IEEE International Symposium on Performance Analysis of Systems and Software, March 2004

A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation

Michael Van Biesbrouck† Timothy Sherwood‡ Brad Calder†

†Department of Computer Science and Engineering, University of California, San Diego
‡Department of Computer Science, University of California, Santa Barbara
{mvanbies,calder}@cs.ucsd.edu, {sherwood}@cs.ucsb.edu

Abstract

Several commercial processors have architectures that include support for Simultaneous Multithreading (SMT), yet there is still not a validated methodology for estimating the performance of an SMT machine that does not rely on full program simulation. To create an efficient sampling approach for SMT we must determine how far to fast-forward each individual thread between samples. The fast-forwarding distance for each thread will vary according to execution phases, thread interactions and changes to the architectural configuration.

In this paper, we examine using individual program phase information to guide SMT simulation. This is accomplished by creating what we call a Co-Phase Matrix. *The co-phase matrix represents the per-thread performance for each potential combination of the single-threaded phase behaviors that can be found when multiple programs are run together. The co-phase matrix is populated by collecting samples of the programs' phase combinations, and is used to guide fast-forwarding between samples. We show for 28 pairs of SPEC programs that using the co-phase matrix provides an average error rate of 4% while requiring that only 1% of the full simulation be performed. The methods are also validated using a variety of architectural configurations and four-threaded workloads.*

1 Introduction

Modern computer architecture research relies heavily on cycle-accurate simulation to help evaluate new architectural features. While the performance of processors continues to grow exponentially, the amount of complexity within a processor grows at an even faster rate. With each generation of processor more transistors are added, resulting in more complex architecture features and cycle times continue to decrease. This has the effect that while the performance of processors is growing, the complexity is growing even faster. The end result is that the time to simulate a constant amount of processor time is growing, and it is already to the point that fully executing programs to completion in a detailed simulator is no longer feasible for architectural studies.

In order to measure cycle-level events and to examine the effect that hardware 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, 13, 7, 10, 11, 16].

The emergence of Simultaneous Multithreading [15] in commercial processors, such as Intel's new Hyper-Threading architecture, increases the importance of improving multi-program workloads 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 the 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 become 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 the distribution of performance between different programs, causing some to execute faster than others. Changing a 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 and creates a whole new set of problems for those interested in accurately and efficiently estimating performance. 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, and between different architecture 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 phase rarely or in predictable ways; others such as `gcc` 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.

Our paper focuses on improving the efficiency of SMT workload simulation, answering the question "Given an SMT multi-program workload with each program starting at a specific starting point, how can we accurately and efficiently estimate performance using sampling?" We focus on an SMT processor with 4 hardware contexts, running combinations of 2 or 4 programs at a time. Our sampling approach to SMT simulation is guided by the phase behavior found in single-program execution. This relies on finding the phase-based behavior of each program using SimPoint [11] to classify fixed-size intervals of execution into phases. For our experiments we selected programs from the SPEC benchmark suite that show a wide variety of single program phase-based behaviors, including many programs with complex structures.

The main contribution of our paper is the creation of a *Co-Phase Matrix* and using it to guide the simulation of an SMT processor for a multi-program workload. The co-phase matrix represents all of the potential phase combinations of a multi-program workload to be examined in an architecture study. The co-phase matrix is used to guide the unique samples we want to focus on capturing performance for during execution. Our simulation approach populates the co-phase matrix with samples during simulation. Once a phase combination has an appropriate sample, we no longer need to simulate that combination and we can just fast-forward execution to the next phase combination. The amount to fast-forward is determined by the performance sample stored in the co-phase matrix. We also examine a method that allows us to sample all co-phase matrix entries in parallel and then, with no additional simulation, determine the IPC of the programs from any relative starting offsets.

Our paper is not the first to point out that the behavior of one program running on an SMT processors can be affected by a different program that is scheduled to run at the same time. Indeed, both the work of Snavely and Tullsen [14] and of Parekh, Eggers and Levy [8] demonstrate that not only does this effect occur, but that it can in some circumstances be exploited for increased schedule efficiency. However, neither of these approaches make use of phases to perform optimizations and instead assume that each of the programs have homogeneous behavior. We extend this idea and show that there is even thread interference that happens at the level of phases, and that this can be used to perform more accurate performance estimation. Our future work will apply this to SMT scheduling.

2 Prior Simulation Techniques

Modern architecture research relies heavily on time-consuming detailed pipeline simulation, leading several researchers to develop ways of reducing simulation time while remaining true to complete simulation. This section provides a brief overview of some of the work in this area as it relates to our SMT sampling technique.

In [11], we used a characterization of a full program's execution created by clustering analysis to find a small set of phases, and for each phase picked a representative sample of the phase's execution. Taken together, these samples can represent the complete execution of a program. The set of chosen samples are called *simulation points*, and each simulation point is an interval on the order of millions of instructions. The simulation points were found by examining only a profile of the basic blocks executed for a program.

To perform the SimPoint analysis, a desired sample size is chosen (we use a sample size of 10 million instructions in this paper). The sample size is the interval size (in dynamic instructions) that the user wants to perform simulation or program analysis at. The program is then broken up into consecutive intervals equal to the sample size, and a profile is gathered for every interval. The profile measures what code was executed during each interval and its frequency. To do this a *Basic Block Vector* (BBV) is used. A Basic Block Vector is an array with one entry for every static basic block in the program. An interval's basic block vector has a count for each basic block, which is the number of times that basic block was executed during the interval. The BBV for an interval is compared to other interval BBVs to see how similar they are, and similar intervals are clustered together. Alternatively, they can be compared to a BBV representing the complete execution of the program in order to find a single interval that is the closest to the complete execution of the program.

Raasch and Reinhardt [9] examined the effects of partitioned resources on SMT execution. To reduce simulation time they used *single simulation points* from SimPoint [11] for each benchmark program-input pair. As a further optimization, principle components analysis and clustering were used to eliminate simulation points that were similar to other simulation points. Their goal was to capture the typical behavior of a workload independent of when each individual program was started. In comparison, our goal is to create an accurate and efficient SMT simulation approach for an SMT workload starting each program at a specific location.

Several different techniques have been proposed for sampling to estimate the program behavior. These techniques take a number of execution samples across the whole execution of the program, which are referred to as clusters in [4] because they are groupings of contiguous instructions. These clusters are spread out throughout the execution of the program in an attempt to provide a representative cross-cut of the application being simulated. Conte *et al.* [4] formed multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC, branch and data cache statistics). They used statistical sampling and confidence intervals to guide the number of samples to use for their simulations.

