In Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS 2006).

Considering All Starting Points for Simultaneous Multithreading Simulation

Michael Van Biesbrouck[†] Lieven Eeckhout[‡] Brad Calder[†]

[†]CSE, University of California, San Diego, USA
[‡]ELIS, Ghent University, Belgium
Email: {mvanbies,calder}@cs.ucsd.edu, leeckhou@elis.UGent.be

Abstract

Commercial processors have support for Simultaneous Multithreading (SMT), yet little work has been done to provide representative simulation results for SMT. Given a workload, current simulation techniques typically run one combination of those programs from a specific starting offset, or just run one combination of samples across the benchmarks.

We have found that the architecture behavior and overall throughput seen can vary drastically based upon the starting points of the different benchmarks. Therefore, to completely evaluate the effect of an SMT architecture optimization on a workload, one would need to simulate many or all of the program combinations from different starting offsets. But exhaustively running all program combinations from many starting offsets is infeasible — even running single programs to completion is often infeasible with modern benchmarks.

In this paper we propose an SMT simulation methodology that estimates the average performance over all possible starting points when running multiple programs concurrently on an SMT processor. This is based on our prior co-phase matrix phase analysis and simulation infrastructure. This approach samples all of the unique phase combinations for a set of benchmarks to be run together. Once these phase combinations are sampled, our approach uses these samples, along with a trace of the phase behavior for each program, to provide a CPI estimate of all starting points. This all starting point CPI estimate is precisely calculated in just minutes.

1 Introduction

Modern computer architecture research relies heavily on cycle-accurate simulation to help evaluate new architectural features. Unfortunately, architectural simulations are extremely time-consuming. There are two primary reasons for this. First, with each processor generation more and more transistors can be integrated to the chip, resulting in more complex microarchitectural features to take advantage of this increased transistor budget. Simulating more complex microarchitectures obviously slows down the simulation. Second, architects use real-life applications in their performance analysis. Current industry-standard benchmarks, such as SPEC CPU2000, execute hundreds of billions of instructions. Even on today's fastest uniprocessor architectural simulators, simulating some of the SPEC CPU2000 benchmarks takes several weeks to complete. Simulating benchmarks to completion on a detailed simulator is obviously no longer feasible.

In order to measure cycle-level events and to examine the effect that microarchitecture optimizations would have on the whole program, architects are forced to execute only a small subset of a whole program at cycle-level detail and then use that information to approximate the full program behavior. The subset of the program that is chosen for detailed study has a profound impact on the accuracy of this approximation, and picking these points so that they are as *representative* as possible of the full program is a topic of several research studies [4, 9, 13, 14, 15, 19].

1.1 SMT Simulation

The emergence of Simultaneous Multithreading [16] in commercial processors, such as Intel's Hyper-Threading architecture and IBM's POWER5, increases the importance of improving multi-program workload simulation. Questions that were simple to answer in a single-threaded environment, such as "which section of execution will represent the complete workload?", are more complex in an SMT environment. When two or more programs share a processor's resources at a cycle-level granularity, as is the case with SMT, the performance of the two applications becomes entangled. If there are multiple programs running at the same time, the behavior of all the programs will affect not only the overall performance of the machine but also the distribution of performance between the different programs, causing some to execute faster than others. Changing a hardware parameter that has an effect on performance may change which parts of the programs execute together. This change, in turn, may mean that the machine is now executing a different mix of behaviors, which will influence the overall performance. This interdependence, or entanglement, makes it difficult to summarize or estimate the overall behavior of the system. The challenge in creating a sampling approach to SMT lies in determining how far to fast-forward each individual thread between samples. This distance will vary as the threads execute through different phases of execution; the distance also varies with different microarchitecture configurations.

Individual programs exhibit phase behavior in which each phase has roughly uniform IPC, misprediction rates, data miss rates, and so forth. Some programs change phases rarely

or in predictable ways; others frequently change phase in complex ways. On an SMT machine, each program affects all of the others in ways determined by its current phase behavior. As a result, the combination of programs can have more complex phase behavior since it is the product of their individual behaviors. This combined behavior determines the relative progress of the threads.

In our prior work, we proposed the *Co-Phase Matrix* [18] to guide the simulation of an SMT processor for a multi-program workload. The co-phase matrix represents all of the potential phase combinations, called a co-phase, of a multi-program workload to be examined in an architecture study. The co-phase SMT simulation approach populates the co-phase matrix with samples during simulation. Once a co-phase has an appropriate sample, we no longer need to simulate that co-phase and we can just fast-forward execution to the next co-phase. The amount to fast-forward for each thread is determined by the performance data stored in the co-phase matrix for that particular co-phase.

1.2 Paper Contribution

The co-phase matrix approach in [18] focused on providing an accurate simulation assuming a single starting position in each binary being simulated. In this paper we show that the architecture behavior and overall throughput can vary drastically based upon the different starting points of the different benchmarks. Therefore, to completely evaluate the effect of an SMT architecture optimization on a workload, we may need to simulate many or all of the program combinations from different starting offsets. But exhaustively running all program combinations from many starting offsets is infeasible — even running single programs to completion is infeasible with modern benchmarks.

This paper proposes an efficient SMT simulation methodology that estimates average performance over all starting points when running multiple programs on an SMT processor. This is achieved by populating the co-phase matrix with performance results for the different co-phases as was done in [18]. But, once this co-phase matrix is populated, we use it to estimate the average performance over all starting points. This is done by randomly picking a number of starting points and by analytically simulating each of these co-phase executions with their given starting points. Since the analytical simulation is done very efficiently, the whole SMT simulation for a set of starting points completes very quickly, in at most a few minutes. Although the fact that co-phases interact with each other is well known in the literature, there is no prior work that proposes a methodology that addresses this issue. To the best of our knowledge, this paper is the first to propose an SMT simulation methodology that estimates average performance for all starting simulation points.

2 Prior Work

Most of the research done on SMT processors use an ad-hoc simulation methodology. Researchers typically pick a number of arbitrary samples (or in many cases just a single sample) from a randomly chosen set of benchmarks. Then they simulate these randomly-picked samples together. In addition, most of these studies have used fairly small samples. Typical numbers are 100M to 300M instructions per sample. There are a number of important pitfalls with these methodologies. First, simulating such a small number of instructions per benchmark is unlikely to be representative for the complete benchmark execution. Second, choosing a single starting point for each benchmark may give a distorted view on how co-executed threads interact with each other in an SMT processor. Third, since these co-samples are chosen arbitrarily, it is unlikely that this gives a faithful image of real SMT behavior for those programs.

