Phase Guided Sampling for Efficient Parallel Application Simulation

Jeffrey Namkung
Dohyung Kim
Rajesh Gupta
University of California
San Diego
jnamkung@ucsd.edu
dhkim@ucsd.edu
rgupta@ucsd.edu

Igor Kozintsev
Jean-Yves Bouget
Intel
igor.v.kozintsev@intel.com
jean-yves.bouguet@intel.com

Carole Dulong
Google
cdulong@google.com

ABSTRACT

Simulating chip-multiprocessor systems (CMP) can take a long time. For single-threaded workloads, earlier work has shown the utility of *phase analysis*, that is identification of repetitive program behaviors, in reducing overall simulation time while maintaining an acceptable loss in accuracy. To cope with multithreaded workloads, a combination of phases from all executing threads must be taken into consideration since inter-thread interference may distort the homogeneity of each phases' true performance. Unfortunately, phase analysis does not work for multithreaded (MT) workloads because the possible phase combinations in an inherently nondeterministic execution model grows exponentially with the number of threads. To this end, we propose a new technique to reduce the number of simulation samples by **synthesizing** samples from similar phase combinations. We present a simple cost function for measuring the similarity between phase combinations and by using the individual thread samples from the similar phase combinations, a new sample can be constructed. This cost function provides a convenient control knob for exploiting tradeoffs between simulation speed and accuracy. Our experimental results show that in most cases, properly setting the cost function's threshold can yield a reduction in sampling by 90%, while maintaining error to less than 5%.

1. INTRODUCTION

Chip multi-processor (CMP) systems are gaining attention due to their ability to provide increasing system performance through silicon integration capabilities. Exploring and evaluating different architectural designs for such systems depends crucially upon the ability to simulate applications that use multithreaded software models quickly and accurately. This is an increasingly difficult task: in case where prototype systems are not available, simulation models in use are orders of magnitude slower.

Phase analysis [12] was shown in [10] to provide an efficient solution for reducing time spent simulating single-threaded workloads. Biesbrouck et al [4] showed how to extend phase analysis for simulating a simultaneous multithreading (SMT) processor model running multiple workloads by using the *co-phase matrix*, a matrix representing phase information across multiple threads. The co-phase matrix addresses the need to consider the combination of phases that can occur together during execution. Simulating CMPs running MT workloads differs from simulating SMTs running multiple single-threaded workloads in three aspects. First, threads running on separate processors have less shared resource contention than threads running on a SMT processor. Whereas SMT processors share functional units, branch prediction units, all caches, memory, and the communication subsystem, CMPs only share the shared caches, main memory, and the communication subsystem. Second, the threads in an MT workload communicate and share data with each other, whereas separate independent workloads do not. Thus, inter-thread interference may also be caused by cache coherency behavior and data dependencies between threads. Lastly, CMPs can scale to a much larger number of threads than SMT processors. The target of our simulation technique is CMP.

We first show that a straightforward extension to phase analysis using the co-phase matrix for simulating MT workloads does not scale well with the number of threads. Our experiments show that the possible reduction in time spent simulating quickly diminishes beyond 16 threads due to a rapid growth in the number of unique phase combinations. To attack this problem, we propose a new technique for reducing the amount of simulation required by dynamically synthesizing samples from previously collected samples during simulation. Our experimental results show that our technique scales well, which further sustains the potential usefulness of phase analysis in the context of MT workloads.

The remainder of this paper follows in Section 2 with a background on phase analysis and how it can be extended to simulate multiple threads using the co-phase matrix. Section 3 identifies the main problem of explosion in samples required when applying this straightforward extension to phase analysis for MT workloads. We propose a new tech-

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

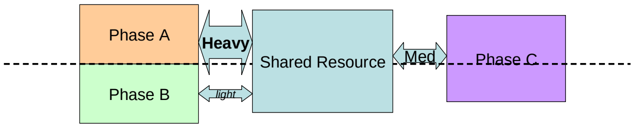


Figure 1: Example on how phases may affect each other through shared resource contention. Thread 1 (left) transitions from phase A, which requires heavy access to the shared resource, to phase B, which has light access. Thread 2 (right) concurrently accesses the same shared resource with medium access.

nique called **Sample Synthesis** in Section 3.2 to address this problem and show in Section 4 that our technique gives promising results in both reducing simulation time while maintaining acceptable accuracy. A discussion of related work is presented in Section 5. We conclude with a summary and future directions in Section 6.