Wunderlich *et al.* [16] provide an accurate simulation infrastructure using statistical sampling. The paper provides contributions to statistical sampling, including how to efficiently warm up the state of the pipeline to accurately gather results using samples of size 1000 instructions. This procedure involves updating caches and branch predictors during execution and then executing 10000 instructions of warmup. In addition, after one sampling run, they show how to use statistical analysis to determine how many samples are needed to achieve a desired error bound and confidence.

To use sampling, simulation infrastructures usually fast-forward between samples. In doing this one has to address the issue of how to deal with the status of large structures such as the cache when switching from one sample to the next. One option is to warm up the structures before gathering detailed simulation results for the sample. Haskins and Skadron [5, 6] have examined finding the minimum distance to fast-forward before a simulation point based upon the working set size of the structure to be warmed up. They use their reuse analysis to accurately determine how long to warm up (start simulating) different structures (*e.g.*, branch predictors and caches) prior to detailed simulation.

2.1 Problems We Address

The techniques summarized above effectively solve the problems they address, but have only been fully explored in the single-threaded context. Our goal is to make these techniques usable by researchers working with SMT architectures. Specifically, we show how to collect samples of SMT execution that are representative of the phased behavior that occurs and how to combine the performance statistics from these samples so that the the overall IPC is accurately estimated.

3 Methodology

For this research we focus on eight programs to create a representative workload from the different types of phase behavior we saw in our prior SimPoint research. We use `bzip2`, `equake`, `gcc`, `lucas`, `gzip`, `mesa`, `perl` and `vpr` to examine multi-program phase-based interactions. The first four programs represent the most complicated phase-based behavior found in the SPEC benchmark suite, and the last four programs the average case phase-based behavior. We examine running all 28 combinations of the above eight programs in pairs on a four hardware context SMT processor. Four groups of four threads are also run on the same processor configuration.

For our multi-program workloads we fast-forwarded each program 1 billion instructions and then started co-simulating them on the SMT processor using the ICOUNT fetching heuristic [15]. We terminate simulation of a given multi-program workload as soon as the first program finishes executing 10 billion instructions. We use this to denote the end of the multi-program simulation due to the fact that each SPEC program has a different number of instructions for its ref input run, and each program executes instructions at different rates.

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.

In addition, we verified that the 10 billion instruction section for this multi-program workload is fairly representative of the phases seen over the whole program execution. This also gives us an arbitrary point part way through execution to start each program.

The M5 SMT simulator [2] from Michigan, based on SimpleScalar3.0c [3], was used to collect performance and architecture metrics in the simultaneously multithreaded environment. Although the simulator is capable of full-system simulation we did not use that capability for this paper. The configuration for this simulator is shown in Table 1. It is configured to support an intensive multithreaded workload. Hence the large cache and abundant reservation stations. We simulated SPEC 2000 benchmarks compiled for the Alpha ISA. The binaries we used in this study and how they were compiled can be found at `http://www.simplescalar.com/`.

4 Baseline SMT Sampling

The bulk of the work on efficient cycle-level simulation has considered only single-threaded processors running a single program. In these environments, it is clear when running a single program what it means for a sample to be representative. When examining a multi-program workload for a multithreaded architecture such as Intel's Hyper-Threading processors, it is not as clear what would be a representative workload sample.

4.1 Sampling Challenge for an SMT Processor

Single-threaded sampling methods assume that sampling points can be easily determined independently from detailed simulation, either through random sampling or some heuristic. But on an SMT processor, the threads share the hardware resources, and it is necessary to model the co-execution of the threads to determine which instructions from the programs will be executed at the same time.

Figure 1 shows how IPC changes over time for each program when it is run *by itself* on the baseline SMT processor. The x-axis represents time, and the numbers on the x-axis represents the percent of execution. The y-axis is the IPC for each execution interval. The time-varying IPC behavior is shown for 10 billion instructions, after fast-forwarding for

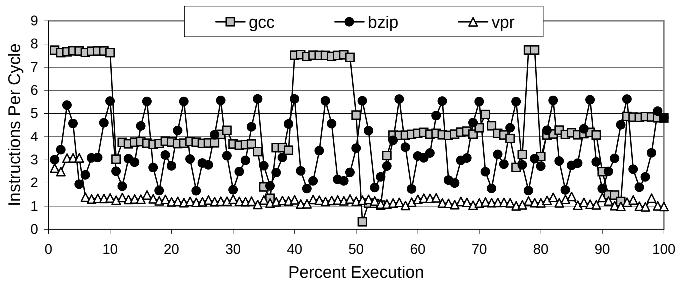

Figure 1: IPC Time-varying behavior for each program when it is run by itself on the SMT processor. The x-axis scale is percentage of execution.

one billion. In `bzip2`, `gcc` and `vpr` we see periodic behavior when these programs are executed by themselves.

Figure 2 shows the time-varying IPC results for all two-program combinations of `bzip2`, `gcc` and `vpr` running on an SMT Processor. These figures show the IPC for each program when co-executing with the others. The per-program IPC is lower than in Figure 1 because now all the programs are fighting for the resources they once had to themselves. The first thing to note is that there is significantly more complex behavior here than was present when the programs were running as individuals. Upon further inspection other patterns begin to take shape. First of all, the IPCs for the programs become somewhat synchronized and the IPCs of individual programs oscillate together through phases of high and low IPC.

As can be seen in Figure 2, the task of determining the relative execution rates for several programs will be complex. For a single pair of programs, such as `gcc` and `bzip2`, at times the programs will execute at the same rate and at others there is an order of magnitude difference in IPC. It is therefore a challenge to determine between samples how much to fast-forward each separate thread in order to arrive at a real sample that would exist in the representative baseline full simulation.

4.2 Using Single Simulation Points

A typical methodology that researchers use is to simulate an SMT processor for 100 to 300 million instructions at a single point for a given multi-program workload. This only exercises the interaction between the programs for a couple of different behavior combinations (serial phases) for each program. To illustrate the problem, we first examine randomly picking an offset in each program to start executing at and measure performance. Each offset combination was simulated until a total of 300 million instructions were executed. This is shown in Figure 3, where IPCs for random simulations are shown for the program pairs `bzip2-gcc`, `bzip2-vpr`, and `gcc-vpr`; the per thread estimated IPC results are shown next to each program combination. The random sampling results are shown as plus (+) signs. We also show the overall and per-program IPCs found when executing the program combination for 10 billion instruction, and the 300 million samples were drawn from this 10 billion instruction execution. The