Raasch and Reinhardt [12] used an improved SMT simulation methodology in their study on how partitioned resources affect SMT performance. They selected a set of diverse co-sample behaviors rather than randomly chosen co-sample behaviors. As a first step, they use single simulation points provided by SimPoint [14]. They then run all possible two-context co-phase combinations on a given microprocessor configuration — in their setup they ran 351 co-phases. For each of those co-phases, they compute a number of microarchitecture-dependent characteristics such as per-thread IPC, ROB occupancy, issue rate, L1 miss rate, L2 miss rate, functional unit occupancy, *etc*. Using the methodology from [5], they then apply principal components analysis (PCA) and cluster analysis (CA) to come to a limited number of 15 two-context co-phases. A potential pitfall with this methodology is that a single simulation point is chosen per benchmark. This could give a distorted view for what is being seen in a real system where programs go through various phases. In addition, they do not consider the issue of multiple starting points as we do in this paper. They consider a single starting point and a single phase per benchmark only.

Van Biesbrouck *et al.* [18] proposed the co-phase SMT simulation approach for accurately predicting SMT performance where each thread starts at a specific starting point. The basic idea is to keep track of the performance data of previously executed co-phases in a co-phase matrix; whenever a co-phase gets executed again, the performance data is easily picked from the co-phase matrix. By doing this, each unique co-phase gets simulated only once, which greatly reduces the overall simulation time. In that work, we did not address the issue of multiple starting points. The point of that paper was to show that the co-phase matrix is an accurate SMT simulation approach for predicting performance where each program starts at a single starting point. That work is being extended in our paper by showing that the co-phase matrix can also be used for estimating average performance over all starting points.

Kihm *et al.* [8] also showed that SMT performance is sensitive to starting points. They profiled a number of co-program executions with different starting points and observed that different performance results were obtained. Their study was done on real hardware, namely on an Intel

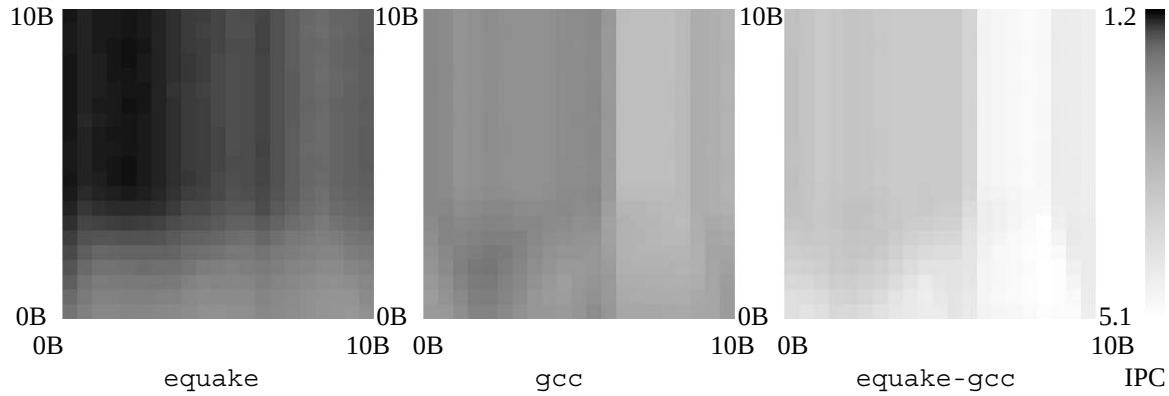

Figure 1: The graphs show the IPC when `equake` *and* `gcc` *are run together from various starting offsets. There are graphs for each program's IPC and their combined IPC. The shade of grey at (x, y) indicates IPC when simulation starts with* `gcc` *x instructions from the start of its execution and* `equake` *y instructions from the start of its execution. Simulation completed after a total of 10 billion instructions were committed.*

Pentium 4 processor. Unlike this paper however, they did not provided a simulation methodology that allows for capturing the average performance for all starting points.

Alameldeen and Wood [1] showed that for multi-threaded workloads running on real systems, performance can be different for different runs from the same initial state. This variability is not modeled in deterministic simulations. To account for this variability, they argue to inject randomness into the simulation environment, and to apply statistics for making viable conclusions. In this work, Alameldeen and Wood do not address the variability in performance results due to multiple starting points in multi-program workloads.

Ekman and Stenström [6] use random sampling for simulating multiprocessor systems and they use the well-established matched pair statistical method to show that the variability in the system's throughput decreases with an increasing number of processors when running multi-program workloads. As a result, fewer samples need to be taken in order to estimate performance on a multiprocessor system through sampled simulation. They only considered highly-synchronized parallel programs, so they do not consider multiple starting offsets and changes in co-phase behavior due to changes in the microarchitecture. For example, they were unable to simulate `cholesky` with their methodology because the threads were not synchronized.

3 Motivation for Simulating All Starting Points

Most studies use absolute performance estimates based upon a small number of simulation runs to predict the effects of microarchitectural changes. These simulation results are not completely accurate in even the single-threaded case, so some simulation methodologies focus only on the change in performance metrics due to microarchitectural changes, not the absolute numbers produced. We show that both approaches can lead to misleading results when simulating SMT processors.

3.1 Absolute Performance Predictions

A single starting point per thread might not be representative for what one would observe in a real SMT environment. The pitfall of selecting a single starting point is that the performance results that you obtain may be very different from those with different starting points. This is illustrated in Figure 1. This plot shows the average IPC that is obtained when simulating two benchmarks, `equake` and `gcc-166`, from different starting locations. In each of these experiments, we simulate until the threads execute a total of 10B instructions. We show results for 441 different relative offsets; adjacent sample points differ by 500M instructions in one thread's starting offset. The average (aggregate) IPC numbers are encoded by shades of gray: white means an IPC of 5.1 whereas black means an IPC of 1.2; we provide a scale to estimate intermediate values. We clearly observe that the overall performance is very sensitive to the starting points; the overall IPC varies from 1.2 to 5.1. So, the pitfall is that using a single starting point may impact expected performance results significantly.

To examine the behavior of multiple starting points further, we now examine the relative progress of execution of several different starting points using what we call a *Relative Progress Graph*. Figure 2 shows the relative progress graph for `equake` and `gcc`. The relative progress for `gcc` and `equake` are shown along the horizontal axis and vertical axis, respectively. For the line that starts at $(0, 0)$, a point plotted at (x, y) indicates that when thread 0 has executed x instructions, thread 1 has executed y instructions. The other lines start at different points on the graph, since they represent either `gcc` starting at the beginning of execution, and `equake` starting simulation at one of the offsets shown on the vertical axis. Similarly, the lines starting on the horizontal axis represent `equake` starting simulation at the beginning and `gcc` a given number of instructions (shown on the horizontal axis) into `gcc`'s execution.

