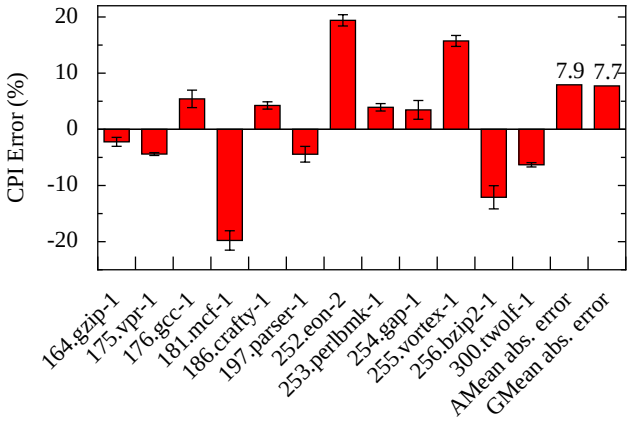


Figure 4. Initial CPI error

5. Aggregate vs. Constituent Error

It is important that models not only be accurate with respect to a baseline, but also that they accurately report the impact of changes to the hardware. In this section, we illustrate that constituent error measurements serve as a better measure of model validity than the pervasively reported aggregate error. We then analyze constituent error in our initial, aggregate-error validated Itanium 2 model.

5.1. An Illustrative Experiment

To evaluate the validity of the initial model, we explore its accuracy in reporting the impact of the addition of instruction prefetching to the hardware. Instruction prefetching was selected because Itanium 2 controls instruction prefetching through software, allowing an exact hardware measurement of the effect of enabling and disabling prefetching. Prefetching is controlled by the `.many` and `.few` completers on branch instructions. We wrote a simple tool that turned off prefetching by rewriting Electron-generated binaries with `.many` branch completers into a binary with only `.few` branch completers.

We compare the speedup achieved by instruction prefetching in the benchmark 186.crafty for the actual hardware and for two models: the initial model described in the previous system, and a model modified to highlight the effects of large offsetting errors. The modified model has higher instruction cache miss and lower data cache miss penalties chosen specifically to offset each other in this benchmark.

Table 1. CPI, speedup, and constituent errors of instruction prefetching for 186.crafty

	Overall CPI			CPI error for baseline / prefetching due to:		
Model	Baseline	Prefetching	Speedup	Front End Stalls	Load-Use Stalls	Other
Actual Itanium 2	0.636	0.623	2.1 %	—/—	—/—	—/—
Initial Model	0.649	0.597	8.7 %	6.2% / 1.2%	-5.1% / -5.8%	0.9% / -0.5%
Modified Model	0.647	0.571	13.3 %	11.7% / 3.4%	-11.0% / -12.4%	1.0% / 0.6%

Table 1 shows the overall CPI, speedup, and errors in performance constituents (relative to hardware CPI) for hardware and both simulation models. The measurement of the performance constituent errors is described more fully in the next section. The baseline overall CPI of both models (first column) is extremely accurate; the modified model is slightly more accurate.

Despite the CPI accuracy of the models for the baseline, both predict too high a speedup (second and third columns) for prefetching. To understand this, we examine the final three columns of Table 1. Both models have too high a penalty for instruction cache misses (here seen as front end stalls) in the baseline case. This excessive cache miss penalty is not seen fully in the overall CPI because it has been "masked out" by offsetting errors in load-use stalls. When the instruction cache misses are reduced through instruction prefetching, the error in front end stalls is also reduced, leaving the error in load-use stalls. This causes the CPI predictions to be too low and the predicted speedup due to instruction prefetching to be too high. Comparison of the initial and the modified model shows that the larger the offsetting errors, the larger the error in the predicted speedup, even though in the baseline case, the model with larger offsetting errors actually has a lower error in overall CPI.

The difference between the speedup reported by the modified model and the speedup of the actual hardware may adversely affect cost/benefit decisions made on proposed hardware or erroneously overstate the impact of published research results. This realization encouraged us to investigate the constituent errors of our initial model in detail and then refine our model to reduce these errors.

5.2. Initial Model Constituent Error Analysis

The Itanium 2 hardware includes performance counters which are able to classify clock cycles as either cycles with completing instructions or bubbles. Bubbles can be further classified by where in the pipeline they originated. These classifications can be used as performance constituents: the total number of cycles is the total number of bubbles of all kinds plus the number of cycles with a completing instruction. By implementing analogous performance counters in the simulation model it is possible to compare the hardware with the simulation model at the level of performance constituents.

The performance constituents are:

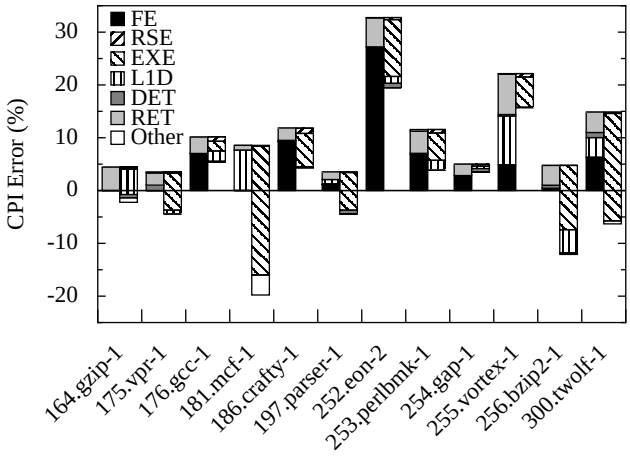


Figure 5. Initial model error constituents

FE - Front-end bubbles – mostly due to instruction cache and instruction TLB misses.

RSE - Register Stack Engine bubbles – cycles in which register spills and fills are being performed.

EXE - Data hazard bubbles – nearly always due to load-use dependencies in these benchmarks.

L1D - L1 data cache pipeline bubbles – from a variety of sources, including hardware page table walks, load-store conflicts, and L2 back-pressure.

DET - Pipeline flush bubbles.

RET - Instruction retirement cycles.

Other - Unaccounted-for differences between the models – due to sampling error and variations in hardware performance between runs.

Figure 5 presents the errors in performance constituents relative to overall CPI for all SPEC CINT2000 benchmarks. In this figure, the size of a bar segment indicates the magnitude of the difference of a constituent of CPI between the two models, while its position (left or right) indicates the sign of the difference. For each benchmark, the left-hand bar contains constituents of CPI where the model is slower than the hardware. The right-hand bar contains constituents of CPI where the model is faster than the hardware. The two bars are aligned at the top so that the distance between the bottom of the rightmost bar and the zero axis indicates the total error in the model. For example, for 181.mcf, the

error in the L1D constituent is approximately +7%, the error in the EXE constituent is about -24%, and the total error is -20%.

The first observation to make from these results is that the performance difference of individual constituents varies widely by benchmark. This indicates that despite similar CPIs among many of the benchmarks, they have quite different behavior (though the converse is not true, similar accuracy does not imply similar behavior.) Some constituents (e.g. L1D) are even positive in some benchmarks and negative in others. This probably indicates that there are offsetting errors within the L1D constituent.

The second observation to make is that there are both positive and negative constituents of error for each benchmark; errors are offsetting, giving better overall results than the individual constituents, indicating that the model quality is not as high as initially believed, despite validation. The data shows that a model validated to a single aggregate metric, in this case CPI, can seem accurate while having substantial error in certain pipeline details.

6. Refining the Itanium 2 Model

Based on the error analysis in Section 5 we iteratively refined the initial model to reduce FE, L1D, and EXE constituent errors. As in the initial model development, the steps consisted of investigating behavior, deciding upon a (new) level of abstraction, updating the model, and evaluating the results.

6.1. Refinements

The largest FE error occurred in 252.eon. Inspection of sub-event counters showed that the L1 instruction cache miss rate was much higher in the model. Inspection of the 252.eon binary showed that it made heavy use of the streaming prefetching hints provided in the ISA. One of the abstractions in the initial model was to ignore the prefetch engine. Thus this error was an abstraction error. The solution to this abstraction error was to implement a simplified (i.e., still somewhat abstracted) prefetch engine. This required less than a day to create a simple state machine to generate prefetch requests, connect the requests to the cache hierarchy, change the L1 instruction cache LRU algorithm to bias against prefetched data, and add some performance and debug monitors.

L1D errors were significant in 255.vortex and 181.mcf. Looking at the sub-events of L1D, we found that the TLB miss rate was too high and the average cost of a miss was too high. The miss rate was a specification error; we had assumed a 4K page size instead of the proper 16K page size. The average cost difference was due to an abstraction error; we modeled TLB misses with a fixed cost. We corrected the page size (a simple parameter change) and replaced the fixed cost TLB-miss model with a less abstract one that performs accesses to the page table. The TLB analysis and fix took less than a day and required changes only to a small portion of the memory hierarchy.

EXE errors were large in several benchmarks, and particularly large in 181.mcf and 300.twolf. Furthermore, after the TLB fix, these same benchmarks had large negative L1D errors, which had been masked previously by offsetting positive TLB errors. Examination of the EXE sub-events showed that it was nearly all due to load-use stalls, while examination of the L1D sub-events showed errors in L2 back-pressure and L2 tag re-reads. L1 data cache miss rates were generally correct. Taken together, this evidence indicated that the L2, L3, and memory models were causing the errors. These errors are abstraction errors.

Reducing these error constituents required an understanding of the memory hierarchy beyond the L1 caches. As before, documents and experiments were used to develop this understanding. Eight days were spent in the investigation. This investigation took so long because the L2 cache subsystem is non-blocking and is able to process requests out of order to improve performance. Bank conflicts, data bypasses, non-LRU cache replacement policies, re-reads of the L2 tag array, and secondary miss processing all combine to make it very difficult to determine what the latency of a particular cache access should be. Furthermore, while the cache controller is richly endowed with performance counters, the documentation of these counters is very limited. Nevertheless, it was possible to understand much of the L2 cache behavior and some of the L3 behavior.

After the eight days of investigation, L2 cache behavior was modeled with a high degree of detail. The total amount of time used in creating the new L2 cache controller was four days. The only changes needed in components other than the L2 cache controller were some minor changes in the L1 data cache to L2 cache interface. The L3 cache and bus interface were modeled in less detail than the L2 cache, but in more detail than in the initial model. Creating the new L3 cache and bus interface model took one day and required changes to no other components of the model.

Note that nearly all the constituent errors in the model were due to specification and abstraction errors, lending support to the claim that structural modeling helps prevent modeling errors. Furthermore, all of the refinements described in this section were achieved in an iterative fashion over the course of 16 days. About half of that time was spent investigating the behavior of the hardware. The reason that modification time was small is due to the structural modeling approach. We found that the natural hardware-based partitioning of the model provided internal interfaces at the locations where changes needed to be made. Thus changes often consisted of simply creating a new portion of

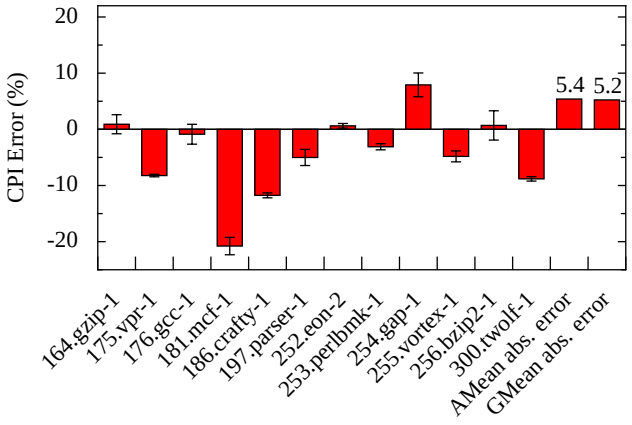


Figure 6. Current model CPI error

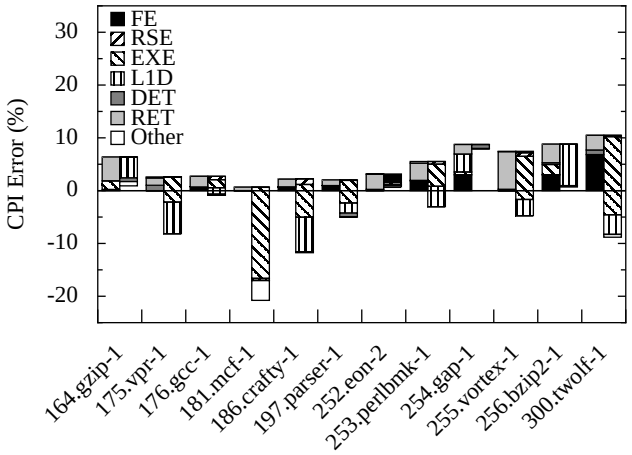


Figure 7. Current model error constituents

the model and hooking it up to the rest of the design in place of the logic it replaced, with no need to even look at other portions of the model. In a few cases where information from other parts of the model was needed, it was accessible and merely needed to be routed to the new portion.

6.2. Current model

The results of the refinement are given in Figure 6. The breakdown of performance constituents is given in Figure 7. These figures show an overall reduction in error, down to 5.4% on average, but more importantly, the targeted constituent errors have decreased significantly. Figure 8 shows the average absolute constituent error across all benchmarks for both the initial and the final model. The FE and EXE error constituents have been significantly reduced. L1D errors have increased slightly because, as discussed before, there were offsetting errors within this constituent. In the 16 days of refinement, overall constituent error decreased by 34%.

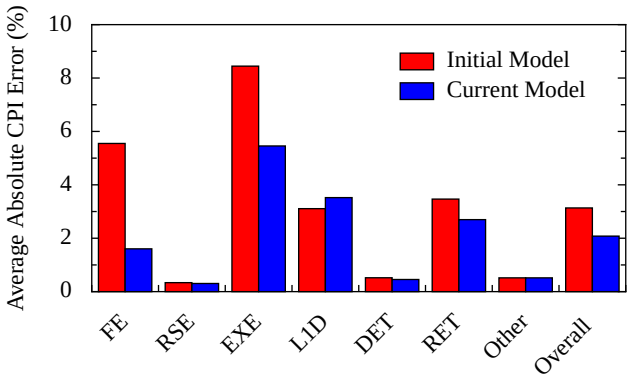


Figure 8. Constituent error change

7. Evaluating flexibility

While the refinement process provides evidence that the Itanium 2 model is flexible, we further show that the model can be used to explore a wider design space. We recount here two independent user experiences.

The first user of the model, the model developer, used the initial model as a starting point for studying modifications to the Itanium 2 processor pipeline that involved adding a limited degree of out-of-order behavior to the pipeline. Instructions were allowed to issue while ignoring load-use dependencies. These instructions would recirculate through the pipeline until the dependencies were resolved. Creating the model for this modified pipeline required modifications only to the portions of the model most closely involved with the changes in the pipeline: the scoreboard logic, the register files, the data cache unit, and the branch resolution logic. This overhaul of the pipeline organization with debugging of the new technique was made in only 6 weeks (including two weeks used to reduce the amount of abstraction used to model the Register Stack Engine), indicating that the initial model could indeed be modified rapidly.

The second user, unfamiliar with the model but familiar with the LSE tools, used the constituent-validated Itanium 2 processor model to explore a novel chip-multiprocessor interconnection mechanism called the synchronization array [14]. This required that he instantiate two Itanium 2 cores and a synchronization array, connect the cores to a shared L2/L3 memory hierarchy, and augment the Itanium 2 datapath to handle the sychronization array instructions. This was accomplished in two weeks; this time includes both the development/debugging time for the model configuration and time spent debugging errors in modified benchmark binaries.

8. Conclusions

Given the central role of simulation in modern processor design and research, a validated baseline model upon which

credible studies can be based is extremely desirable. Unfortunately, validated models are not used because it is widely believed that these validated models are either too time-consuming to develop [4, 2], too difficult to develop due to lack of published information [7], or too time-consuming to modify for wide ranging design-space exploration [4]. Some have claimed that the Liberty Modeling Methodology [17, 16] addresses these issues.

In this work, we present our experience building a validated model of Intel's Itanium 2 processor using the Liberty methodology and the Liberty Simulation Environment (LSE). Though not an ironclad proof or exhaustive study, this experience shows that the Liberty Modeling Methodology and supporting tools can be extremely effective. An initial model was constructed by a single modeler in only 11 weeks. This model predicted hardware cycles-per-instruction (CPI) to within 7.9%. This supports the first set of prior work claims: validated models can be constructed rapidly.

We learned three lessons the hard way. First, we learned that documentation is sometimes in error and often contradictory and vague; any information gathered from it should be validated with quantitative experiments. Second, all approximations should be supported with quantitative experiments that support their validity. Third, and most importantly, we show in this paper that building a model that is validated according to a single aggregate metric does *not* necessarily provide an adequate model for exploring design alternatives. We show that, despite having a high degree of accuracy in an aggregate metric such as CPI, the model can contain significant constituent errors that happen to offset each other.

We were able to apply the Liberty Modeling Methodology to refine our initial Itanium 2 model to reduce the constituent errors with only $2\frac{1}{2}$ additional weeks of effort. This new model predicts overall CPI to within 5.4% with substantially reduced error for the targeted constituents. Furthermore, these Itanium 2 models were modified to explore a novel multiprocessor communication mechanism and novel pipeline organizations for EPIC machines. These significant additional modifications were made in under 8 person-weeks in total. The speed with which these modifications were made supports the second set of prior work claims: validated models built using the Liberty Modeling Methodology can be rapidly modified for design-space exploration.

References

[1] Inside the Intel Itanium 2 processor. Hewlett Packard Technical White Paper, July 2002.

[2] B. Black and J. P. Shen. Calibration of microprocessor performance models. *IEEE Computer*, 31(5):59–65, May 1998.

[3] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, pages 266–277, July 2001.

[4] R. Desikan, D. Burger, S. W. Keckler, L. Cruz, F. Latorre, A. Gonzalez, and M. Valero. Errata on "Measuring Experimental Error in Microprocessor Simulation". *ACM SIGARCH Computer Architecture News*, 30(1):2–4, March 2002.

[5] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad. A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor. In *IEEE Journal of Solid-State Circuits*, volume 37, pages 1433–1440, November 2002.

[6] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–58, November 2000.

[7] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, pages 40–49, February 2002.

[8] Intel Corporation. *Introduction to Microarchitectural Optimzation for Itanium 2 Processors: Reference Manual*. Santa Clara, CA, 2002.

[9] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA, April 2003.

[10] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual, Volume 3: Instruction Set Reference, Revision 2.1*. Santa Clara, CA, 2004.

[11] T. Lyon. Itanium 2 processor microarchitecture overview. Vail Computer Elements Workshop, June 2002.

[12] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE MICRO*, 23(2):44–55, March-April 2003.

[13] Perfmon: An IA-64 performance analysis tool. http://www.hpl.hp.com/research/linux/perfmon.

[14] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[15] D. Soltis and M. Gibson. Itanium 2 processor microarchitecture overview. Hot Chips 14, August 2002.

[16] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 195–206, June 2004.

[17] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pages 271–282, November 2002.

[18] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. Technical Report 2004-003, Computer Architecture Lab at Carnegie Mellon, November 2004.

TraceVis: An Execution Trace Visualization Tool

James Roberts
NVIDIA Corportation
12331 Riata Trace Pkwy
Austin, TX 78727
Email: jroberts@nvidia.com

Craig Zilles
Dept. of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave.
Urbana, IL 61801
Email: zilles@cs.uiuc.edu

***Abstract*— This paper describes TraceVis, a tool for graphically exploring application behavior as it relates to its execution on a microprocessor. Because the human mind can process enormous amounts of visual data–readily identifying trends and patterns–such a tool facilitates performance analysis by allowing raw data to be inspected rather than summaries of pre-selected characteristics. To this end, TraceVis has been designed to enable interactive navigation of many kinds data useful for studying performance problems including: relating the execution to disassembly and high-level language code, annotating the program trace with events (*e.g.*, branch mispredictions and cache misses), and tracing instruction dependencies. In addition, TraceVis provides mechanisms for searching in traces and annotating traces (to bookmark areas of interest or mark regions previously characterized) to allow an analyst to focus their attention on the desired behaviors and regions. Along with exhibiting TraceVis's features, this paper demonstrates how these features can be used in conjunction to analyze the performance of a simulated microprocessor's execution. TraceVis is available in source form for non-commercial use [1].**

I. Introduction

Modern microprocessors have become complex to the point that an intuitive analysis of a program's execution is no longer possible through mere inspection. Innovations such as pipelining, out-of-order execution, memory hierarchies and prefetching have created concurrency and non-determinism in the microarchitecture that obfuscate analysis of even the simplest program. The complexity of modern machines justifies (at least in part) the shift toward qualitative computer architecture research, which has led to the development of a number of high-quality, publicly-available simulation infrastructures.

While these simulators enable collecting performance estimates of novel (micro)architectures, few offer much support into understanding the sources of the changes in performance. Program behavior is often known only in terms of broad generalizations (*i.e.*, summary statistics like average fetch time, IPC, hit rates, etc.). As a result, too many research papers stop short of fully explaining how their proposals impact program execution. We believe this is in part due to a lack of tools that enable the inspection of execution behavior.

A well known alternative to naive aggregation is the use of visualization. Visualization finesses the need for aggregation by exploiting the human's innate ability to efficiently process tremendous amounts of visual information and to identify patterns and anomalies in that information. While a number of high-quality commercially-available tools exist for visualization of parallel program execution (*e.g.*, VAMPIR [2] and PARAVER [3]) and some microprocessor vendors have developed in-house tools (*e.g.*, [4]), the quality of publicly-available visualization tools is not nearly comparable to that of architecture simulators.

To address this need, we have developed TraceVis, a flexible and functionality-rich visualization tool for analyzing microarchitectural simulation data. In designing TraceVis, we had the following eight goals:

1) **Easy access to high-level information.** Coarse-grained information should be available with minimal effort.
2) **Access to increasingly low-level information on demand.** Upon identifying a point of interest at a high-level, users should be able to work down to levels of increasing detail in an intuitive manner.
3) **Interactiveness.** Users should be able to manipulate views and obtain data with minimal latency. Furthermore, the tool should remain responsive regardless of the size of the trace and/or the granularity at which it is being viewed.
4) **Search-ability.** Given a set of user-specified parameters, the tool should be able to locate regions of the trace that match those criteria.
5) **Ability to detect patterns, phases and anomalies.** The tool should aid users in distinguishing regions of similar behavior from those of distinct behavior.
6) **Ability to annotate traces with persistent information.** Users should be able to specify and associate information with a trace as a means for tracking progress, identifying points of interest and conveying their interpretations to collaborators.
7) **Flexibility in usage and extensibility to other systems.** The tool should be general enough that it is capable of visualizing a wide range of microarchitectures.
8) **Visualize everything.** Minimize the need for users to read text-based or numeric results.

This paper demonstrates the features of TraceVis in two ways. First, we briefly overview some of its basic functionality (in Section II). Then, in Section III, we demonstrate a detailed usage scenario that exhibits how these features can be used in performance analysis. Section IV discusses a few implementation issues. Section V provides a brief survey of related work. Lastly, Section VI provides a conclusion.

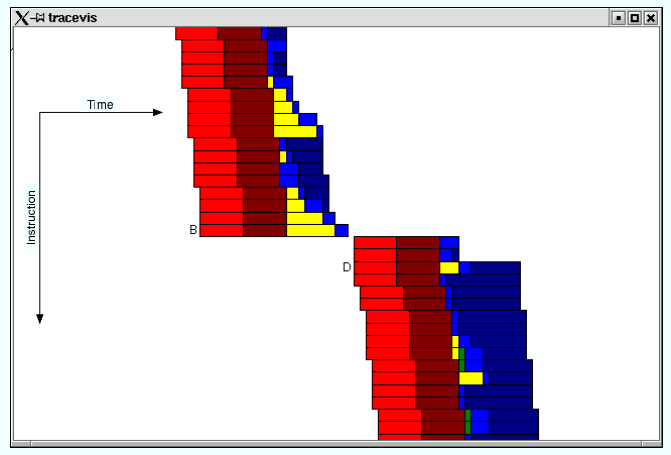


Fig. 1. The basic trace graphing functionality of TraceVis. *Instructions are drawn as horizontal bars and subdivided into color-coded regions each representing a pipeline stage. The 'B' and 'D' denote a branch misprediction and a data cache miss, respectively, for the instruction to its right.*

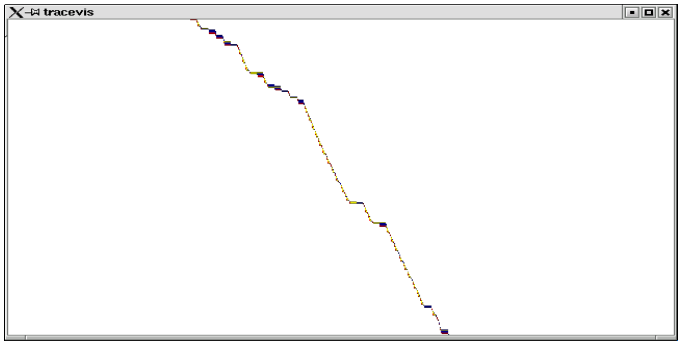


Fig. 2. TraceVis zoomed-out. *TraceVis is capable of rendering at arbitrary levels of zoom. Here 15k instructions are rendered in the image.*

II. Features

In this section we introduce and discuss the key features of TraceVis. We start with the most basic features and proceed to more involved features. Whenever possible we attempt to furnish actual screen shots of the tool to better illustrate its functionality.

A. Basic Trace Graphing

The focal point for TraceVis's functionality is its trace graphing ability. The trace graphing feature enables users to visualize execution traces as instructions flowing through the processor pipeline. Figure 1 shows a sample trace graphing from TraceVis. In the graph, instructions are represented by rectangles whose width denotes the lifetime of the instruction. The instruction bars are sub-divided into color-coded regions which represent different stages in the instruction's lifetime (*e.g.*, fetch-time, decode-time, etc.). The instructions are arranged along the y-axis in program order and along the x-axis according to cycle-time of the instruction's events. All graphed instructions are non-speculative; that is, TraceVis does not display bad-path execution arising from branch misprediction or other misspeculated events.

This representation is similar to the notation used in architecture texts (*e.g.*, [5]) and other pipeline visualization tools [4], [6]. This intuitive representation allows users to quickly determine how instructions flowed through the execution pipeline and how instructions interacted with one another in the course of their lifetimes.

As shown in Figure 1, TraceVis can annotate instructions with a set of events that relate to the instruction. Example events include: branch mispredictions (B), instruction cache misses (I), data cache misses (D) and load/store ordering violation pairs (L,S).

Navigation about the trace (in 2D) can be performed via either keyboard or mouse control. Keyboard control allows users to quickly page up and down a trace. When paging up or down, TraceVis automatically adjusts the x-offset of the trace so that the trace body remains centered in the frame. Mouse-based navigation allows users to interactively wander through the trace by dragging the trace in any direction.

TraceVis is also capable of arbitrary levels of zoom. Figure 2 shows the same trace from Figure 1 pulled back to a much lower level of zoom. From this vantage point it is possible to view large scale program behavior and to quickly survey the trace for regions of particularly interesting phenomena (*e.g.*, low IPC). Interactive zooming then allows users to zoom-in on points of interest and diagnose its cause.

Due to limited pixel space and processing capability, TraceVis reduces the level of detail of its trace rendering as the view is zoomed-out. For instance, beyond a certain level of zoom, the individual event annotations are no longer rendered (in Section II-E we will discuss methods for reclaiming information about these events at this level of zoom).

Also, when zoomed-out beyond a certain zoom-level threshold TraceVis begins rendering only a sampled subset of the instructions within the visible range (*i.e.*, a fixed number of instructions per scan line). This effective sampling of the trace data is necessary in order to maintain interactive frame rates, especially when rendering regions that may include millions of instructions. In practice, we have found that the sub-sampling of the trace data is largely imperceptible at these low levels of zoom.

B. Searching

TraceVis includes a mechanism for searching a trace for a specific instruction address, event (*e.g.*, branch misprediction, cache miss, etc.) or any combination of the two. Figure 3 shows the search feature. The window on the left is where the user specifies the search criteria. Upon initiating a search, TraceVis grays out all the non-matching instructions and moves the first matching instruction to the top of the trace window. Repeating the search function advances the trace to the next matching instruction.

C. Static Code Correlation

TraceVis includes functionality for correlating instructions in the dynamic trace back to the low- and high-level source code. The reverse operation (*i.e.*, correlating static code to

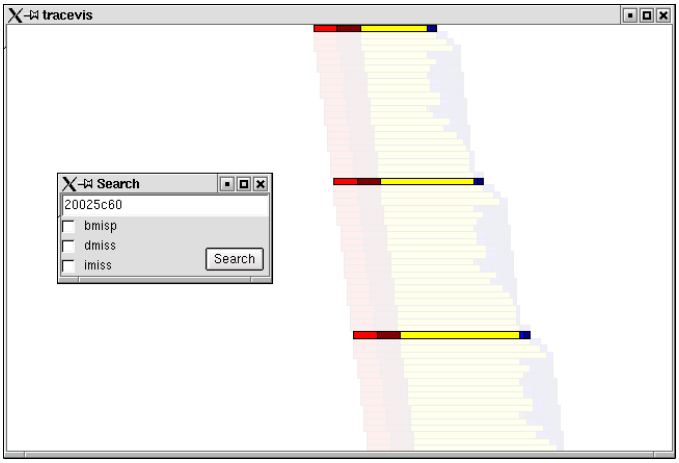


Fig. 3. TraceVis searching for matching instructions. *The first matching instruction is moved to the top. Non-matching instructions are grayed-out.*

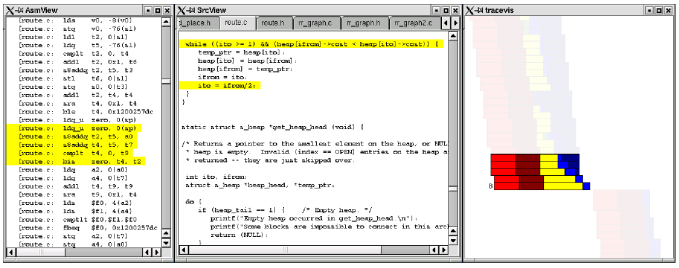


Fig. 4. Dynamic-to-static code correlation. *Highlighted assembly, C source code, and dynamic instructions for the same set of static instructions.*

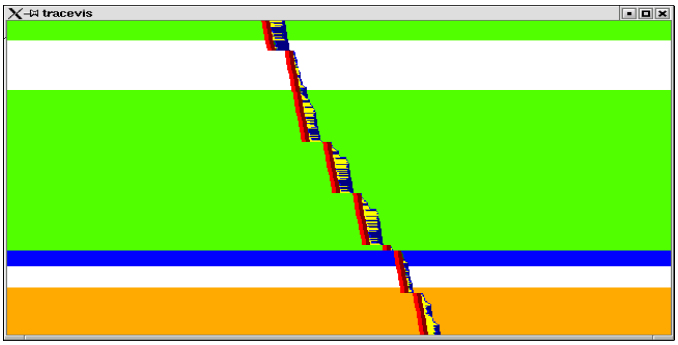


Fig. 5. Static code coloring. *Instructions within a colored region share a common set of instruction address. Uncolored regions contain instructions with addresses that are not in any of the designated sets.*

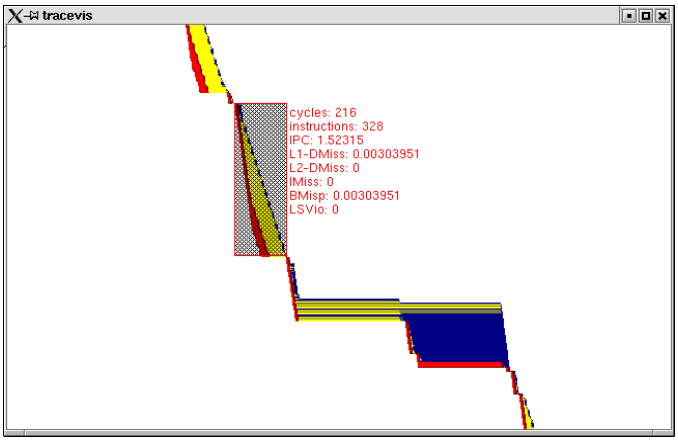


Fig. 6. Regional Statistics. *The statistics pertain to the selected region. The figures shown are fractions, not percentages.*

the dynamic trace) is also possible. Figure 4 shows TraceVis correlating dynamic trace instructions back to both assembly code and C source code.

To use the dynamic-to-static code correlation feature, the user selects a region of the dynamic trace using a bounding box. TraceVis then highlights all lines of high- and low-level source code that match the address of any selected instruction. Since the highlighted regions of text may not be contiguous or even on screen, TraceVis incorporates functionality to allow the user to jump among the highlighted regions of text using keyboard controls.

Correlating static code to the dynamic trace works in a largely symmetric fashion. The user selects lines of source code on either the high-level or low-level views. TraceVis then grays out all but the matching regions in the trace.

D. Static Code Coloring

Figure 5 shows the static code coloring capability of TraceVis. This feature operates in a very similar fashion to the static-to-dynamic code correlation mechanism. The user selects a region of static code and specifies a color for that region. TraceVis then colors regions of the dynamic trace accordingly. Conversely, the user can also select regions of the dynamic trace and request that all static instructions in that region be colored.

The coloring feature differs from the correlating feature in that: 1) multiple colored regions can co-exist on the same trace, and 2) the coloring feature is persistent across TraceVis sessions; that is, the regions and colors are stored to disk for later re-use. On the whole, the correlation mechanism is meant to be a light-weight operation for quick analysis, and the code coloring mechanism is meant to be for less ephemeral annotation.

Region coloring is useful for tracking progress of a user's code exploration. That is, once a section of code has been investigated, the user can color that region to denote that the region (and by extension, all other regions of the same static code) have already been explored. In the same vein, TraceVis provides support for bookmarking so that regions of interest can be annotated. In Section II-E we will discuss how code coloring can be used in conjunction with statistics graphing for the purpose of tracking program phases.

E. Aggregating Statistics

Since TraceVis offers arbitrary levels of zoom, at some point it becomes valuable to summarize data such as cache misses or branch mispredictions into relative rates. TraceVis provides two features for aggregating statistics: region statistics and statistics graphing.

The region statistics feature allows the user to select a

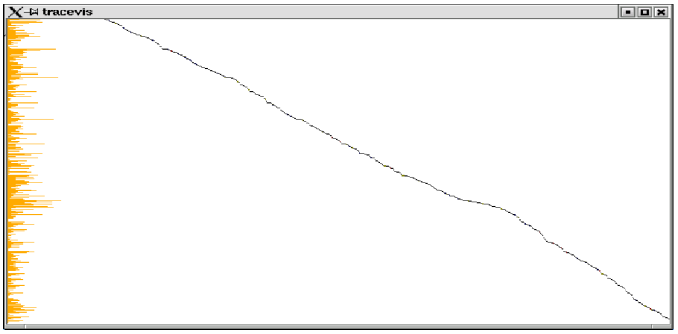


Fig. 7. L2 miss rate visualization. *The histogram depicts the instantaneous L2 miss rate.*

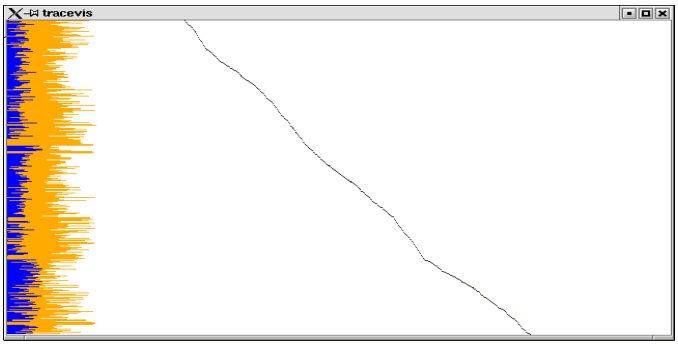


Fig. 8. Code composition visualization. *The colored bars in the histogram reflect the frequency of instructions from the correspondingly colored regions of static code.*

portion of the trace and compute summary information for that portion of the trace. As shown in Figure 6, the region is annotated with the number of cycles and instructions encompassed, the average IPC, and the event rates for the available events. Event rates are reported in number of events per instruction.

The statistics graphing feature plots the frequency of one event type as a histogram along side the program trace. For example, Figure 7 shows TraceVis graphing the relative frequency of L2 cache misses. Each line of the histogram corresponds to the average frequency of L2 cache misses for all instructions represented on that scan line. The statistics graphing feature is particularly valuable when the trace is zoomed out to the a point where individual event annotations (such as those shown in Figure 1) would be too small to be legible.

In addition to plotting event frequencies, TraceVis can plot trace composition with respect to colored static regions, as shown in Figure 8. For each scan line, the corresponding line on the histogram shows the relative makeup of all the instructions represented by that scan line (in terms of the colored regions of static code). For example, a histogram line that shows equal lengths of the two colors implies that the instructions represented on that scan line are composed of an equal amount of static instructions from the two colored regions.

In order to maintain interactive frame rates, statistics graphing currently utilizes a statistical sampling method for obtaining its results. Again, while this methodology adds imprecision to the visualization, we believe the accuracy is sufficient for the purposes of this tool.

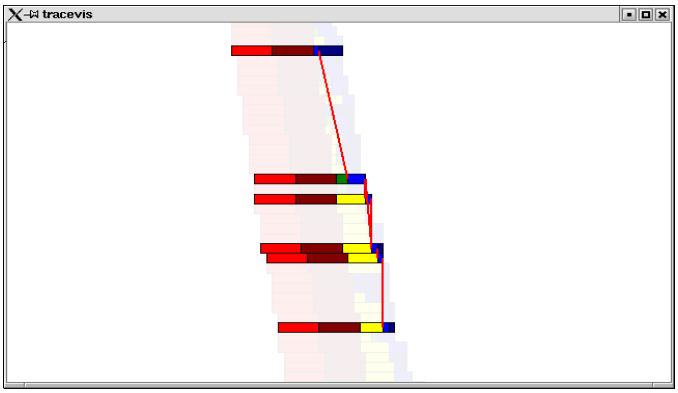


Fig. 9. Dependence Chain Visualization. *The lines between instructions identify the data dependence chain originating from the bottom-most instruction.*

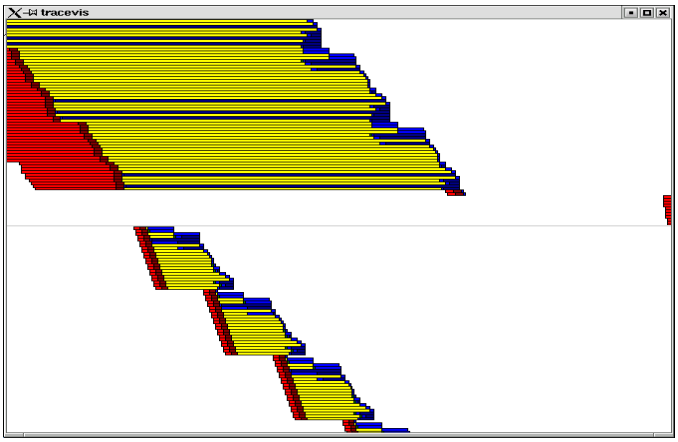


Fig. 10. Multiple Core Visualization. *Two program executions shown simultaneously.*

F. Dependence Graphing

Figure 9 shows the dependence graphing functionality of TraceVis. To use the feature, the user selects a dynamic instruction and requests a dependence graph for that instruction. Then TraceVis graphs out the backward dependence information and grays out the non-involved instructions. Since dependence information potentially extends far back into the trace, the depth of the dependence tree traced is capped by a user-definable setting.

G. Multi-core Traces

Lastly, TraceVis includes support for for visualizing multiple traces simultaneously, which is useful for analyzing parallel or multi-threaded workloads on chip multiprocessors or multi-threaded processors. Figure 10 shows the TraceVis screen split between two separate program traces. The two views are synchronized in such a way that motion or zoom adjustments in either view is automatically mirrored by an commensurate adjustment of the other.

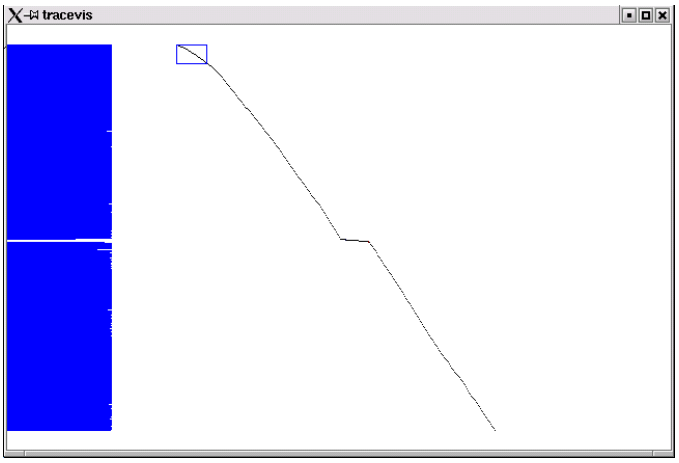


Fig. 11. Visualization of a 10M instruction-long trace of `vpr`. *This trace was captured several billion instructions into the benchmark's execution. The trace with static instruction composition with respect to the boxed region at the trace's beginning; the nearly solid histogram on the left shows that most of the trace uses the same instructions as the beginning of the trace.*

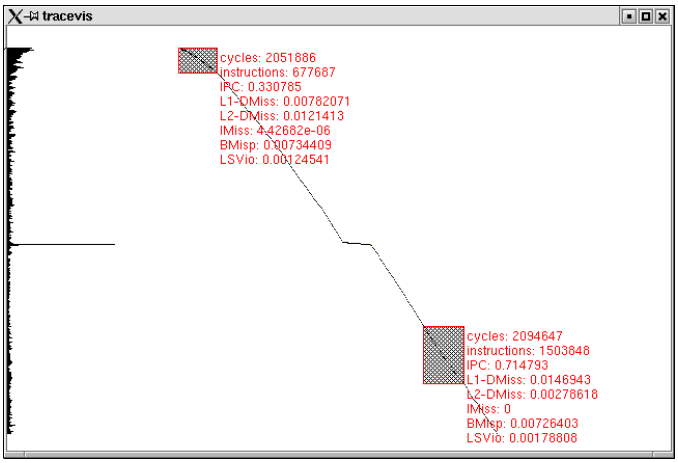


Fig. 12. L2 data cache miss rates and regional statistics (using fractions, not percentages). *Both the histogram of L2 misses and the regional statistics suggest that the warm-up code incurred more performance-degrading events than the steady-state regions.*

III. Use Cases

In this section, we present a sample usage of TraceVis, demonstrating how its features enable an interactive analysis of an execution. For the sake of exposition, we have selected a scenario of modest complexity, but the methodology applies to more complicated examples, as well. We start at a high level and then work down to progressively lower levels of detail.

A. Full-Trace Visualization

Figure 11 shows TraceVis visualizing the full length of a 10M instruction-long trace of the SPEC2000 benchmark `vpr`. This particular trace was recorded several billion instructions into the benchmark's execution. At this high-level view, rendering individual instructions is not feasible or even useful. Rather, instructions are aggregated and rendered as a single summary instruction for each scan line. With 10M instructions and roughly 500 scan lines, it follows that each scan line contains summary information for roughly 20k instructions.

Despite the coarse granularity of the information on this graph, we can gain several key insights into the application's overall behavior. Most obviously, we can gauge the application's IPC (*i.e.*, rate of execution) by observing the slope of the graph's curve. Based on this IPC analysis, we can resolve three distinct phases in the trace's execution: 1) a decreasingly shallow-sloped region at the trace's beginning, 2) a very shallow-sloped and distinctly demarcated region in the trace's center, and 3) the nearly uniform-sloped region that comprises those parts of the trace not in the other two regions.

B. Warm-up Phase

First we investigate the shallow-sloped region at the start of the trace. Here we suspect that the low IPC of this region is the result of not "warming up" the simulated machine before starting trace collection. That is, the low IPC in this region is not a function of the code being executed, but rather is due to training of processor structures such as the branch predictor and the cache (*i.e.*, compulsory cache misses).

Using TraceVis, we can validate our theory by showing that a) the static code being executed in the initial phase is the same as the code executed in the rest of the trace, and b) that performance-degrading events (*i.e.*, branch misprediction rate, cache miss rate, etc.) occur at higher rates in the initial phase than in other regions executing the same static code.

First we prove that the code in the warm-up phase is the same code that is executed in the region of higher IPC. Figure 11 shows TraceVis graphing the composition of the execution trace with respect to colored regions of the static code. The box around the top left portion of the trace is the region from which the instruction addresses were selected. That is, we selected that region of the dynamic trace and colored all the static instructions within the region. Upon graphing out the composition based on our set of selected static instructions, we find that nearly the entire trace is composed of those same instructions. From this, we can conclude that the code in the start-up region is no different from the rest of the trace in terms of the code that it is executing.

Now we show that the low IPC of the warm-up phase can be attributed to warm-up-related events. The TraceVis screen shot in Figure 12 has TraceVis graphing the L2 cache miss rates on the left and two regional statistics on the right. From both the histogram and the regional statistics, we can quickly observe that the L2 miss rate for the warm-up period is significantly higher than that of the rest of the trace (with the notable exception of the horizontal region in the trace's center). Since L2 misses carry a high penalty in this processor model, the degraded IPC could reasonably be attributed these misses.