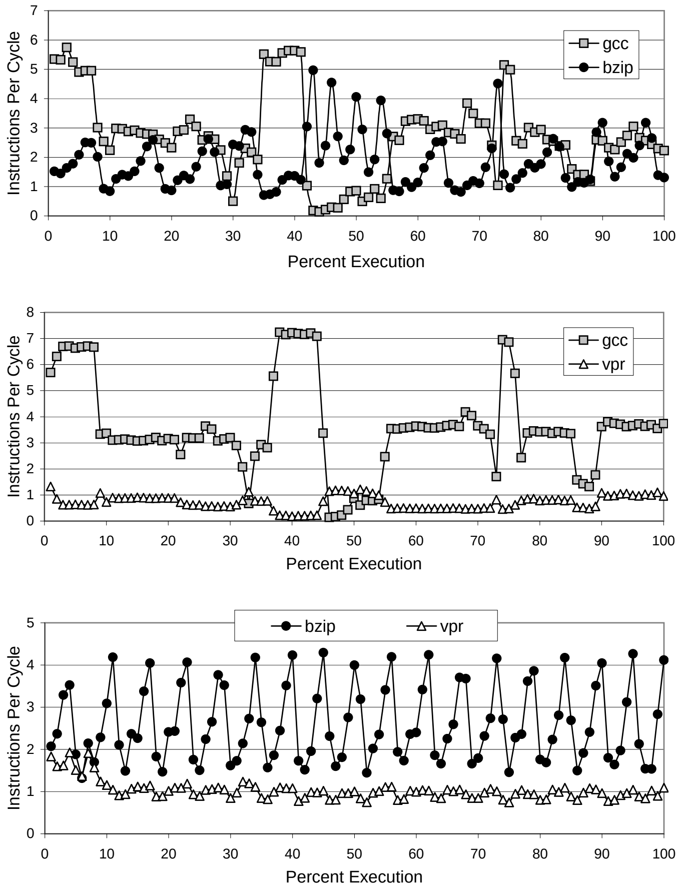

Figure 2: Time Varying IPC when running all the above 2 program combinations at the same time together on a dual hardware context SMT Processor.

results show that arbitrarily taking one sample for a program combination can lead to highly variable IPC estimates.

Also shown in Figure 3 are results that use the best *single* simulation point found by the SimPoint algorithm in [11] for each program and co-simulated those together for 300 million instructions. For a given program/input pair, SimPoint profiles the code usage, broken down into 100 million samples, over the complete execution of the program. It then compares a code profile of the complete execution of the program with the profile from each interval and attempts to pick the more representative set of 100 million contiguous instructions. The circle on the graphs represents using the best single SimPoint for each program when performing an SMT simulation. The results show that for some combinations it has a relatively small error, while for others it can have an overall error of almost 20% (`gcc-gzip`) or even almost 40% (`bzip2` in `bzip2-gcc`).

Single simulation points can be fairly representative of the entire program execution. They typically capture the transition point between the two most dominant phases in the run. Some programs have many phases and the single simulation point will have a higher error rate than if samples are taken from each phase of the program's execution. Therefore, we

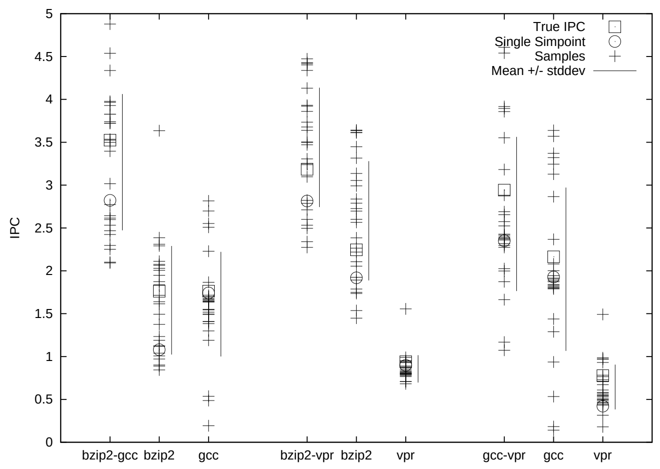

Figure 3: Random sampling results.

focus on obtaining a more accurate picture of the program's execution by taking samples from all of the phase combinations seen for a multi-program workload.

4.3 Guiding Fast-forwarding Using the Last Sample

Figure 2 shows complex fine-grain phase behavior, but general trends of execution are often present for significant intervals. This is caused by the phase nature of programs, where programs tend to execute in the same phase for a given period of time before transitioning to a new phase [1, 12]. Therefore the performance from recent execution can be a reasonable prediction of near future performance.

Using this observation, as a baseline sampling technique we examine a straight-forward sampling approach where we assume that the program's execution will continue to have the same IPC for some time. Then performance from one sample can be used to guide the amount of per-thread fast-forwarding until the next sample occurs. By periodically resampling we can detect changes in per-thread behavior, and then correct the fast-forwarding until the next sampling.

5 Using a Co-Phase Matrix

The technique discussed in the last section, periodically sampling and then assuming the behavior will be stable until the next sample, is simple to implement and works reasonably well in many cases. However, with only an incremental amount of complexity, and leveraging our past work on phase analysis, we can do even better. This new approach anticipates phase changes independently from sampling, allowing samples to be taken at every new phase combination. Figure 1 shows the repetitive phase behavior of single threads, and even when two-program combinations are run in Figure 2. The single-thread phase behavior is still present in multithreaded execution but the phases are now affected by competition for resources from other threads. This leads us to propose the use of phase detection techniques based on phases discovered using our earlier work on single-threaded program analysis.

5.1 Creating a Phase-ID Trace for a Single Program

The phase behavior we exploit is efficiently detected in each of the individual threads using the past techniques discussed in Section 2, and then combined using our new methods.

SimPoint [11] breaks a program's execution up into intervals. An *Interval* is a section of continuous execution (a slice in time) of a program. For the results in this paper all intervals are chosen to be the same size (10 million instructions). A *Phase* is a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency. In this way a phase can re-occur multiple times throughout the execution of the program.

Each interval is represented by a profile of the code (basic blocks) executed during that interval. The algorithm uses k-means clustering to group together intervals into the same phase if the same basic blocks were executed in those intervals with the same frequency. The key to this approach is that the program for a given input is broken into phases based completely on the code that is executed without considering its dynamic behavior. This allows the simulation points to accurately represent the program's behavior across different architecture configurations.

For our approach, the phase behavior is represented by a *Phase-ID trace* representing the complete execution of a single program, where each phase is represented by a unique ID determined by the SimPoint program. This indicates at which instructions in the program's execution phase changes occur and the new phase IDs.

5.2 The Co-Phase Matrix

On an SMT processor, the state of each thread's execution can be represented by the per-program phase-ID it is currently executing in. The key idea of our technique is that, just as in the single threaded version the 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 we need to keep track of a list of all *combinations* of phases that have been seen running together. This combination of phase-IDs, which are executing together, represent a *unique co-phase identifier*. We have found that taking a sample of the simultaneous execution of programs and storing it with its co-phase identifier accurately represents the SMT performance when this same co-phase combination is seen again in the future. We can then store a list of the past combinations we have seen, and we term this list the *Co-Phase Matrix*.