In Figure 2 we see that adjacent starting offsets usually produce similar but not identical executions. The executions

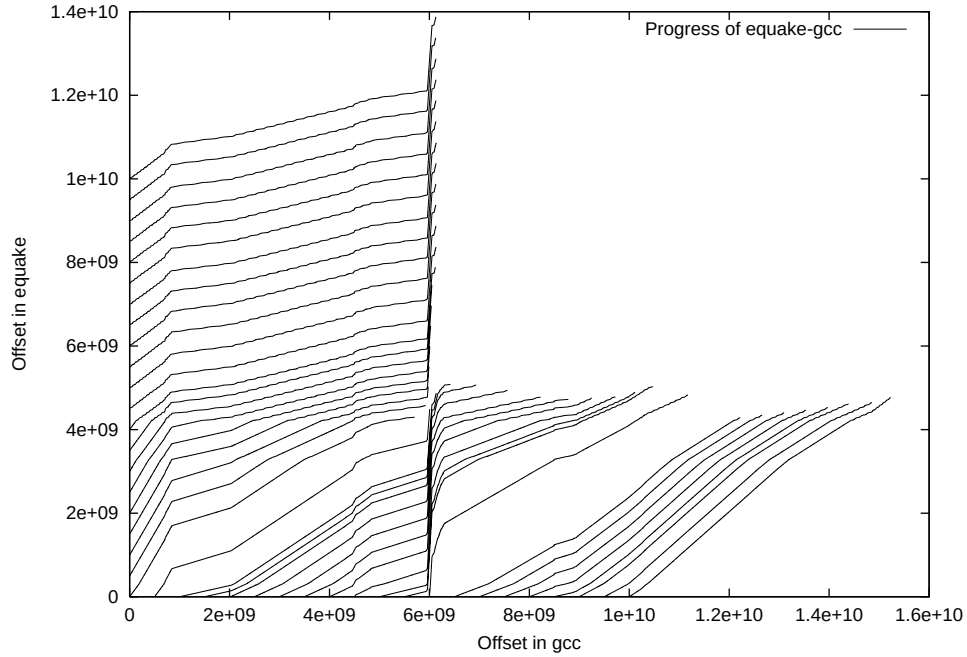

Figure 2: Relative progress of `equake` *and* `gcc`*. Each line represents a single 10B-instruction execution of* `equake-gcc` *from a different starting offset (either equake or gcc is always run from the beginning). Each plotted point represents execution offsets that occur during SMT execution.*

that start near $(0, 0)$ are the ones with the most variety until `gcc` reaches the 6B-instruction mark. At that point `gcc` makes much less relative progress than `equake`. The reason is that `gcc` suffers from a large number of L2 misses at that point. The endings of many executions are thus dominated by progress in `equake`. This extreme phase behavior also appears very clearly in the `gcc` graph in Figure 1 as a sharp increase in IPC for simulations that start `gcc` at an offset of more than 6B instructions. It is less obvious in the `equake` graph because `gcc` is much more affected than `equake`.

The sharp change in `gcc`'s performance is worth additional investigation as it improves understanding of the interactions between programs on SMT processors. In Figure 3 we plot the performance of the programs, measured in instructions per cycle (IPC). The solid lines show SMT behavior and the broken lines show single-threaded behavior. The `equake` lines are marked with squares and the `gcc` ones with triangles. The x-axis indicates the proportion of instructions committed; in the case of the SMT executions this is the combined number of committed instructions between the two threads, so the SMT executions are aligned with each other but the single-threaded and SMT executions of a particular thread are not. The markers are placed every billion instructions of single-threaded execution to help the reader match up parts of execution between singled-threaded and SMT runs. The interesting event that occurs 6B instructions into `gcc`'s execution is at the 50% mark on both graphs; this point is easily identified by dramatic changes in `gcc`. We can see that single-threaded execution also reaches a low IPC at this point, but for a much briefer period. The cause is visible in Figure 4, which uses a log scale to show the average number of L2 cache misses per single-threaded instruction committed. Here, the single-threaded `gcc` briefly misses one cache access per 20 instructions, but the SMT execution misses one of every four instructions due to the increased cache contention. The extremely high miss rate allows `equake` to continue on while `gcc` is nearly stalled. Note that while `equake` often has higher miss rates in SMT execution than when single-threaded, the absolute frequency of misses is much lower so there is less of an impact.

3.2 Relative Performance Predictions

In many practical circumstances, being able to accurately estimate relative performance change is more important that absolute performance estimation [6, 11, 18]. In other words, absolute simulator and methodology accuracy is less important than relative accuracy as long as the relative effects of hardware changes are faithfully tracked. This is especially the case for early design stage studies.

In the case of SMT simulation, this means that the variation in performance for different simulation starting points is not important as long as all of the starting points are equally affected by microarchitectural changes. To examine this, we simulated 20 pairs of starting offsets each for three pairs of programs and executed each of these on eight different hardware configurations, for a total of 480 experiments consisting of 1 billion committed instructions for each run. The program pairs and machine configurations are the same as those used to evaluate relative error in our previous paper [18]. The hardware configurations cover all combinations of small and large L1 caches, L2 caches and branch predictors.

We expect that for any hardware change, some bench-

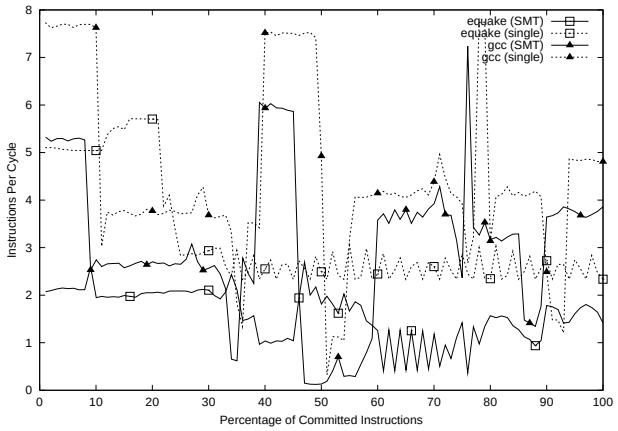

Figure 3: IPC of `equake` *and* `gcc` *running singly and as a pair.*

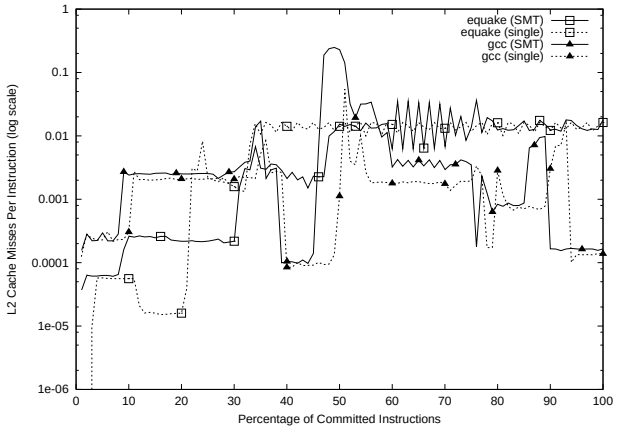

Figure 4: L2 cache miss behavior of `equake` *and* `gcc` *running singly and as a pair.*