C. Mid-trace Plateau

Next we investigate the shallow region at the trace's center. As we saw in Figure 11, the region at the center of the trace executes a set of static code that is different from the rest of

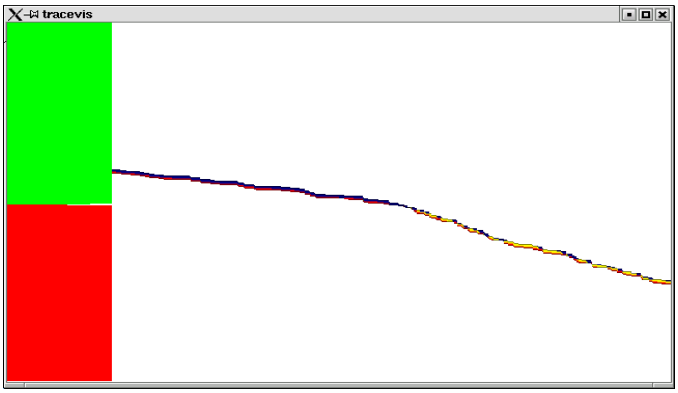


Fig. 13. A detailed view of middle region. *Roughly 6k instructions are represented in this image. The histogram to the left shows the composition of the trace in terms of static instructions. The histogram shows that the middle region is composed of two distinct phases – each one executing a different set of code.*

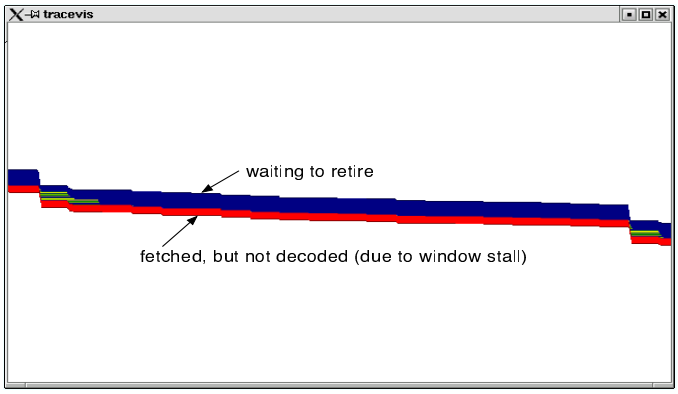


Fig. 14. The top portion of the middle region. *The data in this graph suggests that most of the processor time is spent in fetch stalls and waiting-to-retire stalls. In some cases instructions are in the pipeline for as long as 10,000 cycles. Intuition suggests that the root cause of the stalls is store misses; source code correlation backs up this theory.*

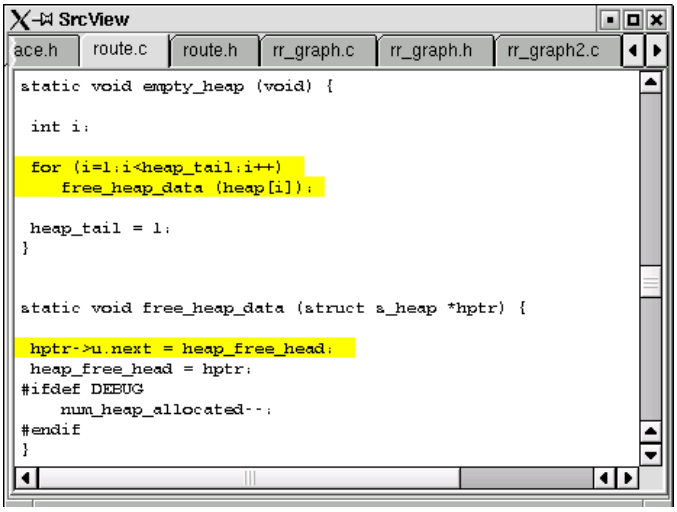


Fig. 15. Source code for the code region of Figure 14. *Re-initializing all of the `hptr` data structure would account for the large number of store misses visible in the trace graph.*

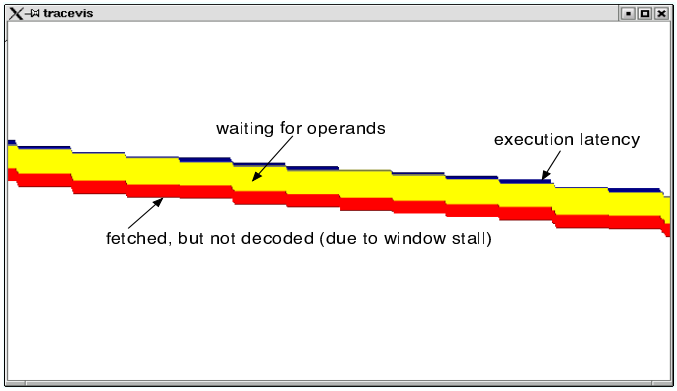


Fig. 16. The bottom portion of the middle region. *The data in this graph suggests that most of the processor time is spent in waiting-for-operand stalls. The stalls are the result of load instructions missing in the L2 cache.*

the trace.

Figure 13 shows a zoomed-in view of the trace's center. The trace visualization shows that the center region itself can be broken down into two distinct regions – each one executing a completely different set of static instructions.

Top Phase: We investigate the top phase of the center region in Figure 14. Here we see that the phase is dominated by two pipeline stages: *fetch* and *waiting-to-retire*. Since our sequentially consistent machine retires instructions in order, we can reason that the long fetch latencies are caused by instruction window back-pressure created by the long *waiting-to-retire* latencies. Using the instruction details information, TraceVis reveals that the long *waiting-to-retire* latencies are themselves caused by L2 D-cache misses. Using the static code correlation mechanism of TraceVis we can determine that the stall inducing instructions arise from the following line of assembly code: `stq a1, 8(a0)`.

This discovery intuitively makes sense: Each store executes in a single cycle – which accounts for the fact that there is very little execution latency observed in this region – but, since the store misses in both the L1 and L2 caches, it can take an additional 400 cycles to retire. Furthermore, from the trace visualization, it appears that the store misses are being somewhat serialized in this region. After careful analysis of the trace, we were able to trace this behavior back to a bug in our simulator which previously went undetected.

Using the same static-to-dynamic code mapping mechanism we can also find the corresponding high-level source code. Figure 15 shows the relevant lines of source code. From the source view we can see that the entire trace region consists of only three lines of source code. We can also see the basic structure of the behavior and the root cause for the performance penalties: the application is freeing a data structure by re-initializing all of that structure's pointers by pointing them to a common target. Assuming that this structure is not within the working set, it is clear why this high-level behavior would cause store misses.

Bottom Phase: Now we investigate the remaining portion

of the trace's middle region. Figure 16 shows this phase to be dominated by *fetch* stalls (again the window has filled) and *waiting-for-operands* stalls. Using the same mechanism as above, we find that the long latencies are caused by load instructions that miss in the L2 cache. This result too makes intuitive sense; the dominance of *waiting-for-operand* behavior in this region can be attributed to long latency load instructions that create chains of stalled dependent instructions.

From the correlated high-level source code, we see that in this phase the application is again re-initializing a data structure. In this case however, re-initialization is performed by walking a linked-list structure. This pointer chasing behavior can reasonably explain the prevalence of L2-missing load instructions, especially if the structure is large and/or not in the working set.

D. Summary

The findings in this section are significant not only in terms of the insight they provide to the workings of the application and the simulator; they are also significant in terms of the manner in which they were obtained. All the conclusions reached were done so using TraceVis's interactive visual environment. Using a high-level summary, we were able to quickly identify regions of distinct behavior. Then, using increasingly low-level information, we were able to identify specific causes of behavior at the level of the microarchitecture, the machine code and finally the programmer-level algorithm.

IV. Implementation

In this section we discuss some of the implementation details of TraceVis. For each case, we state the motivation and and the rationality for our decisions and then present drawbacks and alternatives to these decisions.

A. Language and Windowing Toolkit

TraceVis was written in C++ using the Qt [7] windowing toolkit. The motivation for this choice in language/windowing system was based on the fact that TraceVis needed to be both high performance and cross-platform compatible. In this regard, C++ afforded a good deal of performance, and, since it is available on Windows, Mac and X11-based systems, Qt allowed TraceVis to support multiple platforms easily.

TraceVis does not make use of OpenGL [8] or any other graphics hardware rendering libraries. The decision to use exclusively CPU-based rendering was the result of preliminary tests which showed that the graphics demands of TraceVis could be met by a moderately equipped system (Intel Pentium 4, 2.0Ghz, 512MB RAM) without having to resort to hardware acceleration. If the need arises, we believe that extra performance could be gained by leveraging graphics hardware found on most commodity PCs.

B. Trace File Format

Currently our trace format stores each instruction in 40 bytes: a 64-bit PC field, a 64-bit start cycle, a 16-bit duration for each of seven pipeline stages, three 16-bit fields for holding the distances (in instructions) to producers of its register operands, and a 32-bit vector of event flags. This relatively tight encoding allows a 10 million instruction trace to be stored in 400 MB (or less than 50 MB when compressed with bzip2), which can fit in the physical memory of most workstations. Writing a 10 million instruction trace file only adds 8 seconds to the execution time of our timing simulator.

TraceVis has been designed to be flexible to support retargetting to other simulators and expansion of its capabilities. Specifically, its trace format is encapsulated from the display code, which uses accessor functions to read the trace file, enabling the trace format to be modified with a minimum of effort. Also, we provide a patch for the widely used SimpleScalar simulator [9] to generate our trace format. The statistics counting and display code has been likewise implemented in a generic fashion to allow new metrics to be plotted with a minimum of effort.

C. Multi-View Synchronization

Because TraceVis's internal format–like many other tools–is indexed by instruction, it is not trivial to do the synchronization of multiple views as described in section II-G. The synchronization requires determining the instruction index for a given cycle, when the data structure is set up to do the relation in the other direction. To address this difficulty, we propose an efficient interactive algorithm for view synchronization [10] that avoids the space overhead of building a reverse mapping data structure.

D. Limitations

TraceVis has two main limitations, as it stands. First, no support is available for visualizing wrong path instructions. This could easily be rectified by including the wrong path instructions in the trace, flagged so they could be rendered differently; the drawback of doing so is that then the slope of the rendered trace is no longer a function of average IPC. We believe it would be preferable to display the wrong path instructions next to the correct path instructions, but then there is no longer a linear relationship between the place in the trace and a screen location, something our renderer currently relies on for performance. We are currently exploring a two trace scheme where wrong path instructions are stored in a second trace and rendered as an overlay when zoomed in on the trace.

Second, there is no support to visualize the memory behavior of the program. A few desirable features include: 1) being able to observe memory dependences between instruction in the same way that register dependences are now visualizable, and 2) the ability to relate primary and secondary misses to the same cache block to facilitate distinguishing the number of independent data cache misses for a given region of the program. These features are a focus of future work.

V. Related Work

In this section, we briefly discuss related work. In short, most published tools are either not available or are closed-source products that must be licensed, and and TraceVis compares favorably—in terms of features and performance—with the open source tools.

VAMPIR [2] and PARAVER [3] are representative of the collection of visualization tools designed for performance debugging of parallel applications. These tools focus on the communication and synchronization behavior of parallel programs (typically instrumenting programs by profiling calls to a parallel library (*e.g.*, MPI)), but potentially could be adapted to study questions at the granularity considered by most architecture research. Both of these tools are closed source and require commercial licensing.

Reilly, et al. describe an in-house tool used by Alpha engineers for visualizing Alpha 21264 executions [4] and we suspect that other vendors have comparable tools. While there are similarities with TraceVis, screen shots—the article has few details—show a view where processor state (*e.g.*, which instructions are being decoded) is plotted over time.

Stolte, et al. [11] demonstrate an application of the Rivet Visualization Environment for execution visualization. The Rivet tool offers trace visualization featuring a multi-tiered statistics graph, a view of the high-level source code and a fully-animated reenactment of the instructions working their way through the various processor structures. This tool excels in its ability to simultaneously represent multiple levels of detail (*e.g.*, statistics at different granularities), but lacks a summary view of the program's execution. Neither Rivet nor the described tool has been made available in any form.

GPV [6] is a pipeline visualization tool that, like TraceVis, presents a graphical view of instruction flow through the processor. GPV also provides some degree of statistics graphing (specifically, IPC) as well as assembly-level static information. Additionally, GPV contains functionality to superimpose multiple traces on top of one another, enabling contrasting program executions on two different microarchitectures. GPV, however, is not as feature-rich as TraceVis. GPV does not include methods for searching or annotating traces, nor does it allow for correlation back to the assembly file or high-level code.

TraceVis is also considerably faster and more scalable than GPV. There are two main reasons for this distinction. First, whereas TraceVis is written in C++, GPV is written in Perl/Tk. While Perl/Tk arguably makes GPV more cross-platform compatible, it also comes with a performance penalty versus a compiled language like C++. Second, whereas TraceVis reads its trace information from binary-encoded trace files, GPV uses ASCII-based files as its input. While ASCII is stand-alone human-readable, using it as input to a visualization tool requires an extra parsing stage. Furthermore, ASCII files are necessarily larger than a binary-encoded equivalent.

VI. Conclusion

In creating TraceVis, we strove to meet the standards set forth in the paper's introduction. By this measure, TraceVis largely succeeds in its goals. We have created an interactive program for exploring execution traces and the application/microarchitecture interaction at large.

TraceVis achieves easy access to high-level information through whole-trace visualization which allows users to quickly identify regions of interest. Features such as interactive zooming, regional statistics, code correlation and individual instruction information provide on-demand access to increasingly low-levels of detail—all the way down to the raw trace itself.

TraceVis is able to maintain interactiveness, even when viewing large numbers of instructions, as long as the trace fits in main memory. The key technique employed is sampling. Though sampling is imprecise, users are typically interested in qualitative trends when zoomed out, and these trends can effectively be approximated.

Pattern detection is available in TraceVis via the trace graph and also through statistics graphing. The trace graph offers an opportunity to identify regions of similar IPC, or at a lower level, regions with similar processor activity. Statistics graphing offers the opportunity to investigate otherwise non-visible behavior such as event frequency or code composition.

TraceVis makes persistent annotation available via bookmarking and code coloring. Both of these features enable users to add their own information to the trace and to do so in a way that builds a relationship with a trace that spans multiple sessions.

Finally, the construction of TraceVis was kept general enough that the tool can be applied to configurations other than those presented in this paper. Re-configurable parameters such as the number of pipeline stages and observed events as well as the support for visualizing multiple threads of execution make TraceVis a useful tool for analyzing a wide variety of computing environments.

VII. Acknowledgments

This work was supported by NSF CAREER award 434 CCF 03-47260, grants CCR 03-11340 and EIA-0224453, and equipment donations from AMD and Intel.

References

[1] J. Roberts, "TraceVis: An Execution Trace Visualization Tool," `http://www-faculty.cs.uiuc.edu/ zilles/tracevis`.

[2] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[3] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," European Center for Parallelism of Barcelona, Tech. Rep. UPC-CEPBA-1995-03, 1995.

[4] M. Reilly and J. Edmondson, "Performance simulation of an alpha microprocessor," *Computer*, vol. 31, no. 5, pp. 50–58, 1998.

[5] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, 2nd ed. San Mateo: Morgan Kaufmann, 1998.

[6] C. Weaver, K. Barr, E. Marsman, D. Ernst, and T. Austin, "Performance analysis using pipeline visualization," in *ISPASS*, 2001.

[7] Trolltech, "The Qt Application Development Framework," `http://www.trolltech.com/products/qt/`.

[8] J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide – The Official Guide to Learning OpenGL, Release 1*, 1993.

[9] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," vol. 35, no. 2, pp. 59–67, Feb. 2002.

[10] J. Roberts, "Tracevis: An execution trace visualization tool," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, July 2004.

[11] C. Stolte, R. Bosch, P. Hanrahan, , and M. Rosenblum, "Visualizing application behavior on superscalar processors," in *InfoVis*, 1999.

Evaluating Power Management Strategies for High Performance Memory (DRAM): A Case Study for Achieving Effective Analysis by Combining Simulation Platforms and Real Hardware Performance Monitoring

Karthick Rajamani, Juan Rubio, Wael El-Essawy, Kartik Sudeep and Ram Rajamony
karthick@us.ibm.com, rubioj@us.ibm.com, wrelessa@us.ibm.com, kartik@us.ibm.com, rajamony@us.ibm.com

IBM Austin Research Lab

Abstract

Developing dynamic power management strategies for high performance systems requires a good understanding of the power-performance trade-offs underlining the workload-strategy-system interactions. The feasibility of detailed simulations to help here is severely limited by the range of problem sizes that can be simulated and simulation speed. For our work we want to understand the effectiveness of DRAM power management for a 'realistic' workload - RF-CTH - about 500K lines of Fortran-C benchmark developed at Sandia National Labs. The target architecture is a CMP-based building block for a future high performance computing architecture. We address the scalability and speed limitations for detailed simulation by adopting a multi-step characterization and simulation approach incorporating analysis on current generation hardware using processor performance counters, a fast memory hierarchy simulation engine, a detailed system-level simulator and a power-performance simulator for the DRAM subsystem.

1. Introduction

Detailed full-system simulation is emerging as a key component of computer systems design. Simulators such as Mambo [**1**] and Simics [**2**] provide the ability to model system-level activity in addition to key application activity on a detailed architectural model for the system. Often, such detailed modeling is critical to evaluate the true impact of different design alternatives. While the increased computational power of commercial hardware (that forms the infrastructure for such simulations) makes system-level modeling and simulation feasible, combining the detailed modeling of system-level interactions and detailed timing for architectural-level events still places significant limits on the scale of simulations. Further, when the systems being simulated are larger than the infrastructure for simulations it brings a fresh set of limitations. In this work, we present our approach to tackling these limitations while evaluating the impact of memory (DRAM) power management for a future system design.

The increasing densities from silicon technology and system integration combined with ever-growing demands for computational capacity has brought system power management issues to the forefront in computer design. While processors are dominant power consumers in any computing system, for mid-range and high-end systems, the power consumption of the DRAM sub-system tends to dominate even that of the processors [**3**]. Adopting performance-sensitive power management solutions for the memory system will be a critical component of any system-level power management strategy for larger systems.

Evaluating a memory system power management strategy requires detailed modeling of

- Workload-induced memory system activity,
- System-level and architectural interactions that together with the workload determine the spatial and temporal intensity of DRAM activity,
- DRAM timing and current consumptions, and
- Interaction of all three with the power management strategies.

While the task requires detailed system- and architectural-level modeling it is also critical that real applications be used to drive the simulation as they often exhibit very different behavior from kernel-level benchmarks often used for design-time system parameterization. In this work, we use the RF-CTH code [**4**], a solid mechanics code developed at the Sandia National Labs, as representative of a real workload and

the Stream benchmark as representative of the memory intensive kernel.

Our detailed simulation environment, using Mambo, is limited both in the speed of simulation because of the detail we want to model and in the scale of the problem/system we can simulate as the underlying hardware is significantly small relative to the system we want to model. We address these limitations by adopting a multi-step analysis and modeling approach.

In the first step, we analyze the application on current generation hardware – identify key phases of the application and memory-system related characteristics of each phase. In this step, we collect application characteristics information to drive the next two stages. Based on step 1, we decide on a 'reduced' problem size that can be used to simulate the 'real' problem size. In step 2, we then use a fast memory hierarchy simulator to arrive at the 'revised' system configuration for running the 'reduced' problem size so that it mimics (with respect to activity at the DRAM level) the 'real' problem size on the 'real' system configuration. Finally, in step 3, using the 'reduced' problem size and 'revised' system configuration arrived at in step 2, we evaluate the power management strategy. For this step, we use detailed power and performance simulations by combining the Mambo simulator with a detailed DRAM sub-system power-performance simulator, Memsim [**5**].

There have been a number of efforts to support detailed power and/or thermal simulations by coupling them with detailed architectural level simulators [**6**, **7**, **8**]. While their work is critical to provide a basis for joint power-performance modeling they still face the same limitations with respect to simulation speed and scale as our base infrastructure. An alternative approach to tackling the simulation scale is a proposal for re-scaled benchmarks that presume to capture the key characteristics of real commercial workloads [**9**]. In contrast to their generic solution to the simulation scaling issue, we adopt a more custom approach, where the scale down is tailored to the specific workload(s) and evaluation of interest. This obviously necessitates additional effort but provides the benefits of being a better match with respect to the scenario for evaluation.

The rest of the paper is organized as follows. Section 2 details the issues relevant to investigating DRAM power management – memory power consumption, organization and power management strategy, workloads used for evaluation and the problems faced in evaluating power management strategies. Section 3 gives a brief overview of the infrastructure we use for conducting this investigation. Section 4 outlines our methodology that addresses the problems underlying the investigation. In section 5, we present the results of the analysis by employing our methodology and briefly conclude in section 6.

2. Investigating DRAM Power Management

Independent studies [3] have shown that the DRAM sub-system can be the dominant component of server power consumption. To address this we investigate the usage of the *powerdown* mode of commercial SDRAM. In powerdown, the *CLKE* signal to the DRAM device is disabled and no accesses can be made. As it eliminates the propagation of the *CLK* signal within the device it cuts down the DRAM switching power. Figure 1 shows the estimated relative DRAM power consumption for different states for a DDR3 device (same is modeled in our experiments).

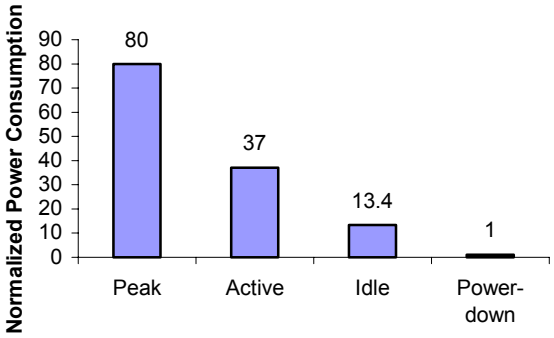


Figure 1: Relative DRAM Power Consumption for Different States.

While powerdown can save considerable power for an idle device it should be used judiciously in performance-sensitive systems as there are entry and exit latencies associated with its transitions and also a minimum duration for each CLKE state (i.e. in powerdown and powered-up). In our evaluations, we model 2 memory cycle latencies for entry and exit and 3 memory cycles for the minimum duration – same as for DDR2, as the DDR3 specifications for these parameters were not yet available. We call the strategy utilizing powerdown that we evaluate in this paper as *queue-aware immediate powerdown* – the memory controller tracks activity in each *rank* (set of devices servicing a single request) and as soon as all the *banks* of a rank are idle and the memory controller queues have no more requests for that rank, the devices of the rank would be transitioned to the *powerdown* mode. Powered-down ranks would be powered up for device *refresh* or for servicing new requests. Our approach provides a simple, easily implement-able solution to DRAM power management.

2.1 System Description and Memory Organizations

The system we are modeling is a chip-multiprocessor (CMP) based building block for a high-performance computing system design. Each CMP chip has a) 4 cores running at 6GHz, b) each with its own 64K L1 D, c) two 2MB L2s (each shared by two cores) and d) a 32MB L3. Each chip has a resident memory controller with 2 4-channel high-speed interfaces connected to fully buffered DIMMS for DDR3-1600 SDRAM (similar to the industry-standard fully-buffered DIMM architecture proposed for DDR2 and beyond), with up to 4 *quad-ranked* DIMMs per channel and a total DRAM capacity of 128GB. All requests to DRAM are at 128-byte cache line size. As memory system performance is closely tied to memory organization, we investigate the impact of powerdown by varying the organizations along two dimensions – the number of channels servicing a single memory request and the number of ranks.

When more channels service a cache request, the data transfer time from DRAM is reduced. However, more devices are utilized for the transfer. The device access energy cost (or active energy) is composed of two main components – activating the row in the DRAM device and the actual read or write operation. The activation energy is solely dependent on the number of devices. The energy for the actual read or write data transfer is roughly the same whether fewer channels (devices) are used to service the request with a longer burst length or more channels are used with a shorter burst length. Thus, fewer channels servicing each request result in lower access energy costs (and active power components) at the cost of slightly longer response times. In our evaluations, we consider two alternatives – 2 channels servicing a request corresponding to a DRAM request burst length of 8 (i.e. 8 transfers are required on the DRAM data pins for a 128-byte transfer) and 4 channels servicing a request corresponding to a DRAM request burst length of 4 (i.e. 4 transfers are required on the DRAM data pins).

When we vary the number of ranks in the system, we vary the available concurrency in the system. In our evaluations, we use the *close page* policy typically followed in server systems – in this policy once the request is serviced the corresponding *bank* in the DRAM devices is pre-charged, which implies that a new request to the same bank would require an activation command to be issued before the actual read or write command. The precharge-activation sequence imposes a minimum restriction of *tRC* between two subsequent requests serviced from the same *bank.* To mitigate this limitation, DRAM devices provide multiple banks on a device, which can be utilized for even a sequential stream by interleaving consecutive addresses across the banks. However, large server systems typically also use multiple ranks of devices with interleaving across the ranks so that the concurrency across the ranks further mitigates the limitation. In general, the ranks (and banks) must be sufficient so that the device bandwidth rather than *tRC* becomes the limiting factor, if any, in servicing memory requests. Note that interleaving across the ranks and banks has an important implication on the simulation approach – a smaller memory sub-system can be effectively substituted in the simulation for a larger real memory system as the interleaving ensures that the intensity of accesses would be similar on both systems as long as the upper level memory hierarchy behavior is the same.

In this evaluation, we consider three different number of ranks – 4, 8 and 16 for each channel. In order to consider the same problem sizes, we match the 4 rank system with 4Gb device size, 8 rank system with 2Gb device size and 4 rank system with 1Gb device sizes – the total memory capacity in all three cases being 128GB. Note that device currents do not increase in proportion to device sizes. So the idle energy component of a system with fewer ranks (larger devices) is lower than the idle energy component of a system with more ranks (smaller devices) for the same total memory capacity.

2.2 Workloads

The key focus of our work is analyzing the impact of power management for real workloads. With our system targeting high-performance computing, we consider the RF-CTH application as a candidate workload. As a comparison point we also evaluate the strategy with the Copy kernel from the STREAM benchmark [**10**].

2.2.1 RF-CTH/CTH

CTH is a multi-material, large deformation, strong shock wave, solid-mechanics code developed at Sandia National Laboratories. It has models for different materials and meshes. It uses second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results. It is used extensively within the Department of Energy laboratory complexes for studying armor/anti-armor interactions, warhead design, high explosive initiation physics and weapons safety issues. CTH is a sensitive application because of its modeling capability. The application we use in this paper, RF-CTH (reduced functionality CTH) is a sanitized version of CTH with the sensitive physics calculations removed. However, it is supposed to capture the essential computing characteristics of the original application.

RF-CTH is implemented in Fortran and consists of approximately 500,000 lines of code. We use MPI for its parallelization with each processor working in its own address space. Given the voluminous nature of this application, it is impractical to analyze it directly on the simulator. Section 4 discusses our approach to this task and section 4.1 provides RF-CTH's characteristics pertaining to our analysis. For the system, described in section 2.1, we decide on the RF-CTH problem size by the largest memory footprint it can accommodate i.e. a problem size of approximately 128GB.

2.2.2 MP STREAM - Copy Microkernel

The STREAM benchmark is widely used to assess system bandwidth limits in high performance computing. It consists of four kernels/loops. We use the Copy loop from the benchmark working on a footprint large enough to miss in all the caches as a stress-test case for the power management strategy. Essentially, the bandwidth requirements of such a workload should offer the minimum idle periods at DRAM device-level and, consequently, the minimum opportunity of employing powerdown. Alternatively, if improperly used i.e. used when there are requests to be serviced and, thus, delaying them, it would also show the maximum performance penalty for employing powerdown. For the 4-way core, we use a multi-processor version of the kernel with each processor handling one-fourth the array space.

2.3 Issues in Evaluating DRAM Power Management

To obtain the performance and power impact of utilizing powerdown it is critical to capture the extent of idleness of each DRAM device and the interval between requests to the same device rank and bank. Capturing this requires an adequately detailed modeling of DRAM level activity. However, activities at the DRAM level are dependent on the entire memory hierarchy. So it is also important to capture the access rates and hit rates for all levels in the memory hierarchy.

The two main limitations for a straightforward detailed simulation to obtain activity at the DRAM level are:

- The problem scale that we desire to model is 128GB while the simulation platform has only 4GB of physical memory.
- The detailed simulation with Mambo has a speed in the range of 500,000 instructions per second (for a 4-way multiprocessor simulation) while running a problem size that is a factor of four smaller than the desired problem size on existing hardware (Power4+ processors at 1.45 GHz) was timed to take over 11 hours (equivalent simulation time of 798 days at 500kips).

In the next two sections, we present the additional infrastructure and methodology used to address these limitations.

3. Infrastructure

As outlined in section 1, we use a variety of tools to assist in our analysis. In this section we give a brief description of them.

3.1 Power4 Performance Counters

Our platform for characterizing CTH is an IBM p650 8-way SMP using 1.45GHz Power4+ processor. The Power4 architecture supports 64 different groups of 8-set performance event counters i.e. they can be programmed to monitor 8 events concurrently. For our analysis, we use a set of groups to obtain the IPC and entire memory hierarchy access information. We use a custom performance monitor library for the Power4 counters on Linux that allows us to sample the counters at up to 10ms intervals with little overhead – we use a sampling interval of 100ms for our analysis. The library allows us to track the IPC with memory hierarchy usage over time enabling us to identify key application phases and their characteristics pertaining to the memory hierarchy such as cache miss rates, access rates etc.

3.2 Mambo

Mambo [**1**] is a full-system simulation toolset for modeling PowerPC-based systems. It provides a range of processor simulators ranging from functional to detailed timing-accurate models. Mambo simulates the entire hardware system: processors, memory hierarchy, disks, network, and other devices, which enable it to boot the operating system. PowerPC applications run on the simulated operating system without modification. Mambo has a large set of configuration parameters, which allows the user to model a wide range of full computer systems. We perform our Mambo simulations on a 2-way Opteron-based Linux box with 4GB of memory.

3.3 E-mambo

E-mambo [**11**] is an execution-driven simulation engine that maintains statistics for applications while executing them natively on a PowerPC machine. Instructions that are executed on the processor are simultaneously simulated by e-mambo thereby maintaining the state of the simulated general-purpose registers and special registers. Architectural studies can be conducted with the help of e-mambo, which provides

a very simple and efficient interface for plugging in any simulated architectural feature. Some of the event handlers that are provided in e-mambo as plug-ins:

- Count - reports simple execution statistics for the program (number of instructions, number of memory references, number of branches, etc.)
- Cache - simulates a cache hierarchy with specified size and associativity for each level and reports hit/miss statistics for the application.
- Locality - simulates an infinite L1-cache to obtain different measures of locality of memory references for the application.
- Tracer - collects instruction traces and memory reference traces for the application.

We use the cache plug-in for tracking the memory hierarchy characteristics. Note that e-mambo is not a full-system simulator. It runs on top of an operating system and we primarily use it for application analysis and preliminary architectural explorations. As instructions are executed on the native hardware it is quite fast and often less constrained with respect to problem sizes that can be modeled compared to detailed full-system simulators.

3.4 Memsim

Memsim [**5**] is a power-performance simulator for SDRAM-based server memory systems. It supports modeling of complex memory system organizations, different interleaving options, different page-mode options, and power management strategies including low-power modes such as powerdown. It supports detailed DRAM timing models and computes DRAM power/energy consumption by tracking activities at the device level accurately. Memsim is highly parameterizable with respect to DRAM device characteristics and memory system organization. It is implemented using an event-driven model and written in C using the CSIM toolkit [**12**]. It can be used in two modes: a memory trace-driven mode and a component-model mode where it can be integrated into a larger simulation setup as a plug-in memory system simulator. In our setup, as Mambo was augmented to model the detailed memory system organization for performance modeling we use Memsim in its trace-driven mode with traces generated by Mambo runs, while ensuring that an identical memory system is modeled in both simulators.

4. Methodology

As mentioned in section 1, we use a multi-step process for our analysis to circumvent these problems. Our steps consist of

- Characterizing the workload on hardware to determine distinct phases, characteristics of the phases, and impact of problem size. Analysis in this step also determines what problem size we choose to use for the detailed simulation.
- Identifying suitable system (cache) parameters so that detailed simulation for the simulator problem size yields similar behavior to the real problem size on the desired system configuration.
- Detailed simulation with power management strategy to quantify impact on power and performance.

4.1 Characterizing Workload

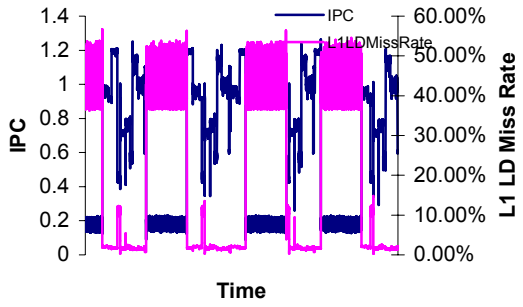


Figure 2: Key Phases for CTH

Figure 2 shows the data from performance counter analysis for a 9GB problem size on one processor – it shows two key characteristics, the IPC and L1D Miss ratio, varying with time. We can identify two distinct phases – a low IPC phase characterized by high L1D Miss ratio and a higher IPC phase characterized by significantly lower L1D Miss ratio. The phase durations are relatively uniform and the phases alternate at regular intervals. Examination of characteristics for other levels in the memory hierarchy provides unique, repeating characteristics relevant to those levels. But, the most distinct phase identification is based on the L1 Miss ratios and associated IPC impact. Based on this behavior, we decide that for detailed simulation we need to capture the memory hierarchy behavior for these two phases accurately. Further, as these phases repeat, we can reduce simulation time by simulating execution of a single "iteration" that contains these two phases (we actually simulate 2 iterations). Note that as RF-CTH employs MPI-based parallelization (with a process per processor) there is little inter-processor sharing of cached data in multi-processor executions. And with equal-sized partitioning that we observed during the hardware execution, the processes share the caches equally which makes it relatively straightforward to infer multi-processor memory hierarchy behavior from a single processor execution.

4.2 Choosing a Simulation Problem Size

Because of the MPI-based parallelization, a 128GB problem size for a 4-way creates similar memory hierarchy activity as a 32 GB problem size for a 1-way with one-fourth the size for the shared cache levels (L2 and L3). So we can use a uniprocessor 32GB problem size execution to obtain the memory hierarchy activity for the 4-way 128GB problem size. However, given our infrastructure, a 32GB uniprocessor execution was also not feasible because of the physical memory size limits of our hardware. We decided to then identify what smaller uniprocessor problem size could emulate the memory hierarchy behavior of a 32GB uniprocessor problem size.

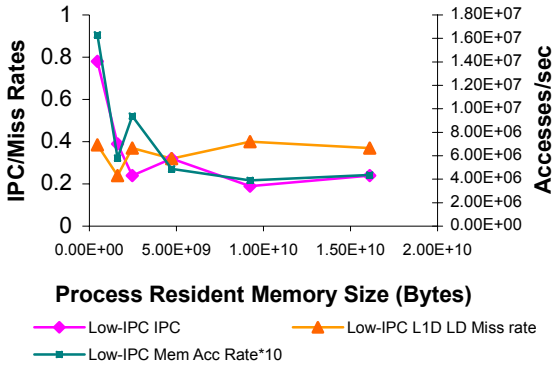


Figure 3: Change in Performance/Characteristics with Problem Size

Figure 3 shows the variation in IPC, L1D LD hit rate and the DRAM access rate with problem sizes on current IBM Power4-based hardware. The different problem sizes are generated by using a different set of valid xyz dimension size for each. As the problem size increases we see the statistics stabilize (the behavior is shown for three statistics for the low-IPC phase, a similar trend is seen for other statistics and for the high-IPC phase). We see that the statistics are essentially similar for problem sizes beyond the 4.7GB range (the fourth point from left in each statistic's line). We conclude that we can emulate the memory hierarchy behavior of a 32GB uniprocessor problem size with any problem size larger than 4.7GB. From this, we decide to use a 9GB problem size for a uniprocessor execution to obtain the memory hierarchy characteristics (this is the 'standard' problem sizes recommended for analyzing the RF-CTH code by its authors).

By this two-step reduction, we conclude that we can faithfully capture the memory hierarchy behavior of a 4-way 128GB problem size RF-CTH execution on a target memory hierarchy of 4x64K L1D, 2x2MB L2 and 32MB L3 execution by a 1-way 9GB problem size execution on a memory hierarchy of 1x64K L1D, 1x1MB L2 and a 8MB L3.

4.3 Choosing System Model Parameters for Simulation

For detailed simulation, we are faced with the constraint that the sum of the application and operating system physical memory requirements should fit within the simulator's physical memory (4GB). From the available smaller problem sizes (determined by valid xyz dimension sizes dictated by RF-CTH sizing rules), we choose the largest one that fits this constraint – a problem size of 1.6GB. This is the problem size for a 4-way execution and a 1-way problem size would be about 4 times smaller. However, as seen in Figure 3, such a problem size would not share the same memory hierarchy statistics as a 9GB problem size that we identified as the equivalent 1-way problem size for our real workload. This implies we need to reduce the cache sizes for the chosen simulator problem size of 1.6GB.

We use e-mambo for identifying the revised cache sizes. For this, we first simulate the 9GB problem size with the cache sizes identified at the end of last section - 1x64K L1D, 1x1MB L2 and a 8MB L3. We obtain detailed time-wise variation of the memory hierarchy statistics – access rates (accesses per instruction) and miss rates for loads and stores at each level. Figure 4 shows a representative L1D LD Miss ratio curve (note that the cache organization modeled here is more aggressive than the hardware caches for Figure 3, hence, the statistics are not directly comparable to those given in Figure 3). We then run the smaller problem size, one-fourth the 1.6GB problem size chosen for the 4-way detailed simulation run, on e-mambo varying the cache sizes and plotting it versus the number of completed instructions (as a measure of progress of execution). For each cache level, we identify the size that gives a stat-curve as close as possible for that level to the corresponding curve of the 9GB problem size. The stat-curves are matched on their peak magnitudes, shape and duty cycle of the two phases among other things.

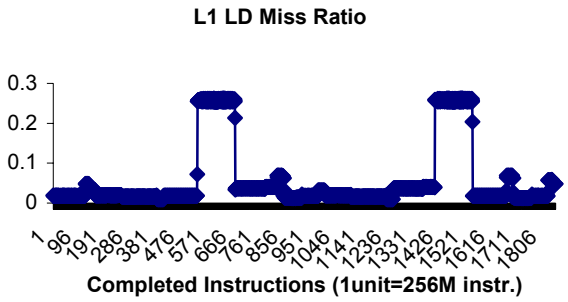


Figure 4: L1 Miss Ratio for the 9 GB problem size

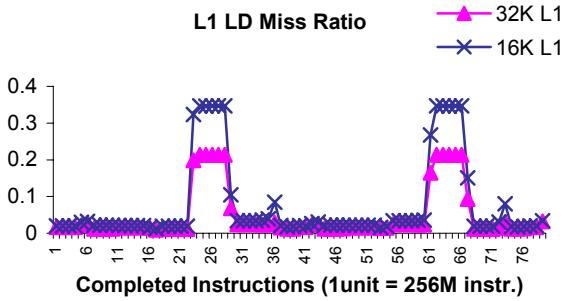


Figure 5: L1 miss ratio for 1-way 400MB (4-way 1.6GB) 'simulator' problem size with different L1 cache sizes.

Figure 5 shows the L1D LD Miss ratio curve for the 400MB problem size for two different L1 cache sizes. Comparing it with Figure 4, one can see that the 32KB size for the L1 D cache provides a better match for this statistic than the 16KB size. A similar matching approach is taken to identify the right size for each of the cache levels by iteratively identifying the right L1 cache size first, then using it when obtaining the L2 size and then using both when obtaining the L3 size. The result of this process is captured in Table 1. Note that this somewhat elaborate but correct procedure results in a not very obvious cache size scale-down for the simulation problem size.

Cache level	Target System (4-way)	Simulation Setup (4-way)
L1D per processor	64KB	32KB
L2 per chip	2x2MB	2x512KB
L3 per chip	32MB	2MB

Table 1: Cache Sizes for Target System and Simulation Environment

4.4 Detailed Simulation Setup

Figure 6 shows the interactions for the detailed simulation. Mambo is fed the execution environment (OS, file system), the application binary, and the appropriate configuration parameters. The execution is driven by an input script file that directs when statistic collection and trace generation are to be done (skipping the non-repetitive initialization phase of the application etc). The memory traces generated from Mambo are fed into Memsim. Memsim simulates the detailed activity and power at the DRAM as a result of the request stream, DRAM timing constraints and power management strategy; and, outputs both the time-varying DRAM power consumption and overall DRAM power and performance statistics.

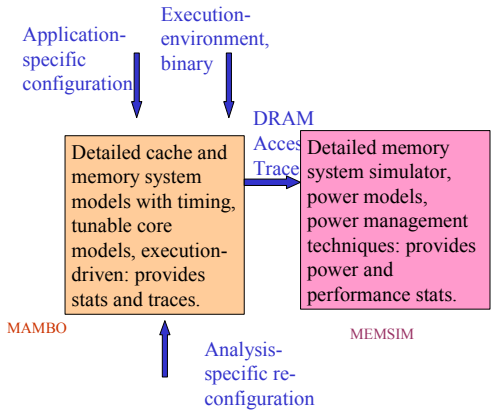


Figure 6: Detailed Power-Performance Simulation with Mambo and Memsim

5. Detailed Simulation Results

In this section we give an overview of the results obtained from the detailed simulation. While these results are the goal for our work, they are not the focus for this paper and so are discussed only briefly.

5.1 Impact of Power Management

Figure 7 shows the impact of power management using *queue-aware powerdown* (see section 2 for description), for RF-CTH. As a contrast, the savings for the STREAM benchmark's Copy kernel is also shown. RF-CTH shows a 72% reduction in power consumption while even the bandwidth-intensive Copy also obtains a 42% reduction in power consumption. Figure 8 provides

the breakdown of the power consumption for the two to illustrate the reason for the difference in benefits seen by the two. In the real workload, RF-CTH, a significant portion of the power consumption is in idle devices which is the component directly addressed by using the powerdown mode. For the benchmark, more than half the power is active that is consumed by the servicing of access requests and so not addressable by the usage of powerdown. The data shown in the figures is obtained for a 16 rank 4-channel wide configuration.

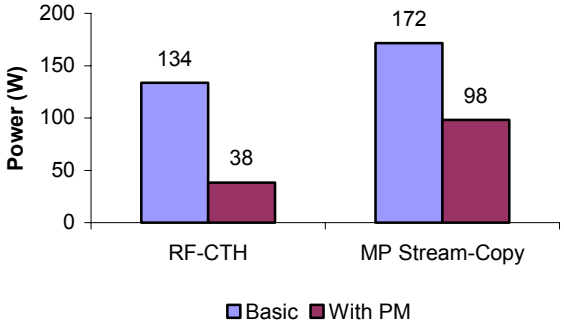


Figure 7: Impact of Power Management

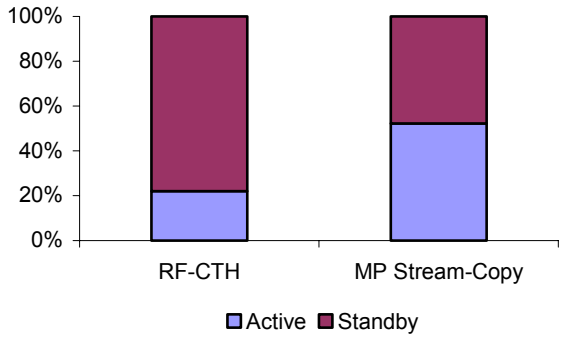


Figure 8: Distribution of Power Components

5.2 Sensitivity to DRAM Organization

In this section, we present the results for different DRAM organizations. Figure 9 shows the performance for RF-CTH and Figure 10 the power consumption. The 1chgrp numbers refer to using all 4 channels in a 4-wide channel group being grouped together to service a request and 2chgrp refers to just 2 channels of a 4-wide channel group being grouped together. Because of RF-CTH's modest bandwidth requirements, the performance is not very sensitive to DRAM organization. However, the larger rank configurations have higher number of devices and so consume significantly higher power (large fraction of which is in idle state). Using powerdown-based power management reduces the power consumption of the larger number of ranks significantly, thus, reducing the difference in power consumption between having larger and smaller number of ranks.