2. BACKGROUND

2.1 Phase Analysis

In [12], single-threaded workloads were shown to exhibit repeated phases of execution, where phases represent similar executed code signatures. Furthermore, the authors note that similar code signatures exhibit approximately homogeneous performance characteristics, such as IPC, branch misprediction rates, etc.

To identify repetitive phases, the code signatures are profiled by first splitting the entire trace of executed instructions into a sequence of intervals. The length of each interval may be fixed or variable length [7] and is typically measured in terms of how many instructions were retired within that interval. For each interval, a lightweight profiler is used to collect statistics of each unique basic block's execution count. In this manner, each interval's code signature can be represented by a *basic block vector* – a frequency vector representing the set of basic blocks that executed within that interval and how often.

After all basic block vectors have been generated for all intervals, the entire set is run through a K-means clustering to group similar intervals together, where each group/cluster represents a unique phase of execution. The basic block vector closest to the center or *centroid* of the cluster can then be used as the *phase representative* for all other intervals assigned to the same cluster.

To reduce simulation time, a detailed timing-accurate simulator (*performance simulator*) can be used to simulate intervals selected as phase representatives, i.e. *sampling*. For the other intervals, a *functional simulator* may be used to *fast-forward* until either the simulation has finished or another phase representative is encountered. Since the functional simulator is much faster than the performance simulator and the number of phase representatives is much smaller than the total number of intervals, total simulation time can be greatly reduced.

2.2 SMT Phase Analysis

Biesbruck and co-authors in [4] proposed extending phase analysis for guiding sampling of multiple workloads running on a SMT processor. The main problem they identified was that phases do not exhibit homogeneous performance characteristics when other phases from other programs/threads exist on a platform with shared resources.

Figure 1 shows an example on how the existence of different phases concurrently executing may affect performance. On the left side of this figure, we see that one thread switches from phase A to phase B. On the right side, the second thread is in phase C. At some point in time before phase C finishes, thread one switches to phase B, which has a different shared resource usage. Thus, we can expect that phase C's performance will be different due to the different shared resource access characteristics of phase A and B.

The authors concluded that the individual phase a particular workload is currently executing is not enough to determine the performance. Rather, the combination of phases that are concurrently running are necessary to determine the performance of each workload. In other words, phase representatives must be redefined as the unique phase combinations that occur during execution.

To capture repeated phase combinations, [4] introduced a mechanism called the co-phase matrix. Because, phase performance behavior is dependent on the phases of other threads, it is important to know *when* and *which* phases co-occur. In the single-threaded context, the intervals execute in a sequential order, allowing static determination of when and which phase representative occur prior to sampling. In contrast, with multithreaded workloads, phase combinations require a notion of time between concurrently executing threads, which is unavailable before sampling. Thus, the co-phase matrix acts as a mechanism that allows us to *dynamically* determine the phase representatives during simulation.

The co-phase matrix acts as a look-up table, which stores the performance samples for each thread-phase pair indexed by each unique phase combination. Every time a new phase combination is seen, the performance simulator is run and a sample for each thread-phase pair is collected. If a phase combination already has an entry, then we can reuse each sample for each thread-phase pair to fast-forward each thread.

3. MULTITHREADED PHASE ANALYSIS

This paper builds upon previous work done in [11], which extended how to perform the first-pass of phase analysis for multithreaded workloads. Intervals are defined on a per-thread basis by only profiling the code signatures for each thread independently.

Frequency vectors are calculated by sampling on a real multi-processor system the hardware performance counters resident on the processors [1]. The hardware performance counters provide a means to sample the program counter, rather than basic blocks, at regular intervals and also provide the value of the time-stamp counter when each sample is taken. In this manner, we are able to collect the frequency vector for each interval along with the performance profile, which was later used for validation and experimentation.

To detect the phases, the entire set of frequency vectors collected for all threads is run through a clustering algorithm. Subsequently, each thread is associated with a sequence of their constituent intervals and the phase each interval was assigned to.