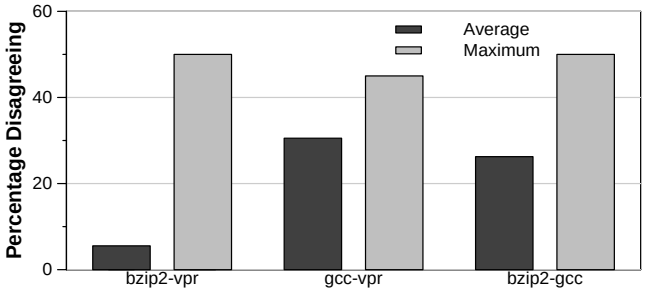

Figure 5: Performance effect disagreement after hardware configuration change. 0% indicates that all starting offsets improve (or degrade) due the change; 50% indicates that half improve and half degrade, the worst possible result.

mark pairs will see performance gains whereas others will not. If we take a single starting point for a particular benchmark pair, then performance will clearly get better or worse as the hardware configuration changes. Unfortunately, picking a different starting offset might give us the opposite result for the same hardware configurations. To examine this we take all 28 pairs of different hardware configurations, and we see how the performance for each pair of hardware configurations differ for a two-program workload examining 20 different starting offsets for that workload. For a pair of hardware configurations, we take a pair of programs and we vary the starting offsets for those programs 20 times. The result of each run is a ranking of the two hardware configurations saying that one has more throughput than the other. We then calculate for the 20 starting offsets examined with two hardware configurations, what percentage of time did the ordering of the two architectures agree versus disagree. If they were always identical, then we would see 0% disagreement. The worst case result would be 50% disagreement, which means that 50% of the time the first hardware configuration was said to be better than the second, and *vice-versa* and the only cause of this would be using different starting offsets.

Ideally, we want to see 0% disagreement as a result of the experiment. With 8 hardware configurations, there are 28 distinct pairs of configurations that we compare. Figure 5 shows the percentage of offsets that disagree in the direction of improvement with the majority of experiments. Since this number varies over the 28 possible hardware configuration pairings (potential experiments), we report average and maximum disagreement rates. The possible rates range from 0% (best) to 50% (worst).

We find that `bzip2-vpr` consistently favors one hardware configuration over the other in all but a couple possible experiments, whereas the others typically have results divided between the two hardware options. In the architecture comparison that caused the most disagreement, the average performance change was over 2% and frequently much higher. For each pair of programs, there are experiments that would produce misleading results if only a single starting offset were chosen. For most experiments, we can expect that some starting offsets lead to improvements but others do not. We conclude that researchers examining relative performance effects still need to be able to analyze the distribution of the results over many starting offsets rather than starting execution from a single offset.

4 Co-Phase Matrix SMT Simulation

The SMT simulation methodology that we propose in this paper borrows from the static co-phase matrix idea proposed by Van Biesbrouck *et al.* [18], which in turn borrows from the SimPoint simulation approach. These two will be discussed now.

4.1 SimPoint

Sherwood *et al.* [13] proposed using code signatures to break a program's execution into phases. To identify phases, they broke a program's execution into contiguous non-overlapping intervals. An *interval* is a continuous portion of execution (a slice in time) of a program. A *phase* is a set of intervals within a program's execution with similar behavior, regardless of temporal adjacency. This means that a phase may appear many times as a program executes. Phase classification

partitions a set of intervals into phases with similar behavior.

Sherwood *et al.* [7, 11, 14] created a tool called SimPoint that groups intervals with similar code signatures into the same phase. The code signature for each interval is represented by a Basic Block Vector (BBV) [14] to capture information about changes in a program's behavior over time. A *Basic Block Vector* is a one-dimensional array, where each element in the array corresponds to one static basic block in the program. The BBV for each interval represents the frequency of execution of each static basic block in the program for that interval. SimPoint can then perform clustering on BBVs, because each vector contains the frequency distribution of code executed in each interval. By comparing BBVs of two intervals during clustering, SimPoint can evaluate the similarity of two intervals. If the distance between the two BBVs is small (close to 0), then the two intervals spend about the same amount of time in roughly the same code, and therefore we expect the performance of those two intervals to be similar. Code signatures grouped into the same cluster have been shown to exhibit similar CPI, numbers of branch mispredictions, numbers of cache misses, *etc* [10, 14].

After this phase classification algorithm is done, intervals with similar code usage will be grouped together into the same phase. Then from each phase, they choose one representative interval that will be simulated in detail to represent the behavior of the whole phase. Therefore, by simulating *only* one representative interval per phase, SimPoint can extrapolate and capture the behavior of the entire program. This set of intervals, one chosen from each phase, is called the set of *Simulation Points*. This set of simulation points for a given program-input pair are simulated to create an estimated CPI (and other architecture metrics) for each architecture configuration during design space exploration.

For the SMT simulation methodology that we propose in this paper, the phase behavior of a complete program execution is represented by a *phase-ID trace*. The phase-ID trace indicates at which instructions in the program's execution phase changes occur and what the new phase IDs are.

4.2 Co-Phase Matrix

Our prior work [18] built on SimPoint to create an SMT simulation methodology that accurately predicts performance when running multiple threads each from a given starting point. The key idea is that for each thread, the same code signatures will be seen for that thread even on an SMT processor. The code signatures only represent the code executed in that thread, so we will be able to consistently see the per-thread code-based phase behavior no matter if we are simulating the program by itself or simultaneously with other programs on an SMT processor. This allows us to identify what phase a program is in, no matter what other behaviors are going on in the simulated processor.

Taking this one step further, we can use the per-thread phase ID trace to create a unique co-phase signature across all of the running programs on an SMT processor to capture that behavior. Just as in the single-threaded version, where overall behavior does not change within a given phase, in an SMT machine the overall behavior should not change unless *at least one* thread has a phase change. Thus, to estimate performance we need to consider all *combinations* of phases that run together. Each combination of simultaneously running phase-IDs forms a *unique co-phase identifier*. In our previous work [18], we have shown that taking a sample of the simultaneous execution of co-phases and storing it with its co-phase identifier accurately represents the SMT performance. In other words, when the same combination of phases is seen again during the execution, we have found that similar overall performance is seen. This means that we can keep track of a list of combinations of per-thread phases. During SMT simulation, we store a list of co-phase combinations along with their simulated performance. This is called the *Co-Phase Matrix*.