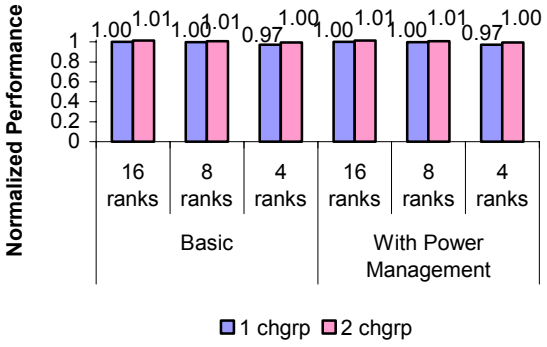


Figure 9: RF-CTH Performance for different DRAM Organizations

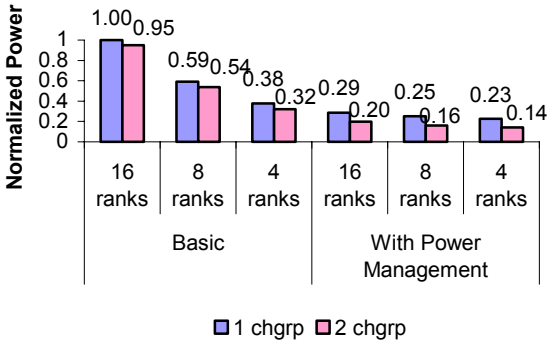


Figure 10: RF-CTH DRAM Power for different DRAM Organizations

Figure 11 and Figure 12 show the performance and power consumption for the MP Stream-Copy kernel executing on all the four processors. Stream has higher bandwidth requirements than RF-CTH. 4 ranks are inadequate to provide this and lower performance compared to 8 or 16 ranks.

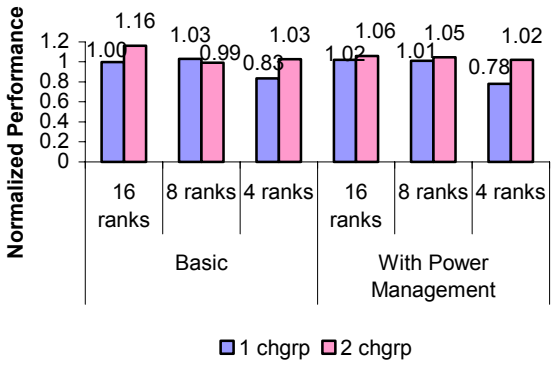


Figure 11: STREAM Performance for different DRAM Organizations

Also with a greater proportion of DRAM power in active state – 2-wide (2chgrp) channel groups has noticeably lower power consumption than 4-wide (1chgrp). Since 2-wide channel group allows more independent requests in flight, it even helps performance a little for the 12 request streams in flight for the benchmark.

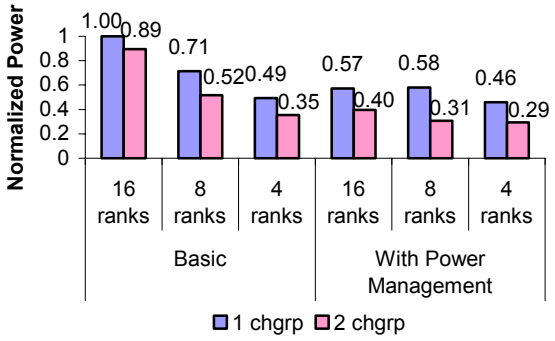


Figure 12: STREAM DRAM Power for different DRAM Organizations

To summarize:

- Power management reduces difference between rank organizations allowing a design with higher available bandwidth (16 ranks) to enjoy power consumption similar to that of a more energy-efficient design (4 ranks).
- Combination of 2-wide channel groups and power-down management attacks both active and idle power consumption yielding the best results.

6. Conclusions

In this paper, we present a methodology for tackling the scaling and speed limitations of detailed simulation-based analysis with real-world workloads. A combination of real hardware analysis and focused (and fast) simpler simulations driven analysis is used to reduce the scale of the simulation problem while retaining the driving characteristics pertinent to the analysis. While we present our approach in the context of evaluating DRAM power management strategies, we believe it to be generally applicable to any memory hierarchy related exploration.

7. Acknowledgments

We gratefully acknowledge Ahmed Gheith, William Speight, Lixin Zhang, Freeman Rawson, Tom Keller, and Mootaz Elnozahy for tools and other help; and, Daniel Phipps, Brett Tremaine, and Paul Coteus for system information that made this work possible. We also thank the anonymous reviewers for their comments and suggestions. This paper is based upon work done in the context of the PERCS project at IBM, which is supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCH3039004.

8. References

[**1**] Mambo -- A Full System Simulator for the PowerPC Architecture - P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and Lixin Zhang, *ACM SIGMETRICS Performance Evaluation Review*, Volume 31 (4), March 2004.

[**2**] SimICS/sun4m: A Virtual Workstation - Peter S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner, *In Proceedings of Usenix Annual Technical Conference*, June, 1998.

[**3**] Energy Management for Commercial Servers - Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kister, and Tom W. Keller, *IEEE Computer*, Volume 36 (12), December, 2003.

[**4**] CTH: A Software Family for Multi-Dimensional Shock Physics Analysis - E. S. Hertel, Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington, *In Proceedings of the 19th International Symposium on Shock Waves*, Volume 1, July 1993.

[**5**] MEMSIM Users' Guide, Karthick Rajamani, *IBM Research Report RC23431*, October 2004.

[**6**] Wattch: A Framework for Architectural-Level Power Analysis and Optimizations - David Brooks, Vivek Tiwari, and Margaret Martonosi, *In Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, June 2000.

[**7**] Microarchitecture-level power-performance simulators - Modeling, validation, and impact on design Z. Hu, D. Brooks, V. Zyuban, and P. Bose., Dec. 2003. *Tutorial at Micro-36.*

[**8**] Temperature-aware microarchitecture - K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, *In Proceedings of the* 30th International Symposium on Computer Architecture (ISCA), April 2003.

[**9**] Simulating a $2M Commercial Server on a $2K PC: Alaa R. Alameldeen, Milo M.K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Mark D. Hill, David A. Wood, IEEE Computer, Volume 36 (2), February 2003.

[**10**] STREAM: Sustainable Memory Bandwidth in High Performance Computers – John McCalpin, *http://www.cs.virginia.edu/stream/*

[**11**] Application Analysis Using Memory Pressure – Kartik Sudeep and Ahmed Gheith, *To appear in Proceedings of the Third Annual ACM SIGPLAN Workshop on Memory Systems Performance (MSP)*, June, 2005.

[**12**] CSIM 19 Simulation Toolkit, Mesquite Software Inc., *http://www.mesquite.com/documentation/.*

A Methodology for Stochastic Fault Simulation in VLSI Processor Architectures

Christian J. Hescott, Drew C. Ness, David J. Lilja
Department of Electrical and
Computer Engineering
University of Minnesota
Minneapolis, Minnesota 55454
{hescott,dness,lilja}@ece.umn.edu

Abstract

We present a simulation methodology for fault analysis within VLSI designs. Our approach uses stochastic fault injection to decrease the required simulation time when performing fault analysis. By using a fraction of the total number of available injection points, we obtain a statistical characterization of the design under test without rigorously testing every gate within the circuit at every point in time. Our approach targets the characterization of fault behavior in large-scale circuits including full CPU architecture designs that would normally be too complex for traditional fault-analysis techniques. We present two methods for performing stochastic fault injection. One requires component library modification and is implemented entirely in VLSI source code. The second does not modify component libraries but rather interacts directly with the simulation environment and is implemented using programming language interface (PLI). We compare the simulation cost of each as well as the trade offs in design analysis. Furthermore, we introduce an optimization strategy when performing fault analysis that utilizes the checkpointing feature typically found in VLSI simulation environments. This optimization eliminates the need for multiple simulations of a single benchmark program with different random seeds. Our techniques can be implemented in any VLSI simulation environment supporting PLI and is compatible with both VHDL and Verilog designs. We present our algorithms and demonstrate their usage on the fully implemented OpenRISC processor [2]. Finally, we validate our method by demonstrating an increase in accuracy when increasing the number of injected faults. We show that even with as few as 250 faults, we see at most an 8 percent difference in the standard deviation and as the number of faults increase, the accuracy quickly approaches 100 percent.

1 Introduction

With the predicted error rate trends in CMOS technology [26] [22], fault tolerance will be pushed further to the forefront of problems in architecture research. It is necessary to have the tools in place for the research community to become more involved without needing to climb the steep learning curves presented by device physics, circuit design analysis and verification. Having a simulation methodology that allows for architecture designers and researchers to begin to look at the problem from their level of abstraction can help to decrease the barrier of entry.

Most research has focused on verification techniques which prove correct functionality or measure the fault coverage of a circuit. However, this can be too stringent when evaluating new fault tolerance techniques especially in the case of studying fault characterization in new architectures and technologies. Typically the circuits are too large to simulate in any reasonable amount of time using the traditional verification techniques. This forces a trade off between level-of-detail and circuit size.

Often, sub circuits must be used for detailed simulations while entire system-level simulations cannot incorporate all of the details such as logical gates and their connecting wires. This can be seen in recently published papers [25],[15],[21] that studied the sensitivity of various super scaler CPU architectures to try to identify which components are the most critical within the design. Due to the large scale of the designs however, simplifications had to be made in order to inject faults and measure their behavior within the system. For example in [25], faults were only injected into latches and not any combinational logic gates. A more inclusive model would use a logical circuit level simulation in which faults can be injected into any gate. Consequently the effects of logical errors as they propagate through the system can be observed in a more realistic manner.

The use of traditional verification techniques becomes

more intractable when complex circuits are studied to inquire the role that each level of design abstraction plays in fault sensitivity. An example application might be determining the most productive abstraction level to target for fault tolerance: circuit, architecture, operating system or software. Performing a study that would incorporate all these levels of abstraction while trying to study the complex interactions of fault propagations within a circuit and how they manifest in each of the abstraction levels would prove to be a truly intractable problem [16].

Statistical fault injection is a simulation methodology that can lessen the total simulation time of a design while still obtaining a characterization to how the entire circuit behaves to faults with a very high level of detail. This is achieved by using a subset of the total fault population. It is assumed that this subset is a good representation of faults encountered under typical operating conditions and consequently the measured circuit's response reflects this.

The remainder of the paper is organized as follows: Section 2 provides motivation for our two statistical fault injection algorithms and a third algorithm for increasing granularity of fault injection. Section 3 provides the details of the algorithm and is focused on implementing them in a simulation environment. In section 4 we provide an example of statistical fault injection on a target design and discuss the trade offs between our two statistical fault injection algorithms. Section 5 presents related work and section 6 concludes the paper.

2 Motivation

2.1 Statistical Fault Injection

We propose a simulation methodology that relaxes the rigid constraints in circuit verification by using a subset of the total possible faults injected into a circuit. By randomly selecting faults from the entire available fault population, we can obtain a statistical characterization of a circuit's behavior to faults and fault propagation. This can lead to shorter simulation times and more rapid testing of newly proposed fault tolerant techniques. Our initial algorithm treats all fault injection points as equally likely, however, this is not required and non-uniform distributions are possible. Our goal is not to obtain a fault coverage metric as in [17], [8], [12], [7], [3], [13] but rather to provide a simulation environment in which a better understanding of fault interaction can be obtained while at the same time providing a method for rapid testing of fault tolerant techniques.

We introduce two algorithms for injecting faults statistically. The first method (referred to as MODLIB) uses a modified component library from a target technology used for synthesis of a design. The MODLIB algorithm injects faults into the outputs of all gates within the design. The decision to inject faults occurs within each synthesized gate and consequently is distributed across the entire design. Our second algorithm (PLInject) uses PLI to create a central fault injector routine that randomly chooses gates to inject faults. This centralized decision routine results in a tremendous performance benefit in terms of simulation time over the MODLIB algorithm. However, the MODLIB algorithm can still have benefits over PLInject when used on smaller circuit sizes as will be discussed in section 4.

2.2 Finer Granularity Using Checkpointing

Characterizing an overall system requires several workloads to be tested while faults are injected. Some type of evaluation criteria is required to determine how a fault effects the system. A typical criteria is to determine weather or not the fault shows up as a software-visible error manifesting itself in one of the architected registers or in memory. Other criteria might include special purpose registers such as the program counter. The extreme case would be costly including every register value within the architecture including memory and caches. A much simpler criteria would be checking that the workload completed successfully. Optimized criteria aim to reduce the amount of storage required by hashing across the memory states and registers of the architecture as in[23].

When performing fault analysis, typically a golden run must be obtained. This golden run constitutes a complete simulation of the workload without any faults injected. Upon completion of the workload, the testing criteria results are recorded as a base case. The simulation is then repeated with fault injection enabled. Upon this second completion of the workload, the results are checked with that of the golden run to determine if the injected fault or faults caused visible errors. Because statistical fault injection is used, several runs of the same workload must be performed with different fault injection points to obtain confidence levels in the results. This can be very costly in simulation time as obtaining good confidence intervals can sometimes take thousands of individual simulation runs for each workload.

The problem becomes even worse when the effects of only a single fault injection are being studied. In this case it can be overkill to simulate an entire workload to see the effects of a single fault injection. Often times, the effects of the fault can manifest as a visible error within a much shorter time period. Consequently the entire workload does not need to be simulated to completion in order to see the effects it has on the system. By shortening the golden run, a single injected fault can be observed on a much shorter time scale. The overall effect is a much more productive simulation.

This is especially true when studying the time sensitivity of an injected fault. An example would be comparing

fault sensitivity during the initialization phase of a workload as opposed to the core processing phase. Measuring sensitivity based on when a fault is injected could conceivably require injecting the fault at every time step during the simulation run. Doing this one fault at a time would require at least $\frac{T}{timescale}$ complete simulations where T is the total simulated time of the benchmark in seconds and $timescale$ is the time scale used in the simulation environment (e.g. 1 nanosecond). It should be noted this does not include the additional runs to obtain confidence intervals. Alternatively, by reducing the time length of the golden run, the total simulation time can be reduced quite significantly while still effectively achieving the same results.

This is exactly the motivation behind our checkpoint algorithm presented in section 3.3. We introduce this algorithm to provide a finer granularity of fault analysis within a DUT that effectively reduces the cost of simulation.

3 Implementation

This section presents our fault injection techniques. The algorithms were intentionally designed to be model-independent so different timing models and fault injection types can be interchanged. Examples include transient pulses, bit flips, stuck-ats and any other injection type that can be modeled as a logical value on a wire.

Section 3.1 presents an algorithm which requires modification to the behavioral specification of a target technology's library. In section 3.2 we show an algorithm that does not require library modification but rather uses PLI to inject faults directly onto wires within the simulated environment.

It is important to note that the two algorithms described here can be used for general fault characterization in a DUT for an entire benchmark run. They are independent of the checkpointing feature described in 3.3. The checkpointing algorithm can be used for higher detailed fault characterization.

3.1 Modified Library Error Injection (MODLIB)

Synthesized designs rely on a behavioral level description of the individual gate types for the target technology when used for simulation. For example a two-input AND gate would look something like that shown in Figure 1. Here the two inputs (A and B) are logically anded together and the output placed onto Z. This along with the path delays constitute the behavior of the gate during simulation.

Our algorithm simply breaks the connection of Z and instead connects it to our own module that is responsible for fault injection (figure 2). The fault injection module can be input sensitive or time sensitive. Time-sensitive fault injection uses a delay parameter that controls how often an fault can be injected. For example a five nanosecond delay would

```
module AND2 (Z, A, B)
output Z; input A,B;
  and (Z, A, B);
  specify // delay parameters
  ...
endmodule
```

Figure 1. Behavioral description of AND2

```
module AND2 (Z, A, B)
output Z; input A,B;
  and (tmp_Z, A, B);
  // Fault inject module
  FAULT_INJECT(Z, tmp_Z);
  specify // delay parameters
  ...
endmodule
```

Figure 2. Fault injected behavioral description of AND2

mean that fault injection would be possible (however not guaranteed) every 5 nanoseconds. Input sensitive fault injection injects faults whenever the inputs of the gate change. The decision to inject a fault is based on the desired model that must be provided to the algorithm. A basic example would use a uniformly distributed pseudo random number generator to sample random values and compare them to the desired fault injection rate. If the random samples fall below the desired rate than an fault is injected.

This method of fault injection is entirely implemented in the VLSI language environment. Consequently no special compilation or simulator modification is required. It does however require a synthesized design with the behavioral source code of the target library.

3.2 PLI Fault Injection (PLInject)

This algorithm targets synthesized gates within a design however it could be applied to behavioral models with slight modification and some limitations. The algorithm assumes the lowest level modules in a design hierarchy to be gate descriptions. This is typically the case in synthesized designs where netlists are created using calls to gates supplied by the target technology's library. Our algorithm injects faults directly onto the wires that connect the gates together.

In order to select which gates are targeted for fault injection, the algorithm performs a depth-first search of a user-supplied module to find all child leaf modules. For each leaf

module, the input and output ports are identified (selectable by the user). The leaf module, ports and external wires connecting to the ports (see Figure 3) are all placed into a hash table for future referencing. Each port of the leaf module is placed in an array and used for selecting a random wire in the fault injection process. Figure 4 shows a design hierarchy represented as a tree. The CPU module is synthesized and consequently only contains netlists of the target library. Its child nodes are leaf nodes and consequently specifying the CPU module targets all its children for fault injection.

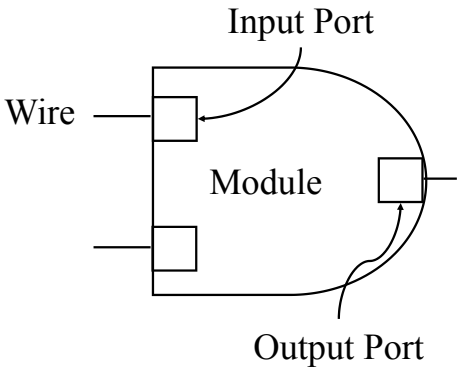


Figure 3. Module, port and wire representations in an AND gate

Targeted fault injection can be accomplished by specifying a subset of the total modules in the hierarchy. Moving higher up the hierarchy selects a larger number of gates for fault injection. Conversely moving down the hierarchy selects a smaller number of gates to the point where specifying a leaf module injects faults into a single gate. Multiple calls can be made to the module selection routine to add additional gates. Any duplicate modules that are found as a result are ignored so each module instance can only be added once.

Gate selection for fault injection is assumed to be uniformly distributed across all ports. However, this assumption can be relaxed by applying distribution weights to the ports in the port array and modifying the port selection routine.

System calls are provided to the VLSI environment allowing easy invoking of a fault injection. A call to the fault injection routine selects a port at random and writes a value onto its connecting wire. Different types of fault injection models can be specified by modifying the fault injection routine.

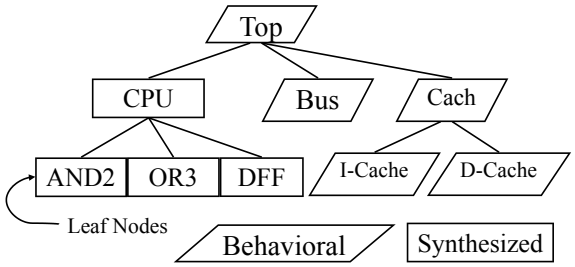


Figure 4. Example design hierarchy with synthesized and behavioral modules

3.3 Checkpoint Algorithm for Fault Injection

In this section we present the algorithm for checkpointing fault injections. The algorithm consists of creating saved checkpoints of the entire simulation environment as a basis to fall back on once the fault injections and testing are complete. Any catastrophic failures or any dormant faults within the system are eliminated when the previous saved state is loaded. With each checkpoint a golden run is performed in which fault injection is disabled. At the end of the golden run, the architected states are saved. These values are used as the "correct" values to compare against once faults are injected into the system.

In figure 5 we show the steps of the checkpointing algorithm. We have broken down the algorithm into three phases: *Checkpoint and Golden Run Creation*, *Injection and Checking*, and *Move to Next Check Phase*. The first phase is responsible for creating an initial checkpoint and performing the golden run simulation. The second phase injects a fault and allows it to propagate before stopping the simulation and checking the results against the golden run. The third phase is responsible for removing the effects of the fault injection and continuing the simulation until the next fault injection point.

The algorithm can be adjusted to control the granularity ($T_{granularity}$) of fault injection and checking. For example, if it is expected that a benchmark is highly sensitive to the time at which a fault is injected , a finer granularity (smaller $T_{granularity}$) can be used and consequently more time steps within the benchmark will be injected with faults and the results checked. A second timing parameter ($T_{latency}$) controls the length of time that faults are allowed to remain in the system before a check is applied. Latent faults may or may not have an impact on the architected state of the program. By adjusting this parameter, checks can be applied to a circuit either shortly after a fault injection or after a relatively long period of time has passed. For example to see immediately visible effects faults have on

the system, a small $T_{latency}$ would be used. Conversely using a larger $T_{latency}$ would measure the long term effects that faults have when they are allowed to propagate through the system and remain dormant for large periods of time.

Checkpoint and Golden Run Creation

1. Create a checkpoint saving entire state of the simulation environment
2. Disable fault injection
3. Run for $T_{latency}$ time steps
4. Create a saved state of all testing criteria (this is the golden run)

Injection and Checking

5. Reload checkpoint created in step 1
6. Enable fault injection
7. Run for $T_{latency}$ time steps
8. Compare architected states with those created in step 4 (record all visible errors)

Move to Next Checkpoint

9. Reload checkpoint created in step 1
10. Disable fault injection
11. Run for $T_{granularity}$ time steps
12. Go to step 1

Figure 5. Steps of the Checkpointing Algorithm

Several fault injection-check pairs can be made for each saved checkpoint by repeating steps 5 through 8 (denoted *Injection and Checking*) and allowing the fault injector to select a new injection point each time. Furthermore, fault injection complexity can be increased by allowing multiple faults to be injected for each pass through the *Injection and Checking* steps. Multiple fault injections can reveal sensitivities in the circuits that are not captured when performing single fault injections [18], [27].

4 Experimental Setup and Results

We demonstrate our algorithms on the OpenRISC processor [2]. This is an embedded scalar processor model that has been successfully implemented on FPGA and ASIC technologies. The design contains a five-stage integer pipeline, instruction and data caches, and virtual memory support. A ported version of the Gnu C compiler is available for compilation of C programs. The OpenRISC 1200 is implemented in behavioral Verilog.

We synthesized the design with Synopsys Design Compiler using Virtual Silicon's UMC 0.18 μm process library available from IMEC [1]. The worst case parameters were used in the compilation. For our MODLIB results, the Verilog behavioral code of the component library was modified as discussed in section 3.1. The random fault injector was set to inject on a 1 ns timescale. For a fair comparison, the PLInject results use the entire synthesized design for random injection with a 1 ns timescale. For the simulation performance comparison (section 4.1), our results are obtained without actually injecting faults as this would not give a fair comparison of simulation cost. Instead, all of the necessary steps for deciding weather or not to inject a fault are measured. The actual injection of a fault is a negligible overhead. For the results presented in section 4.2, however, fault injection is enabled.

The designs were simulated using Cadence's Logic Design and Verification package version 4.1. A 733 MHz Pentium III processor with 512 MB of RDRAM running Linux Redhat Version 8.1 was used for running the simulator.

4.1 Comparison of Two Statistical Fault Injection Algorithms

In this section we compare the advantages of each algorithm presented in section 3. We evaluate the performance cost of each and discuss other factors that contribute to the usability of each algorithm.

The simulation rates (in instructions per second) for the two designs are shown in figure 6 along with three reference cases. The Simplescalar bar is the typical 150 thousand instructions per second mentioned in [6]. The bar denoted RTL is the unsynthesized version of OpenRISC 1200 design and the bar denoted Synth is the synthesized version without fault injection. The Synth result represents the best possible rate we could obtain for our fault injection algorithms. As figure 6 shows, an RTL design simulator is several orders of magnitude below a typical performance simulator (Simplescalar). Moving to a synthesized design results in about a half order of magnitude slowdown compared to the RTL design.

For our PLInject we see another half order of magnitude slowdown compared with the synthesized design. This isn't bad considering that we are adding an additional event every 1 ns of simulated time to check for fault injection. The MODLIB design is almost three orders of magnitude slower than the synthesized case. This is caused from an additional G events occurring every nanosecond where G is

the number of synthesized gates. By distributing the decision mechanism to every gate we incur a tremendous performance cost.

It should be apparent that the PLInject is the only choice when simulating large designs. With a two-order magnitude decrease in simulation performance, the MODLIB method of injection can push simulation times into the range of months or years. However, the MODLIB design can still be usable if the evaluation timescale isn't as small as that used in these results. Furthermore, the MODLIB design can be modified to be input sensitive instead of time sensitive. In this case, a fault is injected with some probability only when the inputs of a gate change. Using this fault injection method is much less costly in terms of simulation time as the decision making only adds a modest amount of overhead and does not create any additional events in the simulator.

The MODLIB design has an advantage over the PLInject method when developing stochastic models for multiple concurrent fault injections. It is more intuitive to map a model onto the MODLIB algorithm as the fault injection decision occurs in each gate. Consequently specifying a model that has a 10% chance of a gate being injected is straightforward and works as expected with multiple fault injections. The algorithm naturally handles multiple faults injected at the same time step without any further specification within the fault injection model.

This is not the case with the PLInject algorithm. Because it uses a single routine for fault injection, the model must specify when more than one fault should be injected concurrently. Developing a realistic model to handle multiple concurrent faults is not entirely straightforward but can be done.

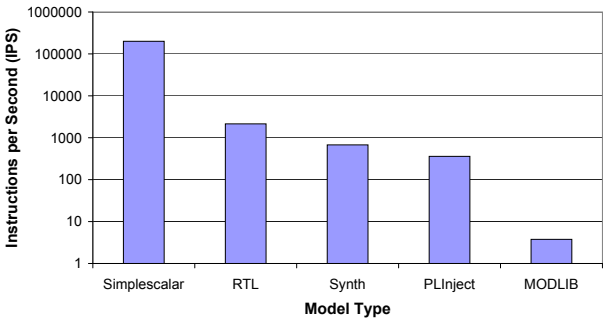


Figure 6. Simulation performance comparison

These results and discussion suggest that neither algorithm is completely superior to the other. Rather the choice is dependent on the experimental setup and the type of information that is desired.

Name	Description	Dyn. Inst.
cbasic	Basic C constructs	15000
dhry	Dhrystone benchmark	50000
except	Exception testing	32000
mmu	Memory management unit test	26000
mul	Multiplication unit test	17000
syscall	System call test	23000
tick	Test tick timer with interrupts	19000
uart	Serial port test	75000

Table 1. OpenRISC Synthetic Benchmarks

4.2 Fault Sensitivity

To demonstrate an example of statistical fault analysis, we conducted an experiment to measure the sensitivity of the OpenRISC processor to injected fault rates. For this experiment we used the PLInject fault injector without checkpointing. A set of 8 benchmark software programs provided with the OpenRISC distribution were used as sample workloads. The different workloads are listed in table 1. Most of the workloads are simple synthetic software benchmarks used to test various functionality of the design. For example the *uart* program tests the serial interface of the OpenRISC processor. A version of the dhrystone benchmark is also included. It should be noted that the OpenRISC simulation environment does not contain an operating system or file I/O. Consequently the benchmarks must use modified versions of printf routines and only static memory assignments. The Linux operating system has been ported to the openRISC environment and provides all suitable routines for standard program simulation. However, a full detailed analysis of fault behavior and its effects on the operating system and industry standard benchmark programs is beyond the scope of this paper. We leave this to future work and instead concentrate on demonstrating the algorithms presented here.

Figure 7 shows the measured response of the openRISC processor to fault injection rates ranging from 10^{-9} to 10^{-1} (horizontal axis). Here the injection rate corresponds to the probability of a fault being injected at every 1 ns of simulated time. For example a benchmark running for 2 ms of simulated time with a 10^{-5} fault injection rate would expect to have roughly 20 total faults injected. Each benchmark was run a total of 10 trials with a different random seed (Note: Random seeds did not vary across benchmarks). Due to the nature of statistical fault injection, both the number of faults injected and their location vary with each different random seed. The measured response is shown on the vertical axis corresponding to the fraction of the eight benchmarks that failed to finish correctly either due to invalid results or runaway program behavior caused by invalid pro-

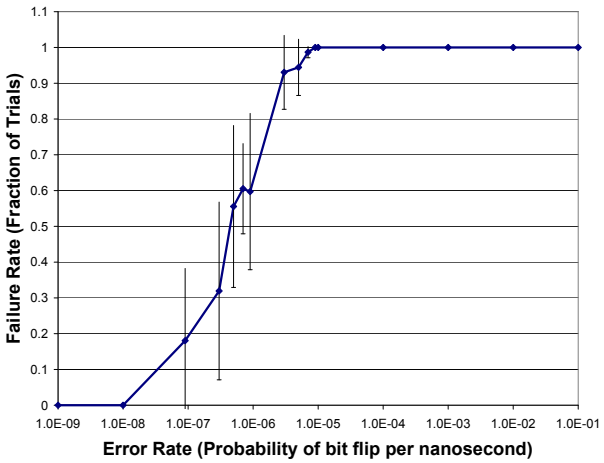


Figure 7. OpenRISC CPU fault sensitivity

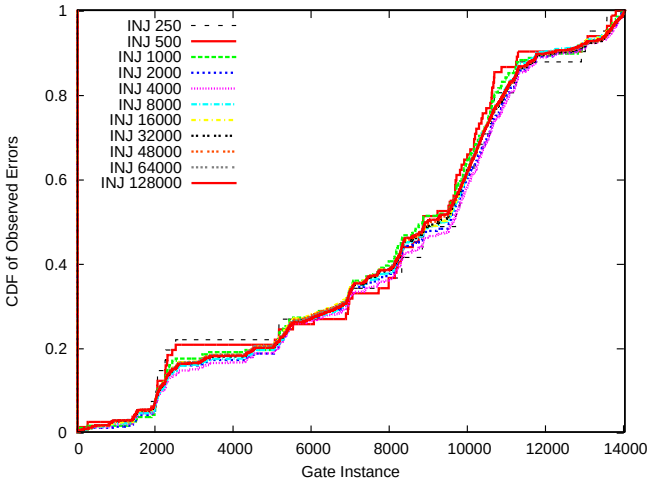


Figure 8. Cummulative error response for each gate running the Dhrystone benchmark

gram counter references. The results are averaged over all trials and the 90% confidence interval is provided in the figure.

These results themselves are not rigorous enough to draw any conclusions about openRISC fault response. However this example demonstrates how this technique can be used to fairly quickly obtain a characterization of a system across different workloads. The total simulation time for each trial of all 8 workloads is on the order of hours.

This type of experiment can help to reveal workloads that are showing higher susceptibility to faults that could possibly be programmatically reduced. For example, a benchmark that fails the experiment every trial at a lower injection rate than the others could suggest that there is something in the design of the program that is extremely sensitive to system errors. Performing a closer analysis of how the faults manifest as errors could pinpoint the weakness of the program and suggest a method to tolerate it.

The response curve in figure 7 could also be used for comparisons of circuit level fault-tolerance techniques. A good fault-tolerance design would shift the curve to the right while poor fault-tolerance designs would be shifted to the left or remain unchanged.

4.3 Validation of Statistical Fault Injection

Statistical fault injection has a distinct advantage requiring fewer fault injections compared to deterministic approaches. However the accuracy of the results is less than what would be obtain with a purely deterministic approach. In this section we derive a measurement of accuracy for our fault injection analysis and show the results for our eight chosen benchmarks.

We define the total fault population available in a given design as the number of gates multiplied by the number of time points at which each gate can receive a fault injection. For example, a design with 50K gates that runs a benchmark for 10 milliseconds and injects faults at a rate of 1 fault/μs would have a total fault population of 500 million injection points.

Figure 8 shows the cumulative distribution function of the error response for a single benchmark. The gates are arranged along the x-axis in increasing order of activity. A single curve in the graph shows the distribution of fault sensitivity within the 14000 gates in the OpenRISC processor design. A gate that is very sensitive to faults will show a steep increase in the response whereas fault-insensitive gates will appear as a flat line. It should be noted, that the shape of the curve is dependent on the ordering of gates along the x-axis, however the results and analysis presented here are independent of this shape.

The plot shows several curves with increasing number of fault injection points. As the number of injection points increases, the response curves tend to converge to a single overall response. Increasing the number of injection points to the total fault population would produce a deterministic experiment. As the number of fault injections increase, the response curve should converge to that of the deterministic fault response. We assume that a larger number of fault injections yields a more accurate measurement and consequently the experiment using the largest number of fault injections is the best and most accurate measurement.

We can quantify the variation seen in figure 8 by measuring the vertical distance from each line to that of the highest measured experiment (128K in figure 8), squaring it, and averaging it over the entire curve (i.e. all gates). Normaliz-

ing this by taking the square root gives us a measurement of standard deviation from the best measured response curve. As the standard deviation approaches zero, the accuracy of the experiment (as compared to a deterministic experiment) approaches 100 percent.

Figure 9 shows this standard deviation versus the fraction of the total population of fault injections. A clear relationship can be seen as the total number of injections increases. For every two orders of magnitude increase in the total number of injected faults the standard deviation decreases by roughly one order of magnitude.

It should be noted that even with our lowest injection fraction (i.e. 1.6×10^{-9} corresponding to 250 total injections) the standard deviation from the best obtained result is only 0.08. In other words even with injecting as few as 250 faults, the overall response of the circuit is within 8 percent of the best measured response. Furthermore, the relationship is an inverse power function and consequently the standard deviation decays very quickly to small values as the number of fault injections increases.

It should be clear from these results that statistical fault injection can be used to approximate fully deterministic fault injection with only a fraction of the total number of injection points and a minimal decrease in accuracy.

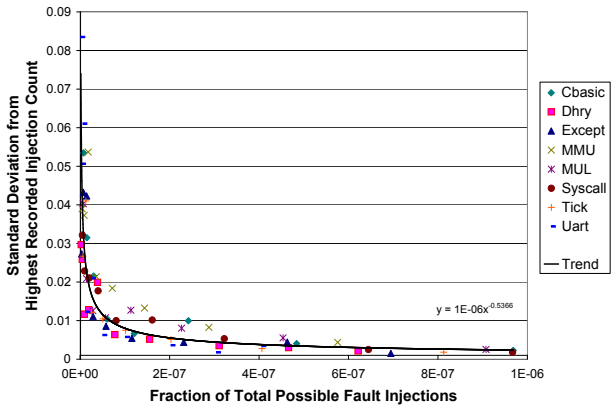


Figure 9. Accuracy quickly approaches 100 percent as the number of injected faults increases

5 Related Work

A fault injection methodology for VHDL is presented in [10]. The use of saboteurs has the same effect as our PLInject technique while the described mutants is somewhat analogous to our MODLIB. However the tool relies strongly on the VHDL language and consequently is not available for Verilog designs. Our methodology presented here can be applied to either VHDL or Verilog designs. Furthermore we incorporate statistical fault analysis into our models.

Alexandrescu et al. present a method for studying fault latching behavior (the likelihood that an injected fault is propagated and latched). Their technique is similar to our checkpointing technique however in [4] the fault injection occurs with latches disabled. Consequently faults are not allowed to propagate beyond the combinational logic being tested. This limits the information obtained during fault injection and prevents the evaluation of overall architecture behavior. Our technique does not have this limitation as faults are allowed to propagate anywhere within the simulated environment.

Mohanram and Touba discuss an approach for implementing targeted concurrent fault detection [14]. They demonstrate their algorithms on a set of synthesized benchmark circuits. However, it is not obvious from the paper if faults are actually injected into the circuits for observation or if the results are only based on error estimation using the presented models.

Mukherjee et al. present a non-statistical approach to characterizing faults in architecture designs [15]. They use a performance simulator to deterministically evaluate fault behavior. The authors note several advantages this has over statistical fault injection including no need to obtain confidence intervals. The authors note the importance of continued use of RTL designs with statistical fault injection when the RTL becomes available in the design cycle. The RTL designs offer more accurate fault-rate estimation as they include all aspects of the design and not just performance-related ones. Our work falls in the category of statistical fault injection with RTL designs. It provides a fairly simple and straightforward implementation of statistical fault injection that to our knowledge has not been introduced into the research community.

Wang et al. perform fault characterization on a custom-designed processor similar to the Alpha 21264 [25]. The processor is implemented as RTL and statistical fault injection used. However, their setup only injects faults into latches and registers. Consequently they're method cannot be used to consider combinational logic effects such as fault masking.

Cha et al. present a simulator for fault injection of transient faults in [9]. It uses two separate simulators: one for detailed timing of a transient pulse propagation and a second for zero-delay logic propagation. The justification is that the detailed timing of a transient pulse only needs to be simulated until it is captured in a latch at which point it resides as a simple logical error. For our implementation, detailed timing of transient pulses must be captured in the model provided to our algorithms. The statistical fault injection algorithms we present are analogous to the zero-

delay logical error simulator in [9] however, our algorithms are easily adapted to current VLSI simulators.

Our technique differs from software fault injection discussed in [5], [11], [19], [24] as we rely on simulation environment to inject faults anywhere into the simulated design. With software fault injection, faults are only injected into portions of real hardware that are directly accessible to software (e.g. registers or memory). This method has its advantages injecting faults onto real machines. However, it is greatly restricted in the types of faults that can be analyzed. Our approach is meant to be a generalized fault injector that is not limited in fault injection locations.

Kim and Somani look at fault characterization of the picoJava-II core processor [21]. They inject faults into behavioral code of the major components (FUBs) of the processor. Consequently they do not inject faults into individual synthesized gates. This has the advantage of less simulation overhead however it limits the type of fault injection experiments that can be performed. Our approach allows a finer granularity both in terms of time sensitive injection and injection location.

6 Conclusions

We've introduced two approaches to performing statistical fault injection into synthesized architecture designs. The PLInject method is clearly superior in terms of simulation time, however the MODLIB can be a more intuitive approach to fault injection if the simulation time permits. Furthermore, the MODLIB is designed entirely within VLSI language requiring no external interfaces.

Statistical fault injection can be largely beneficial for performing rapid analysis of a complex circuit or design. We've shown an example of this where we obtain the fault response to the OpenRISC 1200 processor under varying rates of fault injection. This example demonstrates how the statistical fault injection although not as rigorous as deterministic fault injection still can obtain useful results for circuit characterization. We have shown that the accuracy of our fault injection technique is good even with a relatively small number of fault injections and that the accuracy increases an order of magnitude with roughly two orders of magnitude increase in the number of faults injected.

We've also introduced a method to provide finer granularity of fault analysis which can lead to a more productive simulation environment. By utilizing checkpoint features found in most VLSI simulation software, we can increase the number of fault-check pairs in a circuit simulating an entire workload.

Acknowledgments

This work was supported in part by the Semiconductor Research Corporation under contract no. 2004-HJ-1190 and the Cross-Disciplinary Semiconductor Research Program, National Science Foundation grant CCR-0210197, IBM, the University of Minnesota Digital Technology Center, and the Minnesota Supercomputing Institute.

References

[1] Interuniversity microelectronics center. www.imec.be.

[2] Openrisc 1200 architecture. www.opencores.org.

[3] S. A. Al-Arian and M. A. Al-Kharji. Fault simulation and test generation by fault sampling techniques. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings., IEEE 1992 International Conference on*, pages 365–368, 1992.

[4] D. Alexandrescu, L. Anghel, and M. Nicolaidis. New methods for evaluating the impact of single event transients in ics. In *Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on*, pages 99–107, 2002.

[5] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *Software Engineering, IEEE Transactions on*, 16(2):166–182, 1990.

[6] D. Burger and T. Austin. The simplescalar toolset, version 2.0. www.simplescalar.com.

[7] C. Constantinescu. Estimation of the coverage probabilities by 3-stage sampling. In *Reliability and Maintainability Symposium, 1995. Proceedings., Annual*, pages 132–136, 1995.

[8] M. Cukier, D. Powell, and J. Ariat. Coverage estimation methods for stratified fault-injection. *Computers, IEEE Transactions on*, 48(7):707–723, 1999.

[9] Hungse Cha, E. M. Rudnick, J. H. Patel, R. K. Iyer, and G. S. Choi. A gate-level simulation environment for alpha-particle-induced transient faults. *Computers, IEEE Transactions on*, 45(11):1248–1256, 1996.

[10] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: the mephisto tool. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 66–75, 1994.

[11] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: a flexible software-based fault and error injection system. *Computers, IEEE Transactions on*, 44(2):248–260, 1995.

[12] L. M. Kaufman, B. W. Johnson, and J. B. Dugan. Coverage estimation using statistics of the extremes for when testing reveals no failures. *Computers, IEEE Transactions on*, 51(1):3–12, 2002.

[13] R. M. McDermott. Random fault analysis. In *Proceedings of the 18th conference on Design automation*, pages 360–364. IEEE Press, 1981.

[14] K. Mohanram and N. A. Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pages 893–901, 2003.

[15] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 29–40, 2003.

[16] F. N. Najm and I. N. Hajj. The complexity of fault detection in circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 9(9):995–1001, 1990.

[17] D. Powell, E. Martins, J. Arlat, and Y. Crouzet. Estimators for fault tolerance coverage evaluation. *Computers, IEEE Transactions on*, 44(2):261–274, 1995.

[18] C. Scherrer and A. Steininger. Dealing with dormant faults in an embedded fault-tolerant computer system. *Reliability, IEEE Transactions on*, 52(4):512–522, 2003.

[19] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. Fiat - fault injection based automated testing environment. In *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pages 102–107, 1988.

[20] N. Seifert, Xiaowei Zhu, and L. W. Massengill. Impact of scaling on soft-error rates in commercial microprocessors. *Nuclear Science, IEEE Transactions on*, 49(6):3100–3106, 2002.

[21] Seongwoo Kim and A. K. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *Dependable Systems and Networks, 2002. Proceedings. International Conference on*, pages 416–425, 2002.

[22] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. Proceedings. International Conference on*, pages 389–398, 2002.

[23] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 224–234, New York, NY, USA, 2004. ACM Press.

[24] R. R. Some, W. S. Kim, G. Khanoyan, L. Callum, A. Agrawal, and J. J. Beahan. A software-implemented fault injection methodology for design and validation of system fault tolerance. In *Dependable Systems and Networks, 2001. Proeedings. The International Conference on*, pages 501–506, 2001.

[25] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Dependable Systems and Networks, 2004 International Conference on*, pages 61–70, 2004.

[26] H. S. P. Wong, D. J. Frank, P. M. Solomon, C. H. J. Wann, and J. J. Welser. Nanoscale cmos. In *Proceedings of the IEEE*, volume 87, pages 537–570, 1999.

[27] C. Zhao, X. Bai, and S. Dey. A scalable soft spot analysis methodology for compound noise effects in nano-meter circuits. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 894–899, New York, NY, USA, 2004. ACM Press.

IDDCA: A New Clustering Approach For Sampling

Daniel Gracia Pérez, Hugues Berry, Olivier Temam
gracia@lri.fr,{hugues.berry,olivier.temam}@inria.fr
INRIA Futurs, France

Abstract. Clustering methods are machine-learning algorithms that can be used to easily select the most representative samples within a huge program trace. *k-means* is a popular clustering method for sampling. While *k-means* performs well, it has several shortcomings: (1) it depends on a random initialization, so that clustering results may vary across runs; (2) the maximal number of clusters is a user-selected parameter, but its optimal value can be benchmark/trace-dependent; (3) *k-means* is a multi-pass algorithm which may be less practical for a large number of intervals. To solve these issues, we adapted an alternative clustering method, called DCA, to the issue of sampling. Unlike *k-means*, DCA and its sampling-specific adaptation *IDDCA* do not require the user to be exposed to internal clustering parameters: it dynamically defines the number of clusters for each target program and the method parameters dynamically adapt to the target program. For an ordered input (e.g., a trace of intervals), the method is deterministic. Finally, it is an online and thus single-pass algorithm, resulting in a significant execution time gain over an existing and popular *k-means* implementation. Within the context of a variable-size sampling approach, we show that *IDDCA* can achieve an average CPI error of 1.62% over the 26 SPEC benchmarks, with a maximum error of 5.72% and an average of 403 million instructions.

1 Introduction

Sampling is an accurate and fast solution to increasingly long simulation times (more complex superscalar processors, multi-cores, modular simulation [12, 10], etc). There are two possible approaches for selecting sampling intervals: either (1) pick a large number of uniformly (or randomly) selected small intervals (SMARTS/TurboSMARTS [14, 13]), or (2) pick a few but large and carefully selected intervals (SimPoint [11, 5, 7, 8, 4], EXPERT [6] and our recently proposed budgeted region sampling technique BeeRS [9]). Selected sampling intervals can either have a fixed or a variable size. The sampling accuracy/size tradeoff is more easily and finely adjusted with fixed-size intervals. Variable-size intervals [4], where the interval definition is based on program semantic, may potentially allow even more targeted, and thus accurate, sampling. However, the number of intervals is harder to control and thus potentially large, and the size of intervals can wildly vary. To date, two variable-size sampling methods have been proposed, SimPoint VLI and EXPERT. In the present article, we assume a variable-size sampling method called BeeRS [9], which relies on a simple basic block reuse distance criterion for selecting intervals. The accuracy/size target of BeeRS is to achieve an accuracy of the order of one or a few percents (sufficient for comparing architectures performance) and then to minimize the sampling size. BeeRS also particularly focuses on applicability, i.e., facilitating the use of sampling.