A Co-Phase Matrix represents the combination of all of the phase-IDs from each program in the workload that can execute simultaneously on the SMT machine, where there is an entry in the matrix for each co-phase identifier. If each phase combination were simulated, then the table would be filled with representative samples for all the possible phase combinations in the program. For each combination of the

Instructions (M)	Thread 0 Phase	Thread 1 Phase
0–4	a	x
4–8	a	y
8–12	b	x
12–16	b	x
16–20	a	y
20–24	a	y

Co-Phase	Thread 0 IPC	Thread 1 IPC
ax	2	1
ay	1	1
bx	2	2
by	1	2

Table 2: Phases found in two programs (4M instruction intervals) and a co-phase matrix. The table on the top shows the phase-ID trace gathered from SimPoint. The matrix on the bottom shows an example final co-phase matrix from simulating the two threads together.

Thread 0 Inst (M)	Thread 1 Inst (M)	Co-Phase	Cycles (M)
0–8	0–4	ax	4
8–10	4–8	by	2
10–16	8–14	bx	3
16–20	14–16	ax	2
20–24	16–24	ay	4

Table 3: Example of a co-phase matrix simulation.

phases (*co-phase*) that occurs when co-executing two or more programs we store into the co-phase matrix the per-thread IPC found during a sample of detailed simulation.

5.3 Guiding Fast-Forwarding

We can estimate the co-execution of multiple programs at a given point in time if we have two items: the phase-ID trace and the co-phase matrix entry that corresponds to the phases currently executing. If we have the entry in the co-phase matrix then we can predict per-thread IPC because we have observed and recorded this set of phases executing in the past. From SimPoint we obtain the phase-ID trace, which is used to determine how many instructions each program must execute before it encounters the next phase change. Using the IPC estimates from the co-phase matrix and the phase-ID trace we determine how many cycles it would take for the next phase change to occur for any of the co-executing threads. This is used to guide how far to fast-forward each thread using the estimated IPCs from the co-phase matrix. The following is an overview of the algorithm:

1. Co-Phase Matrix Lookup - The current co-execution thread combination represents a co-phase identifier, which is looked up in the co-phase matrix. If a sample exists, retrieve the per-thread IPC for each thread. If a sample does not exist, then perform detailed simulation for a specified sample size (see Section 6.2), and store the per-thread IPC into the co-phase matrix to reuse later.
2. Determine Number of Cycles to Fast-Forward - Using each program's phase-ID trace, calculate the number of instructions until the next phase change for each thread. Use this and the per-thread IPC from the co-phase matrix to calculate the number of cycles to reach that phase change. The thread with the smallest number of cycles until the next phase change determines how far to fast-forward.
3. Fast-Forward to Next Phase Change - Take the number of cycles from step 2, and multiply this by the per-thread IPC from the co-phase matrix to determine how many instructions to fast-forward each thread. Fast-forward each thread that many instructions. This results in a new phase-combination, and go back to step 1.

An example of using this approach can be found in Tables 2 and 3. The top part of Table 2 shows the per-thread phase-ID trace. The bottom part of the Table shows the resulting co-phase matrix built up using detailed samples during simulation.

In Table 3's example, both programs are executed from the start. From the phase-ID trace we see that thread 0 is in phase a for the first 8M instructions of its execution. Thread 1 will remain in phase x for only 4M instructions. Collectively they are in co-phase ab, in which thread 0 has an IPC of 2 and thread 1 has an IPC of 1. After 4M cycles thread 0 executes 8M instructions and thread 1 executes 4M instructions, at which point both change phase, making the new co-phase by. Now thread 0 has an IPC of 1 and thread 1 has an IPC of 2. At this rate thread 0 will not change phase for 8M cycles but thread 1 will change phase after only 2M cycles. So after 2M additional cycles thread 0 has executed 10M instructions and thread 1 has executed 8M instructions. At this point thread 0 stays in phase b but thread 1 returns to phase x; the new co-phase is bx. At this point the co-phase matrix entry bx is examined to determine the IPC for both of the threads, and the rest of the example proceeds in the same manner.

Next we examine two different approaches for guiding simulation using the contents of the co-phase matrix.

5.4 Estimating Performance with a Dynamic Co-Phase Matrix

The first approach we propose for guiding simulation dynamically populates the co-phase matrix as we simulate. For this technique, we start a simulation from a desired initial offset of each program with an empty co-phase matrix. The start of execution represents the first co-phase matrix entry needed to execute, so we perform a detailed simulation of that co-phase at the current execution and fill in the corresponding co-phase matrix entry. From this matrix entry we estimate how many instructions are going to execute from each program before the next phase change, using knowledge of how many instructions we have executed from each individual program and the single program phase-ID trace. We then fast-forward each program thread by that many instructions, and examine the new co-phase that the workload is at. If this co-phase is not in our co-phase matrix (or, for example, is based on too few samples), we perform detailed SMT simulation and fill it in. If it is in our matrix, we use the existing entry to estimate the number of instructions executed for each program and fast-forward each program. This process is repeated until we have completed simulation. When we are done we have a co-phase matrix filled out with all of the co-phases that were observed during this workload's execution, and a weight is assigned to each co-phase matrix entry corresponding to the fraction of time each co-phase occurred during this process. The performance results gathered in this co-phase matrix are then combined to achieve an overall estimated IPC of the combined run, and a per thread IPC.

The results labeled First Phase, 1% Phase and 5% Phase use this dynamic co-phase matrix approach. First Phase uses only the first sample found. The results labeled 1% and 5% add new samples at regular intervals. To sample 5% of execution, for example, a new sample is taken every 20 times (5 out of 100) that a specific co-phase matrix entry occurs during simulation.

5.5 Estimating Performance with a Static Co-Phase Matrix

We now describe using a static co-phase matrix to guide simulation. For each co-phase that could occur in the matrix we simulate the programs together at representative simulation points. We used the SimPoint algorithm to find the representative simulation points from each phase. The simulation points might not actually co-execute during the baseline comparison we are trying to model, but they embody an average behavior for that co-phase that makes them representative. For this approach, only a single sample is used for each co-phase matrix entry from running the SimPoint simulation point combinations together.

To arrive at an overall IPC, the multi-program workload will start its execution in one of the co-phase matrix entries. Using that co-phase matrix entry and the individual program phase-ID trace information, we predict how many cycles until the next phase change and the number of instructions to be executed from each program. We then advance to a new co-phase simulation matrix entry and advance each program by its number of estimated instructions executed, and we repeat this process. Once we reach the stopping criteria for our simulation (*e.g.*, reaching a total number of simulated instructions or a minimum number from each thread) we have the total number of cycles to execute the workload and the number of instructions executed per thread. Note, this analytical simulation is done *after* we have the co-phase matrix. It requires neither functional nor detailed simulation after the static matrix has been created. At this point the analytical simulation is completely driven by the co-phase matrix.