4.3 Building the Co-Phase Matrix

The *static co-phase matrix* is a co-phase matrix that is populated with information about all possible co-phases for the concurrently running threads prior to estimating the overall CPI. For every benchmark pair that we wish to simulate, we must create a co-phase matrix. We generate the static co-phase matrix using phases and simulation points determined by SimPoint 3.0 [7]. The single-threaded phases and the combinations of programs that we wish to investigate determine the set of co-phases that we need to sample. Each program is run once using a functional simulator to create checkpoints for each simulation point. These checkpoints are machine-configuration independent and can be reused for all subsequent experiments. Disk space and checkpoint loading time can be greatly reduced using techniques that we describe in [17].

For each co-phase entry, we perform a few million instructions of detailed simulation, starting each thread at the SimPoint simulation point from the single-thread phase analysis, using the checkpoints to avoid fast-forwarding. The result of this detailed simulation forms the co-phase entry's sample, which represents the performance for a pair of programs on a single hardware configuration. Since we use checkpoints, it is possible to perform all of the simulations in parallel. Once the static co-phase matrix is built, the SMT simulation itself is done through analytical simulation driven by the co-phase matrix, as described in the next section.

4.4 Using the Co-Phase Matrix to Estimate a Single Starting Point Combination

We now describe in more detail how the co-phase matrix can be used to guide SMT simulation for a set of programs from a single starting point. We will use Figure 4 as an example. In Figure 4, the solid line on the graph represents the relative progress of the two threads. The phase-ID traces of the two threads (thread 1 on the x-axis and thread 2 on the y-axis) can be seen adjacent to the axes. Thread 1 goes through the phase sequence xyxyx with phase changes at 5M, 10M, 20M and 25M instructions, respectively; thread 2 goes through the phase sequence abab with phase changes at 10M, 20M and

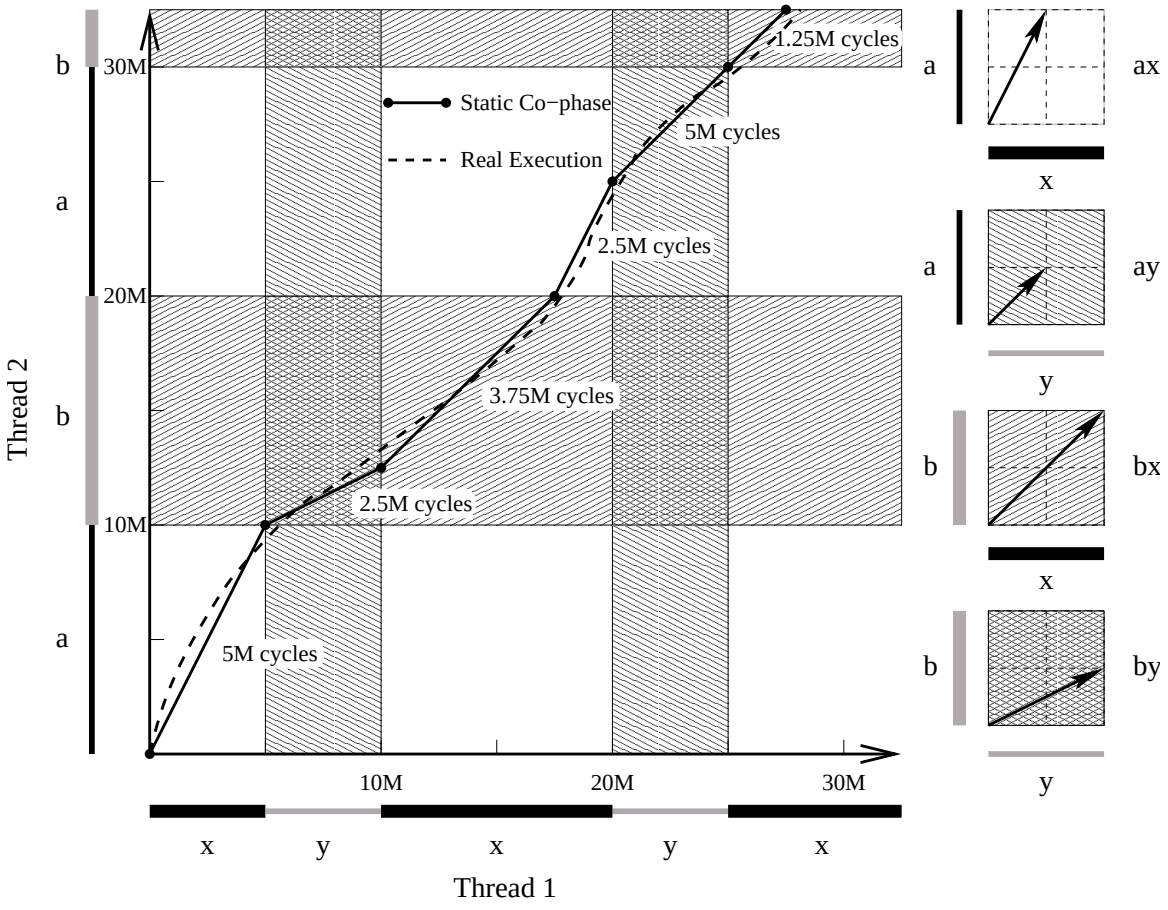

Figure 6: Approximating detailed execution with the static co-phase matrix.

30M instructions, respectively. Each rectangular section of the graph is shaded according to the co-phase that will be used while the execution of the two threads is within the rectangle. For example, the white-shaded rectangles represent the co-phase ax, where phase x for thread 1 is co-executing with phase a from thread 2; the heaviest shaded rectangles represent the co-phase by, *etc*. The graphs on the right of the relative progress graph visually represent the co-phase matrix. For each co-phase we show a vector indicating the relative progress of the threads. For co-phase ax, the co-phase matrix shows that phase a in thread 2 makes twice as fast progress as phase x in thread 1. For by, phase y makes twice as fast progress as phase b. For co-phases ay and bx, both phases make both equally fast progress, but ay has half the throughput of co-phase bx.

For this example, we assume that the execution begins at $(0, 0)$, the start of both programs (we can apply the same procedure for any other starting point). Note that the co-phase matrix would contain actual IPC rates; we will assume in this example that one small square in the co-phase matrix representation shows an IPC of 1. For example, for phase ax, thread 1 (phase x) has an IPC of 1 whereas thread 2 (phase a) has an IPC of 2.