A key step of selecting sampling approaches is naturally how they select sample intervals. The principle is to group together similar program intervals using so-called clustering methods. Currently, SimPoint, the most popular selecting sampling approach, relies on a clustering method called *k-means*. In the present article, we present a new clustering method for sampling, within the context of BeeRS, that addresses three applicability shortcomings of *k-means* (when applied to sampling): (1) the method works by randomly selecting intervals at the start-up phase, so that several runs of the method on the same trace may not provide the same sampling intervals, and thus the same accuracy results; (2) the number of clusters is a user-selected parameter, but its optimal value can be benchmark/trace-dependent, so that inappropriately setting this parameter can degrade either simulation time or accuracy; (3) the method requires multiple passes which may be impractical for a large number of intervals.

The clustering method introduced in this article is called *IDDCA* (*Interleaved Double DCA*), and it is derived from the *Dynamical Clustering Analysis* (DCA) [1] clustering method, and adapted to sampling. We show that *IDDCA* provides consistent results across runs (no randomization phase when the input set is ordered as a set of trace intervals), it automatically determines the appropri-

ate number of samples, and it is about two orders of magnitude faster than *k-means*. The clustering technique currently used in BeeRS would fail on 3 SPEC benchmarks (it would only identify a single cluster, breeding large CPI errors). BeeRS uses a clustering technique which is inspired, but distinct, from *k-means* or its iterative variant *X-means*, and called *Unweighted X-Means*, *UXM*. Plugging *IDDCA* in BeeRS results in less than 6% CPI error on all 26 SPEC benchmarks with an average 1.62% CPI error (assuming 20M instructions warm-up before each interval, which is almost perfect warm-up).

Section 2 presents the BeeRS sampling method for partitioning the program trace into regions. Later on, all clustering techniques are only applied to this trace partitioning method. Section 3 introduces the DCA clustering algorithm, and highlight its differences with *k-means*. Finally, Section 5 presents sampling accuracy and size results and how/why *IDDCA* was derived from DCA.

2 Program Partitioning Into Regions

Beers (*Budgeted Region Sampling*) is a recently proposed sampling method [9] for partitioning the program trace into variable-length intervals. This method is easy to implement and tolerates irregular program control flow; other recent variable-length interval partitioning includes EXPERT [6] and SimPoint VLI [4]. In this section, we introduce the trace partitioning of BeeRS, and in the next sections we investigate the behavior of DCA and *IDDCA* on these variable-size intervals.

Region-Based partitioning. Our program partitioning approach is based on the principle that programs can exhibit complex control flow behavior, even within phases. More precisely, the very principle of phases means that programs usually "stay" within a set of static basic blocks for a certain time, then move to another (possibly overlapping) set of basic blocks, and so on. This set of basic blocks can span overall several parts of multiple subroutines and loops. Moreover, the order and frequency with which these basic blocks are traversed may be very irregular (think of `if` statements with very irregular behavior, think of subroutines which are called infrequently within looping statements, etc...). We call such sets of basic blocks where the program "stays" for a while **regions**. These regions capture the program *stability* while accommodating its *irregular* behavior. We propose a simple method, composed of two rules, for characterizing these basic block regions:

1. Whenever the reuse distance between two occurrences of the same basic block (expressed in number of basic blocks) is greater than a certain time T, the program is said to leave a region.

2. After the program has left a region, application of rule 1 is suspended during T basic blocks, in order to "learn" the new region.

Implicitly, we progressively build a pool of basic blocks: whenever a new basic block is accessed, we examine whether this basic block has been recently referenced (less than T ago); if so, we assume the program is still traversing the same region of basic blocks; if not, we assume the program is leaving this region; then, the second rule gives time for the program to build the new pool of basic blocks.

Selecting T. The only really important parameter in this region partitioning method is T. T represents a trade-off: a too large T and the region sizes can be fairly large, degrading simulation time (in the extreme case where T yields a single region containing the whole code, the accuracy is the best possible and the simulation time is the worst possible); a too small T and the region breakdown is too fine-grained to unveil characteristic groupings of basic blocks, degrading accuracy (if $T = 1$, regions contain a single block and accuracy is very poor). While this trade-off seems delicate at first sight, we found the accuracy of the region partitioning method was largely, if not remarkably, tolerant to variations of T.

Still, we want to find T in a manner that is practical for the user, i.e., architecture-independent. Thus, we need to set T only once for each benchmark/data set pair (independently of the target architecture), much like SimPoint provides architecture-independent simulation points; the final recommended T values are indicated in Table 1. Let us now explain how we find these T values.

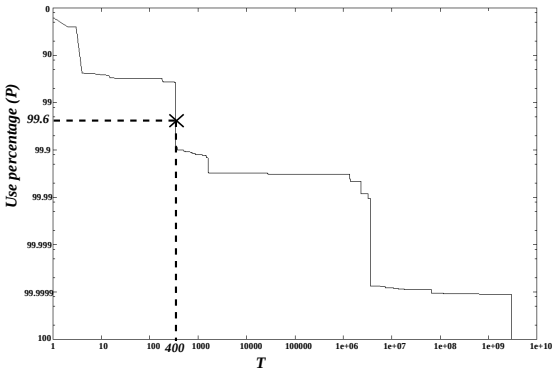


Figure 1: *Selecting T from P for* `swim`.

Since T determines what reuse distances are captured by regions, a fixed value of T can potentially miss key

SPEC	Number of Instructions	T	Num. Regions	Insn. per Region
ammp	326,548,908,728	45,000	183,558	1,778,996
applu	223,883,652,707	1,500	187,278	1,195,462
art	41,798,846,919	1,500	112,350	372,041
bzip2	108,878,091,744	25,000	170,903	637,075
crafty	191,882,991,994	100,000	199,499	961,824
eon	80,614,082,807	20,000	194,912	413,592
equake	131,518,587,184	2,000	196,991	667,637
facerec	211,026,682,877	35,000	196,206	1,075,536
fma3d	268,369,311,687	15,000	184,667	1,453,260
galgel	409,366,708,209	70,000	111,399	3,674,779
gap	269,035,811,516	90,000	192,658	1,396,442
gcc	46,917,702,075	20,000	95,529	4,911,357
gzip	84,367,396,275	30,000	170,966	493,475
lucas	142,398,812,356	100	187,849	758,049
mesa	281,694,701,214	80,000	187,916	1,499,046
mgrid	419,156,005,842	2,500	54,440	7,699,412
parser	546,749,947,007	300,000	177,738	3,076,157
perlbmk	39,933,232,781	100,000	41,866	953,834
sixtrack	470,948,977,898	9,500	183,823	2,561,970
swim	225,830,956,489	400	75,740	2,981,660
twolf	346,485,090,250	200,000	161,142	2,150,184
vortex	118,972,497,867	80,000	190,722	623,806
vpr	84,068,782,425	8,500	193,173	435,199
wupwise	349,623,848,084	200,000	13,696	25,527,442
Average	231,987,140,463	61,130	151,915	2,712,371

Table 1: *Region statistics and T.*

reuses in certain programs or conversely insufficiently discriminate regions in other programs.[1] A more benchmark-tolerant way to capture "enough but not too many" reuses is to set T for each benchmark such that a fixed percentage P of reuse distances are captured in regions. Then, we can deduce the corresponding value of T based on the basic block reuse distance distribution, see Figure 1 for benchmark `swim`. During a training run (emulation only, no simulation, the run is architecture-independent, performed once for each benchmark/data set pair), we record the basic block reuse distance distribution, and the region partitioning for a large range of T values.

The issue now is to find the appropriate value for the percentage of reuse P. P influences accuracy and time; since time (number of simulated instructions) is an architecture-independent characteristic, we select P based on a time target (rather than an accuracy target) in order to fulfill our architecture-independence constraint. We have currently set this time target at $\simeq$200 million instructions, but it is a user-adjustable toggle that can be increased/decreased at the benefit/expense of accuracy. Based on the time target, we now want to find P. Recall the same value of P is used across all benchmarks, so the time target will be fulfilled in average, across all benchmarks. Since we know how to find the T value for each benchmark based on P and the training run, and since we know the number of instructions (time) for each value of T thanks to the training run, we can compute the average time over all benchmarks for each P value; conversely, for a given time target, we can deduce P. Based on the training run and the average time target of 200 million instructions, we found $P = 99.6\%$. The corresponding values of T for each benchmark, along with other region statistics, are indicated in Table 1.

This heuristic allows to appropriately set T and to achieve a good accuracy/time tradeoff, but we do not claim it is optimal. We intend to investigate better T selection heuristics in the future.

Note that the value of P (and consequently T) depends on the clustering method. In this article, we have estimated P using *UXM*, and we use this value throughout the article, whatever the clustering method.

***UXM* and motivation for *IDDCA*.** In order to apply SimPoint to variable-size intervals of BeeRS, the new SimPoint VLI approach should be used, which weights the intervals with their size (number of instructions) during the clustering method as described by Lau et. al. [4]. This weighting is important for accuracy because it affects the relative position of the centroid (center of mass) of a cluster of intervals with respect to all its intervals (for each cluster, the interval that is located the closest to the centroid is chosen to represent the cluster; if the centroid location is incorrectly estimated, a sub-optimal representative interval may be selected). Weighting implicitly ensures that the importance of an interval is correlated to how many instructions it contains. Since SimPoint VLI is not yet released, we instead compare *IDDCA* with *Unwighted X-means*, and we use SimPoint 2.0 as the implementation for *UXM*.

However, BeeRS aims at reducing size as much as increasing accuracy, not privileging accuracy over size at all costs. For that reason, we experimented with a method that works like *k-means* but which ignores the size of the intervals, i.e., not weighting the intervals with their size. This simple trick has the effect of avoiding that larger intervals are systematically privileged, though it may come at the expense of accuracy. This method is no longer *k-means*, hence the *UXM* name, but we found it almost achieved the intended BeeRS target accuracy with a reasonable sampling size of 160 millions instructions in average. However, the effect of not weighting the cluster can sometimes badly skew centroids locations (the unweighted centroid may be far from the more representative weighted centroid) resulting in inappropriate or wrongly chosen intervals. For 3 benchmarks, `apsi`,

[1] Note however that we did observe very good average accuracy/time tradeoffs for the same T value applied across all benchmarks, sustaining the above mentioned tolerance to T variations.

`gzip` and `mcf`, this effect particularly shows since only a single cluster is identified by *UXM*, resulting in very poor accuracy for two of the benchmarks (CPI error of 26.35% for `apsi`, 3.65% for `gzip` and 93.19% for `mcf`, with a 20-million instruction warm-up before each interval).

Thus, the motivation for *IDDCA* was twofold: the shortcomings outlined in the introduction, and finding a clustering method with a reasonable accuracy/size trade-off and no inaccuracy singularity as with *UXM*.

3 Dynamic Cluster Analysis (DCA)

Once a program execution has been divided into intervals (called regions in BeeRS) we can cluster them on the basis of their similarities. In our case, a data point is a region, i.e., a vector of execution frequencies, one per region basic block. All the clustering results of this article are applied to the BeeRS intervals.

Briefly, the aim of clustering algorithms is to classify data points into separate clusters obeying the following rule: a given data point must be more similar to any data point belonging to the same cluster than to any data point picked in a distinct cluster. Because the number of basic blocks per region can be high, the vector dimension can be large, making clustering a fairly time-consuming task. In our case, the number of static basic blocks ranges from 1,236 for `art` to 35,202 for `gcc`. In order to reduce computation time, clustering methods usually pre-process data points by reducing the vector dimension using projection. *Random linear projection* [3, 11] is a fast and simple method to reduce the dimensions without degrading too much the information of the input data. SimPoint uses this technique to reduce the vector dimension down to 15.

3.1 Algorithm

DCA is an alternative clustering method recently proposed by Baune et al. [1]. DCA dynamically defines the number of clusters and their centroids, and constantly revisits intermediate decisions. This dynamic process relies upon three different parameters: Θ_{new}, Θ_{merge} and Θ_{step_factor}.

DCA starts with no cluster and the list of regions to cluster (called R), and executes the following steps:

1. Pick a region (r) from the list of regions R; if there is no cluster yet, create a first cluster containing region r and go to step 5.

2. Find the cluster (c_i) with the closest centroid to the current region r and compute the distance (d) between r and the centroid of c_i.

3. If d is greater than Θ_{new}, then create a new cluster containing the current region r.

4. If d is less or equal to Θ_{new} then:

   - Add r to cluster c_i and update c_i centroid accordingly.
   - Find the cluster (c_j) with the closest centroid to that of c_i. If the distance between the centroids of c_i and c_j is less or equal to Θ_{merge} then merge the two clusters into a unique one and compute its centroid.
   - Update Θ_{new} and Θ_{merge} thresholds so as to make cluster creation and merger more difficult. For that purpose, increase Θ_{new} and decrease Θ_{merge} as follows: $\Theta_{new} = \Theta_{new}/\Theta_{step\_factor}$ and $\Theta_{merge} = \Theta_{merge} \times \Theta_{step\_factor}$ (for $\Theta_{step\_factor} < 1$).

5. Remove r from the list of regions R. If there are no more regions in R, then the process terminates, otherwize go to step 1.

At the end of this process DCA has created a set of clusters. The sampled points are the closest regions to the clusters centroids.

Intuitively, Θ_{new}, Θ_{merge} and Θ_{step_factor} control the creation and merging of clusters. Their initial values are benchmark specific. We first compute the centroid of all the regions (taken as a whole); Θ_{new} and Θ_{merge} are then initialized to 10% of the distance between this global centroid and the farthest region. Θ_{step_factor} determines the rate at which the probabilities of creating or merging clusters changes. We found that Θ_{step_factor} depends on the number of data points to cluster, but that the method was robust enough to tolerate the same value across all benchmarks. We empirically found $\Theta_{step\_factor} = 1 - 10^{-5}$ to be appropriate for the SPEC size range.

3.2 DCA versus *k-means* and *UXM*

In this section, we compare the algorithmic assets of DCA over *k-means* and its iterative variant *UXM*.

Exposing the user to internal parameters. In *UXM*, the maximum number of clusters is defined by the *max_k* parameter, but this parameter is arbitrarily set by the user. While many codes perform well (good accuracy) with the same *max_k* (or *k*) values, some codes require high *max_k*, or conversely work well with a low *max_k* (*k*) (and thus require few sampling intervals). Table 2 shows the number

SPEC	max_k 10	50	100
ammp	10	45	100
applu	10	10	10
apsi	1	1	1
art	10	10	10
bzip2	10	32	100
crafty	1	17	100
eon	10	31	100
equake	10	17	17
facerec	10	28	28
fma3d	1	20	100
galgel	10	10	10
gap	10	32	100
gcc	4	4	4

SPEC	max_k 10	50	100
gzip	1	1	1
lucas	1	13	13
mcf	1	1	1
mesa	10	19	19
mgrid	10	16	16
parser	10	12	100
perlbmk	1	41	100
sixtrack	10	37	100
swim	10	24	24
twolf	10	50	100
vortex	10	49	100
vpr	10	29	100
wupwise	10	16	16
Average	7	22	53

Table 2: *Number of clusters obtained with different max_k values using UXM for BeeRS intervals.*

of clusters for different values of *max_k*. Obviously some benchmarks require a significantly higher number of clusters than others. SimPoint 2.0 uses a back-off heuristic to systematize the setting of *max_k*: if the number of clusters found is equal to *max_k*, *max_k* is doubled and the clustering run again.

The DCA algorithm also has internal parameters (Θ_{new}, Θ_{merge} and Θ_{step_factor}), but the initial values of these parameters have less impact on the algorithm behavior because their values are dynamically adjusted during the algorithm execution. In practice, for none of be experiments we had to tailor these parameters to the benchmarks.

Clustering speed. Even though the complexity of *UXM* and *IDDCA* are similar, the implementation of *IDDCA* is 130 times faster than a widely used implementation of *k-means*, i.e., the SimPoint 2.0 implementation [8].[2] For instance, clustering the BeeRS intervals for one benchmark with all of the same parameters in `runsimpoint` script except *max_k* = 100 requires 21 hours in average on a modern PC, and up to two days (`crafty`). In average, *IDDCA* requires 9 minutes for the same interval list, and 44 minutes on `crafty`.

The SimPoint group has apparently developed a new version of its clustering method [4], still based on *k-means*, and adapted to variable-size intervals, but it has not been released yet. This new version also proposes to speed up the clustering process on very large inputs (very large numbers of intervals) by sub-sampling the set of intervals to cluster, and run *k-means* on only this sample. While this technique can greatly improve the clustering speed, it can also degrade the clustering quality. Note that this technique can also be applied to DCA, so it shifts rather than bridges the performance gap.

[2]Recall that *UXM* is based on SimPoint 2.0.

SimPoint has also sped up the iterative clustering process by limiting the number of iterations of the *k-means* algorithm, even if it has not converged (100 iterations by default). We have found no case on the 26 Spec benchmarks where this fixed *max_k*=100 threshold would not be sufficient, but it is also difficult to ensure that a fixed parameter will be compatible with any benchmark.

While *k-means* must compute multiple times the distance between the cluster centroids and the regions, DCA only needs to compute this distance once for each region. We can take advantage of this higher speed to improve DCA accuracy, when applied to sampling, by using a large dimension after the *random linear projection* (100).

Clustering stability. The clustering quality achieved by the *random* assignment of the initial centroids. Thus, clustering quality can vary from one run of the clustering method to another; therefore, it may be hard for researchers to replicate the experiments of other researchers. Note that, to address this *k-means* issue, SimPoint 2.0 uses a modified initialization method, called *furthest-first*, that allows to partly reduce this source of variability. This initialization technique randomly selects a centroid from the region space, and then recursively selects new centroids from the region space such that their distance to already chosen centroids is maximized. Additionally, SimPoint proposes to execute *k-means* multiple times (5 by default) to maximize the likelihood that the best clustering is obtained.

Section 5 presents the simulation accuracy results for DCA.

Instruction Cache	16K 4-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	16K 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Predictors	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 8 operations per cycle, 64 entry re-order buffer
Memory Disambiguation	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Functional Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 3: *Baseline simulation model.*

4 Methodology

We used the SimpleScalar [2] 3.0b toolset for the Alpha ISA and experimented with all 26 SPEC CPU2000 benchmarks. To create the regions we used the *sim-fast* emulator. Table 3 shows the microarchitecture configuration used for our experiments.

Warm-up. Even though BeeRS [9] proposes a budget-aware warm-up technique, the issue of warm-up is out of the scope of this study. Therefore, to minimize the impact of warm-up, we add a significant warm-up interval of 20 millions instructions before each region, corresponding to a total warm-up size of several billions instructions per benchmark (almost perfect warm-up). However, we have demonstrated that the BeeRS warm-up technique can achieve a similar or better accuracy with a warm-up size of about 100 millions instructions per benchmark.

5 Adapting DCA to sampling

5.1 DCA

Table 4 and Figure 2 respectively show the number of clusters and the sampling size for DCA. While DCA created a reasonable quantity of clusters for most of the benchmarks, it generated more than 100 of them for `bzip2`, `galgel`, `gcc` and `parser`, resulting in very large sampling sizes. For instance, DCA detected ≈ 500 clusters for `parser`, which is the origin of the large number of instructions simulated for this benchmark.

More subtle clustering issues also appear in Table 4. For instance, for `galgel` and `lucas`, DCA produced several clusters that mainly contained very large regions (of more than 400 million instructions). Simulation of these clusters greatly increased the number of instructions required for these benchmarks.

Figure 3 shows the accuracy results for DCA. Even though DCA exhibited no CPI error as strong as *UXM*, it still fails significantly for 3 codes: `gap`, `perlbmk`, and in a lesser way, `equake`. In the next section, we further analyze the behavior of DCA on these benchmarks, and we present enhanced DCA algorithms.

5.2 Interleaved DCA

As explained in Section 3.1, when processing the first regions of a program, DCA easily creates and merges clusters, but as more regions are processed, the probability to create new clusters decreases, keeping the number of clusters stable. This method works well when the overall variability of the regions is experienced from the beginning of the program. But whenever distinct regions appear only late in the program execution, creating a new cluster for these regions becomes more difficult.

SPEC	DCA	IDCA	IDDCA
ammp	45	49	49
applu	33	37	37
apsi	50	44	44
art	46	42	42
bzip2	145	318	318
crafty	20	10	527
eon	20	92	92
equake	14	17	17
facerec	24	22	22
fma3d	70	73	73
galgel	138	140	140
gap	16	92	92
gcc	318	323	323
gzip	59	167	167
lucas	60	56	56
mcf	36	54	54
mesa	8	16	16
mgrid	36	32	32
parser	495	507	507
perlbmk	33	27	129
sixtrack	46	46	46
swim	53	54	54
twolf	22	21	28
vortex	29	31	31
vpr	91	155	155
wupwise	18	16	16
Average	74	94	118

Table 4: *Number of clusters obtained using DCA, IDCA and IDDCA.*

This problem is mainly due to the fact that DCA clusters the regions in the order of their appearance along the program execution. To overcome it, a simple method consists in picking the regions to cluster at regular intervals along the execution trace. We have used this option and set the interval to one tenth of the total number of regions. Let N be this number. Hence, we first cluster the first region, then region number $N+1$, followed by region $2N+1$,..., then region 2, then region $N+2$, region $2N+2$... We call this variation of DCA *Interleaved DCA* (IDCA). Figure 4 shows the clustering generated by IDCA for `equake`, and compares it to the clustering generated by DCA. Clearly, IDCA created a new cluster at the end of the program that DCA did not detect. The presence of this cluster at the end of the program is further confirmed by a SimPoint clustering using fixed-size 10-million instructions intervals.

Figure 3 shows the accuracy results with IDCA, and compares them to DCA. We can see that the CPI errors for `equake` and `gap` are significantly reduced by IDCA (respectively from 9.45% to 0.33% and from 18.10% to 5.07%), thus validating our approach. But as a side effect, IDCA increased the number of instructions to simulate due to the larger number of clusters created, see Figure 2 and Table 4. We are currently working on methods for reducing the number of simulated instructions in DCA and IDCA, by biasing the cluster representative to the smallest regions.

5.3 Interleaved Double-DCA

Still, the accuracy of `perlbmk` remains poor, see Figure 3. An analysis of the clusters shows that there are

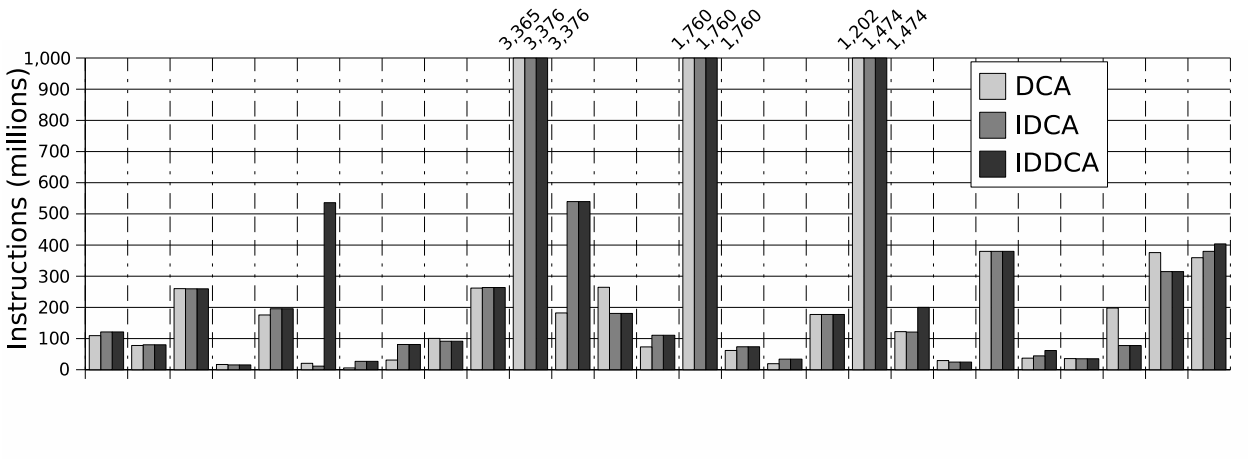


Figure 2: *Simulated instructions with DCA, IDCA and IDDCA.*

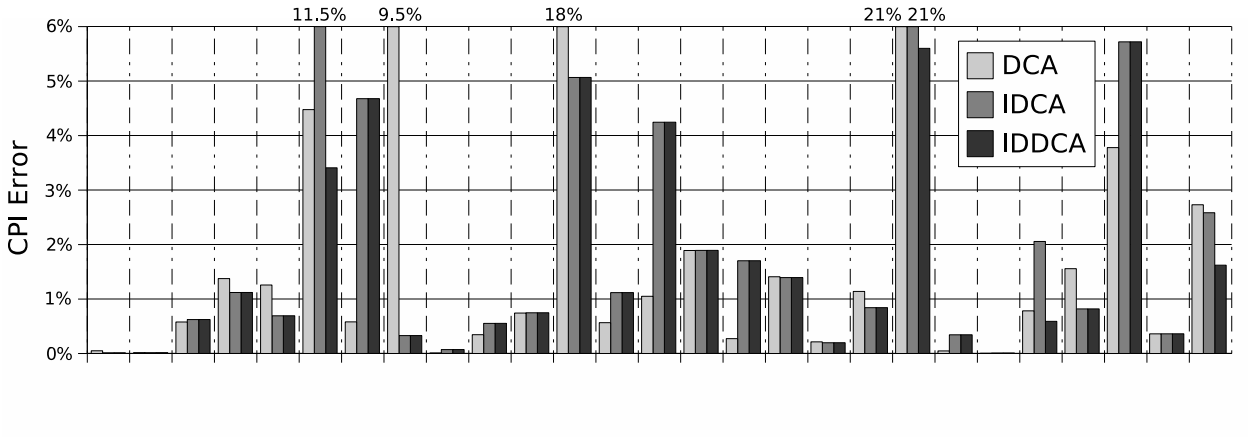


Figure 3: *Simulation accuracy with DCA, IDCA and IDDCA.*

"giant" clusters, i.e., clusters containing more than 90% of the program regions. `crafty` and `twolf` clustering also contain such "giant" clusters, albeit their detrimental influence on simulation accuracy was less pronounced than for `perlbmk`.

Intuitively, a first solution would consist in decreasing the initial value of Θ_{new}, but this solution would remove much of the practicality of DCA by forcing the user to carefully set a parameter. An alternative solution, called *Interleaved Double-DCA* (*IDDCA*), consists in creating an initial clustering using IDCA, then checking if one of the clusters contains more than 90% of the regions. If such a "giant" cluster is present, *IDDCA* performs a second IDCA clustering, restricted to these "giant" clusters.

Table 4 and Figure 2 show the number of clusters and simulated instructions generated by *IDDCA*. Note that the *IDDCA* double clustering was necessary only for `crafty`, `perlbmk` and `twolf`. As expected, the number of clusters and instructions increased for these three benchmarks, but while the number of clusters for `crafty` and `perlbmk` rose from 10 to 527 and from 27 to 129 respectively, for `twolf` the number of clusters varied much more moderately, increasing from 21 to 28.

Figure 3 shows *IDDCA* clustering accuracy results, and compares them to DCA and IDCA. *IDDCA* reduced the CPI error of the three previously mentioned benchmarks (from 11.5% to 3.4% for `crafty`, from 21% to 5.6% for `perlbmk` and from 2% to 0.6% for `twolf`) and obtained the lowest average CPI error with 1.62%. Hence, *IDDCA* allowed to efficiently cluster every of the 26 SPEC benchmark studied.

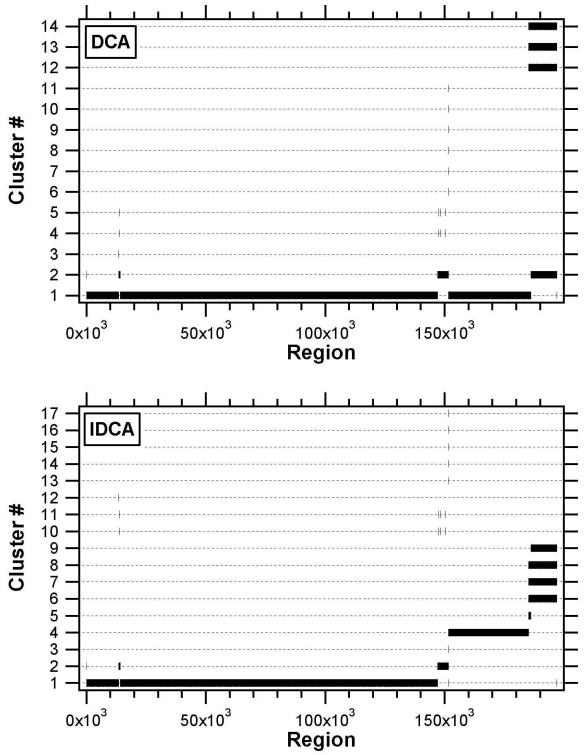


Figure 4: *equake clustering.*

5.4 Unweighted vs. Weighted *IDDCA*

In all the above DCA variants, the intervals are not weighted by their size during the clustering. As for *UXM*, not weighting the clusters is a choice targeted at reducing sampling size rather than privileging accuracy. The *IDDCA* results show that an effort is still needed on size rather than accuracy, so the choice falls on the right side of the tradeoff.

Still, in order to investigate the effect of not weighting the clusters, we have run *IDDCA* clustering with weighted clusters. As expected, the total sampling size increases, and more surprisingly, significantly (34%), see Figure 5. Even more surprising, the accuracy is lower with weighted clusters than with unweighted clusters: 2.00% CPI error for weighted clusters versus 1.62% for unweighted clusters, see Figure 6. The stiff sampling size increase validates the approach of not weighting the interval size during clustering. The performance of unweighted clustering with respect to weighted clustering, both in terms of size and accuracy, suggests that there are many intervals of very different sizes with similar behavior. Thus, not weighting the intervals will still allow to pick an interval representative of the overall behavior, and it will provide an opportunity to pick the cluster representative in an area with smaller intervals.

6 Conclusions and Future Work

In this article, we present *IDDCA*, a clustering method adapted to sampling and based on DCA. *IDDCA* improves DCA by reducing the effect of heterogeneous regions distributions and "giant" clusters. *IDDCA* achieves a CPI error of only 1.62% with 400 million instructions per benchmark in average. This rather large sampling size is especially due to 3 codes with samples that excess 400 million instructions (up to 1 billion instructions sample in the case of `lucas`). We are now working on controlling such sampling size excesses within *IDDCA*. We are also investigating methods for biasing the selection of the representative regions in order to reduce the total number of simulation samples, while preserving accuracy. Furthermore, we are working on the integration of *IDDCA* into BeeRS, in place of *UXM*.

References

[1] A. Baune, F. T. Sommer, M. Erb, D. Wildgruber, B. Kardatzki, G. Palm, and W. Grodd. Dynamical Cluster Analysis of Cortical fMRI Activation. In *NeuroImage 6(5)*, pages 477 – 489, May 1999.

[2] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, 1996.

[3] Sanjoy Dasgupta. Experiments with random projection. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 143–151, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[4] Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. Motivation for Variable Length Intervals and Hierarchical Phase Behavior. *ISPASS '05: IEEE International Symposium on Performance Analysis of Systems and Software*, 2005.

[5] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Structures for Phase Classification. *ISPASS '04: IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.

[6] Wei Liu and Michael C. Huang. EXPERT: expedited simulation exploiting program behavior repetition. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 126–135. ACM Press, 2004.

[7] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, 2003.

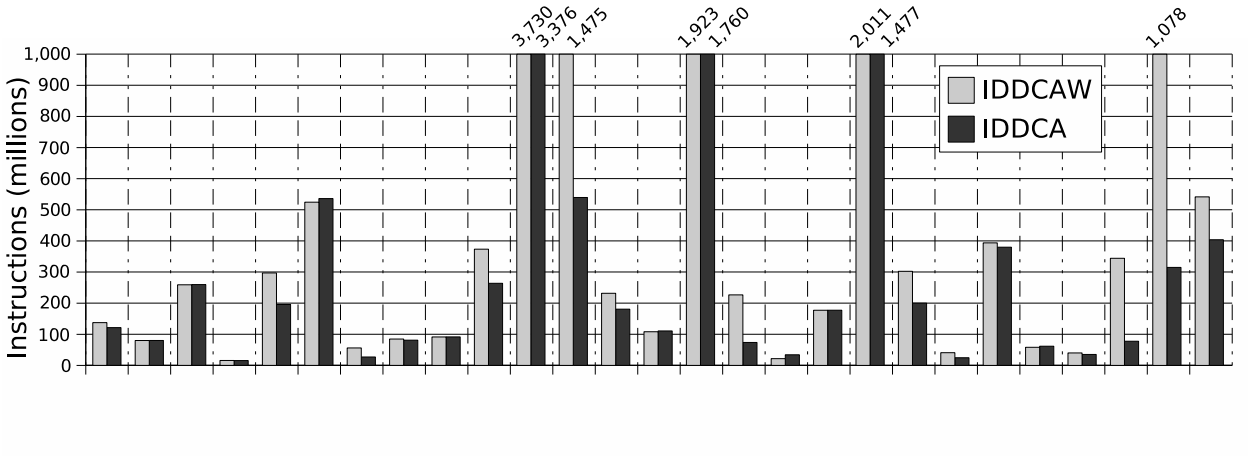


Figure 5: *Simulated instructions with IDDCA and weighted IDDCA (IDDCAW).*

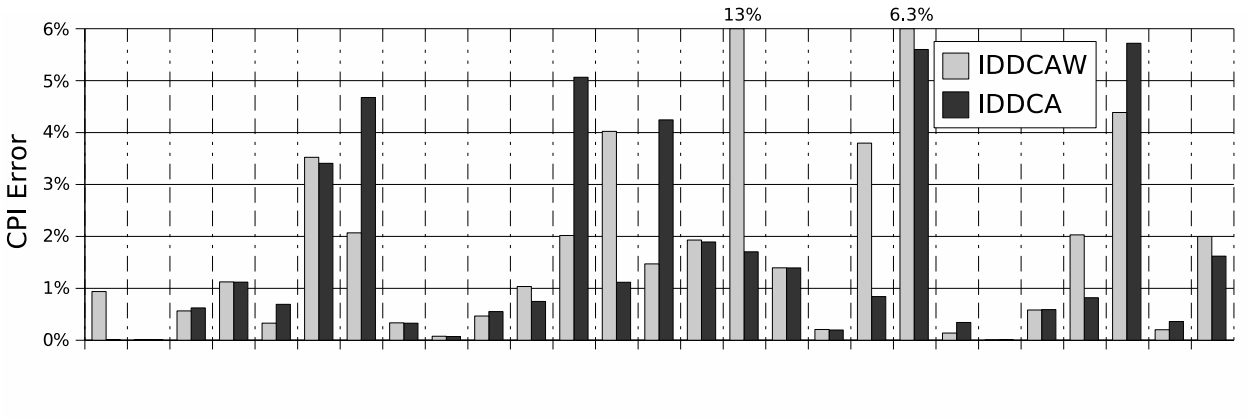


Figure 6: *Simulation accuracy with IDDCA and weighted IDDCA (IDDCAW).*

[8] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 244. IEEE Computer Society, 2003.

[9] Daniel Gracia Perez, Hugues Berry, and Olivier Temam. Budgeted Region Sampling (BeeRS): Wisely Allocating Simulated Instruction for a Better Accuracy/Speed/Applicability Tradeoff. *Submitted for publication*, 2005.

[10] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms. In *MICRO-37: Proceedings of the 37th International Symposium on Microarchitecture*, pages 43–54. IEEE Computer Society, 2004.

[11] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, 2002.

[12] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. Microarchitectural Exploration with Liberty. In *the 34th Annual International Symposium on Microarchitecture*, Austin, Texas, USA., December 2001.

[13] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. *SIGMETRICS '05*, June 2005.

[14] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97. ACM Press, 2003.

Sampling and Stability in TCP/IP Workloads

Lisa R. Hsu, Ali G. Saidi, Nathan L. Binkert, and Steven K. Reinhardt
Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
{hsul, saidi, binkertn, stever}@eecs.umich.edu

Abstract

To overcome the performance cost of simulating detailed timing models, computer architects often augment these models with fast functional simulation for fast-forwarding, checkpointing, warm-up, or sampling. Our experience with simulating network-intensive workloads has shown us that researchers must exercise caution when using such techniques in this environment. Because the TCP/IP protocol is self-tuning, the difference between the effective performance of the functional and detailed models can lead to artificial results.

In this study, we examine the effects of applying conventional simulation speedup techniques to two networking benchmarks on the M5 simulator. We find that short samples taken immediately after switching to detailed models may capture only the period of time where TCP is tuning itself to the new model. In some situations, the transition is so disruptive that no meaningful results can be obtained. However, stable simulation results can be attained with detailed simulations of moderate length in most cases.

1. Introduction

Detailed timing-accurate computer system simulators typically run thousands of times more slowly than the hardware they model. As a result, simulation of complete runs of realistic workloads is impractical. In practice, architects simulate one or more small samples of each workload with the assumption that the data collected during these samples will be representative of the workload as a whole.

A key requirement for this sampling approach is the ability to create the initial system state for a sample interval relatively cheaply. These initial states are typically generated using functional (non-timing) simulation.[1] Even the most basic functional simulator is typically orders of magnitude faster than detailed timing simulation. Using advanced techniques such as direct execution and dynamic code generation, functional simulators can execute code only a few times slower than hardware [11, 4].

A fundamental assumption underlying this technique is that the system's state does not have any significant dependence on the system's timing. Otherwise, the lack of accurate timing in the functional simulation would result in a system state that is unrealistic for the configuration being studied. While this assumption often holds true, it is not axiomatic. In particular, this assumption does not hold for self-tuning workloads, which adapt their behavior to observed system performance.

In this paper, we examine an important class of self-tuning workloads—those based on the TCP/IP network protocol—to understand the impact of this timing dependence on common program sampling techniques. TCP adapts dynamically to the available end-to-end bandwidth by varying the number of outstanding packets (known as the "window size") at the sender. When the system under test is a bottleneck, as is not uncommon, its performance will determine end-to-end bandwidth. During functional simulation, TCP will set its window size according to the largely meaningless effective system performance of the functional model, affecting the architectural state of the system (including TCP's self-tuned connection parameters and the occupancy of network buffers). Because of these effects, this state is not a realistic state for the system being modeled. After switching to the detailed timing model, TCP must adjust its window size accordingly before an accurate network bandwidth reflecting actual system performance is reached. Note that this effect is orthogonal to the issue of behavioral variation along the time axis of a program. This tuning period exists regardless of the point at which the switch from functional to detailed simulation occurs.

1. Initial states can also be extracted from actual hardware systems, but this process is complex, expensive, and rarely used outside of industry.

We use functional simulation to generate architectural checkpoints of TCP/IP-based workloads as well as to warm up cache state in preparation for detailed simulation and measurement. In both of these cases, the effective performance of the simulated system differs significantly from the performance seen using detailed timing simulation. We observe that the behavior of networking workloads depends on whether the connection bandwidth is limited by sender performance or by the network or receiver. We will refer to these scenarios as sender limited and receiver limited, respectively.

- In sender-limited situations, instantaneous performance changes are unlikely to cause packet loss. Instantaneous changes to the sender mean immediate changes to send rate, since there is no feedback required. Instantaneous changes to the receiver may yield different results. If the receiver increases in performance, there would be no change since sender determines overall performance anyway. If the receiver decreases in performance, tuning would only be required if the receiver's performance decreased to the point that it became the bottleneck.
- In receiver-limited situations, instantaneous increases in sender performance or decreases in receiver performance can lead to packet loss that may induce unstable behavior. Though TCP will eventually recover from packet losses, the recovery may not occur within a window of time that is feasible to simulate.

The primary contribution of this paper is that it is the first, to our knowledge, to identify and describe the issues that arise due to the interaction of self-tuning workloads and functional simulation. We also provide a detailed analysis of the impact of functional simulation on TCP-based networking workloads. While TCP itself is of great practical importance, these issues may apply to an increasing number of future workloads, as run-time profile-based software optimization becomes more widespread. Furthermore, many of the recent advances in simulation sampling have relied implicitly on workload timing independence; these techniques must be revisited and perhaps revised or even discarded when analyzing self-tuning workloads.

The next section presents a more detailed discussion of workload timing dependence and its impact on simulation techniques, including coverage of related work. The following sections describe the TCP protocol and our simulation environment, respectively. We then present the results of our experiments, and finally discuss our conclusions and directions for future work.

2. Discussion and Related Work

This section elaborates on the notion of workload timing dependence discussed in the introduction and relates this paper to prior work in the area.

Single-threaded applications were the first and are still the most prevalent type of workload used in architecture studies. Because individual threads in isolation have deterministic architectural behavior (including architectural register and memory contents and committed instruction order), and because application-only simulation does not model potentially non-deterministic interactions with the operating system (such as preemptive scheduling), the architectural behavior of these workloads is completely unaffected by execution timing. Only these workloads are truly timing independent. In this case, the architectural state generated by a fast functional simulator for a given program point is guaranteed to be identical to the architectural state of any (correct) detailed timing simulator at the same program point.

In contrast, multithreaded applications running on multiprocessors or dynamically scheduled onto uniprocessors (e.g., using SMT) are not fully timing independent. System timing effects, such as cache hits and misses, will lead to different rates of progress for different threads, causing variations in the interleaving of thread execution. These variations can have subtle architectural effects even in deterministically scheduled programs, such as the number of iterations a thread sits in a spin loop. More commonly, inter-thread synchronizations are not fully deterministic, and timing variations that, for example, change the order in which threads acquire locks may have significant effects. Goldschmidt and Hennessy [7] showed that in some situations this timing dependence could lead to incorrect results from trace-driven simulations, and recommended the (now widespread) use of execution-driven simulation. Alameldeen and Wood [1] showed that this dependence can be particularly severe for complex multiprocessor server workloads on full-system simulation (i.e., including the operating system), where the combination of different processor interleavings can lead to different OS scheduling decisions and different service orders among client transactions. They recommend combining multiple runs randomized using small perturbations to gain statistical confidence in the simulation results.

Although the timing dependence of these multiprocessor workloads is a significant issue, it is qualitatively distinct from the timing dependence we explore in this paper. In the former case, timing variations can lead the system down different but equally valid archi-

tectural execution paths. Specifically, each of the potential paths followed under one timing scenario is also a legitimate potential path under other timing scenarios. Alameldeen and Wood's randomization technique samples the space of valid paths so that statistically significant conclusions can be drawn about architecturally invisible changes in a system's configuration (e.g., in cache associativity or coherence protocol) without interference from the perturbations in architectural execution they cause. We call these workloads *weakly timing dependent*.

In contrast, we are concerned with workloads that explicitly tune themselves to the performance of the underlying system. These workloads take performance feedback from the underlying system (directly or indirectly) and incorporate this feedback into the control flow of the workload. For example, if TCP packets arrive at a host faster than they can be processed, kernel buffers will eventually fill and packets will be dropped. These packet drops will be detected by TCP, causing not only retransmissions but also a geometric decrease in the sender's window size, fundamentally restricting the connection's bandwidth. Thus timing variations lead to changes in the architectural execution in such a way that the path taken under one set of timing conditions may be substantially different from any execution path that the system would take under another set of timing conditions. As such, the architectural states generated along the former path may be states that could never be reached under other timing conditions. We call such workloads *strongly timing dependent*. Lim and Agarwal's reactive synchronization algorithms [9] are another example of a strong timing dependence.

The distinction between strongly and weakly timing-dependent workloads is not strict. A multiprocessor server running a TCP-intensive workload would exhibit both strong and weak timing dependencies. This paper focuses solely on the strong timing dependence aspects of self-tuning workloads. The interaction of these types of dependences is a topic for future research.

The key point of this paper is that strongly timing-dependent workloads may not be compatible with both traditional and recently proposed techniques for accelerating architectural simulations. Even a purely functional simulation of a timing-dependent workload must have some notion of time. For example, the rate of TCP packet arrivals must be specified outside of the architectural behavior of the system, leading to some relative timing relationship between network bandwidth and CPU execution rate. During functional simulation, a strongly timing dependent workload will adapt its behavior to the effective observed performance of the system. If a fast functional simulator is used to generate initial checkpoints for detailed modeling or to fast-forward between detailed sampling points for such a workload, the architectural state so generated may not be valid for the detailed system timing that the simulator is modeling.

As we will show, functional simulation is still useful as long as the detailed timing model can be run long enough for the workload to re-tune itself to the actual performance of the simulated system. However, this requirement directly conflicts with the approach of the SMARTS system of Wunderlich et al. [12], which advocates performing detailed modeling on numerous but extremely brief samples separated by fast functional execution. Another possible approach to strongly timing dependent workloads which would be compatible with SMARTS is to attempt to match the effective performance of the functional simulation with the detailed model. However, it is not clear how accurately this goal could be achieved, and we do not explore it further in this paper.