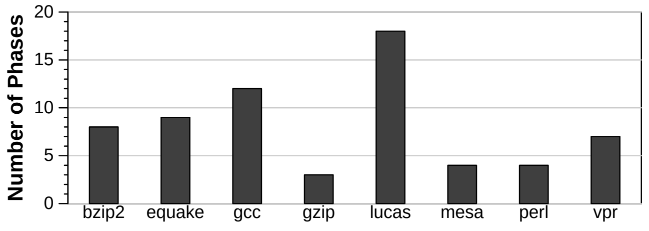

Figure 4: Number of phases found for each program.

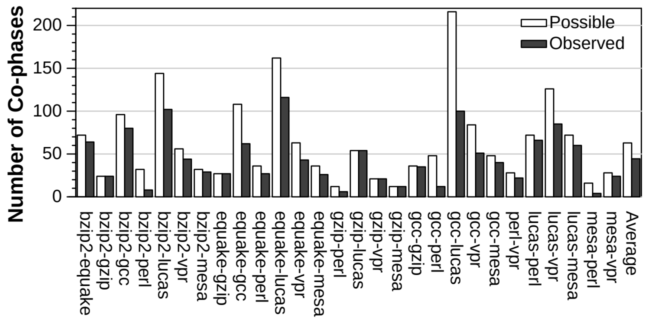

Figure 5: Number of phase combinations that could have occurred and the number that actually occurred during detailed simulation.

This method is extremely efficient when used with checkpoints. One set of checkpoints suffices for each combination of programs, and can be used for every architectural change to be examined. For a particular architecture configuration, after simulating each phase combination once for just tens of millions of instructions, we can then arrive at the estimated performance results for all possible thread starting offsets. With checkpoints, these simulations may be done in parallel if there are sufficient resources. Unfortunately, if the number of threads is large or they have complicated phase behavior, there will be too many potential co-phase matrix entries to simulate. In this case, the dynamic approach described above for filling in the co-phase matrix to guide sampling and fast-forwarding would be preferred. Alternately, the static co-phase matrix can be collected as each entry is needed during simulation and the partial static co-phase matrix can be shared and updated by several simulations concurrently.

6 Results

All of our results are compared against complete detailed simulation of the 10 billion instruction interval described in Section 3. Each workload is labeled with thread 0 first and thread 1 second, so thread 0 in `gcc-vpr` is `gcc`, and thread 1 is `vpr`. We examine both the overall and per-thread IPC. Although the overall IPC is of importance, it is also critical that the per-thread performance be accurate so that our simulation model can be used to study throughput, fairness, perceived user time and scheduling. Additionally, if a simulation method weights two executing threads differently than would occur in practice, then it may be effectively simulating a dif-

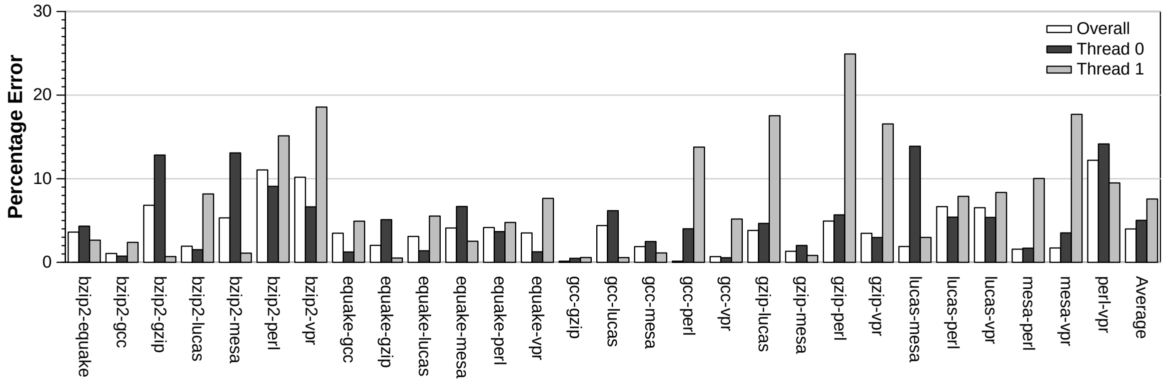

Figure 6: Error in IPC for co-phase matrix simulation using the dynamic co-phase matrix with 1% Phase sampling.

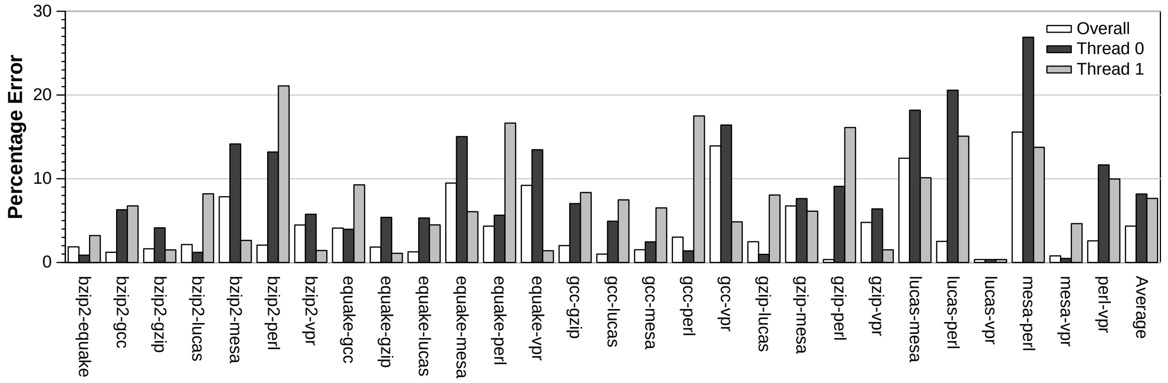

Figure 7: Error in IPC for co-phase matrix simulation using the Static co-phase matrix.

ferent workload than would occur naturally.

The warmup technique we use for the start of a sample in this paper is what we call *Cold Start Hit*. Each cache block uses a warmup bit to that indicates the first time an entry is used. When starting the simulation of a sample, the first access to a cache block is assumed to be a hit. This a very simple method but it provides reasonable warmup accuracy, since the miss rates for caches is usually fairly low.

6.1 Phase Selection

For each program to be run in the multi-program workloads, we gathered the Basic Block Vectors for that program and identified the phases and simulation points as described in [11]. To ensure that fine-grained behavior would be evident we used 10 million instruction intervals. We used the SimPoint tools to find up to 20 phases (max K was set to 20) in each program. In Figure 4, we show the number of phases that were actually found for each program.