In the example given in Figure 4, the starting point is in co-phase ax. From the co-phase matrix entry corresponding to ax, we see that thread 2 progresses twice as fast as thread 1. Execution continues at that rate until both programs change phases at $(5M, 10M)$. Thus it takes 5M cycles to exit the first co-phase. The new co-phase is by with IPCs of 2 and 1 for threads 1 and 2, respectively. respectively. After just 2.5M cycles the horizontal thread leaves phase y at point $(10M, 12.5M)$. Progress beyond this point will be in co-phase bx. Both threads will now make equal progress, *i.e.*, we assume both threads run at an IPC of 2, until one of the thread hits a new phase ID (phase change). The next co-phase change will occur at point $(17.5M, 20M)$ after 3.75M cycles. The new co-phase then is ax, *etc*. This process is repeated until our target execution length is achieved, closely following the real execution of the two threads (shown with a broken line).

The above summarizes how we arrive at an estimated CPI using the static co-phase matrix. Given a static co-phase matrix, the starting points for two threads, and the phase-ID trace, we quickly estimate the overall CPI for the pair of programs running on an SMT processor using the above approach. For the results in this paper, we estimate the CPI for running two threads together until a total of 1 billion instructions have been simulated.

5 Providing an Estimate of All Possible Workload Starting Locations

As extensively discussed in Section 3, using a single starting point in SMT simulation could be misleading. This observation argues for an approach in which all starting points are considered for estimating overall SMT performance. In this section, we now describe how this can be done using the co-phase matrix described in the previous section.

To generate a performance estimate of all combinations of starting offsets, we use the same method for a single pair of starting points as we described in the previous section, but run it for many starting points. This creates a metric, called the *All Combination* (AC) performance number, which represents the average performance of a pair of benchmarks, independent of particular starting offsets. In this paper we only examine the AC in terms of overall CPI, but any other processor statistic can be collected in the same way. The AC is the average performance found when executing both programs for one billion instructions from every combination of possible program offsets. If a program reaches the end of its execution, it is restarted at the beginning to avoid bias near the start and the end of programs. Fixing an execution length is necessary to make averages meaningful and easy to compute. Choosing a long execution length ensures that the weighting of co-phases will match that of continuous SMT execution.

Although the static co-phase method for a single combination of starting addresses is fast, running it for all possible starting points obviously is infeasible. In our setup, this would require 10^{23} analytical simulations using the static co-phase matrix per pair of SPEC benchmarks. Although an analytical simulation using the static co-phase matrix is extremely fast, simulating that many runs would be impossible to do. Thus we propose to sample the set of possible starting offsets. We examine two sampling strategies, namely random and stratified sampling. For both approaches, we assume a populated static co-phase matrix to start from as described in Section 4.3.

We use random sampling to pick starting points for both threads. For each pair of randomly selected starting points, the static co-phase method is used to estimate performance when executing both threads from the given starting point. By doing this for a sufficiently large number of randomly selected starting points, called *samples*, an average performance estimate can be computed for *all* possible starting points. We always simulate for 1B instructions of combined execution, so the average CPI over all samples is just the simple average of all collected CPI rates. (Other metrics may require more complicated computations.) An interesting property of random sampling is that we can estimate the variability of the samples, which allows us to provide confidence bounds for average performance estimates. In addition to pure random sampling, we also consider stratified random sampling to ensure even coverage of possible starting points.

I-Cache	64kB 2-way set-associative, 64-byte blocks, 1-cycle latency
D-Cache	64kB 2-way set-associative, 64-byte blocks, 3-cycle latency
Unified L2	1 MB 4-way set-associative, 64-byte blocks, 10-cycle latency
Memory	100-cycle latency
Branch Pred	21264-style hybrid predictor with 13-bit global history indexing a 8k-entry global PHT and 8k-entry choice table; 2k 11-bit local history entries indexing a 2k-entry local PHT
OOO Issue	out-of-order issue, 256-entry re-order buffer
Width	8 instructions per cycle (Fetch, Decode, Issue and Commit)
Func Units	6 Integer, 2 Integer Multiply, 4 FP Add, 2 FP Multiply

Table 1: SMT processor configuration.

6 Experimental Setup

We use the M5 simulator [2] from the University of Michigan, which is based on SimpleScalar3.0c [3] as our SMT simulation environment. The configuration for this simulator is shown in Table 1. It is configured to support an intensive multithreaded workload. Hence the large caches and the abundant reorder buffer and processor width. We simulated SPEC 2000 benchmarks compiled for the Alpha ISA.

As in our prior study [18], we use the following benchmarks: `bzip2-graphic`, `equake`, `gcc-166`, `gzip-graphic`, `lucas`, `mesa`, `perlbmk-splitmail` and `vpr-route`. These benchmarks were chosen based on their phase behavior in order to present a wide diversity of phase behavior interactions. The first half of the above mentioned programs show the most complex phase behavior observed in SPEC CPU2000; the second half shows average case phase behavior.

We analyzed the benchmarks using SimPoint 3.0 [7]. The phase analysis for our results covered all of program execution and used intervals of 5M instructions. We limited the maximum number of phases found by SimPoint to 30, *i.e.*, $MaxK = 30$. The fewest number of phases found was `mesa` with 25 phases. So, all of the benchmarks used between 25 to 30 phases. For each of these phases, SimPoint chooses a representative interval, called a simulation point.

Since our all-starting-points SMT simulation methodology is based on the static co-phase matrix, we also need to populate the static co-phase matrix with performance numbers for the various co-phases. These performance numbers are computed by simulating all possible combinations of simulation points for each benchmark-pair as described in the prior section. For increased accuracy, we need to deal with the cold-start problem when simulating all of these co-phases. The cold-start problem refers to the fact that the hardware state is unknown at the beginning of each sample. This is particularly a problem for large hardware structures such as caches. In order to deal with the cold-start problem we assume the hit-on-cold warmup technique along with the following strategy to further reduce the impact of cold-start effects. We ignore the first 1.5M instructions of execution and use the remaining instructions to calculate the contents of the Co-Phase Matrix. Data collection ends once one thread com-

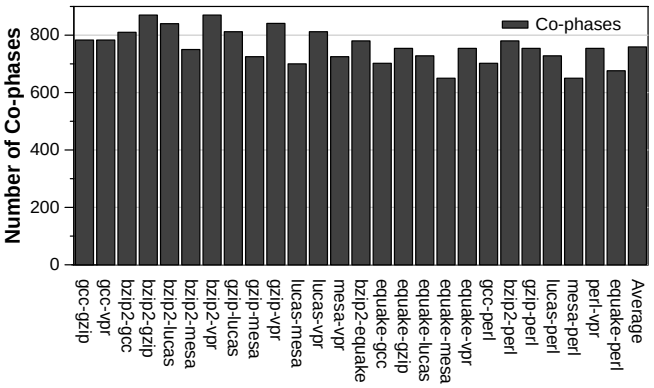

Figure 7: Number of co-phases per benchmark pair.