Sherwood et al.'s SimPoint technique [10] is another recent advance in simulation methodology. Their approach identifies a priori a set of representative portions of program execution. These portions are simulated using a detailed model and combined with appropriate weights to generate an estimate of the full program's performance. The benefit of the technique comes from the fact that the representative portions are identified once for all timing models using a fast functional execution of the entire program. This approach is clearly useful for timing-independent workloads. However, the applicability of SimPoint to strongly timing-dependent workloads depends on whether the portions of execution that are representative under one timing scenario (e.g., the implicit timing of the functional simulator) are also representative of other executions under other timing configurations. While this condition may hold true, further investigation is required. SimPoint has been applied to simulation of weakly timing-dependent workloads consisting of multiple single-threaded applications on an SMT processor [5]. This work exploits the timing-independent nature of the individual threads to simplify the analysis of potential dynamic interactions between them.

3. TCP/IP Overview

In this section we discuss the TCP/IP protocol and the way in which it tunes itself to the available end-to-end bandwidth capability of a connection.

3.1 Basic TCP/IP Operation

TCP/IP is a ubiquitous network protocol used to transfer data across the Internet. The sender transmits data in packets to the receiver. As the receiver processes data, it sends back acknowledgement packets (ACKs) to the sender to indicate the data it has received. Each packet contains a TCP header that identifies the connection and includes other information about the connection's status.

3.2 TCP Flow Control

TCP flow control is the attempt at the sender to match its sending rate to the processing rate of the receiver. In the header of each TCP packet, the receiver indicates the amount of kernel buffer space it has available for the connection. The sender's network stack maintains this value internally in a parameter called the send window, and ensures that the number of outstanding (unACKed) packets for that connection does not exceed this send window.

3.3 TCP Congestion Control

The terms congestion control and flow control are often used interchangeably, but they are two distinct algorithms in TCP. Congestion control refers to the sender's attempt to match its rate to the capability of the network through which packets are transmitted. If packets from the sender are being lost en route to the receiver (dropped by routers, etc.), the sender will reduce its sending rate regardless of the amount of buffer space available at the receiver. Because IP packets may be silently dropped at any point in the route, packet losses are inferred from receiver ACKs and timeout events. Loss events cause adjustments to a sender-internal parameter called the congestion window. The sender may not have more unACKed data than specified by the congestion window.

The congestion control algorithm has three main phases: (i) slow start, (ii) additive increase/multiplicative decrease (AIMD), and (iii) timeout event recovery. Connections always begin with the slow start phase, where the congestion window is initialized to one packet and grows exponentially with every ACK received. When the congestion window reaches a certain threshold, the protocol moves into the AIMD phase, where every successful ACK received results in an additive increase in the congestion window. If the sender observes from the receiver's ACK information that a packet appears to have been dropped, the congestion window is cut in half (i.e., a multiplicative decrease), but the connection stays in the AIMD phase. If the sender does not receive any ACKs from the receiver for a sufficient period, a timeout occurs, and the connection attempts to recover by returning to the slow start phase.

All common TCP implementations adhere to this general framework. There are many additional details that vary across specific protocol implementations (e.g., Tahoe, Reno, and Vegas), but these details are not critical to this paper. The important thing to note is that packet losses can result in severe throttling of the sender's transmission rate.

3.4 Tuning Latency

Except for timeouts, TCP flow and congestion control at the sender is based on information contained in the ACK packets from the receiver. As a result, the time required for a TCP connection to tune itself is primarily a function of the round-trip time (RTT) between a sender and receiver. In this paper, most of our simulations involve two systems directly connected by a single low-latency link. This setup minimizes the TCP self-tuning interval. We also do a few experiments with a higher latency link as well, in order to demonstrate the effects of RTT on tuning delay.

4. Methodology

We performed our experiments with M5 [6], a full-system simulator designed to support network-oriented workloads. It implements models for CPUs, caches, memory, I/O devices, and bus-based interconnects. M5 has both a detailed out-of-order CPU model and a simple in-order functional CPU model which can operate with or without caches. For the network interface, M5 models the National Semiconductor DP83820 network interface controller (NIC). We boot a Linux 2.6.8.1 kernel on the simulated hardware.[1]

4.1 System configurations

In this paper, we model three different CPU/cache combinations:

- The *pure functional* (PF) model uses the simple CPU model with no caches and a single-cycle memory latency. This model is used primarily for generating checkpoints.

1. Our simulated DP83820 fixes a bug in the actual device that prevents unaligned DMA transfers; we also patched the device driver to eliminate its workaround, allowing DMA transfers that automatically word-align payloads. We also patched a bug in the Linux kernel that does not enable checksum offloading correctly in some situations.

Table 1: System Under Test Parameters

	Parameter	Value
Detailed CPU	CPU width	4 insts/cycle
	Branch Predictor	hybrid local/global
	BTB	4k entries, 4-way assoc
	Instruction Queue	64 entries
	ROB	128 entries
Memory System	DL1/IL1	128kB, 2-way assoc, 64 byte blocks IL1: latency 1 cycle, 8 MSHRs DL1: latency 3 cycles, 32 MSHRs
	L1/L2 Bandwidth	64 bytes/CPU cycle
	L2/Memory Bandwidth	16 bytes/25 CPU cycles
	HT Controller Latency	750 cycles
	Memory/HT Bandwidth	16 bytes/25 CPU cycles

- The *functional with caches* (FC) model uses the same simple CPU model but adds a two-level cache hierarchy. Because the CPU issues instructions in order, the caches are effectively blocking. This model is intended for cache warm-up.
- The *detailed* (D) model uses our detailed out-of-order CPU with non-blocking caches. This model is intended for gathering detailed performance statistics.

See Table 1 for detailed parameter values. To avoid I/O bus bottlenecks, we model a NIC attached to a HyperTransport-like bus which is attached to the memory controller.

The simple functional CPU has a multiplier option that allows the user to specify a maximum number of instructions to execute per cycle. We vary this parameter, setting it at either 1 or 8 to get lower or higher performance. We label the PF models as PF1 and PF8 to indicate the multiplier setting. In this paper, the FC model always uses a multiplier of 1.

In terms of effective simulated system performance, the PF8 and PF1 models are the fastest due to their idealized single-cycle memory latency. The FC model is slowest, because it incurs cache miss penalties and cannot overlap them with its simple 1-CPI blocking CPU. The D model has intermediate performance; although it pays cache miss penalties, its non-blocking caches and out-of-order superscalar execution mitigate their impact relative to FC.

Our network simulations all involve multiple systems (at least a sender and a receiver, and in some cases a gateway system). We model all systems within a single simulator process to guarantee accurate, realistic, and repeatable timing for the network protocol. However, in any experiment only one system is of interest, the system under test. To both reduce simulation time and drive the system under test at maximum load, we always use the PF8 model for all systems other than the system under test.

4.2 Benchmarks

Netperf [8] is a collection of network micro-benchmarks developed by Hewlett-Packard. Included are several benchmarks for evaluating the bandwidth and latency characteristics of various network protocols. We selected TCP stream, a transmit benchmark; and TCP maerts, a receive benchmark. In both of these benchmarks, the client (the system under test) connects to a server. The appropriate system then attempts to send data as fast as possible over the connection.

SPEC WEB99 (specweb) is a popular benchmark for evaluating the performance of webservers. It simulates multiple users accessing a mix of both static and dynamic content over HTTP 1.1 connections. It includes CGI scripts to do dynamic ad-rotation and other services a production webserver would normally handle. For our simulations, we use the Apache webserver [2], version 2.0.52. We also use the mod_specweb99 module, available on the SPEC and Apache websites, which replaces the reference Perl CGI scripts with a more optimized C implementation. We also use a custom client based the Surge [3] traffic generator. This modified client uses the same statistical distributions as the standard SPEC WEB99 clients, but is lighter weight (to reduce simulation time) and attempts to saturate the webserver rather than maintain a fixed bandwidth as the standard client does.

In addition to the back-to-back configuration of the two previous benchmarks, we also created a network address translation (NAT) configuration for the netperf maerts benchmark, where the client communicates to the server through a NAT gateway. The NAT gateway masks the IP address of the client system from the server. NAT gateways are commonly used to connect an entire private network through a single public IP address. In this configuration, the NAT gateway machine is the system under test.

4.3 Checkpointing

We use checkpointing to avoid the overhead of booting multiple simulated machines and initializing the benchmarks for each experiment. Special instructions embedded in the benchmark scripts cause M5 to save

off all architectural state. Later simulations can then begin from the specified checkpoint, using whatever operating mode is desired by the user. For all configurations, checkpoints are generated after connections are established and network traffic has ramped up.

We generated four checkpoints for each of our benchmarks. The differences varied on two axes: whether the system under test was checkpointed in PF1 or PF8 mode, and whether the wire latency was 0 or 400 µs. All systems not under test used PF8 mode. All CPUs were clocked at 4 GHz.

4.4 Experiments

This paper focuses on the effects of traditional simulation acceleration techniques on networking workloads. To that end, we copied two common techniques, both starting from a purely functional (PF) checkpoint: running a detailed (D) timing simulation directly, and using a functional-with-caches (FC) interval to warm up cache state before the detailed timing simulation (FC→D). We varied the length of the FC cache warmup to be 125 million, 250 million, 500 million, or 1 billion cycles. Prior experimentation with showed that netperf performance is extremely stable, so we ran the detailed portion for 100 million cycles. We ran specweb for 1 billion detailed cycles because of its lack of steady behavior. Statistics were sampled every 10 million cycles in both cases.

5. Results

The results for the netperf stream benchmark on a zero-latency network are shown in Figure 1. For this and most remaining figures, the graphs plot the total wire bandwidth of the system under test for each 10 million cycle interval from the point immediately after loading the checkpoint. The legend indicates whether the checkpoint was taken with a PF1 or PF8 CPU. For space reasons, we show only the 500 million cycle cache warmup results for FC→D.

In looking at the stream graphs, there is no significant bandwidth settling period after any CPU model transition. There is also no apparent difference as to whether the checkpoint was taken with a PF1 or PF8 CPU. Recall that netperf stream exercises the client's ability to send data. Because the client (the system under test) is the bottleneck, this scenario is sender limited (as discussed in Section 1). The client's send rate is not constrained by the state of the network or receiver's free buffer count, so it can adapt quickly when its performance suddenly changes. Thus for netperf stream (and likely other sender-limited workloads), there is no need for a significant TCP tuning interval to reach the new steady-state performance.

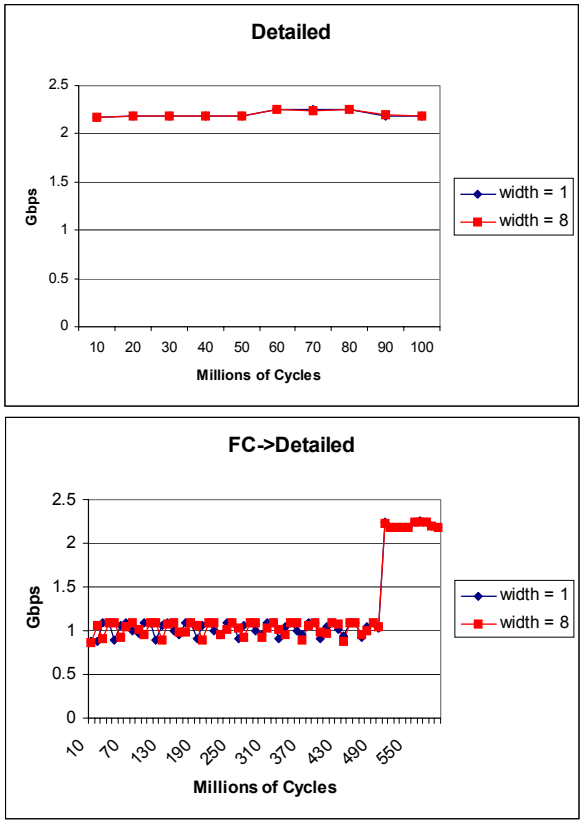


Figure 1: Netperf stream bandwidths. The legend indicates the width of the system under test CPU during the checkpoint. Since stream is a sender-limited benchmark, the transitions are smooth and stable.

The netperf maerts benchmark (Figure 2) tests the client's receive performance. When the client system changes instantaneously, it takes time for the effects of this change to propagate to the server sending the data. In the detailed case, the slowdown from the PF8 checkpointing CPU to the detailed CPU is drastic enough to require some tuning. Longer runs show that the true steady state for this benchmark under our configuration is about 5.7 Gbps. The PF8→D transition requires 70 million cycles to stabilize to this level. The bandwidth gets noticeably greater while it is tuning due to retransmission traffic. The PF1→D transition also requires a bit of time, but is harder to see in the graph. With some statistical analysis, we find the coefficient of variation (CoV) of Gbps for the 10 samples of the detailed run is 1.66%. However, considering the first sample as the tuning period, and only measuring the latter 9 samples drops the CoV to 0.5%. Further lengthening the tuning period does not reduce the CoV any more. Thus we can

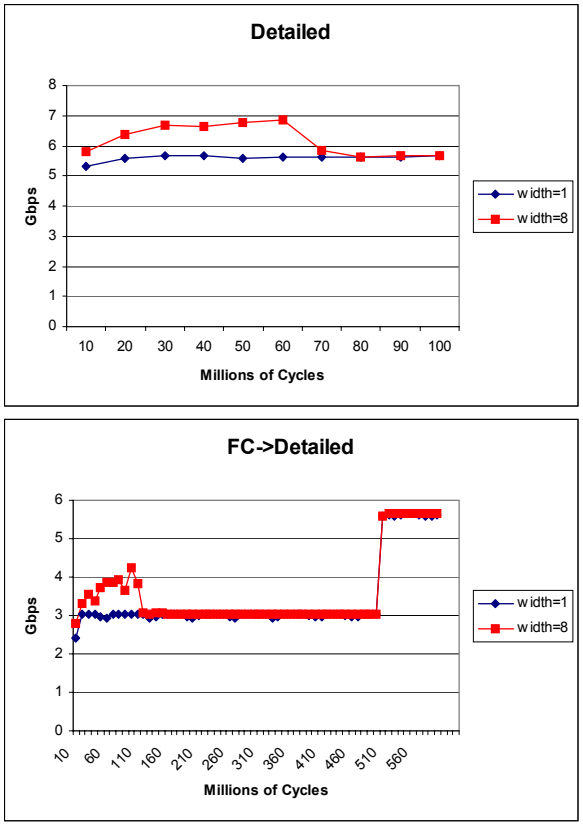


Figure 2: Netperf maerts bandwidths. The legend indicates the width of the system under test CPU during the checkpoint run. Since maerts is a receive-limited benchmark, CPU transitions require some tuning before becoming stable.

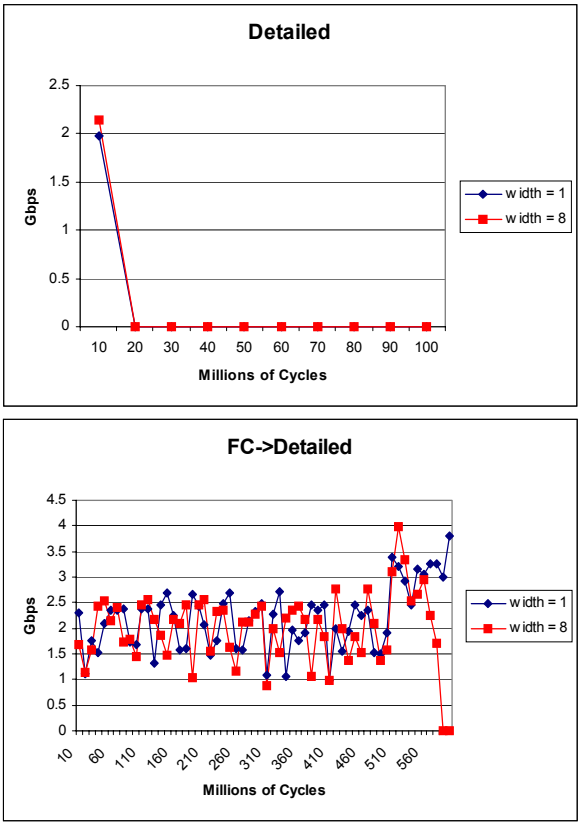


Figure 3: Bandwidth for netperf NAT configuration. In the top graph, the drastic transition from PF to D is exacerbated by the long network latency caused the by NAT gateway, causing the simulation to become unstable and effectively come to an end.

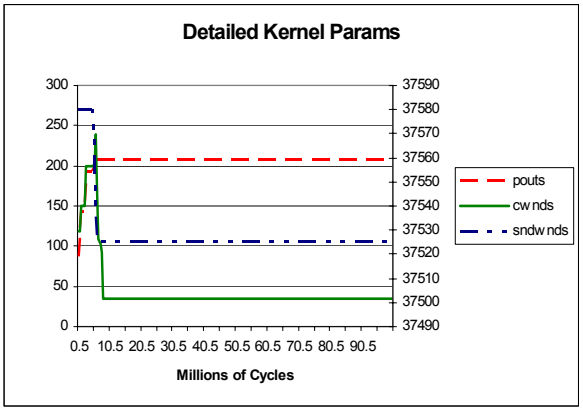
conclude that this tuning period requires no more than 10 million cycles.

In the FC→D case, there is a much lengthier stabilization period going from the PF8 checkpoint into the FC mode than from the PF1 checkpoint. The PF8 CPU is faster than PF1, so the performance mismatch is greater and requires greater adjustment. However, both cases stabilize well before the change from FC to D. The FC to detailed transition is quite smooth, likely since it involves a speedup of the CPU rather than a slowdown. The transitions in the graph are nearly instantaneous; the first sample is not measurably different from all the remaining samples.

We conclude that, for receiver-limited benchmarks such as netperf maerts, some significant transition time is necessary for TCP to retune itself whether it is transitioning directly from a pure functional checkpoint (or fast forward) to detailed simulation or to a functional cache warm-up period. The length of this transition time is a function of the severity of the CPU mismatch between the two phases: from 10 to 70 million cycles for PF→D and 10 to 125 million cycles for PF→FC, with the larger values corresponding to the PF8 checkpoint. TCP tuning time after the FC→D transition is negligible because the effective receiver performance is increasing rather than decreasing. With M5, using a minimal FC phase for tuning yields a shorter total simulation time than transitioning directly to detailed from the FC checkpoint.

For the NAT configuration of netperf maerts, the system under test is the NAT gateway. If coming out of the checkpoint moves the system to a slower CPU model, the gateway slows and injects more delay into an already high-latency network between the server and client. Sometimes the gateway slows to the point that it loses packets, which takes an extremely long time for TCP to detect and recover from.

Figure 3 shows the bandwidths of all the NAT maerts runs on a zero-latency network. When transitioning from PF directly to detailed, the simulation effectively ends. The sender, not knowing about the degradation of the network, continues to send packets at a high rate



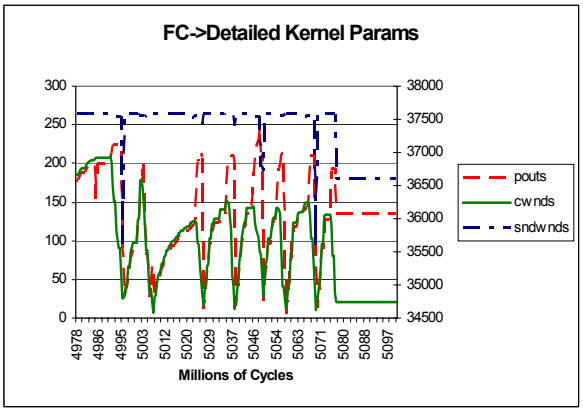
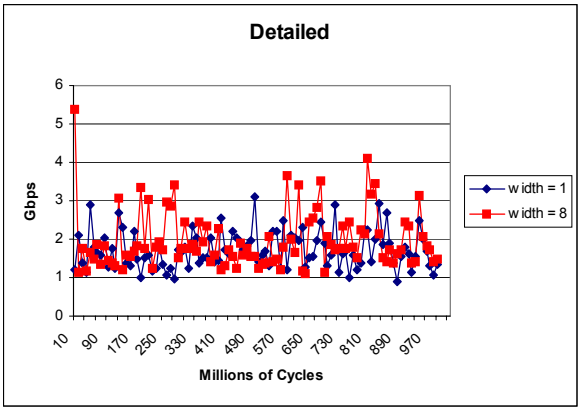


Figure 4: TCP kernel parameters for NAT maerts runs that manage send rate. Pouts and cwnds are in packets, and go with the left y-axis, while sndwnds are in bytes and go with the right y-axis.



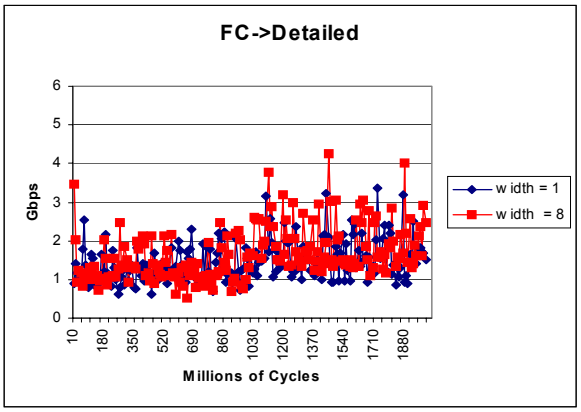


Figure 5: Specweb bandwidths.

that the newly slowed gateway cannot handle. The gateway thus drops packets. In looking at the packet traces, we found they were riddled with retransmissions and lost packets. Towards the end, there was a long stream of duplicate ACKs sent by the receiver. Duplicate ACKs indicate that the receiver has missed a packet in the sequence and requires a retransmission.

In contrast, the simulation continues transmission through the FC phase after the PF to FC transition with both checkpoints, though without stabilizing. However, the PF8 checkpoint run ceases transmitting at the FC→D transition. The cause for this behavior is not clear since the FC→D transition actually represents a speed increase.

To gain insight to what occurred on a TCP level, we tracked some TCP stack parameters for the runs starting from the PF8 checkpoints. Figure 4 shows some TCP parameters maintained by the server to manage its send rate to the client. Cwnd is the congestion window, sndwnd is the send window (which represents the amount of kernel buffer space available in the receiver that is sent via TCP header), and pouts is the number of packets in flight at the time (packets out). Notice that in the top graph (showing the PF→D case), the receiver's advertised kernel buffer space and the sender's congestion window size plummet immediately after transition. The slowdown from PF to D is so drastic that these parameters actually drop below the packets-in-flight count. However, the number of packets in flight is not supposed to exceed the congestion window, preventing the sender from transmitting additional packets. Meanwhile, the receiver cannot make progress without the retransmission it is asking for with the duplicate ACKs. In a real system, these situations are solved by TCP timeouts and resets, but this process takes far too long to simulate.

A similar parameter drop-off happens in the bottom graph at the FC→D. We are unsure why this fall-off occurred at the transition from a slower CPU to a faster one, and not at the PF→FC transition as might be expected. We plan to look into this situation further as future work.

Results from the specweb benchmark are shown in Figure 5. Recall that our specweb runs differed from netperf in that the detailed simulations ran for 1 billion cycles. Even on this time scale, the benchmark is inher-

ently less stable than netperf, though on even larger time scales it can be quite stable [1].

With specweb the system under test is the server, and it serves web pages requested by the client. It is difficult to see with the naked eye whether there are TCP tuning effects. We combined the detailed bandwidth samples into 150-million-cycle intervals and measured the CoV of these aggregated samples. For the direct PF→D runs, the CoV drops from roughly 8% to 3% when taking a 300 million cycle transition phase into account. The FC→D case transitions more quickly, needing only a 150 million cycle warmup to reduce the CoV near minimum. This implies that, like with the maerts benchmark, the FC phase takes some bite out of the TCP tuning period. However, since the total simulation is only 1 billion cycles long, it is difficult to be confident that these samples are truly representative. Specweb is sufficiently more complex than netperf that further study is needed to make definitive conclusions. Being a sender-limited case, it may not require much tuning time at all (as with netperf stream).

The above graphs were all of simulations using zero-delay wires. When link delays are thrown into the mix, the problem of tuning time is exacerbated further. Figure 6 shows results from the netperf maerts FC→D run with a 400 µs wire delay. This wire delay increases the round-trip time (RTT) by 800 µs, increasing the tuning time significantly.

The top graph of Figure 6 compares measured bandwidth between the zero-delay and 400 µs-delay runs. The simulation with link delay has not reached steady state by the end of the simulation. The bottom graph shows the TCP stack parameters from the 400 µs-delay run. The beginning part of the graph shows how the CPU transition disrupts the simulation and forces the geometric decrease of the congestion window. Packets in flight must follow. The steep increase that follows is the TCP slow-start phase, where the congestion window increases exponentially with each ACK. The next phase is the AIMD phase of TCP described in Section 3.3. Recall that in this phase of operation, the congestion window increases by one packet with each ACK. Since ACKs are delayed by the RTT, the increase in cwnd does not happen nearly as quickly as in the previous experiments.

In reality, 800 µs is a modest RTT. While the RTT from our desktop machines to local University of Michigan servers is roughly 300 µs, RTTs for non-local servers are typically tens of milliseconds, with well-connected international sites (e.g., www.cern.ch) over 100 ms. Thus simulating realistic Internet link delays requires particular care, and may not be practical without novel techniques. Maerts is capable of achieving much higher bandwidths, and taking data from these simulations would not give accurate results.

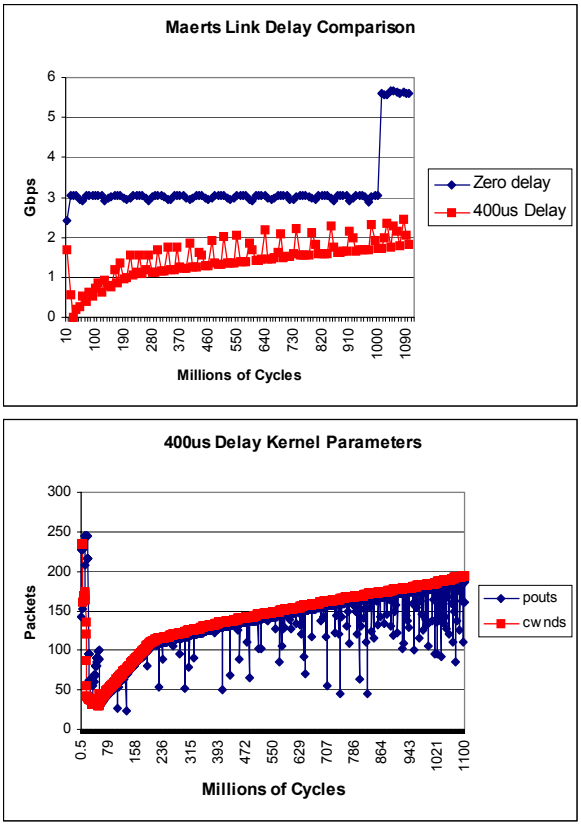


Figure 6: Top: Contrasts between zero delay wires and 400us delay wires. Bottom: TCP congestion windows increase by one with every ACK received. When ACKs are delayed by the link, the cwnd parameter is slow to increase, thus slowing TCP tuning time.

6. Conclusions and Future Work

It is clear that for many cases of networking benchmarks, conventional methods of simulation acceleration via functional simulation may induce artificial behavior. TCP/IP is a self-tuning protocol; when run on a purely functional system model, it will tune its performance parameters to the meaningless effective performance of the functional system. The effects of this artificial tuning range from incorrect data to completely unstable behavior.

We observed that there are two classes of simulations: sender-limited and receiver-limited. The netperf stream benchmark and specweb are examples of the sender-limited case, whereas netperf maerts and NAT are examples of receiver-limited cases. Spontaneously changing sender performance in sender-limited cases

generally does not require noticeable TCP tuning time, as the sender can autonomously adjust its transmission rate. However, spontaneously changing receiver or network performance in receiver-limited cases can lead to dropped packets because of the propagation delay in informing the sender of the new environment. Often, allowing TCP to tune itself before taking measurements is sufficient. However, in some cases, these dropped packets can cause a simulation to yield no meaningful results.

Where retuning is needed, the time required is significant and can vary dramatically depending on the situation (from 10 to 150 million cycles in our experiments, given zero wire delay). Thus, fast-forwarding paradigms like the SMARTS method of simulation acceleration (which advises 1000 cycles of detailed simulation following a period of fast forwarding and functional warming) would very likely result in invalid data when testing networking benchmarks. The retuning period is also a strong function of network round-trip time; when realistic network latencies were added between the sender and receiver, the retuning time quickly became significantly larger than is practical for simulation.

We also observed that TCP adjusts more quickly and with more stability when effective system performance increases rather than decreases. We thus had fewer problems when checkpoints were generated using a purely functional model with lower effective performance (PF1 vs. PF8). Intermediate warm-up using the functional-with-caches (FC) model also reduced tuning time, as this mode is not as slow to simulate as detailed, and the next transition from FC to detailed is an effective performance increase.

The problem of packet drops leading to completely disrupted simulations does not appear to have a simple solution. We have found that often just taking the checkpoint at a different point in execution can sometimes prevent this situation, but we do not have a rigorous method for distinguishing such points. Understanding and addressing these pathological cases is an area of future work.

Other promising directions for research include characterizing needed warmup and transition times in more detail and finding techniques to deal with scenarios that involve realistically large RTTs.

While many of our detailed conclusions are tied to fundamental characteristics of TCP/IP, the issues raised in the paper applies to other strongly timing dependent workloads as well. We expect that future sophisticated run-time systems will employ dynamic optimization techniques aggressively. Architects must take care in these situations as well that the software layers do not tune themselves inappropriately to the artificial effective performance of functional system models.

7. References

[1] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 7–18, February 2003.

[2] Apache Software Foundation. Apache HTTP server. http://httpd.apache.org.

[3] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[4] Robert C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proc. 1995 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 14–24, 1995.

[5] Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proc. 2004 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, March 2004.

[6] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2003.

[7] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulation of multiprocessors. In *Proc. 1993 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 146–157, May 1993.

[8] Hewlett-Packard Company. Netperf: A network performance benchmark. http://www.netperf.org.

[9] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 25–35, October 1994.

[10] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proc. Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 45–57, October 2002.

[11] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proc. 1996 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[12] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. 30th Ann. Int'l Symp. on Computer Architecture*, pages 84–97, June 2003.

Using A Multiscale Approach to Characterize Workload Dynamics

Tao Li

Department of Electrical and Computer Engineering
University of Florida, Gainesville, Florida, 32611
E-mail: taoli@ece.ufl.edu

Abstract

Workload characterization has been essential ingredients for the design of computer systems. As computer architecture becomes adaptive and reconfigurable, its efficiency increasingly depends on the dynamic behavior of workloads.

Program execution manifests wildly varied changes at run time. As software becomes large and sophisticated, such variation can span a wide range of time scales. As current workload characterization methodologies have paid less attention to the scaling behavior of program, we are often at a loss as to how to interpret and forecast the changing of workload dynamics over time. To address this issue, we advocate a multiscale workload characterization methodology that can capture the dynamic nature of workloads across different time scales.

We apply this technique to study the scaling properties of the SPEC2000 integer benchmarks. The on-line program scaling estimator proposed in this paper allows one to capture both short-term and long-term execution characteristics of large program in-flight.

1. Introduction

Workload characterization is at the foundation of computer architecture design and optimization. By understanding workload behavior, both hardware and systems can be tuned to better suit the needs of the applications. There have been many studies [1, 2, 3, 4] that evaluated the performance of workloads and attempted to characterize it with various architectural features. A majority of these studies use aggregate metrics (e.g., average IPC, total cache misses) to represent the specific architectural characteristics for the entire program execution.

As computer architecture becomes adaptive and reconfigurable, its efficiency increasingly depends on the workload dynamic behavior. For example, reconfigurable microarchitecture periodically profiles program execution characteristics and uses them as the feedbacks to adapt its resource (e.g. instruction queues, branch predictors and caches) to meet the need of applications. Previous studies [5, 6] revealed that program execution manifests wildly varied changes at run time. As software becomes large and sophisticated, such variation can span a wide range of time scales. Current workload characterization methodologies, however, have paid less attention to reveal how the workload characteristics change across different time scales. As a result, we are often at a loss as to how to interpret and forecast the changes in workload behavior over time. Such limitations can become serious with the increasing workload execution time.

A better understanding of program multiscale behavior can benefit computer architecture design and performance evaluation in many aspects. For example, reconfigurable architecture can exploit program scaling properties that capture both small time period and large time period characteristics to fine-tune its multi-configuration hardware units across various time scales. In another scenario, for a program that shows the well-defined scaling properties (e.g., *self-similarity*), its large time scale behavior can be accurately projected by using its small time scale behavior.

The goal of this paper is to improve our current understanding of the changing nature of the workload dynamics over time. Differing from the previously proposed methodologies [1, 5] (e.g., aggregate measurement, phase characterization), this paper introduces the using of multiscale models and metrics to describe program dynamics. The discrete wavelet transforms are used to discover program behavior at multiresolution levels. The proposed multiscale workload characterization methodology allows one to "zoom in" and "zoom out" over a wide range of program execution periods to capture the workload dynamic behavior. This paper proposes an on-line program scaling estimator that allows one to capture both short-term and long-term execution characteristics of large software in-flight without storing huge traces. The proposed estimator can be integrated with current performance monitoring systems.

The remainder of this paper is organized as follows. Section 2 introduces the basic scaling models and metrics. Section 3 presents a wavelet-based scaling analysis method. Section 4 describes the experimental setup. Observations from the program scaling analysis are presented and discussed in Section 5. We first look at program second-order scaling statistics and then study its high-order scaling properties. Section 6 presents an on-line program scaling estimator that allows the performance monitoring systems to discover the changes in workload behavior on the long-run programs. Section 7 discusses the related work. We conclude this paper and point out some directions of our future work in the Section 8.

2. Background: Scaling Models

The notion of scaling can be loosely defined as the absence of special characteristic time or space scales. As a result, the whole and its parts can not be statistically distinguished from each other. As a critical phenomenon in the nature, scaling manifests itself in many real world objects (e.g. complex forms and patterns such as clouds, mountains and coastlines) [7].

Since our main focus in this paper is to characterize the scaling of workload behavior in the time domain, we present vari-

ous scaling models in the context of time series in the following subsections.

2.1 Self-similarity

The purest formal framework for scaling is that of exact self-similarity. Self-similarity means that the sample paths of the process $X(t)$ and those of a rescaled version $c^H X(t/c)$, obtained by simultaneously dilating the time axis by a factor $c > 0$, and the amplitude axis by a factor c^H, can not be statistically distinguished from each other (Figure 1). The Hurst parameter H, is used to measure the degree of self-similarity. The closer H is to 1, the stronger self-similarity the process exhibits.

Figure 1 Self-similarity: a dilated portion of the sample path of a process can not be statistically distinguished from the whole.

Exact self-similarity fulfils the intuition of scaling in a perfect way. However, the model is overly rigid. The remainder of this section details more flexible models that enable some deviations from the exact self-similarity.

2.2 Long-Range Dependence

For a time series $x(t)$ contains the data of interest, the basic features of this process are its mean $u_x = E[x]$, variance $\sigma_x^2 = E[(x-u_x)^2]$, and correlation function $r_x(k) = E[(x(t+k)-u_x)(x(t)-u_x)]$. The process $x(t)$ displays *long-range dependence* (LRD) if its correlation function $r_x(k)$ behaves like a power-law of the time lag k, i.e.

$$r_x(k) \sim c_r |k|^{2H-2} \quad \text{as} \quad |k| \to \infty$$

where c_r is a positive constant and the Hurst parameter $1/2 < H < 1$. In such a case, the correlations decay so slowly that they sum to infinity. When the Hurst parameter $H = 1/2$, the process manifests *short-range dependence* (SRD). The SRD is characterized by quickly decaying correlations.

3. Scaling Analysis Techniques

In this section, we introduce a wavelet-based methodology as the key tool for scaling discovery. The wavelet analysis [8] acts as a mathematical "microscope" which allows one to zoom in on fine structures of a signal or, alternatively, to reveal large scale structures by zooming out.

3.1 Discrete Wavelet Transform (DWT)

Consider a series $X_{0,k}, k = 0,1,2,...$, at the finest level of time scale resolution 2^{-n}. This time series might represent the measured workload dynamic characteristics (e.g., the number of instructions retired per cycle, cache misses per thousand cycles etc.) during each sampling interval. We can coarsen this event series by averaging (with a slightly unusual normalization factor) over non-overlapping blocks of size two

$$X_{1,k} = \frac{1}{\sqrt{2}}(X_{0,2k} + X_{0,2k+1}) \quad (1)$$

and generates a new time series X_1, which represents a coarser granularity picture of the original series X_0. The difference between the two, known as *details*, is

$$d_{1,k} = \frac{1}{\sqrt{2}}(X_{0,2k} - X_{0,2k+1}) \quad (2).$$

Note that the original time series X_0 can be reconstructed from its coarser representation X_1 by simply adding in the details d_1; i.e., $X_0 = 2^{-1/2}(X_1 + d_1)$. We can repeat this process (i.e., write X_1 as the sum of yet a coarser version X_2 of X_0 and the details d_2, and iterate) for as many scale as are present in the original time series

$$X_0 = 2^{-n/2} X_n + 2^{-n/2} d_n + ... + 2^{-1/2} d_1 .$$

We refer to the collection of details $d_{j,k}$ as the *discrete wavelet coefficients*. Note that for the purpose of illustration simplicity, we choose the Harr wavelet as the mother function here. The calculations of all $d_{j,k}$, which can be done iteratively using the equations 1 and 2, make up the so called *discrete wavelet transform* (DWT).

As wavelet transform divides data into a low-pass approximation and a high-pass detail at any level of resolution and analyzes each component with a resolution matched to its scale, the coefficients of wavelet decomposition can be directly used to study the scale dependent properties of the data. In the following subsections, we describe a set of wavelet-based scaling analysis techniques.

3.2 Energy Function and Logscale Diagram

Given a time series $X_{0,k}, k = 0,1,2,...$, and its discrete wavelet coefficients $d_X(j,.)$, the average energy at resolution level 2^j is then defined as:

$$E_j = \frac{1}{n_j} \sum_{k=1}^{n_j} |d_X(j,k)|^2 \quad (3),$$

where n_j is the number of wavelet coefficients. The *log-scale diagram* (LD) is the plot of E_j as a function of resolution level 2^j (together with the confidence intervals) on a $\log_2 - \log_2$ scale; i.e., $y_j = \log_2(E_j)$ as a function of scale j. Asymptotically, the

energy function is expected to be linear with time scale j for self-similar processes, i.e.

$$y_j = \alpha j + \log_2(E_0), \text{ where } \alpha = 2H - 1$$

The slope of the LD plot provides an estimate of the Hurst parameter H. If $H = 0.5$, $\alpha = 0$ (a flat slope points to a short-range dependent process) while $H > 0.5$ yields $\alpha > 0$ (a positive slope indicates a long-range dependent process).

Therefore, the LD plot allows the detection of scaling through observation of strict alignment (linear trend) of the confidence interval of the y_j within some octave range. If a strict alignment is detected, the scaling parameters can be estimated. The LD is a second-order statistics (sample variance) of the wavelet coefficients, hence it does not capture the higher-order properties of the processes.

3.3 Partition Function and Multiscale Diagram

Multifractal analysis compares the scaling of different wavelet moments q to estimate the local regularity in processes. Its main tool is the wavelet partition function $S(q, j)$, a generalization of the wavelet energy function (3), defined as

$$S(q, j) = \sum_k \left| 2^{-j/2} d_X(j,k) \right|^q \quad (4).$$

By computing the partition function, one can characterize the statistics of the local behavior of workload dynamics. This is because a wavelet is an oscillating function and the values of the wavelet coefficients are proportional to the size of the irregularity. The partition function raises the wavelet coefficients to an exponent and magnifies the importance of the largest coefficients that arise due to a local irregularity. On the contrary, it reduces the importance of small coefficients. Therefore, the smaller the wavelet coefficients for the scale j, the smaller the value of the partition function for $S(q, j)$ for large q. This permits to study the importance of the local irregularities at time scale j.

The *multi-scale diagram* (MD) is the plot of $S(q, j)$ as a function of resolution level 2^j on a $\log_2 - \log_2$ scale; i.e., $\log_2(S(q, j))$ as a function of scale j, for a range of values of the moments q.

A relatively smooth process (second-order process) that shows no particularly large local irregularity will have values of the $\log_2(S(2, j))$ larger than $\log_2(S(q, j))$ (for $q > 2$). On the opposite, an irregular process (high-order process) will tend to have larger values of $\log_2(S(q, j))$ for values of $q > 2$.

4. Experimental Setup

In this paper, we demonstrate the using of wavelet-based multiscaling analysis techniques to characterize the changing nature of workload dynamics over time. This section describes the experimental methodology, including the simulated machine architecture, the studied workloads and the collected benchmark traces. An on-line estimator without requiring the off-line trace analysis will be described in Section 6.

4.1 Machine Configuration

The statistics of workload dynamics are measured on the SimpleScalar 3.0 [9] *sim-outorder* simulator. The simulated microarchitecture configuration in this study is an 8-way superscalar model. The details of the machine configurations are summarized in Table 1.

Table 1 Processor configurations

Parameter	Configuration
Processor Width	8
ITLB	128 entries, 4-way, 200 cycle miss
Branch Prediction	combined 8K tables, 10 cycle misprediction, 2 predictions/cycle
BTB	2K entries, 4-way
Return Address Stack	32 entries
L1 Instruction Cache	32K, 2-way, 32 Byte/line, 2 ports, 4 MSHR, 1 cycle access
RUU Size	128 entries
Load/ Store Queue	64 entries
Store Buffer	16 entries
Integer ALU	4 I-ALU, 2 I-MUL/DIV
FP ALU	2 FP-ALU, 1FP-MUL/DIV
DTLB	256 entries, 4-way, 200 cycle miss
L1 Data Cache	64KB, 4-way, 64 Byte/line, 2 ports, 8 MSHR, 1 cycle access
L1 Cache	unified 1MB, 4-way, 128 Byte/line, 12 cycle access
Memory Access	100 cycles

4.2 Program Traces

We use the twelve SPEC2000 integer programs as the experimented workloads. To analyze the workload dynamics at large time scales, we use the reference input data sets and simulate 9 out of 12 workloads until completion. Benchmarks *mcf, parser* and *twolf* take extremely long time to complete. For these three benchmarks, we stop the simulation after the benchmark execution cycles reach the time scales that are comparable to those of other completely simulated benchmarks.

During the simulations, the statistics of the architectural characteristics of interest are collected and recorded in traces. The instructions-per-cycle (IPC) metric, which indicates how efficiently a microprocessor performs its functions, is mainly used in this study. Each collected trace contains a sequence representing the program IPC rate measured on every ten-thousand elapsed cycles. The program traces statistics are summarized in Table 2. One can see that these collected traces allow the study of the changing nature of workload dynamics over a period of hundreds of billion cycles.

5. Characterizing Program Scaling Behavior

This section provides a detailed characterization of the SPEC2000 integer benchmarks scaling behavior using the methodology described in Section 3.

Table 2 Benchmark traces description

Benchmark	Input	Duration (Cycles)
mcf	/ref/inp.in	570,689,841,862
gcc	/ref/166.i	33,578,085,795
crafty	/ref/crafty.in	337,250,101,460
gzip	/ref/input.graphic	52,867,265,321
bzip2	/ref/input.source	70,644,828,028
eon	/ref/chair.cook.ppm	93,485,005,275
gap	/ref/ref.in	355,758,277,267
parser	/ref/ref.in	247,035,615,983
perlbmk	/ref/splitmail.pl	49,931,474,883
twolf	/ref/ref	274,987,890,000
vortex	/ref/lendian1.raw	93,677,830,341
vpr	/ref/net.in	122,267,820,515

5.1 Visualization of the Traces

We begin our analysis of multiscaling characteristics with a demonstration of the time-varying behavior of two programs from SPEC2000, *crafty* and *gzip*. Figure 2 visualizes the traces across 5 orders of magnitude of time scales. Each plot within the figure represents the IPC rate, enumerated as the number of instructions retired per time unit. Successive plots are refinements of the previous plots; the top plot in each column has a time unit of 4.096×10^7 cycles, the second of 5.12×10^6 cycles, and so one. The left column illustrates the IPC dynamics of benchmark *gzip*, and the right column illustrates the IPC dynamics of benchmark *crafty*.