During the detailed simulation of a two-program workload, in the worst case, the number of possible co-phases between the two different programs is the product of the number of simulation points for each program. For example, in Figure 4 we see that `gcc` had 12 phases and `vpr` had 7, so the total possible number of co-phases is 84. Figure 5 shows the maximum possible number of co-phase identifiers for each two program combination we examined, as well as the number observed during simulation. Even though `gcc-vpr` had 84 possible co-phases, only 51 of the possible pairings occurred during the baseline simulation we performed. Figure 8 shows that the overall, and per-thread IPC of each of the program combination. The pairs varied greatly in performance, with overall IPC between 2.2 and 4.5 and single-threaded performance between 0.7 and 3.2. Depending on its partners, a program's IPC can vary by as much as 1.3.

6.2 Picking a Sample Size

For each sample gathered during SMT simulation, we considered the following three sampling methods. In these sampling techniques, a sample stops when one of the conditions below is met:

Total 5M: total 5M instructions are committed,

First 5M: a single thread commits 5M instructions,

Both 5M: both threads commit at least 5M instructions.

The Total 5M method performed worst because it executes the fewest instructions. There is a similar problem with First 5M, which may be dominated by the best-performing thread. In cases where one thread significantly outperforms the other this could lead to unbalanced sampling. Both 5M was the overall best method with only `gcc`'s frequent phase changes as notable weakness, so in our dynamic co-phase matrix experiments we sample long enough that each thread executes at least 5M instructions during the sampling. When populating a co-phase matrix entry, it is not always possible to sample that long before hitting a co-phase change. When this occurs,

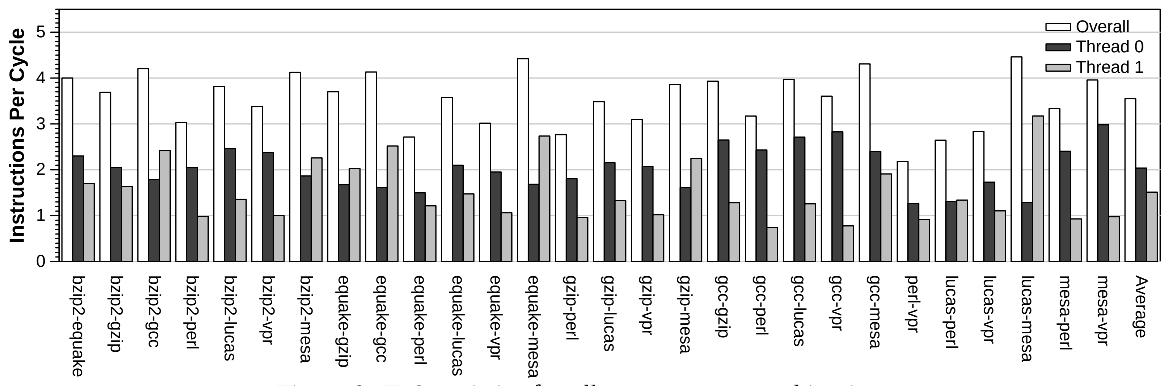

Figure 8: IPC statistics for all two-program combinations.

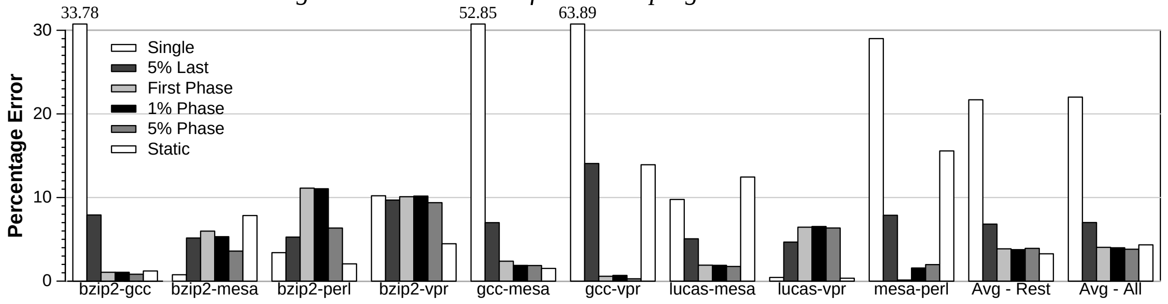

Figure 9: Overall IPC error comparing the different SMT sampling techniques.

we take additional samples when the co-phase re-occurs and combine them with the earlier sample results. This avoids the problems with frequent phase changes. With resampling techniques, the additional samples do not need to contain 5M instructions for each thread.

The experiments with static co-phases matrices do not have the advantage of sampling from multiple locations, so they use Both 10M. The faster thread can overrun the end of the co-phase but longer samples seem to be an advantage overall. Experiments with many different ways to terminate a sample (including all of the 5M methods just mentioned) demonstrated that no technique is best for all pairs of programs.

6.3 Pairwise Simulation Results

We first examine the error in IPC seen using the dynamic and static co-phase matrix simulation approach described in Section 5. Figure 6 shows the error in IPC when guiding simulation dynamically building up the co-phase matrix using 1% sampling as described in Section 5.4. Figure 7 shows the error in IPC when using the static co-phase matrix to guide simulation. Results are shown for overall IPC error and per-thread error. Overall, for both techniques, the average error rate was 4% and the average per-thread error rate was below 8%. In the worst case (`mesa-perl`), the overall error was 16%, and the per-thread error was 27%.

Figure 9 compares our dynamic and static co-phase matrix approaches with alternate sampling techniques described in Section 4, concentrating on overall error in IPC. The first bar shows the result of co-simulating the best Single simulation points until one thread executes 100M instructions. 5% Last shows the estimated performance, where we regularly sample the per-thread IPC to predict how far to fast-forward each thread to the next sample. Each sample simulates until both threads execute at least 5M instructions. 5% of the workload has detailed simulation performed on it, and the rest of execution is skipped via fast-forwarding.

The next bar, First Phase, shows our dynamic co-phase matrix approach, where sampling is done for a co-phase matrix entry until a total of 5M instructions are simulated for each thread. This represents taking the 1st sample for each co-phase matrix entry, and using that to guide fast-forwarding for that entry for the rest of the simulation. The N% Phase methods resample each co-phase, where detailed simulation is performed for 1% or 5% of the workload's execution, and the rest is fast-forwarding. For this approach new samples are combined with the old ones. Finally, Static uses the static co-phase matrix.

The Single SimPoint method produced errors much greater than would be expected when using single SimPoints in the single-threaded case. Even though the intervals are representative of the overall instruction mix, the pairing of single simulation points cannot capture the complexity of ongoing phase interactions. The high single-thread errors show that pairings can be quite atypical. In terms of average error rate, phase-based sampling techniques are significantly better than the Single and 5% Last techniques.