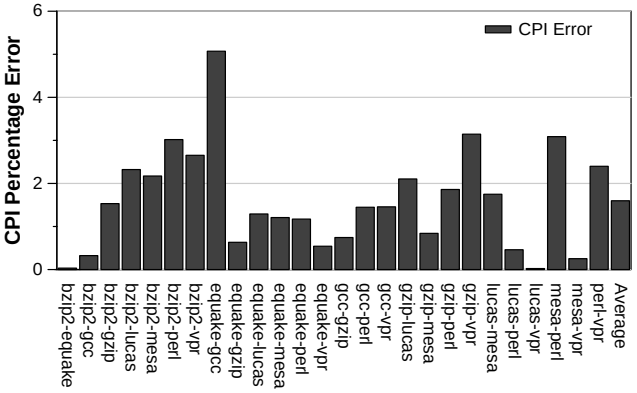

Figure 8: Error in CPI for static co-phase method simulation.

mits 3.5M instructions beyond the detailed warmup.

7 Evaluation

In this section we evaluate our newly proposed SMT simulation methodology. This is done in two steps. We first show that the static co-phase matrix is an accurate method for estimating SMT performance for two threads starting from a set of starting points. We subsequently show that the co-phase matrix can also be used to estimate SMT performance for all starting points.

7.1 Single starting point Co-Phase Matrix driven SMT simulation

Figure 7 shows the number of co-phases in the static co-phase matrix for the various benchmark pairs. Populating the static co-phase matrix with performance numbers requires computing performance numbers for all of these co-phases. The number of co-phases varies from 650 to 870, with 759 co-phases on average.

Figure 8 validates the static co-phase matrix approach with respect to the full detailed simulation runs for 81 starting offset pairs. For this result we simulated each starting offset for 1 billion detailed instructions to get the baseline CPI. We then used the static co-phase matrix, with the approach described in Section 4.4 to create the estimated CPI until both programs have reach 1 billion instructions of estimated execution. This then gives us an estimated error for each pair of starting points. We then report the average error across all starting points in Figure 8 for a set of programs. For this result, nine starting offsets were chosen for each program, which resulted in 81 simulations for a benchmark pair.

The results in Figure 8 show the average performance prediction error over those 81 simulation runs. We observe an average error of 1.6% and a maximum error of 5.1%. The highest error is for `equake-gcc`; others have error of at most 3.1%. As shown previously, `equake-gcc` presents challenges due to dramatically different execution rates of the two programs at a particular point, which can magnify small errors in sampling. Additional experiments have shown that this combination of programs benefits from a longer sampling period. Our low error rates validate our changes to interval size, sampling and phase analysis. They also demonstrate that the co-phase matrix can be used to accurately estimate performance when running a combination of programs on an SMT processor from a single starting point.

7.2 Convergence of All Combination Performance Estimates

We now examine the all combination performance for 26 benchmark pairs using the static co-phase method and two sampling techniques, random sampling (Figure 9) and stratified random sampling (Figure 10). For this approach, we first fill our static co-phase matrix with estimated CPIs as described earlier.

For each sample CPI, a starting location is randomly chosen separately for each program. Then a single run is performed of the static co-phase method of those starting points for 1 billion instructions using the static co-phase matrix approach in Section 4.4. This is repeated as we collect samples for many possible starting point combinations. The graphs show the effects on estimated All Combination CPI as the number of samples is increased. As the results show, the accumulated samples eventually converge to an All Combination CPI, however the techniques converge at different rates.

For the results, we did this 1024 times and calculated the cumulative average at each step. The estimates for most benchmark pairs converged after about 400 samples. Given a desired confidence level, it is possible to sample until that confidence level is achieved and report error bounds along with the estimate. We examined improving the convergence rate of the random sampling method using stratified random sampling. Each program was divided into four parts, giving 16 partitions per benchmark pair. We sampled equally from each of the 16 partitions of possible starting offsets. This had the effect of decreasing the variation when the number of samples was small (up to 200), but did not provide much help for greater numbers of samples. Note that plotted lines in Figure 10 are less noisy because points were plotted every 16 samples (due to stratification); only the variation in magnitude is important.

Random sampling allows us to adjust the number of samples taken so that the error due to the limited number of samples has a high probability of being within a small margin. Figure 11 shows, with 95% confidence, a bound on the differ-

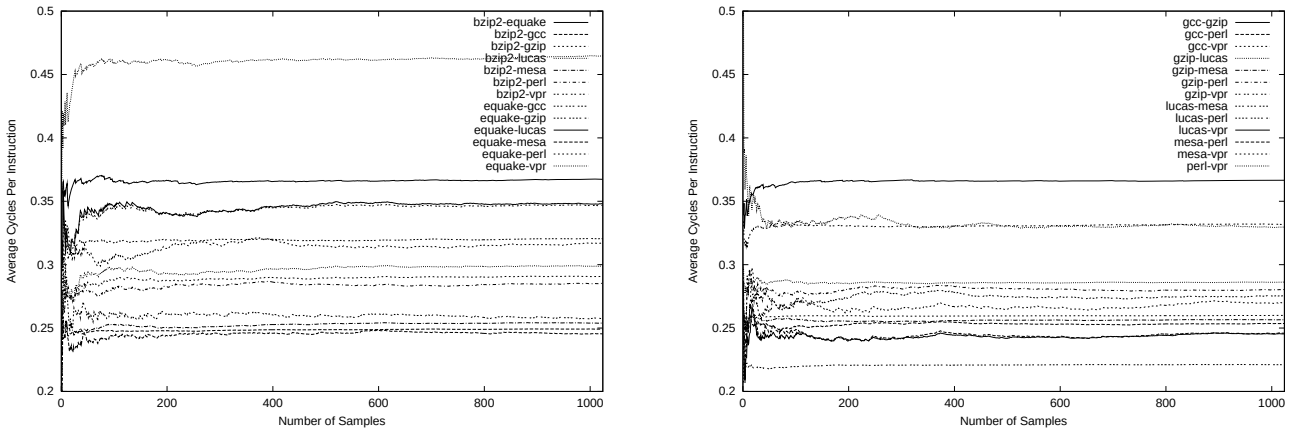

Figure 9: All combination CPI convergence using random sampling.