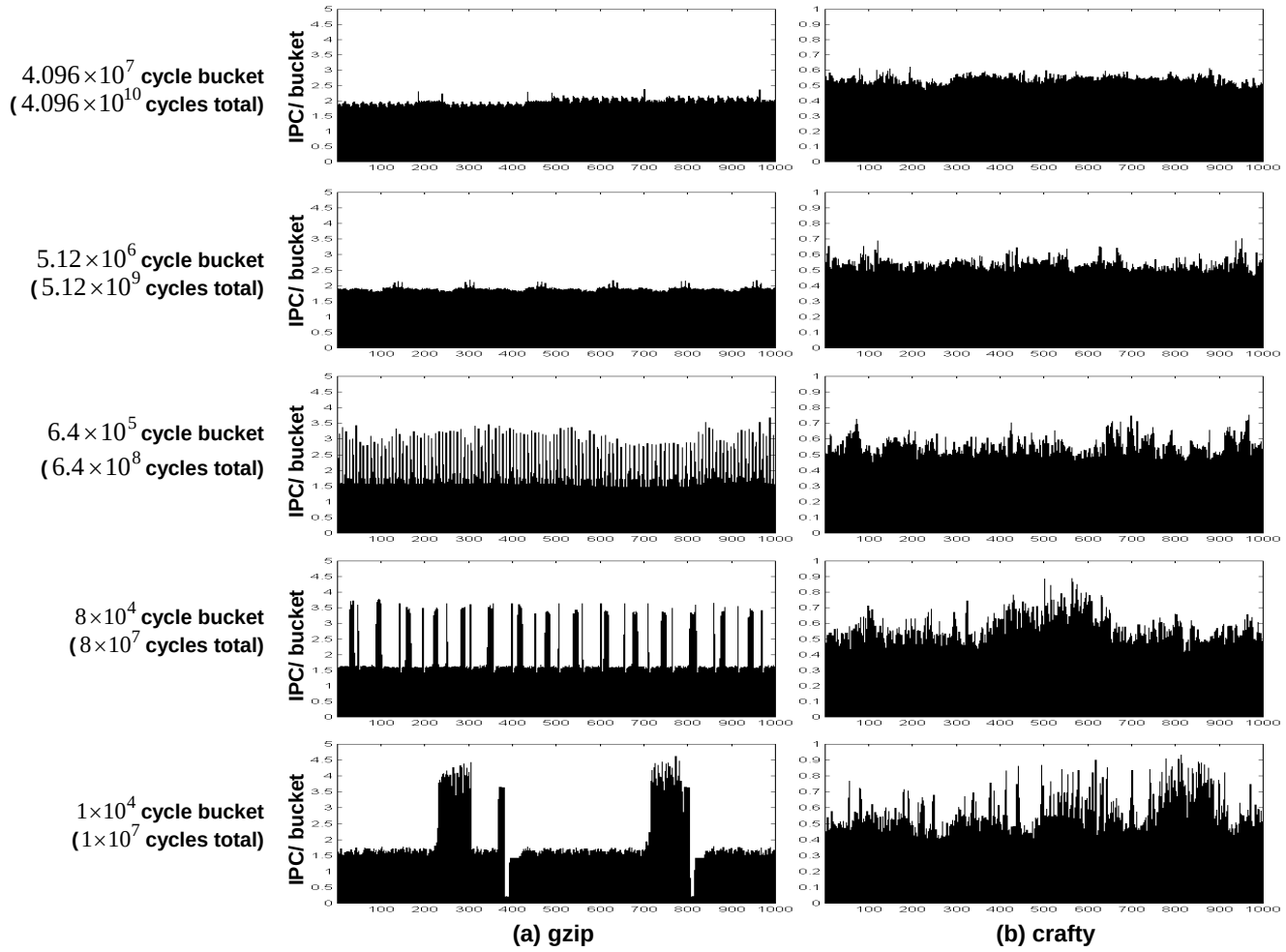


Figure 2 Visualization of the IPC dynamics: programs IPC per unit time (or "bucket") are plotted for 5 increasingly refined time units, varying by a total factor of 4,096. Column (a) shows benchmark *gzip*, and column (b) shows benchmark *crafty*. The x-axis of all graphs is the number of buckets; the y-axis shows the IPC during that bucket.

One can see that the IPC dynamics of both programs change across different time scales. For the *gzip* trace, the pronounced phases occur at fine time granularities (1×10^4 and 8×10^4 cycle buckets). For coarse granularities (6.4×10^5 cycle bucket), the trace manifests very bursty and unpredictable behavior. On the large time scales (5.12×10^6 and

4.096×10^7 cycle buckets**),** the trace becomes extremely smooth, although a small number of spikes can be seen to interrupt this smoothness. Therefore, the workload dynamics of *gzip* changes over its execution time. Characteristics observed at small time scales will not represent those at large time scales.

The IPC dynamics on benchmark *crafty*, on the other hand, appears bursty at all time scales, although the burstiness appears to smooth out at coarser time granularities. A closer investigation reveals that the trace exhibits scale invariance, i.e. the workload dynamics appear to be similar at all time scales. Viewing the collected traces as time series, we then use the multiresolution analysis techniques described in the Section 3 to investigate the program scaling behavior.

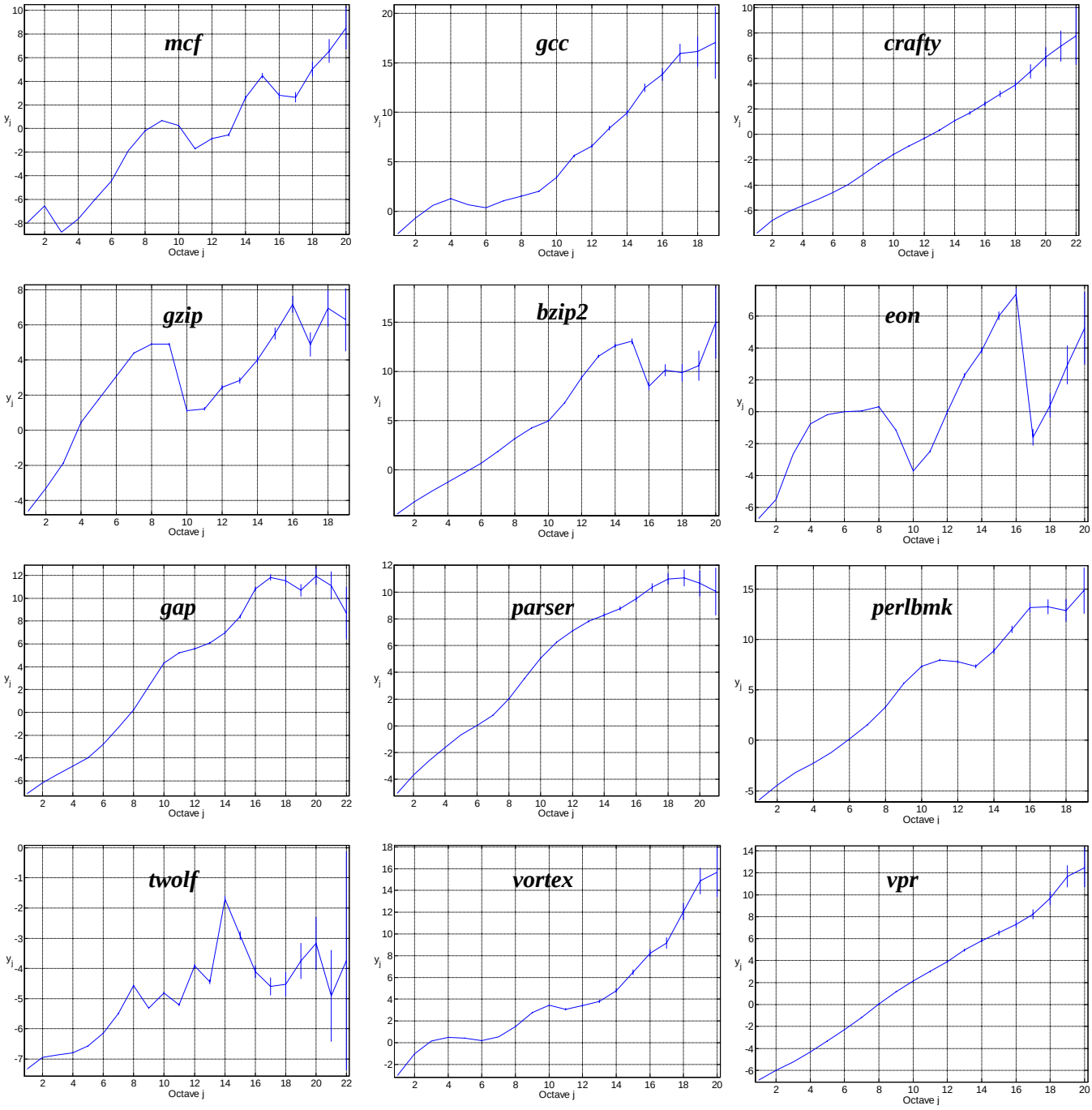


Figure 3 the LD plots of SPEC2000 integer benchmark traces

5.2 Second-order Scaling Analysis

In this section, we study the second-order scaling properties of the experimented workloads. The second-order scaling properties over the total length of the traces are analyzed.

Figure 3 presents the LD plots for the SPEC 2000 integer benchmarks. The x-axis shows time scales, varied from fine to coarse resolutions. The vertical bars at each octave give the 95% confidence intervals for the y_j. Confidence intervals about the y_j increase monotonically with j as one moves to larger and larger scales, as seen in each of the diagram in Figure 3.

A property of the LD concerns the theoretical possibility to distinguish the type of process present within some octave range based on the value of the slope of the logscale diagram. As described in Section 3, asymptotically self-similar (or, equivalently, long-range dependence) will result in a linear relation between y_j and the scale j. If the trace is exactly self-similar, a plot of y_j vs. j will show a linear relationship for all scales.

Looking at Figure 3, one can see that the SPEC2000 integer programs show heterogeneous scaling characteristics. For benchmarks *crafty*, *vpr*, and *parser*, there is a clear evidence of strict scaling across all octaves, i.e. the traces of these benchmarks exhibit self-similarity. More frequently, one sees the LD plots with scaling over a limited range of time scales, two or multiple different scaling regions, a flat slope regions and dips. A flat slope region in the LD plots indicates that the program dynamics statistics look like the uncorrelated "white noise" across the spanned time scales. A dip in the LD plots implies a periodic component. This is because the highly regular or periodic structures (e.g. loops) in time series reveal themselves in terms of small wavelet coefficients and subsequently the low values of the energy function.

For small time scales between octaves 1 and 6, the studied workloads seem to exhibit various characteristics. For example, periodicity occurs on benchmarks *mcf* and *gcc* by showing the dips in their LD plots. Octaves in [2,4] for *twolf* and octaves in [3,6] for *vortex* show evidence of SRD.

For time scales between octaves 6 and 16, the linear relation between y_j and scale j are observed on the majority of the studied workloads, implying that the corresponding traces are consistent with self-similarity or long-range dependence. Typical scaling regions extend several orders of magnitudes, with upper cutoffs that vary from one benchmark to another. *Mcf*, *gzip*, *eon* and *twolf* lack LRD characteristics on these time sales. For these benchmarks, the linear scaling properties are interrupted by the dips due to periodic execution spanned across these time scales.

Note that benchmark *vortex* trace, with some "bumps" around scales 9-11, exhibits some indications for departure from exact LRD. The *perlbmk* trace shows this departure much clearer: a non-trivial scaling behavior for small time scales (octaves below 10) and a distinctly different large-time scaling behavior (octaves above 13), with a "change-region" that shows up very clearly as a pronounced "knee" in the graphs, approximately at scales [10,13].

It is observed that the dips at large time scales occur on benchmarks *mcf*, *gzip*, *bzip2*, *eon*, *gap* and *twolf*, indicating the those programs contain periodic structures even at very large time scales. In general, the time-varying slope of the LD's on Figure 3 for the largest time scales (octave in [16 20]) reveal that these time-scales contain a mix of correlation and scaling.

5.3 High-order Scaling Analysis

To discover the high-order scaling properties of the collected traces, we use the partition function and multiscale diagram described in subsection 3.3. While the LD reveals the energy (variance) of the increments of a process, the partition function and MD give information about the distribution of large and small wavelet coefficients at each time scale. We plot $\log(S(q, j))$ as a function of the scale j, for a range of the moments q ($1 \le q \le 8$). Discriminating between second-order (smooth) and high-order (irregular) process can be done by determining for which values of q the $\log(S(q, j))$ is the largest. A second-order process will have larger $\log(S(q, j))$ for $q \le 2$ while a high-order process for $q > 2$.

Figure 4 presents the logarithm of the partition function $\log(S(q, j))$ for all benchmarks. It shows that at large time scales, most of the studied benchmarks show the second-order scaling behavior ($\log(S(q, j)) < \log(S(2, j))$ for $q > 2$). This indicates that the second-order scaling properties are sufficient to describe the workload dynamic behavior at large time scales. Benchmarks *gcc*, *parser*, *perlbmk*, and *vortex* have large irregularities at small time scales. This implies that these traces contain large wavelet coefficients at small time scales, hence large irregularities. *Bzip2* and *gap* contain irregularities even at coarser time scales. For those programs, although at large time scales, all traces are second-order processes, for the short time scales behavior of workloads, multiplicative processes described it well.

6. On-line Program Scaling Estimation

So far, our workload dynamics scaling analysis has heavily relied on the trace collection and off-line processing. The demand on huge trace storage and off-line processing makes this method incapable of handling large programs that run long time. Moreover, for many applications, such as resource allocation/adaptation and performance monitoring/prediction, run-time program scaling measurement is necessary.

The Hurst parameter H, which measures the degree of self-similarity, holds a central place in the description of scaling behavior. Its accurate measurement is therefore of considerable importance. In this section, we present an on-line program scaling estimator that allows one to characterize program scaling behavior in-flight.

To calculate H, (1) a discrete wavelet transform (DWT) of the input data is first performed to generate the details $d_{j,k}$ over the time scales. (2) Then, at each fixed octave j, the details are squared and averaged across time k to produce the energy function E_j. (3) Finally, the Hurst parameter H is determined from the scaling regions in the LD plot.

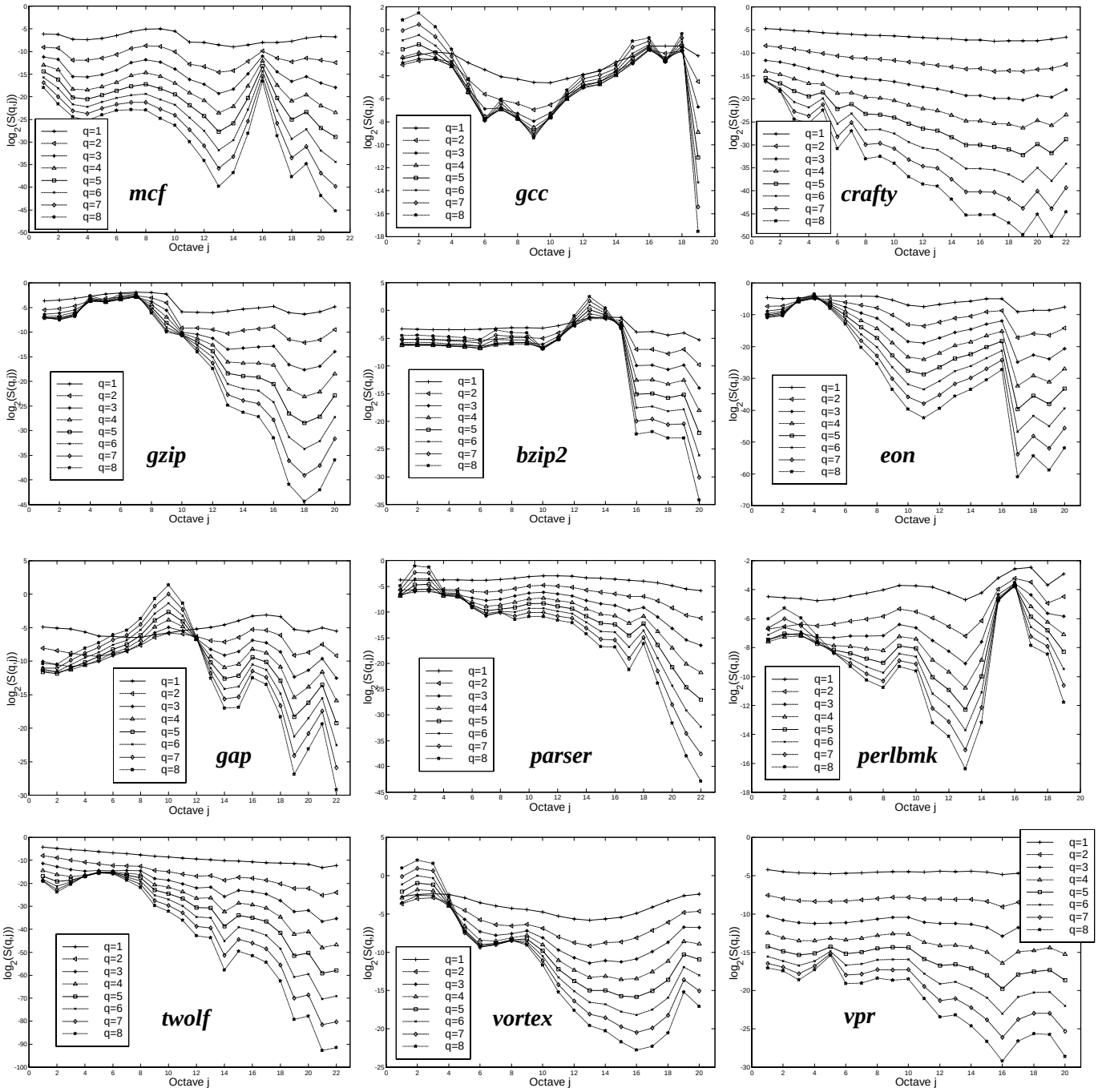


Figure 4 the MD plots

6.1 DWT and Filter-banks Design

The on-line discrete wavelet transform plays the key role in scaling estimation. Steps (2) and (3) are trivial once the details are computed. As described in Section 3, the DWT can be viewed as the multiresolution decomposition of an input sequence. The decomposition processes is implemented with two digital filters: high-pass filter G and low-pass filter H. After the process of filtering, downsampling operation is used to decimate half of the filtered results. To achieve multiresolution decomposition, the multilevel DWT can be implemented by the Mallat's pyramid algorithm [10]. Figure 5 shows the steps of Mallat's pyramid algorithm for DWT computation.

Figure 6 shows a three-level DTW using the pyramid algorithm and filter-banks. The output coefficients at each resolution level are calculated by using the low-pass coefficients of the previous level and are decimated by two.

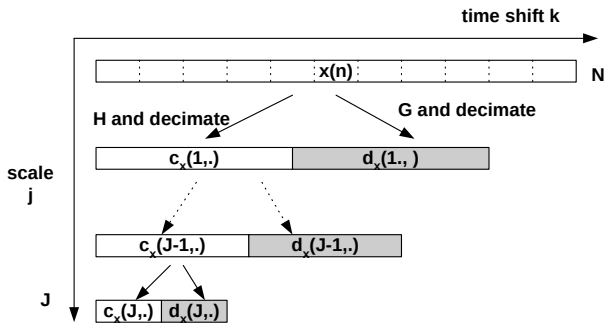


Figure 5 Mallat's pyramid algorithm for DWT computation

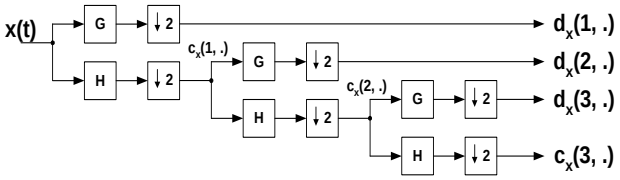


Figure 6 A three resolution level wavelet decomposition. At each level in the recursive structure, the high-pass filter (G) output $d_x(j,.)$ and the low-pass filter (H) output $c_x(j.,)$, occur at half the rate of the low-pass filter input $c_x(j-1,.)$

All filters in the pyramid tree shown in Figure 7 are constructed using FIR filters [11]. The structure of a FIR filter of length N is shown in Figure 7. As can be seen, each filter tap consists of a delay element, an adder, and a multiplier.

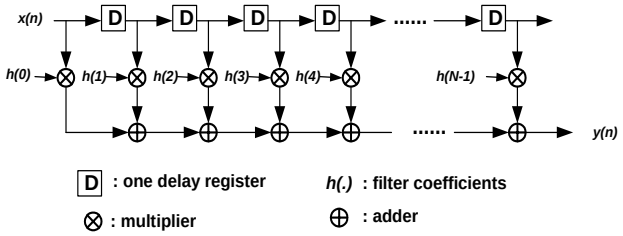


Figure 7 FIR Filter Structure

The filter coefficients are derived from the scaling functions of the corresponding wavelets. In this study, we use Daubechies wavelets to implement the DWT. This wavelet type is known for its excellent special and spectral localities [12]. The coefficients of the 3 wavelets from the Daubechies wavelets family are summarized in Table 3.

The Daubechies filters have the following relationship between its low-pass filter and high-pass filter coefficients:

$$g_{M-1-i} = (-1)^i h_i \quad (M > 2)$$

In this study, we use Daubechies 6-tap wavelet as the filter coefficients.

Table 3 Filter Coefficients of Daubechies Wavelets

		Daubechies Wavelets		
Coefficients i		M=6	M=8	M=10
i=0	*h(0)*	0.332671	0.230378	0.160102
i=1	*h(1)*	0.806892	0.714847	0.603829
i=2	*h(2)*	0.459878	0.630881	0.724309
i=3	*h(3)*	-0.135011	-0.027984	0.138428
i=4	*h(4)*	-0.085441	-0.187305	-0.242295
i=5	*h(5)*	0.035226	0.030841	-0.032245
i=6	*h(6)*		0.032883	0.077571
i=7	*h(7)*		-0.010597	-0.006241
i=8	*h(8)*			-0.012581
i=9	*h(9)*			0.003336
$\sum\|h(n)\|$		1.85118	1.865446	2.000938
$\max\{\|h(n)\|\}$		0.806892	0.714847	0.724309

6.2 Scaling Estimation Methodology

In a system designed to measure program dynamics on-line, there are two key components: the workload dynamics capture process and the on-line estimation itself.

Figure 8 illustrates the overall structure of scaling estimation and the interface between CPU and the modules. As can be see, performance counters and clock are used to generate the events of interest during program execution. The on-line estimator can be integrated into the operating system kernel or mapped with the highly efficient VISI architecture [23]. For the simulation based studies, the one-line scaling estimation module can be integrated into architectural simulators to provide estimates. In this work, we have integrated the proposed estimator into the SimpleScalar *sim-outorder* simulator.

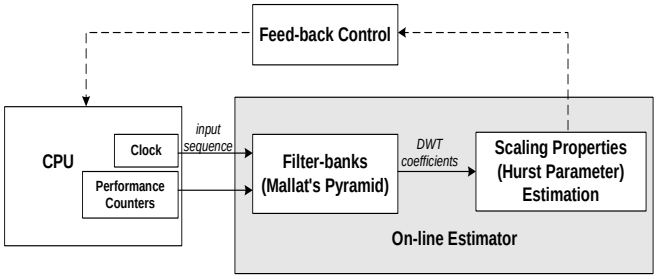


Figure 8 Overall Structure of Scaling Estimation

6.3 Performance

The performance of the estimator as a function of the length of data processed is demonstrated using execution driven simulations on the machine model described in Section 4. The IPC of program for every ten thousands elapsed cycles, generated by the *sim-outorder* simulator, is piped to the on-line estimator one at a time. The interval chosen between the actual estimations of H is every 2^8 data points.

Figure 9 shows the three examples of on-line estimation. The graphs illustrate typical behavior of the estimator in time. The dashed line shows the true Hurst parameter while the solid lines show examples of the one-line Hurst parameter estimates. As can be seen, there is a warm up period at the beginning of

the measurement run to wait for the octaves required for the analysis to become available. Figure 9 shows that the on-line estimator can accurately estimate the program scaling behavior.

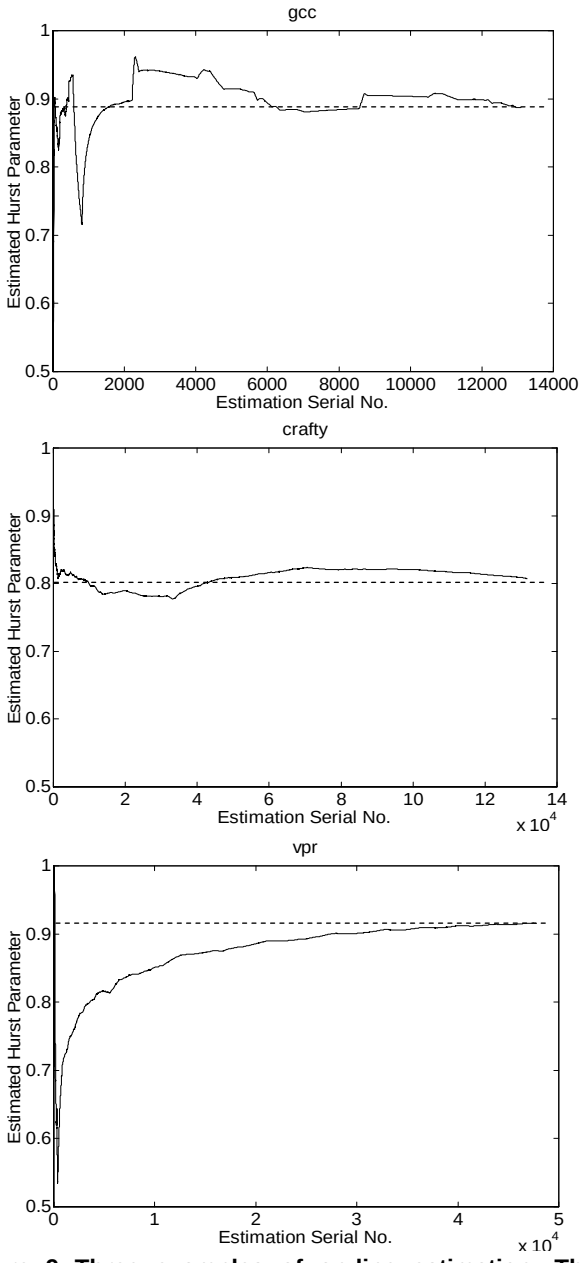


Figure 9 Three examples of on-line estimation. The dashed lines show the true Hurst parameter while the solid lines show examples of the on-line Hurst parameter estimates.

7. Related Work

The analysis of workload dynamic/phases behavior and then apply to run-time optimizations have recently received considerable attention [5, 6, 13, 14, 15]. For example, the SimPoint framework proposed by Sherwood and Calder [5] uses a set of representative instruction chunks to represent the entire program execution. In [24], variable length intervals are used to study hierarchical phase behavior of programs. Scaling models have been applied in the past to diverse fields, such as the study of strange attractors of certain dynamic systems [16] and modeling of traffic in modern communication networks [17]. Self-similarity, long-range dependence, and multifractal behavior have been studied and convincingly matched to real network traffic [18, 19, 20]. In [21], the authors present a first study to apply wavelet analysis to the computer architecture field. The VLSI architectures for implementing the DWT are summarized in [22, 23].

8. Conclusions and Future Work

The efficiency of advanced hardware and system optimizations increasingly depend on the dynamic behavior of workloads. Program execution manifests changing nature at run time. As software execution cycles become larger, such variation can span across a wide range of time scales. Understanding and characterizing the time-varying behavior of workload dynamics lays a foundation for an informed investigation on these long-run systems. Moreover, knowledge of program scaling properties is important to support tasks such as resource adaptation, performance monitoring, and anomaly detection at different time periods.

We show in this paper that various scaling properties (e.g., self-similarity, long-range dependence, and multiscaling) and metrics (e.g., energy and partition functions), combined with a wavelet-based analysis technique, can be used as a useful tool for unraveling the program dynamics over different time periods. We then apply this methodology to study the scaling behavior of SPEC2000 integer benchmarks, that is, to identify regions where the scaling property holds, to detect changes in scaling behavior, and to find ranges of time scales with more complex scaling patterns. An on-line program scaling estimator of the Hurst parameter is developed to allow the measuring of program scaling properties at run-time.

The compact and parsimonious models that exploit scaling phenomena can capture both long-term and short-term program behavior. In the future work, we plan to investigate the feasibility of using scaling models for benchmark statistics simulations and benchmark synthesis. The using of program scaling properties for resource adaptation, performance prediction, QoS control, and bottleneck and anomaly detections will also be investigated.

References

[1] A. Maynard, C. Donnelly, and B. Olszewski, Contrasting Characteristics and Cache Performance of Technical and Multi-user Commercial Workloads, In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[2] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael and W. E. Baker, Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads, In the Proceedings of the International Symposium on Computer Architecture, 1998.

[3] L. A. Barroso, K. Gharachorloo, E. Bugnion, Memory System Characterization of Commercial Workloads, In

the Proceedings of the International Symposium on Computer Architecture, 1998.

[4] L. Eeckhout, R. Bell, B. Stougie, K. D. Bosschere and L. K. John, Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies, In the Proceedings of the International Symposium on Computer Architecture, 2004.

[5] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, Automatically Characterizing Large Scale Program Behavior , In the Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.

[6] E. Duesterwald, C. Cascaval, and S. Dwarkadas, Characterizing and Predicting Program Behavior and its Variability, In the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2003.

[7] D. Sornette, Critical Phenomena in Natural Sciences, Springer-Verlag, 2000.

[8] I. Daubechies. Ten Lectures on Wavelets. Capital City Press, Montpelier,Vermont, 1992.

[9] D. Burger and T. M. Austin, The SimpleScalar Tool Set, Version 2.0, Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

[10] S. Mallat, Multifrequency Channel Decompositions of Images and Wavelet Models, IEEE Transactions on Acoustic, Speech, and Signal Processing, vol. 37, page 2091-2110, 1989.

[11] A. Oppenheim and R. Schafer, Discrete Signal Processing, New Jersy: Prentice Hall, 1999.

[12] I. Daubechies, Orthonomal bases of Compactly Supported Wavelets, Communications on Pure and Applied Mathematics, vol. 41, pages 906-966, 1988.

[13] T. Sherwood, S. Sair and B. Calder, Phase Tracking and Prediction, In the Proceedings of the Annual International Symposium on Computer Architecture, 2003.

[14] A. S. Dhodapkar and J. E. Smith, Managing Multi-configuration hardware via Dynamic Working Set Analysis, In Proceedings of the International Symposium on Computer Architecture, 2002.

[15] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures, In Proceedings of the International Symposium on Microarchitecture, 2000.

[16] D. Auerbach, B. O'Shaughnessy, and I. Procaccia, Scaling Structure of Strange Attractors, Physical Review A, vol. 37, issue 6, page 2234–2236, Mar. 1988

[17] A. Feldmann, A. C. Gilbert, W. Willinger, and T. G. Kurtz, The Changing Nature of Network Traffic: Scaling Phenomena, ACM Computer Communication Review, vol. 28, page 5-29, Apr. 1998.

[18] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, On the Self-Similar Nature of Ethernet Traffic, IEEE/ACM Transactions on Networking, vol. 2, no. 1, page 1-15, Feb. 1994.

[19] A. Feldmann, A. C. Gilbert, and W. Willinger, Data Networks as Cascades: Investigating the Multifractal Nature of Internet WAN traffic, ACM Computer Communication Review, vol. 28, page 42-55, Sept. 1998.

[20] P. Abry and D. Veitch, Wavelet Analysis for Long-range Dependent Traffic, IEEE Transactions on Information Theory, 44 page 2-15, 1998.

[21] R. Joseph, Z. Hu, and M. Martonosi, Wavelet Analysis for Microprocessor Design: Experiences with Wavelet-Based dI/dt Characterization, In the Proceedings of the International Symposium on High Performance Computer Architecture, 2004.

[22] C. Chakrabarti, M. Viswanath, R. M. Owens, Architectures for Wavelets Transforms: A Survey, Journal of VLSI Signal Processing, vol. 41, page 171-192, Dec. 1996.

[23] P. Y. Chen, VLSI Implementation for One-Dimensional Multilevel Lifting-Based Wavelet Transform, IEEE Transactions on Computers, vol. 53, no. 4, Apr. 2004.

[24] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder, Motivation for Variable Length Intervals and Hierarchical Phase Behavior, In the Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2005.

The Case for Automatic Synthesis of Miniature Benchmarks

Robert H. Bell, Jr. [‡†]
[‡]IBM Systems and Technology Division
Austin, Texas
robbell@us.ibm.com

Lizy K. John [†]
[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
ljohn@ece.utexas.edu

Abstract

There are two parts of the design process that can benefit from reduced, miniature benchmarks that behave like longer-running applications during simulation: 1) the early design phase before an implementation exists, and 2) the later design phases when cycle-accurate functional models exist. In the early design phase, many hundreds or thousands of potential design tradeoffs must be evaluated rapidly at a high-level. In the later design phases, design changes are costly to undo, so potential changes need to be accurately evaluated using a performance model that has been validated against a functional model. Applications typically run too long to be executed completely for either early design tradeoffs or late performance model validation.

In this paper, we explore the potential for automatically synthesizing reduced miniature benchmarks from the execution characteristics of actual applications. We discuss the advantages and challenges of automatic synthesis and present evidence that miniature benchmarks can reproduce the machine behavior of much longer running applications. We present one approach to benchmark synthesis and show that IPC and many average characteristics of the executing synthetic benchmarks are similar to those of the applications that the synthetic benchmarks are derived from. We also show that an early design task like identifying performance trends due to design changes can be carried out while still reducing runtimes significantly. The synthetic benchmarks converge to results rapidly, enabling performance model validation.

1. Introduction

Over two decades ago, researchers used synthetic benchmarks like Whetstone[10] and Dhrystone[30] to approximate the performance of applications on their designs. However, the early synthetic benchmarks fell out of favor because they were difficult to maintain and upgrade in the face of ever-evolving languages, libraries and programming styles. In the early phases of the pre-silicon design process, researchers turned to simulation of real applications, and some applications (like SPEC [26]) have became benchmarks of computer performance. However, long runtimes for the latest benchmarks make full program simulation for early design studies impractical [22] [33].

In the later phases of the pre-silicon design process, the validation of a performance model against a functional model or hardware is necessary at various times in order to minimize incorrect design decisions due to inaccurate performance models [5]. As functional models are improved, accurate performance models can pinpoint with increasing certainty the effects of particular design changes. This translates into higher confidence in late pre-silicon or second-pass silicon design performance.

Prior validation efforts have focused on handwritten microbenchmarks or short tests of random instructions [6] [27] [4] [18] [17] [16] [34]. Black and Shen describe tests of up to 100 randomly generated instructions [4], not enough to approximate many characteristics of applications. Desikan *et al.* use microbenchmarks to validate an Alpha 21267 simulator to 2% error [11], but the validated simulator still gives errors from 20% to 40% when executing the SPEC2000 benchmarks.

Ideally, SPEC and other applications would be used for performance model validation, but, again, this is limited by their long runtimes on functional simulators [5]. In [36], only one billion simulated cycles *per month* are obtained. In [34], farms of machines provide many cycles in parallel, but individual tests on a 175 million-transistor chip model execute orders of magnitude slower than the hardware emulator speeds of 2500 cycles per second.

Sampling techniques such as SimPoint [22], SMARTS [33] and Intrinsic Checkpointing [35] can reduce application runtimes, making early design studies feasible, but it is still necessary to execute tens of millions of instructions. Statistical simulation creates representative synthetic traces with less than one million instructions [8] [19] [12], but traces are not useful for functional model validation.

Sakamoto *et al.* combine a modified trace snippet with a memory image for execution on a specific machine and a logic simulator [21], but the method is machine-specific

and there is no attempt to reduce the total number of simulated instructions. In [14], assembly programs are generated that have the same power consumption signature as applications. However, all workload characteristics are modeled as microarchitecture-dependent characteristics, so the work is not useful for studies involving design trade-offs [13]. Wong and Morris [32] investigate synthesis for the LRU hit function to reduce simulation time, but no method of simultaneously incorporating other workload characteristics is developed. The research community recognizes the need for a general synthesis method [23], but none has been forthcoming.

In this paper, we discuss the problem of synthesizing reduced, miniature benchmarks for early design studies and performance validation. We describe an example synthesis system that uses the workload characterization and graph analysis of statistical simulation in combination with specific memory access and branching models as in [2][3]. A miniature benchmark is generated as C-code with low-level instructions instantiated as *asm* statements. When compiled and executed, the synthetic code reproduces the dynamic workload characteristics of an application, and yet it can be easily executed on a variety of performance and functional simulators, emulators, and hardware, and with significantly reduced runtimes.

The rest of this paper is organized as follows. Section 2 presents properties necessary for the miniature benchmarks and some of their benefits. Sections 3 and 4 gives an overview of the synthesis approach and some experimental results. Section 5 gives some discussion, and the last sections present conclusions and references.

2. Representative Miniature Benchmarks

Automatic benchmark synthesis is most useful if the synthesized benchmark has the following two properties:

1) The benchmark reproduces the machine execution characteristics of the application upon which it is based.

2) The benchmark converges to a result much faster than the original application.

If the first property holds, the benchmark is said to be *representative* of the original application, at least over some range of instructions or workload characteristics. The workload characteristics can be categorized into two classes [13]: *microarchitecture-independent* metrics such as instruction mix, dependency distances, basic blocks, and temporal and spatial locality; and *microarchitecture-dependent* metrics such as cache miss rates and branch predictability. If the second property does not hold to some degree, there is no good reason to use the synthetic benchmark over the original application.

Prior work usually focuses on one of the properties at the expense of the other, or on both properties but over a narrow range. The hand-coded tests and automatic random tests in Black and Shen [4] converge quickly (property 2), but they provide limited or inefficient coverage of all the instruction interactions in a real application (property 1). The *reverse-tracer* system [21] achieves accurate absolute performance for a short trace (property 1), but no runtime speedup is obtained. In [14], both properties are achieved, but workload characteristics that are important to performance, like the instruction sequences and the dependency distances [1], are not maintained. Intrinsic Checkpointing [35] achieves both properties but is only incrementally faster than SimPoint [22].

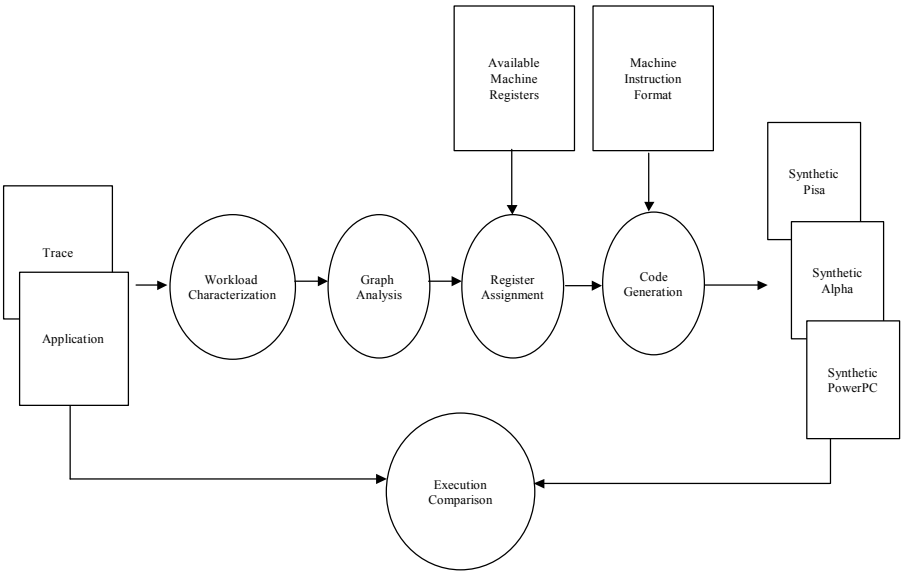


Figure 1: Miniature Benchmark Synthesis and Simulation Overview

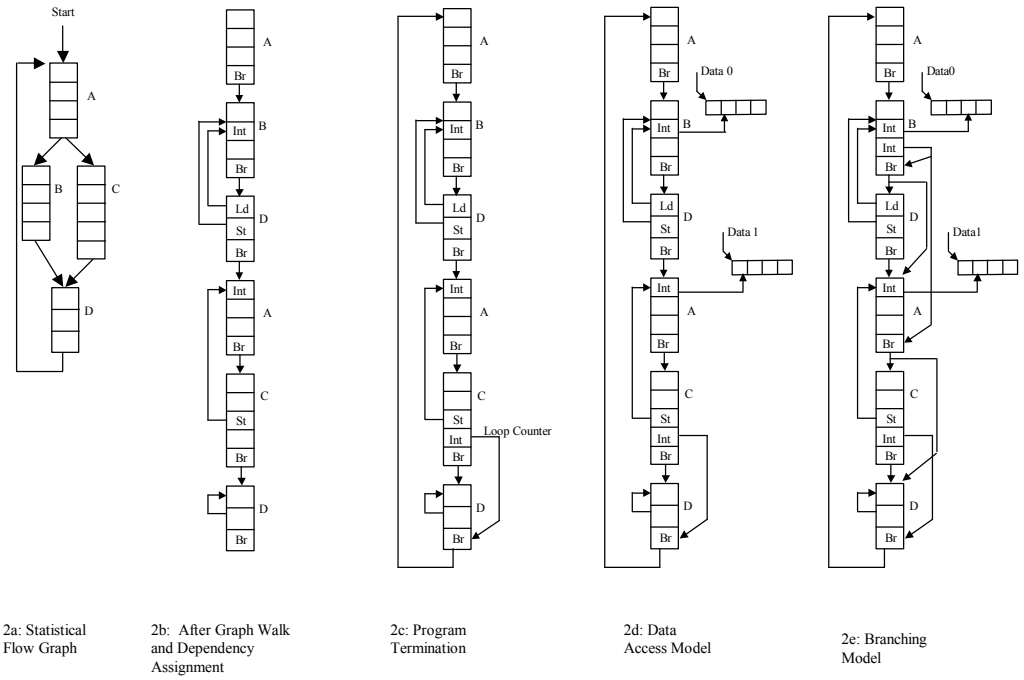


Figure 2: Benchmark Synthesis Illustrated

In practice, achieving both properties for the representative phases of an entire workload is difficult, but in most cases it is not necessary. For validation purposes, a reduced synthetic benchmark need represent only specific application features of interest, not all features. For early design studies, many prominent workload features must be represented, but absolute accuracy [12] need not be high as long as performance trends from design changes are visible, i.e. *relative* accuracy [12] is high.

Any miniature benchmark synthesis system that satisfies the two properties will introduce errors because the application size, in terms of the numbers of instructions executed, has been scaled down in order to obtain a runtime speedup. Ideally, the benchmark will be generated in a high-level language like C. Assuming that the errors can be kept at acceptable levels, such high-level benchmarks have several advantages over sampled traces. These include: portability to a variety of machines, emulators, and execution-driven simulators; and flexibility with respect to easier modification of the code to study changes in workload characteristics or future workloads.

In the next section, we present a synthesis system that takes a step toward automatically generating reduced, miniature benchmarks that can satisfy both properties simultaneously for many workload characteristics. We discuss the modeling abstractions and the errors that are introduced by the synthesis process.

3. Example Synthesis System

Figure 1 depicts the synthesis process at a high level. There are four major phases: *workload characterization; graph analysis; register assignment* and *code generation*. In this paper we give an overview of the synthesis process and philosophy. Additional detail, as well as exact synthesis parameters and algorithms for *Pisa* and *Alpha* code targets, can be found in [2] and [3].

Figure 2 gives a step-by-step illustration of the synthesis process, which we describe below. At a high level, we start with the statistical flow graph from statistical simulation [12][1], which is a reduced representation of the control flow instructions of the application. The graph is walked, giving a representative synthetic trace. We then apply algorithms to instantiate low-level instructions, specify branch behaviors and memory accesses, and generate code, yielding a simple but flexible program.

3.1. Workload Characterization

The dynamic workload characteristics of the target program are profiled using a functional simulator, cache simulator and branch predictor simulator as in [12][1]. The characterization system currently takes input from fast functional simulation using SimpleScalar [7] or trace-driven simulation in an IBM proprietary performance simulator. We characterize the basic block instruction

sequences, the instruction dependencies, the branch predictabilities, and the L1 and L2 I-cache and D-cache miss rates at the granularity of the basic block. Instructions are abstracted into five basic classes: integer, floating-point, load, store, and branch. Long and short execution times for integer and floating-point instructions are distinguished. We also track the IPC of the original workload to compare to the synthetic result.

The statistical flow graph [12][1] is assembled from the workload characterization. An example is given in Figure 2a. Basic blocks A, B, C and D each has various probabilities of branching to one or more basic blocks.

3.2. Graph Analysis

We use the workload characterization to build the pieces of the synthetic benchmark. The statistical flow graph is walked using the branching probabilities for each basic block, and a linear chain of basic blocks is assembled, as in Figure 2b. This chain will eventually be emitted directly as the central operations of the synthetic benchmark. The number of instantiated basic blocks is equal to an estimate of how many blocks are needed to match the I-cache miss rate of the application given a default I-cache configuration [2][3]. We then tune the number of synthetic basic blocks to match the I-cache miss rate and instruction mix characteristics by iterating through synthesis a small number of times. In practice, anywhere from one to 1000 basic blocks may be necessary to meet the I-cache miss rate of a particular application. Typically less than 4000 instructions are synthesized.

For each basic block, we assign instruction input dependencies (also Figure 2b). The starting dependence for each instruction is taken from the average found for the instruction during workload characterization. If the dependency is not compatible with the input operand type of the dependent instruction, then another instruction is chosen. The algorithm is to move forward and backward from the starting dependency through the list of instructions until the dependency is compatible. In practice, the average number of moves per instruction input is small, usually less than one. For loads and stores, the data address register must be of integer type. When found, it is attributed as a memory access counter for special processing during the code generation phase.