For our experiments we choose the NAS OpenMP benchmark suite [5] shown in Table 1 as our MT workloads. These benchmarks were chosen due to their ability to scale to a high number of threads and their high degree of data par-

Benchmark	SAV	Samples/interval	# phases
bt.A	10^6	100	6
cg.B	10^6	100	7
ep.B	10^5	1000	5
lu.B	10^6	100	4
mg.B	10^5	1000	8
sp.A	10^6	1000	9

Table 1: Benchmarks used and experimental settings

allelism; OpenMP primarily targets loop-level parallelism. The settings used for hardware sampling are also shown in Table 1. The number of phases chosen for clustering are the same as those used in [11] and were determined through visual inspection of the clusterings. The Sample-After-Value (SAV) represents when the hardware performance counters were sampled based upon a specified number of instructions retired. The values were selected based on the length the benchmark and was chosen to (1) provide enough samples for clustering and (2) allow clustering to finish in reasonable time. For most benchmarks, we split the execution into intervals of 10 million instructions and we used the sampled program counters for building our frequency vectors. Column four shows the number of phases used for clustering during phase analysis. The machine used for hardware sampling consisted of 16 Intel Xeon processors running at 3 GHz in a clustered configuration.

We implemented a simple timing model to gain an understanding of inter-thread phase behavior. The timing model is implemented by loading for each thread a phase trace representing the sequence of intervals for that thread and the phases those intervals were clustered to. Additionally, each interval is assigned the IPC profile measured for that interval during the hardware sampling mentioned earlier. This allowed us to correctly align each threads' intervals with respect to each other in time and identify all phase combinations that occur during execution.

3.1 Phase Combination Growth

To get a notion of how much time would be required for simulation (only sampling unique phase combinations), Figure 2 plots the ratio of unique vs. total phase combinations seen as we varied the number of threads from 1 to 16. When the number of threads is small, the number of unique phase combinations is much smaller than the total number of phase combinations, because the combinatorial growth with only two threads is small. On the other hand, when the number of threads is increased to 16, we can see that out of all the phase combinations seen, a much larger percentage of those combinations are unique. As a result, if we were required to sample all unique phase combinations, the amount of time simulating would remain quite long and in some cases almost as long as a full detailed simulation, for instance in example mg.B.

3.2 Sample Synthesis

The intuition behind our technique is that the degree of resource sharing - thread interference - on a CMP system will be much less than SMT, which share most of the processor's resources. Accordingly, the performance characteristics of a phase is **not** always dependent on **all** other concurrently executing phases and does not *always* change when a new phase combination is encountered. Put in another way, phase combinations that are similar are likely to have homogeneous performance characteristics for the thread-phase pairs that are the same. Therefore, if we can determine which phase combinations are similar, we can avoid sampling by reusing individual thread-phase pair samples from similar phase combinations.

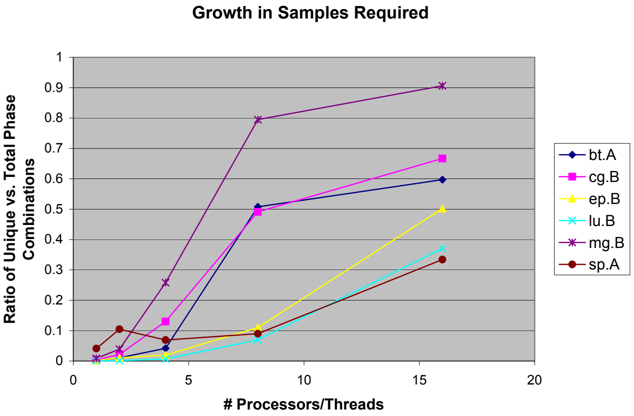
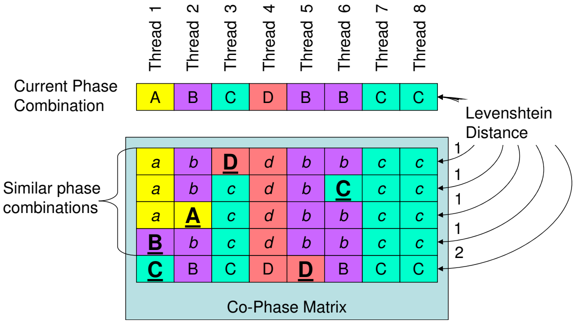


Figure 2: Percentage of sampling required when increasing number of threads

3.2.1 *Similar Phase Combinations*

To calculate the similarity between phase combinations, we compute the Levenshtein distance [8], which is similar to a Hamming distance, where we count the number of different thread-phase pairs between two combinations. Figure 3 shows an example of how we compute similarity using the Levenshtein distance.