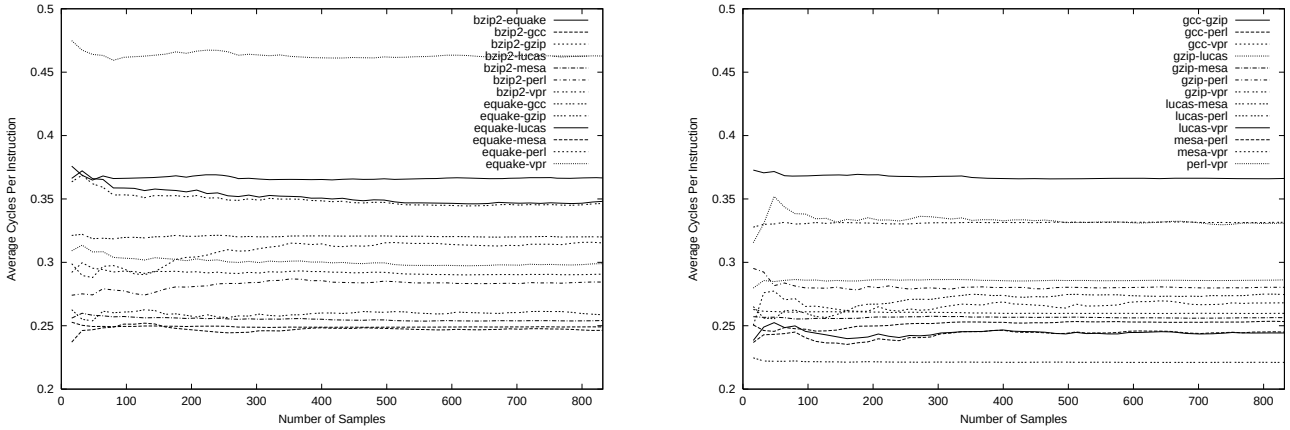

Figure 10: All combination CPI convergence using stratified random sampling.

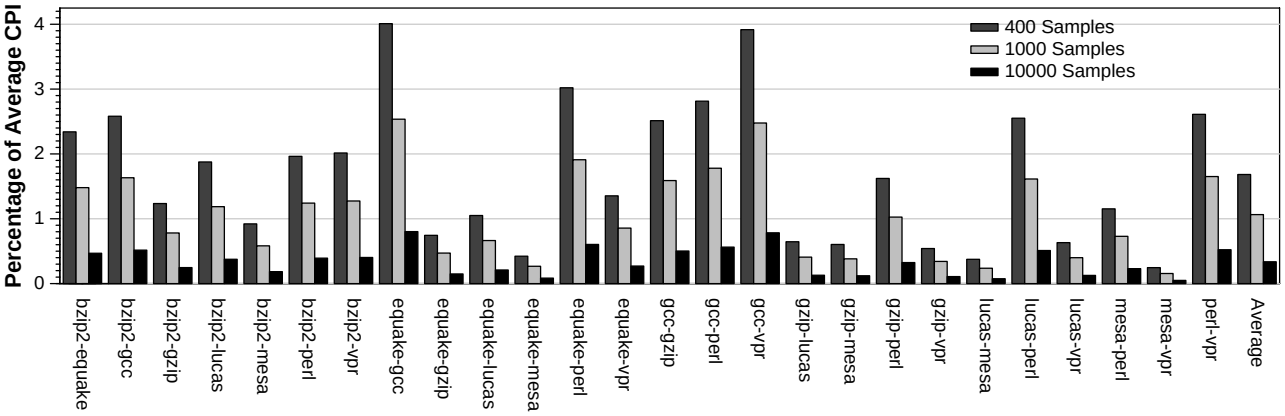

Figure 11: Confidence intervals for varying numbers of random samples.

ence between the actual average CPI of all possible starting combinations and our estimated CPI when using 400, 1000 and 10000 samples. The difference between the actual and estimated CPI is represented as an absolute percentage from the estimated CPI in Figure 11.

The size of the confidence intervals is dependent upon the natural variability of the benchmark combinations, the number of samples and the confidence desired. Although random sampling appears to converge after 400 samples for most benchmark pairs, the size of the confidence interval is still quite high. The results show that using 1000 samples and 95% confidence, the real CPI should be within just 1% of the estimated average CPI. All confidence intervals are under 0.8% of average CPI when using 10000 samples (just 0.34% on average). When using 10000 random samples, less than five minutes of static co-phase matrix simulation time per benchmark pair is sufficient to calculate the AC CPI.

8 Summary

Simulating an SMT system is a challenging task because of the tight entanglement between the different threads that co-execute on an SMT processor. Because of this entanglement, using a single starting point for each thread is unlikely to give a reliable performance number. This is an important pitfall that was not addressed in previous work.

In this paper we showed that it is important to consider multiple starting points in order to obtain a reliable SMT performance number. Moreover, we presented an efficient SMT simulation methodology for achieving this. By building up a co-phase matrix that summarizes the performance of all the co-phase executions, we are able to quickly estimate the average performance for all possible starting points. This is done by sampling over all possible starting points and by analytically simulating those randomly selected starting points over the co-phase matrix. Due to the use of the static co-phase method we were able to show an average sample collection bias of under 1.6%.

We evaluated two sampling approaches, random sampling and stratified random sampling. We observed that both sampling strategies resulted in around 400 starting points that need to be simulated in order to get stable performance estimates, but using 1000 or more samples allows strong confidence bounds. Since each sample can be collected in a fraction of a second, and a confidence interval on sampling error below 0.8% can be obtained in just a few minutes. The end result is an SMT simulation methodology that estimates average SMT performance over all combinations of starting points in the order of minutes once the co-phase matrix is populated with samples.

Acknowledgments

We would like to thank the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by NSF grant No. CCF-0342522, NSF grant No. CCF-0311710, a UC MICRO grant, and a grant from Intel and Microsoft. Lieven Eeckhout is a Postdoctoral Fellow with the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen) and is also supported by Ghent University, IWT, HiPEAC and the European SCALA project No. 27648.

References

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded commercial workloads. In *Annual International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[2] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Feb. 2003.

[3] 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.

[4] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, Oct. 1996.

[5] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload design: Selecting representative program-input pairs. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT-2002)*, pages 83–94, Sept. 2002.

[6] M. Ekman and P. Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–99, Mar. 2005.

[7] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[8] J. L. Kihm, T. Moseley, and D. A. Connors. A mathematical model for accurately balancing co-phase effects in simulated multithreaded systems. In *Workshop on Modeling, Benchmarking and Simulation (MoBS) held in conjunction with ISCA*, June 2005.

[9] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications, Kluwer Academic Publishers*, Sept. 2000.

[10] 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 a nd Software*, Mar. 2005.

[11] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *PACT'03*, pages 244–256, Sept. 2003.

[12] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2003.

[13] 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*, Sept. 2001.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming*, Oct. 2002.

[15] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–81, Nov. 1999.

[16] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA*, pages 191–202, 1996.

[17] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *2005 International Conference on High Performance Embedded Architectures and Compilation (HiPEAC)*, pages 47–67, Nov. 2005.

[18] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of the 2004 IEEE International Symposium on Performanc e Analysis of Systems and Software (ISPASS'04)*, Mar. 2004.

[19] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *30th International Symposium on Computer Architecture*, June 2003.