When all instructions have compatible dependencies, a search is made for an additional integer instruction that becomes the loop counter (Figure 2c). The branch in the last basic block in the program checks the loop counter to determine when the program is complete. The number of executed loops is chosen to be large enough to assure IPC convergence given the memory accesses of the load and store instructions in the benchmark. In practice, the number of loops does not have to be very large to characterize simple stream access patterns. Experiments have shown that the product of the loop iterations and the number of instructions must be around 300K to achieve low branch predictabilities and good stream convergence. The loop iterations are therefore approximately *300K/4000* for most benchmarks.

The data access counter instructions are assigned a stride based on the D-cache hit rate found for the corresponding load and store accesses during workload characterization (Figure 2d). The memory accesses for data are modeled using the sixteen simple stream access classes shown in Table 1. The table was generated based on a default cache configuration [2][3], and the stride is shown in four byte increments. The stride for a memory access is determined first by matching the L1 hit rate of the load or store that is fed by the access counter, after which the L2 hit rate for the stream is predetermined.

By treating all memory accesses as streams, the memory access model is kept simple. This reduces changes to the instruction sequences and dependencies, which have been shown to be critical for correlation with the original workload [1]. On the other hand, there can be a large error in stream behavior when an actual stream hit rate falls between the hit rates in two rows of the table. Section 4 shows that the simple model is responsible for correlation error when the cache hierarchy changes from the default. More complicated models might walk cache congruence classes or pages (to model TLB misses), or move, add, or convert instructions to implement specific functions. Adding a few instructions to implement a more complicated model will not impact instruction mix and behavior in most cases. There are many models in the literature that can be investigated as future work [24][29][9][15].

In some cases, we found that additional manipulation of the streams was necessary for correlation of the benchmarks because of the cumulative errors in stream selection. Parameters were added to adjust the basic block

Table 1: L1 and L2 Hit Rates versus Stride

L1 Hit Rate	L2 Hit Rate	Stride
0.0000	0.000	16
0.0000	0.0625	15
0.0000	0.1250	14
0.0000	0.1875	13
0.0000	0.2500	12
0.0000	0.3125	11
0.0000	0.3750	10
0.0000	0.4375	9
0.0000	0.5000	8
0.1250	0.5000	7
0.2500	0.5000	6
0.3750	0.5000	5
0.5000	0.5000	4
0.6250	0.5000	3
0.7500	0.5000	2
0.8750	0.5000	1
1.0000	N/A	0

and overall miss rates of the synthetic data accesses to compensate. A small number of synthesis iterations is usually necessary. Details are given in [2][3].

We superimpose a branch predictability model onto the set of basic blocks that already represent the instruction mix, dependencies and data access patterns of the original workload (Figure 2e). A number of branches in the trace are configured to branch past the next basic block or a number of instructions based on the global branch predictability of the original application. An integer instruction that is not used as a data access counter or a loop counter is converted into an *invert* instruction that operates on a particular register every time it is encountered. If the register is set, the branch jumps past the next basic block. The *invert* mechanism causes a branch to have a predictability of 50% for predictors that use 2-bit saturating counters. Benchmarks like *mgrid* and *applu* have average basic block sizes much longer than other benchmarks. In those cases, parameters are used to adjust the synthetic branch predictability. Details are given in [2][3].

The configured branches, invert instruction, and loop counter must not be skipped over by a taken branch, or loop iterations may not converge, or the branch predictability may be incorrect. Code regions containing these instructions are carefully avoided.

In practice, there are many synthetic benchmarks that more or less satisfy the metrics obtained from the workload characterization and overall application IPC. As mentioned in several places above, the usual course of action is to iterate through synthesis a number of times until the metric deltas are as small as desired. Usually less than ten iterations are needed to obtain reasonably small errors.

3.3. Register Assignment

All architected register usages in the synthetic benchmark are assigned exactly during the register assignment phase. Most ISAs specify dedicated registers that should not be modified without saving and restoring. In practice, not all registers need be used to achieve a good synthesis result. In our experiments, only 20 or so general-purpose registers divided between data access counters and code use are necessary.

Data access streams are pooled according to their stream access characteristics and a register is reserved for each class. All data access counters in the same pool increment the same register, so new stream data are accessed similarly whether there are a lot of counters in the pool and few loop iterations or few in the pool but many iterations. The exact numbers of data access and stream registers assigned for each benchmark are given in [2][3].

For applications with large numbers of stream pools, synthesis consolidates the least frequent pools together until the total number of registers is under the register use limit. A roughly even split between code registers and pool registers improves benchmark *quality*. High quality is defined as a high correspondence between the instructions in the compiled benchmark and the original synthetic C-code instructions. With too few or too many registers available for code use, the compiler may insert stack operations into the binary. The machine characteristics may not suffer from a few stack operations, but for this study we chose to synthesize code without them.

The available code registers are assigned to instruction outputs in a round-robin fashion.

3.4. Code Generation

The code generator takes the representative instructions and the attributes from graph analysis and register assignment and outputs a single module of C-code that contains calls to assembly-language instructions in the target language [2][3]. Figure 1 shows the three targets currently supported. Each instruction in the representative trace maps one-to-one to a single *asm* call in the C-code. Ordinary C-code is emitted for functions not related to the trace, as, for example, to instantiate and initialize data structures and variables.

We emit a C-code *main* header and variable declarations to link output registers to data access variables for the stream pools, the loop counter variable, and the branching variable. Pointers to the correct memory type for each stream pool are declared, and *Malloc* calls for the stream data are generated with size based on the number of loop iterations. Each stream pool register is initialized to point to the head of its *malloced* data structure.

The loop counter register initialization is emitted, and the instructions associated with the original graph walk are emitted as volatile calls to assembly language instructions. The data access counters are emitted as integer additions of its output register value to the associated stride for the stream. The loop counter is emitted as an integer subtraction of one to its output register. The basic blocks are analyzed and code is generated to print out unconnected output registers depending on a switch value. The switch is never set, but the print statements guarantee that no code is eliminated during compilation. Code to free the *malloced* memory is generated, and, finally, a C-code footer is emitted.

Additional detailed synthesis information for the SPEC95, SPEC2000 and STREAM benchmarks can be found in [2] and [3].

4. Synthesis Results

In this section we present the SPEC2000 results for the benchmark synthesis system described in the last section.

4.1. Experimental Setup and Benchmarks

We start with an experimental system that exhibits good synthetic simulation correlation against actual application simulations. Our system is derived from the statistical simulation system HLS [19][20], which we updated with the statistical flow graph to improve correlation [1][12]. SimpleScalar 3.0 [7] was downloaded and *sim-cache* was modified to carry out the workload characterization. The twelve SPECint2000 and fourteen SPECfp2000 *Alpha* binaries were executed in *sim-outorder* on the first reference dataset for the first billion instructions. Single-precision versions of eight STREAM and STREAM2 benchmarks [16] with a ten million-loop limit were also simulated. We use the default SimpleScalar configuration in Table 2, as in [19]. SimpleScalar does not model an L3, but the memory latency estimates a fast L3.

Code generation was enabled and C-code was produced using the synthesis methods of Section 3. The synthetic benchmarks were compiled using *gcc* with optimization level *–O2* and executed to completion in SimpleScalar on an IBM p270 (400 MHz).

4.2. Synthesis Results

The synthetic benchmarks have an execution speed advantage over the original applications. Most simulations of the SPEC2000 synthetics take less than four seconds to execute an average of 325K instructions, compared to about 25K seconds for the original codes. On average, the applications simulate 6000 times slower than the synthetics [3].

The STREAM synthetics must also execute for a large number of instructions (283K on average) in order to represent to be represent the data access patterns of the original codes. As a result, the original codes of 10M dynamic instructions execute only 35 times slower than the synthetic codes. Dynamic executions of at least 300K instructions provide data access convergence and also limit the code overhead of the synthetic to less to 1% of the total dynamic instructions.

Table 2: Default Simulation Configuration, Alpha ISA

Instruction Size (bytes)	4
L1/L2 Line Size (bytes)	32/64
Machine Width	4
Dispatch Window/LSQ/IFQ	16/8/4
Memory System	16K 4-way L1 D, 16K 1-way L1 I, 256K 4-way unified L2
L1/L2/Memory Latency+transfer (cycles)	1/6/34
Functional Units	4 I-ALU, 1 I-MUL/DIV, 4 FP-ALU, 1 FP-MUL/DIV
Branch Predictor	Bimodal 2K table, 3 cycle mispredict penalty

Table 3: Percent Execution Error, Synthetics vs. Applications

Metric	Avg. %Error	Max. %Error
IPC	2.4	8.0 (facerec)
Instruction Frequencies	3.4	7.3 (branches)
Dependency Distances	11.1	34.9 (integers)
Dispatch Occupancies	4.1	8.7 (floats)
Basic Block Sizes	7.2	21.1 (mgrid)
L1 I-cache Miss Rate (>1%)	8.6	22.9 (sixtrack)
L1 D-cache Miss Rate (>1%)	12.3	55.7 (mgrid)
L2 Cache Miss Rate (>15%)	18.4	61.2 (applu)
Branch Predictability	1.5	6.4 (art)

Table 3 compares the execution characteristics of the synthetic benchmarks to those of the SPEC2000 and STREAM codes. The average percent errors for all the metrics are generally less than 15%, with most below 10%, although some of the maximum errors are high. The error in IPC remains low because errors in the metrics offset each other for particular benchmarks, or the absolute values of the metrics are very low and have little effect. As an example, the large percent error for the *mgrid* L1 D-cache miss rate corresponds to a reduction for the synthetic that is offset by a 0.3% increase in I-cache miss rate and a 21.1% decrease in basic block size. As another example, the 22.9% L1 I-cache miss rate error for *sixtrack* is a decrease taken against a miss rate of just 1.1%, so the effect of the error is small. Also, the large increase in L2 cache misses for *applu* is offset by a 31.5% decrease in its L1 D-cache miss rate, to 6.5%. The various errors in Table 3 are broken out for each benchmark in [3].

The overall average IPC error for the synthetic benchmarks is 2.4%, with a maximum error of 8.0% for *facerec*. The error in IPC expresses the average effect of small or offsetting errors among the workload characteristics of the synthetics as described below.

The average error in instruction frequencies over the five classes of instructions for the synthetic benchmarks is 3.4% with a maximum of 7.3% for branches. The basic block size varies per synthetic with an average error of 7.2% and a maximum error of 21.1% for *mgrid*. The errors are caused by variations in the fractions of specific basic block types in the synthetic benchmark with respect to the original workload, which is a direct consequence of selecting a limited number of basic blocks during synthesis. For example, *mgrid* is synthesized with a total of 30 basic blocks made up of only six different unique block types. *Applu* is synthesized with 19 basic blocks but 18 unique block types.

The average I-cache miss rate error is 8.6% for benchmarks with miss rates above 1%. However, the number of synthetic instructions is within 2.8% of the expected number given the I-cache configuration. The errors are due to the process of choosing a small number of basic blocks with specific block sizes to synthesize the

workload. For miss rates close to zero, a number of instructions less than the maximum number that fits in the default cache is typically used, up to the number needed to give an appropriate instruction mix for the benchmark. For the STREAM loops, only one basic block is needed to meet the instruction mix and miss rate requirements. For all synthetic benchmarks there is a small but non-zero miss rate, versus an essentially zero miss rate for some of the applications. This is because the synthetic benchmarks are only executed for about 300K instructions, far fewer than necessary to achieve a very small I-cache miss rate. However, since the miss rates are small, the impact, when combined with the miss penalty, is also small.

The average branch predictability error is 1.9%, with a maximum error for *art* of 6.4%. *Mgrid*, with its large basic block size error, has the third largest error at 4.9%.

For L1 data cache miss rates greater than 1%, the average error is 12.3%. Despite this error, the trends in D-cache miss rates generally correspond with those of the original workloads [3]. There is some variation for smaller miss rates, but, as with many I-cache miss rates, the execution impact is also small.

The unified L2 miss rates have a large average error of 18.4%. The large error is due to the simple streaming memory access model. However, the errors are often mitigated by small L1 miss rates. A good example is *gcc*, which has a 15% L2 miss rate but only a 2.6% L1 miss rate. The 61.2% L2 miss rate error for *applu* is offset by I-cache and L1 D-cache miss rates that are smaller than those of the original workload. *Art* and *ammp* have large L1 miss rates (41% and 44%), but their L2 miss rates are offset by relatively larger I-cache miss rates and smaller branch predictabilities. The main cause of the errors is the fact that the current memory access model focuses on matching the L1 hit rate, and the L2 hit rate is simply predetermined as a consequence. A large L2 miss rate error for *ammp* (46%) is explained by the fact that our small data-footprint synthetic benchmarks have data-TLB miss rates near zero, while the actual *ammp* benchmark has a data-TLB miss rate closer to 13%. As a consequence, the synthetic version does not correlate well when the dispatch window is increased and tends to be optimistic.

The average dependency distances have 11.1% error on average. The largest components of error are the integer dependencies (at 34.1%), caused by the conversion of many integer instructions to data access counters. A data access counter overrides the original function of the integer instruction and causes dependency relationships to change. Another source of error is the movement of dependencies during the search for compatible dependencies. The movement is usually less than one instruction position, as mentioned earlier, but *mgrid* and *applu*, the benchmarks with the largest average block sizes at 100.1 and 93.4, respectively, show significant movement. The branching model also contributes errors to the integer instruction class.

In spite of the dependency distance errors, the average dispatch window occupancies are similar to those of the original benchmarks with an average error of 4.1%.

Table 4: Design Change Percent Error and Percent Relative Error Synthetics vs. Applications

Design Change	Avg. %Error	Avg. %Rel. Err.
Dispatch Window 8, LSQ 4	2.8	2.4
Dispatch Window 32, LSQ 16	3.7	2.1
Dispatch Window 48, LSQ 24	4.9	3.8
Dispatch Window 64, LSQ 32	6.1	5.1
Dispatch Window 96, LSQ 48	8.3	7.5
Dispatch Window 128, LSQ 64	9.0	8.3
Machine Width 2	2.7	1.6
Machine Width 6	2.6	1.1
Machine Width 8	2.6	1.1
Machine Width 10	2.6	1.1
Issue Width 1	1.9	2.3
Issue Width 8	2.7	1.0
Commit Width 1	2.8	2.1
Commit Width 8	2.4	0.2
Instruction Fetch Queue 8	2.6	0.5
Instruction Fetch Queue 16	2.7	0.8
Instruction Fetch Queue 32	3.0	1.1
Caches 0.25x	20.1	19.4
Caches 0.5x	24.8	23.9
Caches 2x	4.1	3.3
Caches 4x	4.7	3.8
L1 I-cache 2x	3.0	1.3
L1 D-cache 2x	3.1	1.0
L1 D-cache Latency 8	9.5	9.7
BP Table 0.25x	2.5	1.1
BP Table 0.5x	2.3	0.3
BP Table 2x	2.3	0.3
BP Table 4x	2.3	0.4

4.3. Using the Synthetic Benchmarks to Assess Design Changes

We now study design changes using the same synthetic benchmarks; that is, we take the benchmarks described in the last section, change the machine parameters in SimpleScalar and re-execute them. Table 4 gives the average IPC prediction error and relative IPC error [12] when executing various design changes on the benchmarks synthesized from the default configuration. A change in machine width implies that the decode width, issue width and commit width all change by the same amount from the base configuration in Table 2. When the caches are increased or decreased by a factor, the number of sets for the L1 I-cache, D-cache and L2 cache are increased or decreased by that factor. Likewise, when the bimodal branch predictor is multiplied by a factor, the table size is multiplied by that factor from the default size. The *L1 D-cache 2x* and *L1 I-cache 2x* specify a doubling of the L1 D-cache (to 256 sets, 64B cache line, 8-way set associativity), and a doubling of the L1 I-cache configuration (to 1024 sets, 64B cache line, 2-way set

associativity). The numbers here do not include *ammp*; as explained in the last section, *ammp* tends to be optimistic when the dispatch window changes because our small data footprint benchmarks do not model data-TLB misses.

Looking at the *Dispatch Window* rows, when the synthetics are executed on configurations close to the default configuration, the average IPC prediction error and average relative errors are below 5%. However, as the configuration becomes less similar to the configuration used to synthesize the benchmarks, the errors increase. One conclusion is that the synthetics are most useful for design studies and validations closer to the synthesis configuration, and that the miniature benchmarks should be resynthesized when the configuration strays farther away - in this case, when the dispatch window rises to four times the default size.

When the errors are small, the changes in IPC for the applications and synthetics are very similar. If the IPC changes are significantly larger than the errors in IPC due to synthesis modeling, the changes using the synthetics should be large enough to trigger additional studies using a detailed cycle-accurate simulator. For early design studies, chip designers are looking for cases in a large design space in which a design change may improve or worsen a design. Example analyses are given in [2][3].

The *Machine Width* results differ from the dispatch window results in that the errors are small regardless of the width change. For the dispatch studies, the absolute change in IPC from the default configuration for both synthetics and applications is greater than 17% for each case (for a dispatch window of 128 the change is over 56%). Likewise, when the width is reduced to 2, the absolute change in IPC is over 23%, which indicates that the low average prediction error and relative error are meaningful. But when the width increases to 6, 8 and 10, the change is never more than 2.3%, which is on the order of the IPC prediction errors of the synthetics versus the applications. However, the fetch queue size did not change from the default and it supplies too little ILP to stress the wider machine width. These configurations therefore cannot test the accuracy of the synthetics.

Similarly, the absolute IPC change from the default IPC for the *Issue Width 8* and *Commit Width 8* rows never gets greater than 1.4%, and likewise for the instruction fetch queue and branch predictability rows, it never gets greater than 1.6%. Simply changing the IFQ size, issue, or commit width without addressing the other pipeline bottlenecks does not improve performance, as expected. Other workloads may be needed to stress the branch predictor.

The remaining studies yield changes in IPC significantly greater than the error of the synthetics versus the applications, except for the *Caches 0.25x* and *Caches 0.5x* studies. The synthetics underestimate performance when the cache is significantly reduced due to capacity misses among the synthetic data access streams.

For the L1 D-cache latency study, the average absolute IPC error is 9.5% and the relative error is 9.7%, significantly less than the 22.1% change in IPC from the default configuration. On further investigation, the error is mostly due to the SPECint synthetics, with an average relative error of 19.9%, versus 4.2% for the others. Despite these errors, the IPC changes in the actual benchmarks executing one billion instructions are still visibly reflected in the synthetic benchmarks that run in seconds [3].

Again, all of these runs use the same miniature benchmarks synthesized from the initial SimpleScalar configuration, not re-synthesized benchmarks.

5. Drawbacks and Discussion

The main drawback of the approach is that the microarchitecture independent workload characteristics, and thus the synthetic workload characteristics, are dependent on the particular compiler technology used. However, since the process is automatic, resynthesis based on workload characterization from new compiler technology is simplified. It also avoids questions of high-level programming style, language, or library routines that plagued the representativeness of the early hand-coded synthetic benchmarks such as Whetstone [10] and Dhrystone [30].

One objection is that the synthetics are comprised of machine-specific assembly calls. However, use of low-level operations are a simple way to achieve true representativeness in a much shorter-running benchmark, and the *asm* calls are easily retargeted to other ISAs that follow the RISC philosophy.

Another drawback is that only features specifically modeled among the workload characteristics appear in the synthetic benchmark. This will be addressed over time as researchers uncover additional features needed to correlate with execution-driven simulation or hardware, although the present state-of-the-art is quite good [12][1]. In the future, synthesis parameters could be used to incorporate or not incorporate features as necessary.

One consequence of the present method is that dataset information is assimilated into the final instruction sequence of the synthetic benchmark. For applications with multiple datasets, a family of synthetic benchmarks must be created. The automatic process makes doing so possible, but future research could seek to find the workload features related to changes in the dataset and model those changes as runtime parameters to the synthetic benchmark.

Ideally, our miniature programs would be benchmark replacements, but the memory and branching models used in their creation introduce significant errors. This makes

them a solution in the "middle" between micro-benchmarks and applications. However, as shown in Section 4, many of the characteristics of the original applications are maintained, and the synthesis approach provides a framework for the investigation of advanced cache access and branching models each independently of the other.

Our benchmarks use a small number of instructions in order to satisfy the I-cache miss rate. This small number causes variations in workload characteristics, including basic block size, with corresponding changes in instruction mix, dependency relationships, and dispatch window occupancies. One solution is to instantiate additional basic blocks using *replication* [32]. Multiple sections of representative synthetic code could be synthesized and concatenated together into a single benchmark. Each section would satisfy the I-cache miss rate, but the number of basic blocks would increase substantially to more closely duplicate the instruction mix. Similarly, multiple sections of synthetic code, and possibly initialization code, could be concatenated together to recreate program phases [22]. Additionally, phases from multiple benchmarks could be consolidated together and configured at runtime through user parameters.

6. Conclusions

In this paper we explore the possibility of automatically synthesizing reduced, miniature benchmarks from the dynamic workload characteristics of executing applications. Two major uses of the miniature benchmarks are for rapid early design studies and pre-silicon performance validation. We discuss the advantages and challenges of automatic synthesis and describe an example system in which the target application's executable is analyzed in detail and sequences of instructions are instantiated as in-line assembly-language instructions inside C-code.

Unlike prior synthesis efforts, we focus on the low-level workload characteristics of the executing binary to create a workload that behaves like a real application executing on the machine. Multiple synthetic benchmarks are necessary if the application is executed on multiple machines, significantly different ISAs, or multiple datasets, but the automatic process minimizes the cost of creating new benchmarks and enables consolidation of multiple representative phases into a single small benchmark. Other benefits include portability to various platforms and flexibility with respect to benchmark modification. Future work includes more accurate memory access models and branching models.

Aknowledgements

The authors would like to thank Lieven Eeckhout, Koen De Bosschere, and the anonymous reviewers for their detailed comments. This research is supported by the National Science Foundation under grant number 0429806, by the IBM Center for Advanced Studies (CAS), and an IBM SUR grant.

References

[1] R. H. Bell, Jr., L. Eeckhout, L. K. John and K. De Bosschere, "Deconstructing and Improving Statistical Simulation in HLS," Workshop on Debunking, Duplicating, and Deconstructing, June 20, 2004.

[2] R. H. Bell, Jr. and L. K. John, "Experiments in Automatic Benchmark Synthesis," Technical Report TR-040817-01, Laboratory for Computer Architecture, University of Texas at Austin, August 17, 2004.

[3] R. H. Bell, Jr. and L. K. John, "Improved Automatic Testcase Synthesis for Performance Model Validation," to appear in the Proceedings of the ACM International Conference on Supercomputing, June 2005.

[4] B. Black and J. P. Shen, "Calibration of Microprocessor Performance Models," IEEE Computer, May 1998, pp. 59-65.

[5] P. Bose and T. M. Conte, "Performance Analysis and Its Impact on Design," IEEE Computer, May 1998, pp. 41-49.

[6] P. Bose, "Architectural Timing Verification and Test for Super-Scalar Processors," Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing, June 1994, pp. 256-265.

[7] D. C. Burger and T. M. Austin, "The SimpleScalar Toolset," Computer Architecture News, 1997.

[8] R. Carl and J. E. Smith, "Modeling Superscalar Processors Via Statistical Simulation," Workshop on Performance Analysis and Its Impact on Design, June 1998.

[9] T. Conte and W. Hwu, "Benchmark Characterization for Experimental System Evaluation," Proceedings of Hawaii International Conference on System Science, 1990, pp. 6-18.

[10] H. J. Curnow and B.A. Wichman, "A Synthetic Benchmark," Computer Journal, vol. 19, No. 1, February 1976, pp, 43-49.

[11] R. Desikan, D. Burger and S. Keckler, "Measuring Experimental Error in Microprocessor Simulation," International Symposium on Computer Architecture, 2001.

[12] L. Eeckhout, R. H. Bell, Jr., B. Stougie, L. K. John and K. De Bosschere, "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies," International Symposium on Computer Architecture, June 2004.

[13] L. Eeckhout, Accurate Statistical Workload Modeling, Ph.D. Thesis, Universiteit Gent, 2003.

[14] C. T. Hsieh and M. Pedram, "Microprocessor power estimation using profile-driven program synthesis," IEEE

Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 17, No. 11, November 1998, pp. 1080-1089.

[15] T. Lafage and A. Seznec, "Choosing Representative Slices of Program Execution for Microarchitecture Simulations," IEEE Workshop on Workload Characterization, 2000.

[16] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," IEEE Technical Committee on Computer Architecture newsletter, December 1995.

[17] L. McVoy, "lmbench: Portable Tools for Performance Analysis," USENIX Technical Conference, Jan. 22-26, 1996, pp. 279-294.

[18] M. Moudgill, J. D. Wellman and J. H. Moreno, "Environment for PowerPC Microarchitecture Exploration," IEEE Micro, May-June 1999, pp. 15-25.

[19] M. Oskin, F. T. Chong and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," Proceedings of the 27th Annual International Symposium on Computer Architecture, June 2000, pp. 71-82.

[20] http://www.cs.washington.edu/homes/oskin/tools.html

[21] M. Sakamoto, L. Brisson, A. Katsuno, A. Inoue and Y. Kimura, "Reverse Tracer: A Software Tool for Generating Realistic Performance Test Programs," IEEE Symposium on High-Performance Computing," 2002.

[22] T. Sherwood, E. Perleman, H. Hamerly and B. Calder, "Automatically characterizing large scale program behavior," Proceedings of the International Conference on Architected Support for Programming Languages and Operating Systems, October 2002.

[23] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja and V. S. Pai, "Challenges in Computer Architecture Evaluation," IEEE Computer, August 2003, pp. 30-36.

[24] E. S. Sorenson and J. K. Flanagan, "Evaluating Synthetic Trace Models Using Locality Surfaces," IEEE International Workshop on Workload Characterization," Nov. 2002, pp. 23-33.

[25] K. Sreenivasan and A.J. Kleinman, "On the Construction of a Representative Synthetic Workload," Communications of the ACM, March 1974, pp.127-133.

[26] http://www.spec.org

[27] S. Surya, P. Bose and J. A. Abraham, "Architectural Performance Verification: PowerPC Processors," Proceedings of the IEEE International Conference on Computer Design, 1999, pp. 344-347.

[28] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le and B. Sinharoy, "POWER4 System Microarchitecture," IBM Journal of Research and Development, January 2002, pp. 5-25.

[29] D. Thiebaut, "On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio," IEEE Transaction on Computers, Vol. 38, No. 7, July 1989, pp. 1012-1026.

[30] R. P. Weiker, "Dhrystone: A Synthetic Systems Programming Benchmark," Communications of the ACM, October 1984, pp. 1013-1030.

[31] J. N. Williams, "The Construction and Use of a General Purpose Synthetic Program for an Interactive Benchmark for on Demand Paged Systems," Communications of the ACM, 1976, pp.459-465.

[32] W. S. Wong and R. J. T. Morris, "Benchmark Synthesis Using the LRU Cache Hit Function," IEEE Transactions on Computers, Vol. 37, No. 6, June 1988, pp. 637-645.

[33] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," The International Symposium on Computer Architecture, June 2002.

[34] J. M. Ludden, et al., "Functional Verification of the Power4 Microprocessor and the Power4 Multiprocessor Systems," IBM J. Res. Dev., Vol. 46, No. 1, January 2002.

[35] J. Ringenberg, C. Pelosi, D. Oehmke and T. Mudge, "Intrinsic Checkpointing: A Methodology for Decreasing Simulation Time Through Binary Modification," International Symposium on Performance and Simulation Systems, March 2005, pp. 78-88.

[36] R. Singhal, et al., "Performance Analysis and Validation of the Intel Pentium4 Processor on 90nm Technology," Intel Tech. J., Vol. 8, No. 1, 2004.

SimPoint 3.0: Faster and More Flexible Program Analysis

Greg Hamerly[‡] Erez Perelman[†] Jeremy Lau[†] Brad Calder[†]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Department of Computer Science, Baylor University

Abstract

This paper describes the new features available in the SimPoint 3.0 release. The release provides two techniques for drastically reducing the run-time of SimPoint: faster searching to find the best clustering, and efficiently clustering large numbers of intervals. SimPoint 3.0 also provides an option to output only the simulation points that represent the majority of execution, which can reduce simulation time without much increase in error. Finally, this release provides support for correctly clustering variable length intervals, taking into consideration the weight of each interval during clustering. This paper describes SimPoint 3.0's new features, how to use them, and points out some common pitfalls.

1 Introduction

Modern computer architecture research requires understanding the cycle level behavior of a processor during the execution of an application. To gain this understanding, researchers typically employ detailed simulators that model each and every cycle. Unfortunately, this level of detail comes at the cost of speed, and simulating the full execution of an industry standard benchmark can take weeks or months to complete, even on the fastest of simulators. To make matters worse, architecture researchers often simulate each benchmark over a variety of architecture configurations and designs to find the set of features that provides the best trade-off between performance, complexity, area, and power. For example, the same program binary, with the exact same input, may be run hundreds or thousands of times to examine how the effectiveness of an architecture changes with cache size. Researchers need techniques to reduce the number of machine-months required to estimate the impact of an architectural modification without introducing an unacceptable amount of error or excessive simulator complexity.

At run-time, programs exhibit repetitive behaviors that change over time. These behavior patterns provide an opportunity to reduce simulation time. By identifying each of the repetitive behaviors and then taking only a single sample of each repeating behavior, we can perform very fast and accurate sampling. All of these representative samples together represent the complete execution of the program. The underlying philosophy of SimPoint [16, 17, 14, 3, 10, 9] is to use a program's behavior patterns to guide sample selection. SimPoint intelligently chooses a very small set of samples called *Simulation Points* that, when simulated and weighed appropriately, provide an accurate picture of the complete execution of the program. Simulating only these carefully chosen simulation points can save hours to days of simulation time with very low error rates. The goal is to run SimPoint once for a binary/input combination, and then use these simulation points over and over again (potentially for thousands of simulations) when performing a design space exploration.

This paper describes the new SimPoint 3.0 release. In Section 2 we present an overview of the SimPoint approach. Section 4 describes the new SimPoint features, and describes how and when to tune these parameters. Section 4 also provides a summary of SimPoint's results and discusses some suggested configurations. Section 5 describes in detail the command line options for SimPoint 3.0. Section 6 discusses the common pitfalls to watch for when using SimPoint, and finally Section 7 summarizes this paper.

The major new features for the SimPoint 3.0 release include:

- **Efficient searching to find the best clustering.** Instead of trying every value, or every Nth value, of k when running the k-means algorithm, we provide a binary search method for choosing k. This reduces the execution time of SimPoint by a factor of 10.
- **Faster SimPoint analysis when processing many intervals.** To speed the execution of SimPoint on very large inputs (100s of thousands to millions of intervals), we sub-sample the set of intervals that will be clustered. After clustering, the intervals not selected for clustering are assigned to phases based on their nearest cluster.
- **Support for Variable Length Intervals.** Prior versions of SimPoint assumed fixed length intervals, where each interval represents the same amount of dynamic execution. For example, in the past, each interval represented 1, 10, or 100 million dynamic instructions. SimPoint 3.0 provides support for clustering variable length intervals, where each interval can represent different amounts of dynamic execution. With variable length intervals, the weight of each interval must be considered during clustering.
- **Reduce the number of simulation points by representing only the majority of executed instructions.** We provide an option to output only the simulation points whose clusters account for the majority of execution. This reduces simulation time, without much increase in error.

2 Background

To ground our discussion in a common vocabulary, the following is a list of definitions we use to describe the analysis performed by SimPoint.

- Interval - A section of continuous execution (a slice in time) of a program. All intervals are assumed to be non-overlapping, so to perform our analysis we break a program's execution into contiguous non-overlapping intervals. The prior versions of SimPoint required all intervals to be the same size, as measured in the number of instructions committed within an interval (e.g., interval sizes of 1, 10, or 100 million instructions were used in [14]). SimPoint 3.0 still supports fixed length intervals, but also provides support for *Variable Length Intervals* (VLI), which allows the intervals to account for different amount of executed instructions as described in [8].
- Phase - A set of intervals within a program's execution with similar behavior. A phase can consist of intervals that are not temporally contiguous, so a phase can re-appear many times throughout execution.
- Similarity - Similarity defines how close the behavior of two intervals are to one another as measured across some set of metrics. Well-formed phases should have intervals with similar behavior across various architecture metrics (e.g. IPC, cache misses, branch misprediction).
- Frequency Vector - Each interval is represented by a frequency vector, which represents the program's execution during that interval. The most commonly used frequency vector is the basic block vector [16], which represents how many times each basic block is executed in an interval. Frequency vectors can also be used to track other code structures [10] such as all branch edges, loops, procedures, registers, or opcodes, as long as tracking usage of the structure provides a signature of the program's behavior.
- Similarity Metric - Similarity between two intervals is calculated by taking the distance between the corresponding frequency vectors from the two intervals. SimPoint determines similarity by calculating the Euclidean distance between the two vectors.
- Phase Classification - Phase classification groups intervals into phases with similar behavior, based on a similarity metric. Phase classifications are specific to a program binary running a particular input (a binary/input pair).

2.1 Similarity Metric - Distance Between Code Signatures

SimPoint represents intervals with frequency vectors. A frequency vector is a one dimensional array, where each element in the array tracks usage of some way to represent the program's behavior. We focus on code structures, but a frequency vector can consist of any structure (e.g., data working sets, data stride access patterns [10]) that may provide a signature of the program's behavior. A frequency vector is collected from each interval. At the beginning of each interval we start with a frequency vector containing all zeros, and as the program executes, we update the current frequency vector as structures are used. A frequency vector could be a list of static basic blocks [16] (called a Basic Block Vector (BBV)), or a list of static loops, procedures, number of registers in the ISA, or opcodes as described in [10].

If we are tracking basic block usage with frequency vectors, we count the number of times each basic block in the program has been entered in the current interval, and we record that count in the frequency vector, weighted by the number of instructions in the basic block. Each element in the frequency vector is a count of how many times the corresponding basic block has been entered in the corresponding interval of execution, multiplied by the number of instructions in that basic block.

We use basic block vectors (BBV) for the results in this paper. The intuition behind this is that the behavior of the program at a given time is directly related to the code executed during that interval [16]. We use the basic block vectors as signatures for each interval of execution: each vector tells us what portions of code are executed, and how frequently those portions of code are executed. By comparing the BBVs of two intervals, we can evaluate the similarity of the two intervals. If two intervals have similar BBVs, then the two intervals spend about the same amount of time in roughly the same code, and therefore we expect the behavior of those two intervals to be similar. Prior work showed that loop and procedure vectors can also be used, where each entry represents the number of times a loop or procedure was executed, performs comparably to basic block vectors [10], while using fewer dimensions.

To compare two frequency vectors, SimPoint 3.0 uses the Euclidean distance, which has been shown to be effective for off-line phase analysis [17, 14]. The Euclidean distance is calculated by viewing each vector as a point in D-dimensional space, and calculating the straight-line distance between the two points.

2.2 Using k-Means for Phase Classification

Clustering divides a set of points into groups, or clusters, such that points within each cluster are similar to one another (by some metric, usually distance), and points in different clusters are different from one another. The k-means algorithm [11] is an efficient and well-known clustering algorithm which we use to quickly and accurately split program behavior into phases. The k in k-means refers to the number of clusters (phases) the algorithm will search for.

The following steps summarize the phase clustering algorithm at a high level. We refer the interested reader to [17] for a more detailed description of each step.

1. Profile the program by dividing the program's execution into contiguous intervals, and record a frequency vector for each interval. Each frequency vector is normalized so that the sum of all the elements equals 1.
2. Reduce the dimensionality of the frequency vector data to a smaller number of dimensions using random linear projection.
3. Run the k-means clustering algorithm on the reduced-dimension data for a set of k values.

I Cache	16k 2-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 20 cycle latency
Main Memory	150 cycle latency
Branch Pred	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mem Disambig	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Func Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Mem	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

4. Choose from among these different clusterings a well-formed clustering that also has a small number of clusters. To compare and evaluate the different clusters formed for different values of k, we use the *Bayesian Information Criterion* (BIC) [13] as a measure of the "goodness of fit" of a clustering to a dataset. We choose the clustering with the smallest k, such that its BIC score is close to the best score that has been seen. Here "close" means it is above some percentage of the range of scores that have been seen. The chosen clustering represents our final grouping of intervals into phases.
5. The final step is to select the simulation points for the chosen clustering. For each cluster (phase), we choose one representative interval that will be simulated in detail to represent the behavior of the whole cluster. By simulating *only* one representative interval per phase we can extrapolate and capture the behavior of the entire program. To choose a representative, SimPoint picks the interval in each cluster that is closest to the *centroid* (center of each cluster). Each simulation point also has an associated weight, which is the fraction of executed instructions in the program its cluster represents.
6. With the weights and the detailed simulation results of each simulation point, we compute a weighted average for the architecture metric of interest (CPI, miss rate, etc.). This weighted average of the simulation points gives an accurate representation of the complete execution of the program/input pair.

3 Methodology

We performed our analysis for the complete set of SPEC2000 programs for multiple inputs using the Alpha binaries on the SimpleScalar website. We collect all of the frequency vector profiles (basic block vectors) using SimpleScalar [4]. To generate our baseline fixed length interval results, all programs were executed from start to completion using SimpleScalar. The baseline microarchitecture model is detailed in Table 1.

To examine the accuracy of our approach we provide results in terms of CPI error and k-means variance. CPI error is the percent error in CPI between using simulation points from SimPoint and the baseline CPI of the complete execution of the program.

The k-means variance is the average squared distance between every vector and its closest center. Lower variances are better. When sub-sampling, we still report the variance based on every vector (not just the sub-sampled ones). The relative k-means variance reported in the experiments is measured on a per-input basis as the ratio of the k-means variance observed for clustering on a sample to the k-means variance observed for clustering on the whole input.

4 SimPoint 3.0 Features

In this section we describe how to reduce the run-time of SimPoint and the number of simulation points without sacrificing accuracy.

4.1 Choosing an Interval Size

When using SimPoint one of the first decisions to make is the interval size. The interval size along with the number of simulation points chosen by SimPoint will determine the simulation time of a binary/input combination. Larger intervals allow more aggregation of profile information, allowing SimPoint to search for large scale repeating behavior. In comparison, smaller intervals allow for more fine-grained representations and searching for smaller scale repeating behavior.

The interval size affects the number of simulation points; with smaller intervals more simulation points are needed than when using larger intervals to represent the same proportion of a program. We showed that using smaller interval sizes (1 million or 10 million) results in high accuracy with reasonable simulation limits [14]. The disadvantage is that with smaller interval sizes warmup becomes more of an issue, but there are efficient techniques to address warmup as discussed in [6, 2]. In comparison, warmup is not really an issue with larger interval sizes, and this may be preferred for some simulation environments [12]. For all of the results in this paper we use an interval size of 10 million instructions.

4.1.1 Support for Variable Length Intervals

Ideally we should align interval boundaries with the code structure of a program. In [7], we examine an algorithm to produce variable length intervals aligned with the procedure call, return and loop transition boundaries found in code. A *Variable Length Interval* (VLI) is represented by a frequency vector as before, but each interval's frequency vector can account for different amounts of the program's execution.

To be able to pick simulation points with these VLIs, we need to change the way we do our SimPoint clustering to include the different weights for these intervals. SimPoint 3.0 supports VLIs, and all of the detailed changes are described in [8]. At a high level the changes focused around the following three parts of the SimPoint algorithm:

- Computing k-means cluster centers – With variable length intervals, we want the k-means cluster centers to represent the centroid of the intervals in the cluster, based on the

weights of each interval. Thus k-means must include the interval weights when calculating the cluster's center. This is an important modification to allow k-means to better model those intervals that represent a larger proportion of the program.

- Choosing the Best Clustering with the BIC – The BIC criterion is the log-likelihood of the clustering of the data, minus a complexity penalty. The likelihood calculation sums a contribution from each interval, so larger intervals should have greater influence, and we modify the calculation to include the weights of the intervals. This modification does not change the BIC calculated for fixed-length intervals.
- Computing cluster centers for choosing the simulation points – Similar to the above, the centroids should be weighted by how much execution each interval in the cluster accounts for.

When using VLIs, the format of the frequency vector files is the same as before. A user can either allow SimPoint to determine the weight of each interval or specify the weights themselves (see the options in Section 5).

4.2 Methods for Reducing the Run-Time of K-Means

Even though SimPoint only needs to be run once per binary/input combination, we still want a fast clustering algorithm that produces accurate simulation points. To address the runtime of SimPoint, we first look at three options that can greatly affect the running time of a single run of k-means. The three options are the number of intervals to cluster, the size (dimension) of the intervals being clustered, and the number of iterations it takes to perform a clustering.

To start we first examine how the number of intervals affects the running time of the SimPoint algorithm. Figure 1 shows the time in seconds for running SimPoint varying the number of intervals (vectors) as we vary the number of clusters (value of k). For this experiment, the interval vectors are randomly generated from uniformly random noise in 15 dimensions.

The results show that as the number of vectors and clusters increases, so does the amount of time required to cluster the data. The first graphs show that for 100,000 vectors and $k = 128$, it took about 3.5 minutes for SimPoint 3.0 to perform the clustering. It is clear that the number of vectors clustered and the value of k both have a large effect on the run-time of SimPoint. The run-time changes linearly with the number of clusters and the number of vectors. Also, we can see that dividing the time by the multiplication of the number of iterations, clusters, and vectors to provide the *time per basic operation* gives improving performance for larger k, due to some new optimizations.

4.2.1 Number of Intervals and Sub-sampling

The k-means algorithm is fast: each iteration has run-time that is linear in the number of clusters, and the dimensionality. However, since k-means is an iterative algorithm, many iterations may be required to reach convergence. We already found in prior work [17], and revisit in Section 4.2.2 that we can reduce the number of dimensions down to 15 and still maintain the SimPoint's clustering accuracy. Therefore, the main influence on execution time for SimPoint 2.0 was the number of intervals.

Program	# Vecs × # B.B.	SP2	SP3-All	SP3-BinS
gcc-166	4692 × 102038	41 min	9 min	3.5 min
crafty	19189 × 16970	577 min	84 min	10.7 min

Table 2: This table shows the running times (in minutes) by SimPoint 2.0 (SP2), SimPoint 3.0 without using binary search (SP3-All) and SimPoint 3.0 using binary search (SP3-BinS). SimPoint is run searching for the best clustering from k=1 to 100, uses 5 random seeds, and projects the vectors to 15 dimensions. The second column shows how many vectors and the size of the vector (static basic blocks) the programs have.

To show this effect, Table 2 shows the SimPoint running time for `gcc-166` and `crafty-ref`, which shows the lower and upper ranges for the number of intervals and basic block vectors seen in SPEC 2000 with an interval size of 10 million instructions. The second and third column shows the number of intervals (vectors) and original number of dimensions for each vector (these are projected down to 15 dimensions). The last three columns show the time it took to execute SimPoint searching for the best clustering from k=1 to 100, with 5 random initializations (seeds) per k. SP2 is the time it took for SimPoint 2.0. The second to last column shows the time it took to run SimPoint 3.0 when searching over all k in the same manner as SimPoint 2.0, and the last column shows clustering time when using our new binary search described in Section 4.4.3. The results show that increasing the number of intervals by 4 times increased the running time of SimPoint around 10 times. The results show that we significantly reduced the running time for SimPoint 3.0, and that combined with the new binary search functionality results in 10x to 50x faster choosing of simulation points over SimPoint 2.0. The results also show that the number of intervals clustered has a large impact on the running time of SimPoint, since it can take many iterations to converge, which is the case for `crafty`.

The effect of the number of intervals on the running time of SimPoint becomes critical when using very small interval sizes like 1 million instructions or smaller, where there can be millions of intervals to cluster. To speed the execution of SimPoint on these very large inputs, we sub-sample the set of intervals that will be clustered, and run k-means on only this sample. We sample the vector dataset using weighted sampling for VLIs, and uniform sampling for fixed-length vectors. The number of desired intervals is specified, and then SimPoint chooses that many intervals (without replacement). The probability of each interval being chosen is proportional to the weight of its interval (the number of dynamically executed instructions it represents).

Sampling is common in clustering for datasets which are too large to fit in main memory [5, 15]. After clustering the dataset sample, we have a set of clusters with centroids. We then make a single pass through the unclustered intervals and assign each to the cluster that has the nearest center (centroid) to that interval. This then represents the final clustering from which the simulation points are chosen. We originally examined

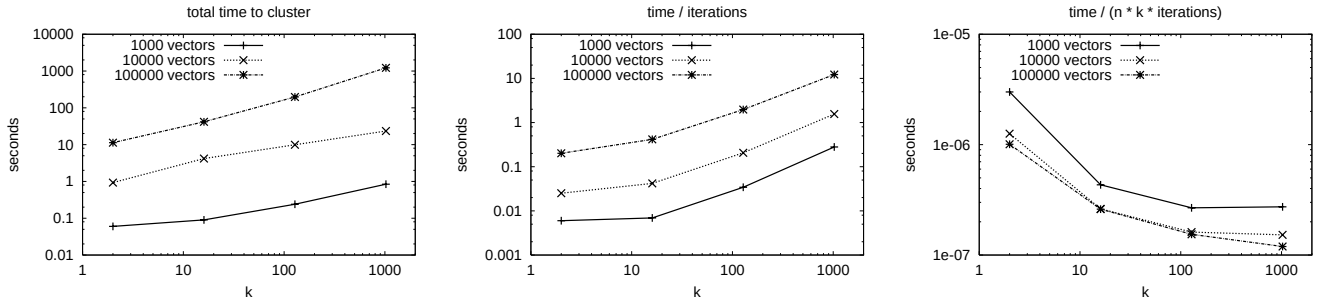


Figure 1: These plots show how varying the number of vectors and clusters affects the amount of time required to cluster with SimPoint 3.0. For this experiment we generated uniformly random data in 15 dimensions. The first plot shows total time, the second plot shows the time normalized by the number of iterations performed, and the third plot shows the time normalized by the number of operations performed. Both the number of vectors and the number of clusters have a linear influence on the run-time of k-means.

using sub-sampling for variable length intervals in [8]. When using VLIs we had millions of intervals, and had to sub-sample 10,000 to 100,000 intervals for the clustering to achieve a reasonable running time for SimPoint, while still providing very accurate simulation points.