Each row in the figure represents a unique phase combination that occurred during simulation. Each column represents the particular thread's phase of execution. The top row represents the current set of co-executing phases for all the threads, while the bottom rows represent previously seen phase combinations and their respective samples stored in the logical co-phase matrix. The Levenshtein distance between the current phase combination and each phase combinations stored in the co-phase matrix is shown by the arcs. Each letter/color represents a unique phase and the bold and underlined letters represent the thread/phase pairs that dif-



	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7	Thread 8	
Current Phase Combination	A	B	C	D	B	B	C	C	
	a	*b*	**D**	*d*	*b*	*b*	*c*	*c*	1
	a	*b*	*c*	*d*	*b*	**C**	*c*	*c*	1
	a	**A**	*c*	*d*	*b*	*b*	*c*	*c*	1
	B	*b*	*c*	*d*	*b*	*b*	*c*	*c*	1
	C	B	C	D	**D**	B	C	C	2

Figure 3: Similarity measured by Levenshtein distance

Algorithm 1 Guided sampling of MT workloads

```
1: while phasesRemain do
2:   $CurrPhaseCombo \Leftarrow GetNextPhaseCombo()$
3:   if HaveExactSample(CurrPhaseCombo) == false then
4:     if SynthesizedSample(CurrPhaseCombo) == false then
5:       $ObtainSample(CurrPhaseCombo)$
6:     end if
7:   end if
8:   $SampleSet = GetSample(CurrPhaseCombo)$
9:   $FastFoward(SampleSet)$
10: end while
```

fer from the corresponding thread in the current phase combination. Thus, for the last row in the co-phase matrix, we can see that thread 1 and 5 are in different phases (C and D respectively) than the current phase combination (A and B respectively); the Levenshtein distance is 2.

Using the Levenshtein distance as our similarity metric, we can now identify which of the previously collected samples are similar by only analyzing their phase combination patterns and using a certain threshold on the distance. For example in Figure 3, if we decided 1 to be the maximum distance allowed for identifying similar combinations, then the top 4 entries in the co-phase matrix are deemed similar. The actual samples for each thread could then be used to synthesize a new sample for the current phase combination without having to run the performance simulator to actually collect the sample. In Figure 3, the samples used for synthesis are in lowercase. In the case we have multiple samples to choose from, we average the samples. On the other hand, if there are not enough samples to fully synthesize every thread/phase pair for the current combination, then we must run the performance simulator to obtain a real sample.

The pseudocode for guided sampling using sample synthesis is listed under Algorithm 1. Lines 1-10 represent the loop that examines each phase combination as they occur during simulation. In Line 2, we check to see if the co-phase matrix has an exact match for the current phase combination. If not, we attempt to synthesize a new sample set (Line 3). If this fails, meaning there were not enough similar phase combinations that have matching thread-phase pairs, then we run the performance simulator (Line 5) and add the sample obtained to the co-phase matrix. Line 8 and 9 simply fast forward each of the threads by the amount specified in the samples.

The main synthesis algorithm (Algorithm 2) is comprised of two separate loops (Lines 2-7 and 8-14). The first loop examines all entries in the co-phase matrix to see which combinations are similar to the current phase combination. These similar phase combinations are added to a set of candidates. In the second loop, we look to see if we can find matched thread-phase pair samples, in the set of candidates, to reconstruct our new sample. If we were able to reconstruct the whole sample, then we return true; otherwise we return false.

Because our timing model did not actually perform simulation, the running time of our experiments reflects the overhead of our technique, which was a matter of a few minutes. In comparison to the lengthy time required for simulation, we believe this overhead to be negligible.

Algorithm 2 Attempt to synthesize sample set for $CurrPhaseCombo$ using Levenshtein distance $threshold$

```
1: $CandidateCombos \Leftarrow \emptyset$
2: for all $Combo$ in $coPhaseMatrix$ do
3:   if LevenshteinDistance($Combo$, $CurrPhaseCombo$) $\leq threshold$ then
4:     add $Combo$ to $CandidateCombos$
5:   end if
6: end for
7: $NewSample \Leftarrow \emptyset$
8: for all $ThreadPhasePair$ in $CurrPhaseCombo$ do
9:   for all $Combo$ in $CandidateCombos$ do
10:    if $ThreadPhasePair \in Combo$ then
11:      $NewSample \Leftarrow ThreadPhasePair.sample$ $ThreadPhasePair$ found in $Combo$ to $NewSample$
12:    end if
13:  end for
14: end for
15: if $NewSample$ is valid then
16:   $coPhaseMatrix \Leftarrow NewSample$
17:   return true
18: end if
19: return false
```