6.3.1 Relative Progress Graphs

To better explain and compare the errors seen in Figure 9, we now examine the relative progress of each through execution using a *Relative Progress Graph*.

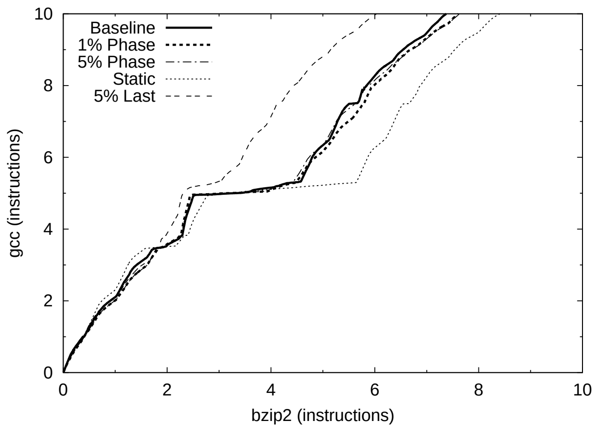

Figure 10: Relative progress for `bzip2-gcc`.

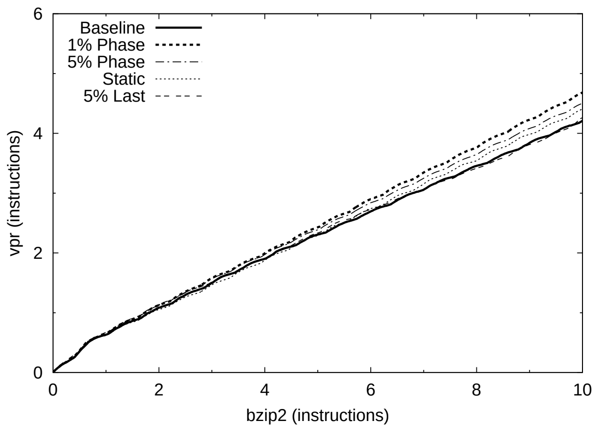

Figure 11: Relative progress for `bzip2-vpr`.

Figures 10 and 11 compare the baseline execution of program pairs with the estimated execution derived from sampling. A point plotted at (x, y) indicates that when thread 0 has executed x instructions, thread 1 has executed y instructions. The solid line represents the baseline detailed simulation and the broken lines represent the progress estimated by our various sampling methods. The slope of the line at a point indicates the relative IPCs of the two programs, but actual IPC values cannot be derived from this graph because it does not show how many cycles have executed. The graph can be used to verify that the sampled runs enter the correct co-phases together. The ideal behavior is for the sampling approach to identically follow the baseline run.

In Figure 10 we examine the co-execution of `bzip2` and `gcc`. This pairing has the most dramatic variation in relative IPCs amongst our test cases, as can be seen by the nearly horizontal and vertical line segments midway through execution. Each sharp bend in the graph represents a transition between significantly different co-phases. It is immediately obvious that the last sample method does not track the behavior of the baseline (the top line in the graph). It does a reasonable job of tracking the sharp bend at 2.5 billion instructions, but because it has already deviated from the baseline it does not match the nearly horizontal phase for even a third of its length, an error which cannot be corrected. The static method (the bottom line in the graph) sampled unfortunately just before the nearly vertical section, causing it to switch to the horizontal too late and hence continue in that co-phase too long. Despite this error it manages to track subsequent phase changes fairly accurately. All of the other methods make better use of phase knowledge, ensuring that they take samples at the phase combinations. The slight deviation from the baseline is determined by how representative of the entire phase are the samples. The close tracking of the baseline indicates that sufficiently many phases were used for both programs.

Figure 11 shows a poorly-performing pairing with about 10% error in the non-static methods. Here the phase behavior settles into an undulating line after the first billion instructions. Although Last 5% sampling looks the best here, it can be seen that this is almost an accident. It crosses the baseline many times but rarely has a similar slope; the frequent variations caused by `bzip2` repeatedly send it back towards the baseline but it does not correct its course again until it is too late. These errors make its IPC estimate no better than that of the other techniques. The phase-based techniques track changes in the baseline admirably, but these runs consistently diverge over time, suggesting that the samples are slightly off from the average trend. The static method gets an advantage in this case by not taking the earliest samples, keeping its error in `vpr`'s execution much lower.

In each case it is clear that attention to phases provides essential information for tracking SMT execution behavior. Regular sampling without phase information succumbs to error for reasonable sampling frequencies. Shorter samples would allow more frequent sampling but would decrease sample accuracy; this will only give reasonable results for program pairs with consistent behavior such as `bzip2-vpr`, but not for complex combinations such as `bzip2-gcc`.

The analysis for `bzip2-vpr` indicates that advanced warmup techniques may improve our results. Functional warming during fast-forwarding, as used by Wunderlich *et al.* [16] is a promising technique. Keeping caches and branch predictors warm has a modest impact on the performance of fast-forwarding, so the increased execution time should not outweigh the anticipated benefits to accuracy. SMT functional warming requires that state updates from the threads be interleaved, so each thread will need to be fast-forwarded in small increments according to their estimated relative IPCs.

6.3.2 Variability of Co-phase Samples

We now examine the benefit of using co-phase information to guide sampling. We find that samples taken from co-phases exhibit less variation than samples taken across all co-phases during our baseline runs. We break the execution of an SMT workload into 10M combined instruction intervals. We then calculate the Coefficient of Variation (CoV) over all of these intervals of execution, as shown in Figure 12. This represents the variation seen when randomly sampling over the complete baseline execution of the workload.

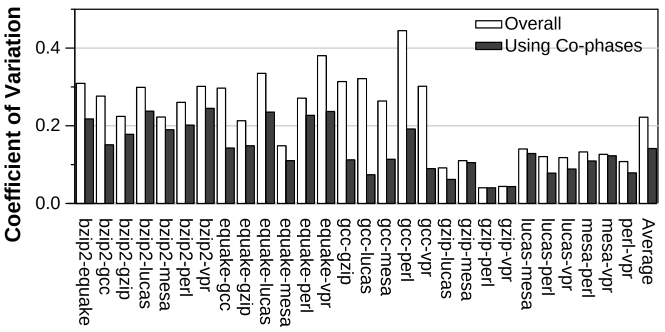

Figure 12: Coefficient of variation improvements through phase separation.

Next, we compare this to the variance in samples seen when guiding sampling with the co-phase matrix. To calculate this, we bin all of the SMT workload intervals into co-phase matrix entries, and we calculate the CoV of each co-phase matrix entry. We then weight each CoV based upon the amount of execution each phase accounted for. These weighted CoVs are then combined to arrive at an average CoV that would be seen when gathering samples based upon the co-phase matrix. The co-phase CoV is also shown in Figure 12. The results show that the co-phase sampling always made an improvement, reducing the variance by one-third on average.