The experiments shown in Figure 2 show the effects of sub-sampling across all the SPEC 2000 benchmarks using 10 million interval size, 30 clusters, 15 projected dimensions, and sub-sampling sizes that used 1/8, 1/4, 1/2, and all of the vectors in each program. The first two plots show the effects of sub-sampling on the CPI errors and k-means variance, both of which degrade gracefully when smaller samples are used. The average SPEC INT and SPEC FP average results are shown.

As shown in the second graph of Figure 2, sub-sampling a program can result in k-means finding a slightly less representative clustering, which results in higher k-means variance and higher CPI errors, on average. Even so, when sub-sampling, we found in some cases that it can reduce the k-means variance and/or CPI error (compared to using all the vectors), because sub-sampling can remove unimportant outliers in the dataset that k-means may be trying to fit. It is interesting to note the difference between floating point and integer programs, as shown in the first two plots. It is not surprising that it is easier to achieve lower CPI errors on floating point programs than on integer programs, as the first plot indicates. In addition, the second plot suggests that floating point programs are also easier to cluster, as we can do quite well even with only small samples. The third plot shows the effect of the number of vectors on the running time of SimPoint. This plot shows the time required to cluster all of the benchmark/input combinations and their 3 sub-sampled versions. In addition, we have fit a logarithmic curve with least-squares to the points to give a rough idea of the growth of the run-time. The plot shows that two different datasets with the same number of vectors may require different amounts of time to cluster due to the number of k-means iterations required for the clustering to converge.

4.2.2 Number of Dimensions and Random Projection

Along with the number of vectors, the other most important aspect in the running time of k-means is the number of dimensions used. In [17] we chose to use random linear projection to reduce the dimension of the clustered data for SimPoint, which dramatically reduces computational requirements while retaining the essential similarity information. SimPoint allows the user to define the number of dimensions to project down to. We have found that SimPoint's default of 15 dimensions is adequate for SPEC 2000 applications as shown in [17]. In that earlier work we looked at how much information or structure of frequency vector data is preserved when projecting it down to varying dimensions. We did this by observing how many clusters were present in the low-dimensional version. We noted that at 15 dimensions, we were able to find most of the structure present in the data, but going to even lower dimensions removed too much structure.

To examine random projection, Figure 3 shows the effect of changing the number of projected dimensions on both the CPI error (left) and the run-time of SimPoint (right). For this experiment, we varied the number of projected dimensions from 1 to 100. As the number of dimensions increases, the time to cluster the vectors increases linearly, which is expected. Note that the run-time also increases for very low dimensions, because the points are more "crowded" and as a result k-means requires more iterations to converge.

It is expected that by using too few dimensions, not enough information is retained to accurately cluster the data. This is reflected by the fact that the CPI errors increase rapidly for very low dimensions. However, we can see that at 15 dimensions, the SimPoint default, the CPI error is quite low, and using a higher number of dimensions does not improve it significantly and requires more computation. Using too many dimensions is also a problem in light of the well-known "curse of dimensionality" [1], which implies that as the number of dimensions increase, the number of vectors that would be required to densely populate that space grows exponentially. This means that higher dimensionality makes it more likely that a clustering algorithm will converge to a poor solution. Therefore, it is wise to choose a dimension that is low enough to allow a tight clustering, but not so low that important information is lost.

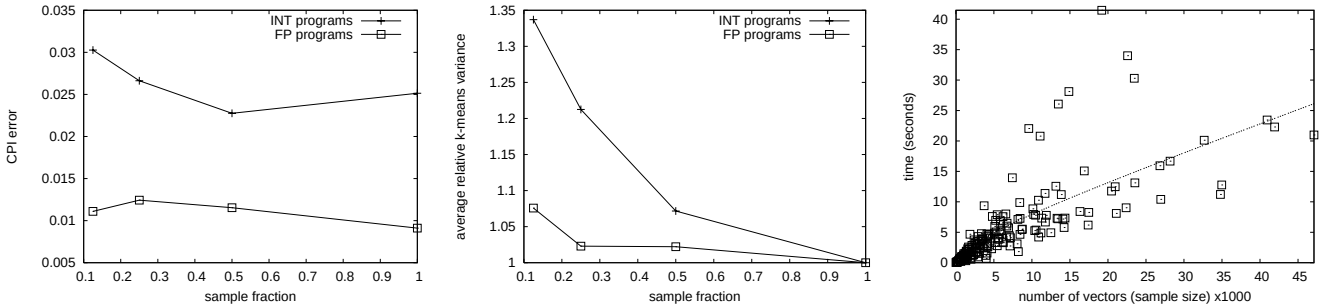


Figure 2: These three plots show how sub-sampling before clustering affects the CPI errors, k-means variance, and the run-time of SimPoint. The first plot shows the average CPI error across the integer and floating-point SPEC benchmarks. The second plot shows the average k-means clustering variance relative to clustering with all the vectors. The last plot shows a scatter plot of the run-time to cluster the full benchmarks and sub-sampled versions, and a logarithmic curve fit with least squares.

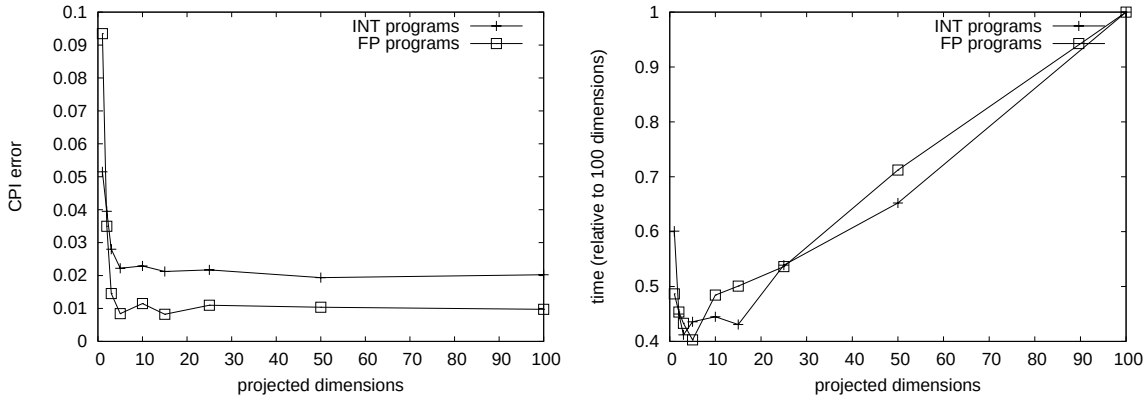


Figure 3: These two plots show the effects of changing the number of projected dimensions when using SimPoint. The default number of projected dimensions SimPoint uses is 15, but here we show results for 1 to 100 dimensions. The left plot shows the average CPI error, and the right plot shows the average time relative to 100 dimensions. Both plots are averaged over all the SPEC 2000 benchmarks, for a fixed $k = 30$ clusters.

4.2.3 Number of Iterations Needed

The final aspect we examine for affecting the running time of the k-means algorithm is the number of iterations it takes for a run to converge.

The k-means algorithm iterates either until it hits a user-specified maximum number of iterations, or until it reaches a point where no further improvement is possible, whichever is less. k-means is guaranteed to converge, and this is determined when the centroids no longer change. In SimPoint, the default limit is 100 iterations, but this can easily be changed. More iterations may be required, especially if the number of intervals is very large compared to the number of clusters. The interaction between the number of intervals and the number of iterations required is the reason for the large SimPoint running time for `crafty-ref` in Table 2.

For our results, we observed that only 1.1% of all runs on all SPEC 2000 benchmarks reach the limit of 100 iterations. This experiment was with 10-million instruction intervals, k=30, 15 dimensions, and with 10 random (seeds) initializations (runs) of k-means. Figure 4 shows the number of iterations required for all runs in this experiment. Out of all of the SPEC program and input combinations run, only `crafty-ref`, `gzip-program`, `perlbmk-splitmail` had runs that had not converged by 100 iterations. The longest-running clusterings for these programs reached convergence in 160, 126, and 101 iterations, respectively.

4.3 MaxK and Controlling the Number of Simulation Points

The number of simulation points that SimPoint chooses has a direct effect on the simulation time that will be required for those points. The maximum number of clusters, `MaxK`, along with the interval size as discussed in Section 4.1, represents the maximum amount of simulation time that will be needed. When fixed length intervals are used, $MaxK * interval\_size$ puts a limit on the instructions simulated.

SimPoint enables users to trade off simulation time with accuracy. Researchers in architecture tend to want to keep simulation time to below a fixed number of instructions (e.g., 300 million) for a run. If this is desirable, we find that an interval

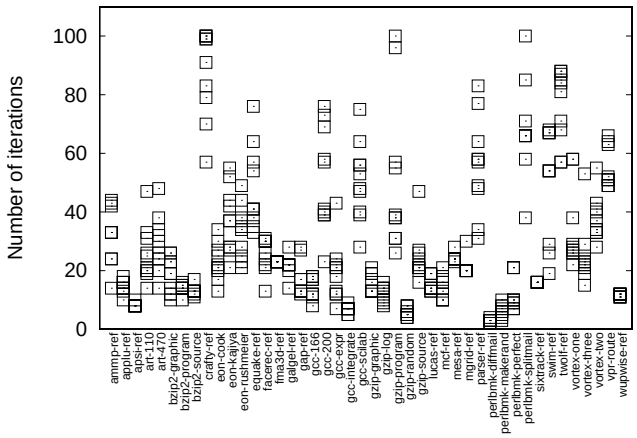


Figure 4: This plot shows the number of iterations required for 10 randomized initializations of each benchmark, with 10-million interval vectors, k = 30, and 15 dimensions. Note that only three program/inputs had a total of 5 runs that required more than the default limit of 100 iterations, and these all converge within 160 iterations or less.

size of 10M with `MaxK=30` provides very good accuracy (as we show in this paper) with reasonable simulation time (below 300 million and around 220 million instructions on average). If even more accuracy is desired, then decreasing the interval size to 1 million and setting `MaxK=300` or `MaxK` equal to the square root of the total number of intervals: $\sqrt{n}$ performs well. Empirically we discovered that as the granularity becomes finer, the number of phases discovered increases at a sub-linear rate. The upper bound defined by this heuristic works well for the SPEC benchmarks.

Finally, if the only thing that matters to a user is accuracy, then if SimPoint chooses a number of clusters that is close to the maximum allowed, then it is possible that the maximum is too small. If this is the case and more simulation time is acceptable, it is better to double the maximum k and re-run the SimPoint analysis.

4.3.1 Choosing Simulation Points to Represent the Top Percent of Execution

One advantage to using SimPoint analysis is that each simulation point has an associated weight, which tells how much of the original program's execution is represented by the cluster that simulation point represents. The simulation points can then be ranked in order of importance. If simulation time is too costly, a user may not want to simulate simulation points that have very small weights. SimPoint 3.0 allows the user to specify this explicitly with the `-coveragePct p` option. When this option is specified, the value of `p` sets a threshold for how much of the execution should be represented by the simulation points that are reported in an extra set of files for the simulation points and weights. The default is `p = 1.0`: that the entire execution should be represented.

For example, if `p = 0.98` and the user has specified `-saveSimpoints` and `-saveWeights`, then SimPoint will report simulation points and associated weights for *all* the non-empty clusters in two files, and *also* for the largest clusters which make up at least 98% of the program's weight. Using his reduced-coverage set of simulation points can potentially save a lot of simulation time if there are many simulation points with very small weights without severely affecting the accuracy of the analysis.

Figure 5 shows the effect of varying the percentage of coverage that SimPoint reports. These experiments use binary search with `MaxK=30`, 15 dimensions, and 5 random seeds. The left graph shows the CPI error and the right shows the number of simulation points chosen when only representing the top 95%, 98%, 99% and 100% of execution. The three bars show the maximum value, the second highest value (max-1), and the average. The results show that when the coverage is reduced from 100%, the average number of simulation points decreases, which reduces the simulation time required, but this is at the expense of the CPI error, which goes up on average. For example, comparing 100% coverage to 95%, the average number of simulation points is reduced from about 22 to about 16, which is a reduction of about 36% in required simulation time for fixed-length vectors. At the same time, the average CPI error increases from 1.5% to 2.8%. Depending on the user's goal, a practitioner can use these types of results to decide on the appropriate tradeoff between simulation time and accuracy. Out of all of the SPEC binary/input pairs there was one combination (represented by the maximum) that had a bad error rate for 95% and 98%. This was `ammp-ref`, and the reason was that a simulation point was removed that had a small weight (1-2% of the executed instructions) but its behavior was significantly different enough to effect the estimated CPI.

Note, when using simulation points for an architecture design space exploration, the CPI error compared to the baseline is not as important as making sure that this error is consistent between the different architectures being examined. What is important is that a consistent relative error is seen across the design space exploration, and SimPoint has this consistent bias as shown in [14]. Ignoring a few simulation points that account for only a tiny fraction of execution will create a consistent bias across the different architecture runs when compared to complete simulation. Therefore, this can be acceptable technique for reducing simulation time, especially when performing large design space exploration trade-offs.

4.4 Searching for the Smallest k with Good Clustering

As described above, we suggest setting `MaxK` as appropriate for the maximum amount of simulation time a user will tolerate for a given run. We then use three techniques to search over the possible values of k, which we describe now. The goal is to try to pick a k to reduce the number of clusters (simulation points), which reduces simulation time by reducing the number of points needed to represent the program's execution.

4.4.1 Setting the BIC Percentage

As we examine several clusterings and values of k, we need to have a method for choosing the best clustering. The *Bayesian*

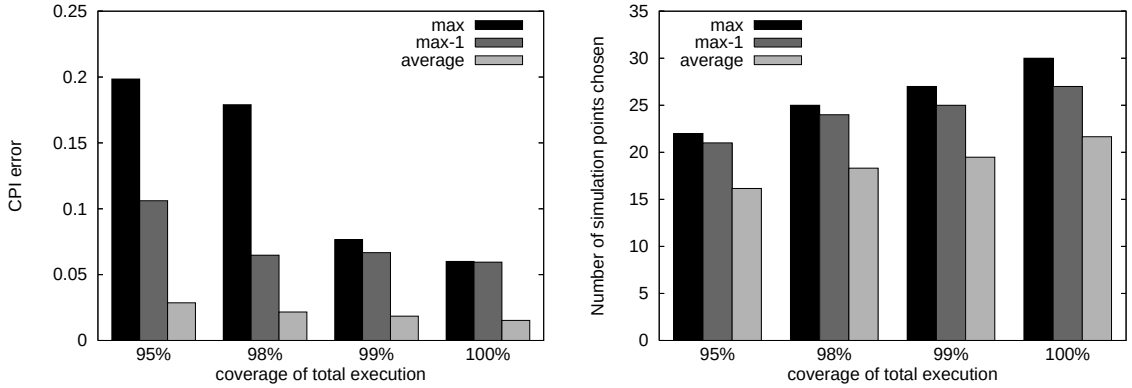


Figure 5: These plots show the CPI error and number of simulation points picked across different percent coverages of execution. For 100% coverage, all simulation points are used, but for less than 100%, simulation points from the smallest clusters are discarded, keeping enough simulation points to represent the desired coverage. Bars labeled "max-1" show the second largest value observed.

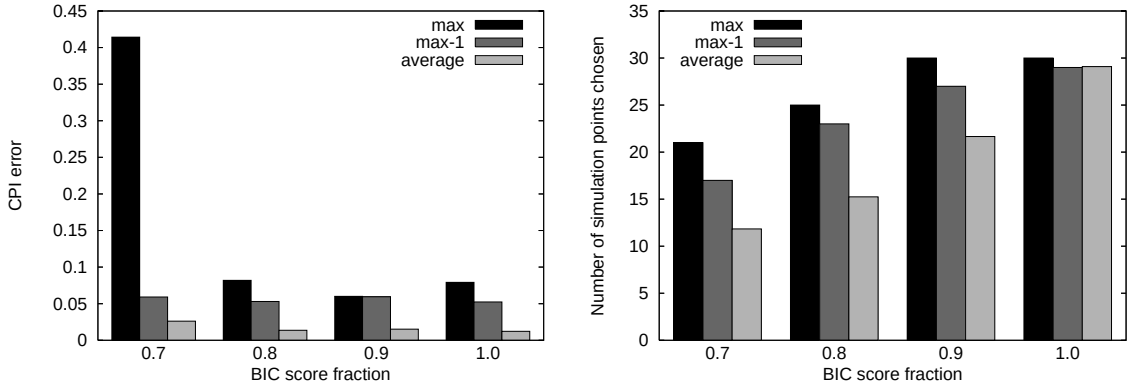


Figure 6: These plots show how the CPI error and number of simulation points chosen is affected by varying the BIC threshold. Bars labeled "max-1" show the second largest value observed.

Information Criterion (BIC) [13] gives a score of the goodness of the clustering of a set of data. These BIC scores can then be used to compare different clusterings of the same data. The BIC score is a penalized likelihood of the clustering of the vectors, and can be considered the approximation of a probability. However, the BIC score often increases as the number of clusters increase. Thus choosing the clustering with the highest BIC score can lead to often selecting the clustering with the most clusters. Therefore, we look at the range of BIC scores, and select the score which attains some high percentage of this range. The SimPoint default BIC threshold is 0.9. When the BIC rises and then levels off, this method chooses a clustering with the fewest clusters that is near the maximum value. Choosing a lower BIC percent would prefer fewer clusters, but at the risk of less accurate simulation.

Figure 6 shows the effect of changing the BIC threshold on both the CPI error (left) and the number of simulation points chosen (right). These experiments are for using binary search with `MaxK=30`, 15 dimensions, and 5 random seeds. BIC thresholds of 70%, 80%, 90% and 100% are examined. As the BIC threshold decreases, the average number of simulation points decreases, and similarly the average CPI error increases. At the 70% BIC threshold, `perlbmk-splitmail` has the maximum CPI error in the SPEC suite. This is due to clustering that was picked at that threshold which has only 9 clusters. This anomaly is an artifact of the looser threshold, and better BIC scores point to better clusterings and better error rates, which is why we recommend to the BIC threshold to be set at 90%.

4.4.2 Varying the Number of Random Seeds, and k-means initialization

The k-means clustering algorithm is essentially a hill-climbing algorithm, which starts from a randomized initialization, which requires a random seed. Because of this, running k-means multiple times can produce very different results depending on the initializations. Sometimes this means k-means can converge to a locally-good solution that is poor compared to the best clustering on the same data for that number of clusters. Therefore conventional wisdom suggests that it is good to run k-means several times using a different randomized starting point each time, and

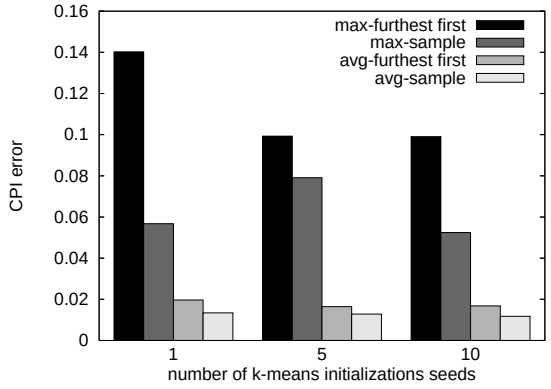


Figure 7: This plot shows the average and maximum CPI errors for both sampling and furthest-first k-means initializations, and using 1, 5, or 10 different random seeds. These results are over the SPEC 2000 benchmark suite for 10-million instruction vectors, 15 dimensions, and k = 30.

take the best clustering observed, based on the k-means variance or the BIC. SimPoint has the functionality to do this, using different random seeds to initialize k-means each time. Based on our experience, we have found that using 5 random seeds works well.

Figure 7 shows the effect on CPI error of using two different k-means initialization methods (furthest-first and sampling) along with different numbers of initial k-means seeds. These experiments are for using binary search with `MaxK=30`, 15 dimensions, and a BIC threshold of .9. When multiple seeds are used, SimPoint runs k-means multiple times with different starting conditions and takes the best result.

Based on these results we see that sampling outperforms furthest-first k-means initialization. This can be attributed to the data we are clustering, which has a large number of anomaly points. The furthest-first method is likely to pick those anomaly points as initial centers since they are the furthest points apart. The sampling method randomly picks points, which on average does better than the furthest-first method. It is also important to try multiple seed initializations in order to avoid a locally minimal solution. The results in Figure 7 shows that 5 seed initializations should be sufficient in finding good clusterings.

4.4.3 Binary Search for Picking k

SimPoint 3.0 makes it much faster to find the best clustering and simulation points for a program trace over earlier versions. Since the BIC score generally increases as k increases, SimPoint 3.0 uses this to perform a binary search for the best k. For example, if the maximum k desired is 100, with earlier versions of SimPoint one might search in increments of 5: $k = 5, 10, 15, \ldots, 90, 100$, requiring 20 clusterings. With the binary search method, we can ignore large parts of the set of possible k values and examine only about 7 clusterings.

The binary search method first clusters 3 times: at $k = 1$, $k =$ max k, and $k = (\text{max k} + 1)/2$. It then proceeds to divide the search space and cluster again based on the BIC scores observed for each clustering. The binary search may stop early if the window of k values is relatively small compared to the maximum k value. Thus the binary search method requires the user only to specify the maximum k value, and performs at most $\log(\text{max k})$ clusterings.

Figure 8 shows the comparison between the new binary search method for choosing the best clustering, and the old method used in SimPoint 2.0, which searched over a large number of k values in the same range. The top graph shows the CPI error for each program, and the bottom graph shows the number of simulation points (clusters) chosen. These experiments are for using binary search with `MaxK=30`, 15 dimensions, 5 random seeds, and a BIC threshold of .9. SimPoint 2.0 performs slightly better than the binary search method, since it searches exhaustively through all k values for `MaxK=30`. Using the binary search, it possible that it will not find as small of clustering as the exhaustive search. This is shown in the bottom graph of Figure 8, where the exhaustive search picked 19 simulation points on average, and binary search chose 22 simulation points on average. In terms of CPI error rates, the average is about the same across the SPEC programs between exhaustive and binary search.

5 SimPoint 3.0 Command Line Options

Clustering and projection options:

- `-k regex`: This specifies which values of k should be searched. The regular expression is

```
regex := "search" | R(,R)*
    R := k | start:end | start:step:end
```

  `Search` means that SimPoint should search using a binary search between 1 and the user-specified `maxK`. The `-maxK` option must be set for `search`. Searching is the default behavior. If the user chooses not to use `search`, they may specify one or more comma-separated ranges of positive integers for k. The argument `k` specifies a single k value, the range `start:end` indicates that all integers from `start` to `end` (inclusive) should be used, and the range `start:step:end` indicates that SimPoint should use values starting at `start` and stepping by interval `step` until reaching `end`. Here is an example of specifying specific values with the regular expression: `-k 4:6,10,12,30:15:75`, which represents searching the k values 4,5,6,10,12,30,45,60,75.

- `-maxK k`: When using the "search" clustering method (see `-k` option), this command line option must be provided. It specifies the maximum number of clusters that SimPoint should use.

- `-fixedLength "on" | "off"`: Specifies whether the frequency vectors that are loaded should be treated as fixed-length vectors (which means having equal weights), or VLI vectors. The default is on. When off, if no weights are loaded (using `-loadVectorWeights`) then the weight of

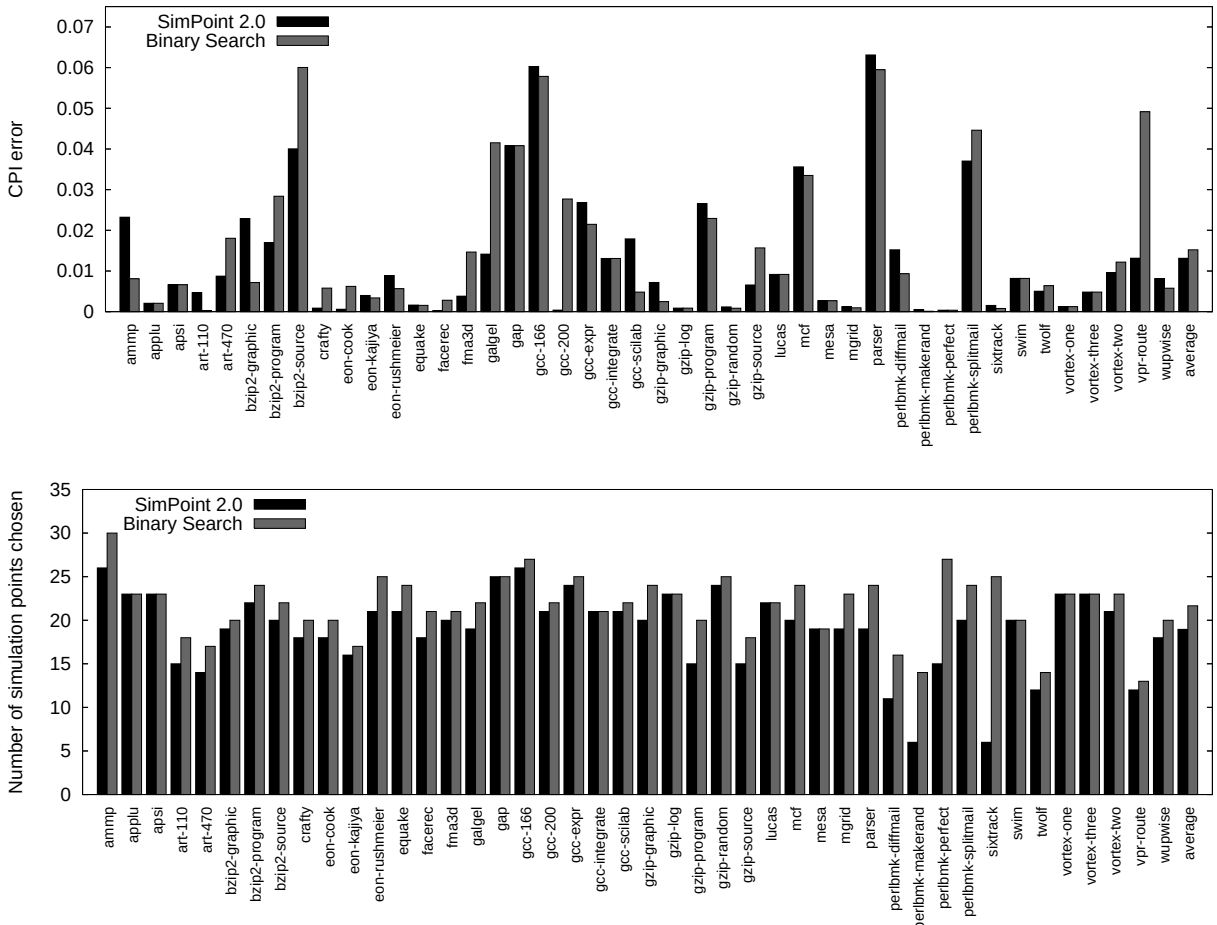


Figure 8: These plots show the CPI error and number of simulation points chosen for two different ways of searching for the best clustering. The first method, which was used in SimPoint 2.0, is searching for all k between 1 and 30, and choosing the smallest clustering that achieves the BIC threshold. The second method is the binary search for `MaxK=30`*, which examines at most 5 clusterings.*

each interval is determined by summing up all the frequency counts in the vector for an interval and dividing this by the total frequency count over all intervals.

- `-bicThreshold t`: SimPoint finds the highest and lowest BIC scores for all examined clusterings, and then chooses the one with the smallest k which has a BIC score greater than `t*(max_score-min_score)+min_score`. The default value for `t` is 0.9.
- `-dim d | "noProject"`: `d` is the number of dimensions down to which SimPoint should randomly project the un-projected frequency vectors. If the string "noProject" is instead given, then no projection will be done on the data. If the `-dim` option is not specified at all, then a default is 15 dimensions is used. This option does not apply when loading data from a pre-projected vector file using options `-loadProjData` or `-loadProjDataBinary`.
- `-seedproj seed`: The random number seed for random projection. The default is 2042712918. This can be changed to any integer for different random projections.
- `-initkm "samp" | "ff"`: The type of k-means initialization (sampling or furthest-first). The default is `"samp"`. Sampling chooses k different vectors from the program at random as the initial cluster centers. Furthest-first chooses a random vector as the first cluster center, then repeats the following $k-1$ times: find the closest center to each vector, and choose as the next new center the vector which is furthest from its closest center.
- `-seedkm seed`: The random number seed for k-means initialization (see `-initkm`). The default is 493575226. This can be changed to any integer obtain different k-means initializations, and using the same seed across runs will provide reproducible initializations.
- `-numInitSeeds n`: The number of random initializations to try for clustering each k. For each k, the dataset is clustered `num` times using different k-means initializations (the k-means initialization seed is changed for each initial-

ization). Of all the num runs, only the best (the one with the highest BIC score) is kept. The default is 5.

- `-iters n | "off"`: The maximum number of k-means iterations per clustering. The default is 100, but the algorithm often converges and stops much earlier. If "off" is instead chosen, then k-means will terminate once it has converged. In running all of the SPEC programs with all of their inputs using the default parameters to SimPoint 3.0 only 1.1% of all runs did not converge by 100 iterations. Clearly, the default number of iterations is usually sufficient, but can be increased if SimPoint is often reaching the limit.
- `-verbose level`: The amount of output that SimPoint should produce. The argument `level` is a non-negative integer, where larger values indicate more output. The default is 0, which is the minimum amount of output.

Sampling options:

- `-sampleSize n`: The *number* of frequency vectors (intervals) to randomly sample before clustering with k-means. Using a smaller number of vectors allows k-means to run faster, at a small cost in accuracy. The vectors are sampled without replacement, so each vector can be sampled only once. For VLI vectors, vectors are chosen with probability proportional to how much of the execution they represent. The default is to use all vectors for clustering.
- `-seedsample n`: The random number seed for vector sampling. The default is 385089224. This can be changed to any integer for different samples.

Load options:

- `-loadFVFile file`: Specifies an unprojected sparse-format frequency vector (FV) file of all of the intervals. Either this argument, `-loadProjData`, or `-loadProjDataBinary` must always be present to provide SimPoint with the frequency vectors that should be analyzed.
- `-numFVs n`, `-FVDim n`: These two options together specify the number of frequency vectors and maximum number of dimensions in the unprojected frequency vector file so the file doesn't need to be parsed twice (both options must be used together).
- `-loadProjData file`: Specifies an already-projected *text* vector file (saved with `-saveProjData`). When loaded this way, SimPoint does not use random projection or otherwise change the vectors.
- `-loadProjDataBinary file`:
  Specifies an already-projected *binary* vector file (saved with `-saveProjDataBinary`). This is the binary-format version of `-loadProjData`. This option provides the fastest way to load a dataset.
- `-inputVectorsGzipped`: When present, this option specifies that the input vectors given by `-loadFVFile`, `-loadProjData`, or `-loadProjDataBinary` are compressed with gzip compression, and should be decompressed while reading.
- `-loadInitCtrs file`: Specifies initial centers for clustering (rather than allowing SimPoint to choose the initial centers with furthest-first or sampling). These centers are points in the same dimension as the projected frequency vectors, but they are not necessarily actual frequency vectors. This option is incompatible with using multiple values of k; only the k corresponding to the number of centers in the given file will be run. This is useful if you want to specify the exact starting centers to perform a clustering.
- `-loadInitLabels file`: Specifies the labels that will be used to form initial clusters (rather than allowing SimPoint to choose with furthest-first or sampling). Like `-loadInitCtrs`, this option is incompatible with multiple k values. This is used if you want to specify the initial starting clusters to perform a clustering based on a set of labels. In doing this, the new starting centers will be formed from these labels and clustering iterations will proceed from there.
- `-loadProjMatrix file`: Specifies a *text* projection matrix to use to project the unprojected frequency vector file (saved from a previous run with `-saveProjMatrix`), rather than allowing SimPoint to choose a random projection matrix. This option also allows users to specify their own projection matrix.
- `-loadProjMatrixBinary file`: Specifies a *binary* projection matrix to use to project the unprojected frequency vector file. This is the binary version of `-loadProjMatrix`.
- `-loadVectorWeights file`: Specifies a text file that contains the weights that should be applied to the frequency vectors. The weights should all be non-negative, and their sum should be positive.

Save options:

- `-saveSimpoints file`: Saves a file of the vectors chosen as Simulation Points and their corresponding cluster numbers. Frequency vectors are numbered starting at 0, which means the first vector in the execution has an index of 0. *Note* that earlier versions of SimPoint started numbering vectors from 1.
- `-saveSimpointWeights file`: Saves a file containing a weight for each Simulation Point, and its corresponding cluster number. The weight is the proportion of the program's execution that the Simulation Point represents.
- `-saveVectorWeights file`: Saves a file with a weight for each frequency vector as computed by SimPoint. The weight of a vector is the proportion that vector represents of the all of the vectors provided. When using VLIs (and the option `-fixedLength off`, this is calculated for a vector by taking the total value of all of the entries in a vector divided by the total value of all of the entries in all vectors. The weights are also stored in projected vector files saved with `-saveProjData` and `-saveProjDataBinary`, so this option is not necessary for just saving and loading projected data.

- `-saveAll`: When this option is `not` specified, SimPoint only saves specified outputs for the best clustering found (according to the BIC threshold). When this option is specified, SimPoint will save the specified outputs for all k values clustered. This option only affects saving labels, simulation point weights, simulation points, initial centers, and final centers.
- `-coveragePct p`: This option tells SimPoint to save additional simulation points and weights that belong to the largest clusters that together make up at least `p` proportion of the vector weights for the entire program. The range of `p` is between 0 and 1; the default is 1. For example, .98 means to output the smallest number of simulation points to account for at least 98% of execution (vectors). This option only affects the saving of simulation points and simulation point weights. The simulation points and associated weights for all clusters will also be saved.
- `-saveProjData file`: Specifies the file in which to save a text version of the projected frequency vectors to enable faster loading later. See `-loadProjData`.
- `-saveProjDataBinary file`: Specifies the file in which to save a binary version of the projected frequency vectors to enable faster loading later. See `-loadProjDataBinary`.
- `-saveProjMatrix file`: Specifies the file in which to save a text version of the projection matrix so it may be reused. See `-loadProjMatrix`.
- `-saveProjMatrixBinary file`: Specifies the file in which to save a binary version of the projection matrix so it may be re-used. See `-loadProjMatrixBinary`.
- `-saveInitCtrs file`: Specifies the file in which to save the initial cluster centers.
- `-saveFinalCtrs file`: Specifies the file in which to save the final cluster centers found by k-means.
- `-saveLabels file`: Specifies the file in which to save the final label and distance from cluster center for each clustered vector.

Table 3 shows all of the default and required options for running SimPoint. The two required parameters for every run of SimPoint are providing the frequency vectors and the setting of $MaxK$ either using the `-k` option or `-maxK` option.

Option	Default Value
-k	"search"
-initkm	"samp"
-numInitSeeds	5
-bicThreshold	0.9
-fixedLength	"on"
-dim	15
-iters	100
-sampleSize	no sub-sampling
-coveragePct	1 (100%)

Table 3: This table gives the standard options that are used with SimPoint and their default values. For every run of SimPoint, the frequency vectors must be provided as an unprojected frequency vector file, or a pre-projected data file given via `-loadProjData` *or* `-loadProjDataBinary`*. When using the* `-k "search"` *method,* `-maxK` *must always be provided.*

6 Common Pitfalls

There are a few important potential pitfalls worth addressing to ensure accurate use of SimPoint's simulation points.

Setting MaxK Appropriately – MaxK must be set based on the interval size used and the maximum number of instructions you are willing to simulate as described in Section 4.3.

The maximum number of clusters and the interval size represent the maximum amount of simulation time needed for the simulation points selected by SimPoint. Finding good simulation points with SimPoint requires recognizing the tradeoff between accuracy and simulation time. If a user wants to place a low limit on the number of clusters to limit simulation time, SimPoint can still provide accurate results, but some intervals with differing behaviors may be grouped together as a result. In such cases it may be advantageous to increase $MaxK$ and with that use the option `-coveragePct` with a value less than 1 (e.g. .98). This can allow different behaviors to be grouped into more clusters, but the final set of simulation points can be smaller since only the most dominant behaviors will be chosen for simulation points.

Off by One Interval Errors – SimPoint 3.0 starts counting intervals and cluster IDs at 0. These are the counts and IDs written to a file by `-saveSimpoints`, where SimPoint indicates which intervals have been selected as simulation points and their respective cluster IDs. A common mistake may be to assume that SimPoint 3.0, like previous versions of SimPoint, counts intervals starting from 1, instead of 0. Just remember that the first interval of execution and the first cluster in SimPoint 3.0 is number 0.

Reproducible Tracking of Intervals and Using Simulation Points – It is very important to have a reproducible simulation environment for (a) creating interval vectors, and (b) using the simulation points during simulation. If the instruction counts are not stable between runs, then selection of intervals can be skewed, resulting in additional error.

SimPoint provides the interval number for each simulation point. Interval numbers are zero-based, and are relative to the start of execution, not to the previous simulation point. So for fixed-length intervals, to get the instruction count at the start of a simulation point, just multiply the interval number by the interval size, but watch out for Interval Drift described later. For example, interval number 15 with an interval size of 10 million instructions means that the simulation point starts when 150 million (15*10M) correct path instructions have been fetched. Detailed simulation of this simulation point would occur from instruction 150 million until just before 160 million.

One way to get more reproducible results is to use the first instruction program counter (Start PC) that occurs at the start of each interval of execution, instead of relying on instruction count. The same program counter can reappear many times, so

it is also necessary to keep track of how many times a program counter value must appear to indicate the start of an interval of execution. For example, if a simulation point is triggered when PC 0x12000340 is executed the 1000th time. Then detailed simulation starts after that PC is seen 1000 times, and simulation occurs for the length of the interval. For this to work, the user needs to profile PCs in parallel with the frequency vector profile, and record the first PC seen for each interval along with the number of times that PC has executed up to that point in the execution. SimPoint provides the interval chosen for a simulation point, and this data can easily be mapped to this PC profile to determine the start PC and the Nth occurrence of it where simulation should start.

It is highly recommended that you use the simulation point Start PCs for performing simulations. There are two reasons for this. The first reason deals with making sure you calculate the instructions during fast-forwarding exactly the same as when the simulation points were gathered. The second reason is that there can be slight variations in execution count between different runs of the same binary/input due to subtle changes in the simulation environment. Both of these are discussed in more detail later in this section.

Interval "Drift" – When creating intervals, the problem may occur that the counts inside an interval might be just slightly larger than the interval size. Over time these counts can add up, so that if you were to try to find a particular fixed length interval in a simulation environment different from where the intervals were generated, you might be off by a few intervals.

For example, this can occur when forming fixed length intervals of X instructions. After X instructions execute the interval should be created, but since this boundary may occur in the middle of a basic block, an additional Y instructions are included into the interval to complete the basic block. Even though Y may be extremely small, it will accumulate over many thousands of intervals and cause a slow "drift" in the interval endpoints in terms of instruction count.

This is mainly a problem if you use executed instructions to determine the starting location for a simulation point. If you have drift in your intervals, to calculate the starting instruction count, you cannot just multiply the simulation point by the fixed length interval size as described above, since the interval lengths are not exactly the same. This can result in simulating the wrong set of instructions for the simulation point. When using the instruction count for the start of the simulation point, you need to keep track of the total instruction count for each interval if you have interval drift. You can then calculate the instruction count starting location for a simulation point by summing up the exact instruction counts for all of the intervals up to the interval chosen as the simulation point.

Accurate Instruction Counts (No-ops) – It is important to count instructions exactly the same for the frequency vector profiles as for the detailed simulation, otherwise they will diverge. Note that the simulation points on the SimPoint website include only correct path instructions and the instruction counts include no-ops. Therefore, to reach these simulation points in a simulator, *every* committed (correct path) instruction (including no-ops) must be counted.

System Call Effects – Some users have reported system call effects when running the same simulation points under slightly different OS configurations on a cluster of machines. This can result in slightly more or fewer instructions being executed to get to the same point in the program's execution, and if the number of instructions executed is used to find the simulation point, this may lead to variations in the results. To avoid this, we suggest using the Start PC and Execution Count for each simulation point as described above. Another way to avoid variations in startup is to use checkpointing [2].

Calculating Weighted IPC – For IPC (instructions/cycle) we cannot just apply the weights directly as is done for CPI. Instead we must convert all the simulated samples to CPI, compute the weighted average of CPI, and then and then convert the result back to IPC.

Calculating Weighted Miss Rates – To compute an overall miss rate (e.g. cache miss rate), first we must calculate both the weighted average of the number of cache accesses, and the weighted average of the number of cache misses. Dividing the second number by the first gives the estimated cache miss rate. In general, care must be taken when dealing with any ratio because both the numerator and the denominator must be averaged separately and *then* divided.

Number of intervals – There should be a sufficient number of intervals for the clustering algorithm to choose from. A good rule of thumb is to make sure to use at least 1,000 intervals in order for the clustering algorithm to be able to find a good partition of the intervals. If there are too few intervals, one can decrease the interval size to obtain more intervals for clustering.

Using SimPoint 2.0 with VLIs – As described in Section 4.1.1, SimPoint 2.0 assumes fixed-length intervals, and should not be used if the vectors to be clustered are variable length. The problem with using VLIs with SimPoint 2.0 is that the data will be clustered with a uniform weight distribution across all intervals, which is not correct for representing the execution properly. This means that the centroids may not be representative of the program's execution in a cluster. This can result in large error rates, since a vector that is not representative of the majority of the cluster could be chosen as the simulation point.

Wanting Variable Length, but not asking for it – If you want variable length weighting for each interval then you need to use the `-fixedLength off` option. You may need to also use `-loadVectorWeights` if your vector weights cannot be automatically calculated from the vector's values.

7 Summary

Modern computer architecture research depends on understanding the cycle level behavior of a processor running an application, and gaining this understanding can be done efficiently by judiciously applying detailed cycle level simulation to only a few simulation points. The level of detail provided by cycle level simulation comes at the cost of simulation speed, but by

targeting only one or a few carefully chosen samples for each of the small number of behaviors found in real programs, this cost can be reduced to a reasonable level.

The main idea behind SimPoint is the realization that programs typically only exhibit a few unique behaviors which are interleaved with one another through time. By finding these behaviors and then determining the relative importance of each one, we can maintain both a high level picture of the program's execution and at the same time quantify the cycle level interaction between the application and the architecture. The key to being able to find these phases in a efficient and robust manner is the development of a metric that can capture the underlying shifts in a program's execution that result in the changes in observed behavior. SimPoint uses frequency vectors to calculate code similarity to cluster a program's execution into phases.

SimPoint 3.0 automates the process of picking simulation points using an off-line phase classification algorithm, which significantly reduces the amount of simulation time required. These goals are met by simulating only a handful of *intelligently* picked sections of the full program. When these simulation points are carefully chosen, they provide an accurate picture of the complete execution of a program, which gives a highly accurate estimation of performance. This release provides new features for reducing the run-time of SimPoint and simulation points required, and provides support for variable length intervals. The SimPoint software can be downloaded at:

`http://www.cse.ucsd.edu/users/calder/simpoint/`

Acknowledgments

This work was funded in part by NSF grant No. CCR-0311710, NSF grant No. ACR-0342522, UC MICRO grant No. 03-010, and a grant from Intel and Microsoft.

References

[1] R. E. Bellman. *Adaptive Control Processes*. Princeton University Press, 1961.

[2] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. Technical Report UCSD-CS2004-0803, UC San Diego, November 2004.

[3] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[4] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[5] F. Farnstrom, J. Lewis, and C. Elkan. Scalability for clustering algorithms revisited. *SIGKDD Explor. Newsl.*, 2(1):51–57, 2000.

[6] G. Hamerly, E. Perelman, and B. Calder. How to use SimPoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review*, 31(4), March 2004.

[7] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. Technical Report UCSD-CS2004-0804, UC San Diego, November 2004.

[8] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[9] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[10] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[11] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[12] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *International Symposium on Microarchitecture*, December 2004.

[13] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734, 2000.

[14] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[15] F. J. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, 3(2):131–169, 1999.

[16] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming*, October 2002.