4. EXPERIMENTS AND RESULTS

To examine our technique's efficacy, we utilized the same thread timing model described earlier. Incorporating the co-phase matrix, our timing model processes all threads simultaneously, while collecting samples for phase combinations not previously seen. If a matching entry in the matrix existed or we were able to synthesize a sample, we replaced the performance samples of each of the corresponding intervals with the matching/synthesized sample and adjusted the timing between threads accordingly. This model, while unable to handle direct handling of synchronization and inter-thread data dependencies, provided a good initial estimate on the effectiveness of our technique. Furthermore, we have integrated our technique in a simulation environment, which does handle synchronization and data dependencies, and our preliminary results show similar trends. However, due to space constraints, this paper only focuses on solving the sample explosion problem.

Using the similarity between phase combinations as a knob, we can see the tradeoff between reduction in simulation time and the amount of error in the performance projection. Figures 4, 5, and 6 show experimental results for our benchmarks running on 16 processors/threads configuration, while varying the threshold for the Levenshtein distance. Figure 4 shows how our technique can reduce the total number of samples required while adjusting (by Levenshtein distance similarity) which phase combinations in the co-phase matrix can be used to reconstruct a new sample. The points plotted at a Levenshtein distance of zero, show the required amount of samples when no synthesis is performed.

We can see that for most benchmarks, we can reduce the total number of samples required below 10% when using a Levenshtein distance of 4. Furthermore, the samples are reduced fairly quickly before leveling off. At a certain point, increasing the Levenshtein distance has a much less effect on reducing the number of samples.

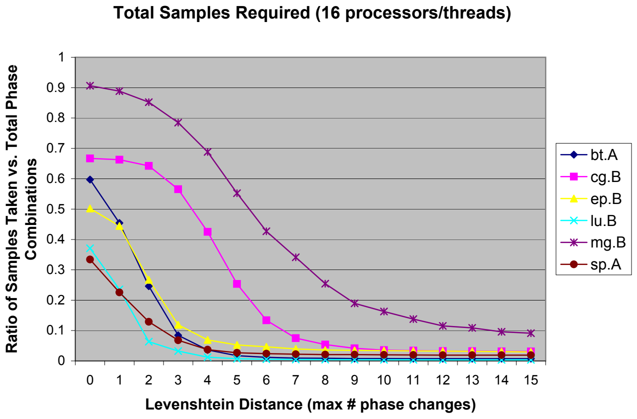


Figure 4: Reduction in samples required as a function of phase combination similarity

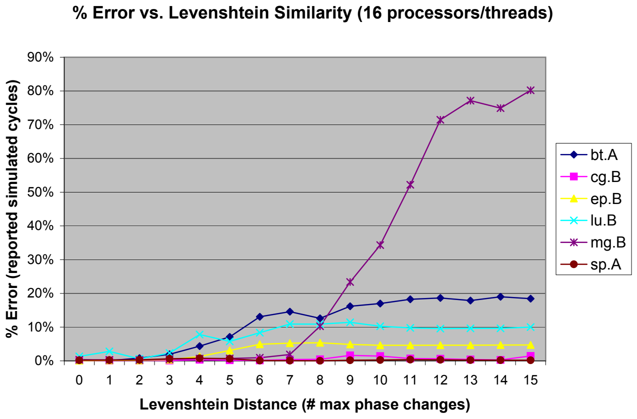


Figure 5: Error in total workload execution time as a function of phase combination similarity

Reducing samples will most likely lead to a greater amount in error, which we measured in terms of reported simulated cycles. Figure 5 shows the error as we vary the Levenshtein distance. The difference in error characteristics for each of the benchmarks as a function of Levenshtein distance, especially mg.B, indicates that beyond a certain point, the individual threads samples used to synthesize new phase combination samples are inappropriate. This further implies that each benchmark has different characteristics in terms of how much inter-thread interference may be present. For example, if each thread in the benchmark has a very low cache miss rate, the amount of interference between threads should be low since the shared resources have less contention.