We also verified that the co-phase methods are suitable for comparing different simulator configurations. We ran different pairs of programs through eight configurations, varying the size of the L1 caches, the L2 cache and the branch predictor tables. Timing parameters were changed so that the smaller structures had lower latency. The `bzip2-gcc` pair was particularly sensitive to architectural changes so in Figure 13 we show real and estimated overall and per-thread IPC. The Static Phase method is particularly consistent in its error, and the magnitude of the error is relatively constant over most of the configurations. It changes at the 5th processor configuration, where the error drops to nearly zero.

6.4 Four-Context Simulation Results

We also examined applying our co-phase matrix approach to a four-context SMT processor. Table 4 shows the the overall IPC and per-thread IPC for the four program combinations we examined. The last two columns show the total possible number of co-phase combinations and the number observed during simulation.

Figure 14 shows the overall IPC error rates for the four program combinations we examined. We compare the results for Single, 1% Phase, 5% Phase and Static methods. The results show that when using all four contexts the machine is completely resource constrained so errors in different threads balance out, making the overall IPC error negligible.

Figure 15 shows the detailed breakdown of per-thread error for one combination. Results are shown across the different sampling techniques for the overall error, and per-thread error for `equake`, `gzip`, `lucas`, and `perl`. As in the two-threaded case, `perl` is a significant source of error because of the limited number of phases found. Static errors are more pronounced than the other methods, unlike the situation with two threads, because the larger number of programs produce shorter co-phases, increasing the probability that a the static sample will not match the dynamic instances of the co-phase. This is similar to the problem noted with `gcc`'s frequent phase changes, but worse. Nonetheless, the magnitude of the errors is smaller than in many two-threaded cases because more of the simulated machine's resources are used at capacity, allowing less variation in performance.

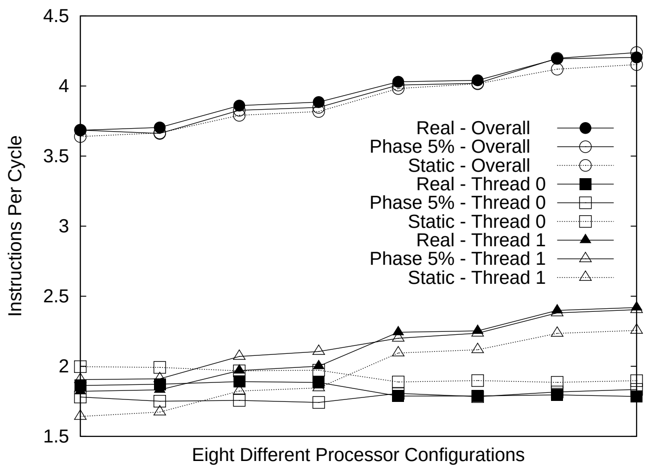

Figure 13: Overall and per-thread performance for `bzip2-gcc` *under different architecture configurations.*

7 Summary

Simultaneous Multithreading architectures are appearing in commercial processors, yet there is still relatively little support for sampling or determining where to simulate to achieve representative simulation results. The challenge in creating a sampling approach to SMT is in determining how far to fast-forward each individual thread between samples. This distance will vary between different architecture configurations and as the threads execute through different phases of execution.

In this paper, we present the co-phase matrix method for sampling the execution of Simultaneous Multithreading machines. Our simulation approach builds a co-phase matrix and uses it to guide fast-forwarding between samples. In performing detailed simulation using the co-phase matrix, we were able to estimate the IPC for multi-program workloads with an average error of 4% when using the 1% co-phase sampling approach. Our static co-phase method allows parallel simulation and can estimate workload performance from all possible thread starting positions with just 4.3% error. We also showed that the co-phase matrix can be used to estimate the progress

Programs	Instructions Per Cycle					Co-phases	
	Overall	Thread 0	Thread 1	Thread 2	Thread 3	Possible	Observed
bzip2-equake-gcc-lucas	5.09	1.41	1.07	1.68	0.93	15552	469
bzip2-gcc-mesa-vpr	4.56	1.20	1.64	1.19	0.53	2688	305
equake-gzip-lucas-perl	4.03	0.96	1.31	0.99	0.77	1944	375
gzip-mesa-perl-vpr	3.65	1.15	1.41	0.61	0.48	336	62

Table 4: IPC and number of co-phases found for each set of four programs.

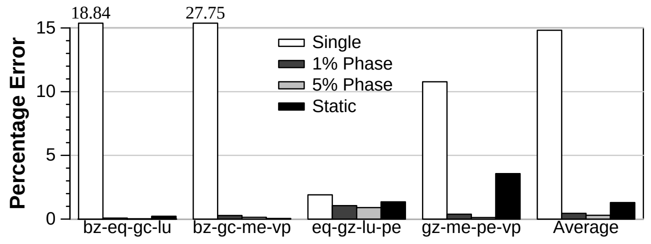

Figure 14: Overall IPC error rates for four four-threaded combinations.

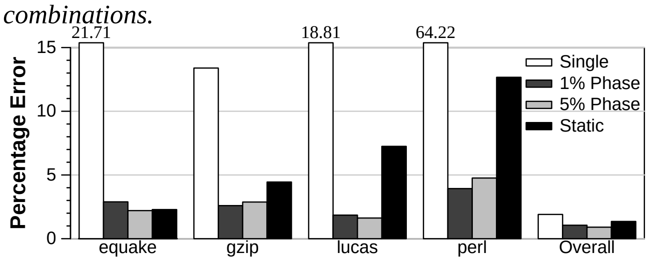

Figure 15: Per-thread IPC accuracy for the `equake-gzip-lucas-perl` *combination.*

of up to four threads on an SMT machine. This has lead to future work including using our co-phase matrix to guide the symbiotic scheduling of threads for SMT.

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. CCR-0311710, NSF grant No. ACR-0342522, UC MICRO grant No. 03-010, and a grant from Intel and Microsoft.

References

[1] James E. Smith Ashutosh S. Dhodapkar. Comparing program phase detection techniques. In *36th International Symposium on Microarchitecture*. ACM, December 2003.

[2] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-oriented full-system simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, February 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)*, October 1996.

[5] J. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 2001 International Conference on Computer Design*, September 2001.

[6] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003.

[7] 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*, September 2000.

[8] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, University of Washington, 2000.

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

[10] 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.

[11] 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.

[12] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, pages 336–349. ACM, June 2003.

[13] 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, November 1999.

[14] 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.

[15] 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.

[16] Roland E. Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *30th International Symposium on Computer Architecture*, June 2003.