The main limitation in using Levenshtein distance as a cost function for measuring similarity is that all thread-phase pairs that are different are equally distant from each other by a value of one. Cost functions that provide more indicative measures of similarity between different phases is an area of further investigation, however for this study, the Levenshtein distance showed relatively good results and was simple to implement.

In terms of error, we can see that all of the benchmarks maintain an error rate below 10% at a Levenshtein distance of five and most of them never produce error above 20%. At a distance of three, all of the benchmarks were contained to under 5% error and most of these benchmarks required only 10% of the samples; indicating that re-using previous samples from similar phase combinations can effectively reduce simulation time and still maintain good accuracy.

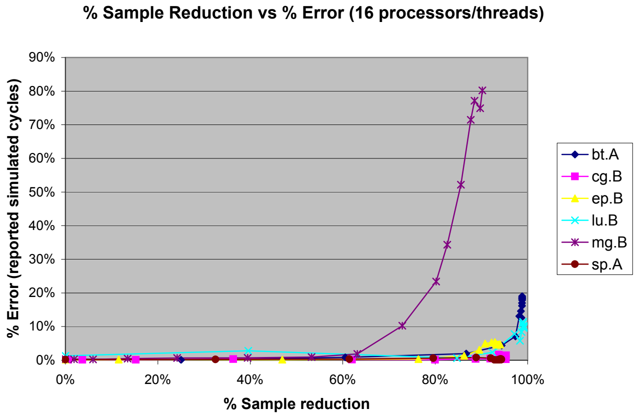


Figure 6: Tradeoff between accuracy and simulation time

It is also beneficial to understand the tradeoffs between how much accuracy we can obtain while reducing the time spent simulating. Figure 6 displays how one might use this chart to estimate how much time they can save at the cost of a loss in accuracy. The points plotted in Figure 6 show the reduction in samples required when compared to the number of samples required without synthesis.

When compared with sampling all unique phase combinations, we can see that synthesis allows all benchmarks to achieve 40% reduction in samples and for most of the benchmarks 90%, while maintaining error rates below 5%. These results confirm that previous samples may be used to synthesize new samples as an effective solution to the sample explosion problem.

5. RELATED WORK

Recent advances in simulation technology have achieved much higher simulation speeds [2] – a high speed commercial simulator indicating an increasing demand for fast and accurate system-level simulation tools. Interestingly, sampling techniques, including our proposed technique, are independent of the simulator. Thus, we believe that our technique is complementary to other simulation environments and may provide even faster performance estimations of MT workloads. More importantly, our technique directly addresses the slowdown in simulation speeds due to modeling a large number of processors, i.e. scalability, which is a problem inherent to CMP simulation.

Most sampling techniques fall into three major categories. The first category contains techniques that simply select a particular point in the benchmarks execution to start sampling. This is the simplest and (unfortunately) most common technique[14]; typically providing poor accuracy since a benchmark's performance behavior commonly has various different phases of execution. The second category of techniques are known as statistical sampling [13], which uses confidence intervals and other statistical properties of the benchmark for finding the optimal sampling length and period. The last category is targeted/guided sampling [12].These techniques utilize characteristics of the benchmark that are independent of the processor being modeled to help determine when to sample. [15]provides a more thorough comparison of these three categories and show that while statistical sampling and guided sampling both provide a high degree of accuracy, guided sampling using phase analysis is more time efficient.

While sampling techniques have been heavily explored for uni-processor/single-threaded benchmarks, only a few recent works have shifted the target platform/application to multi-processor/multithreaded benchmarks. [6] investigated extending statistical sampling methods from [13] to handle multiple threads. In [4] and [3], phase analysis was applied to multiple single-threaded benchmarks running on a SMT processor by introduction of the co-phase matrix.

In [9], the benchmarks are natively executed and through dynamic binary instrumentation via just-in-time compilation methods, program behavior may be extracted for phase analysis. [11] on the other hand utilize hardware performance counters resident on the processors to extract program behavior at high speed. In both works, phase analysis for MT workloads is shown to be able to detect the different phases across threads. The phases are determined by running the same clustering algorithm used in [10] on the cumulative set of basic block vectors for all intervals on all threads, where intervals are defined per-thread. Their experimental methodology enabled them to show low variance of several metrics, e.g. CPI and L3 hit rates, for the phases detected. The authors do suggest the potential utility in using phases to reduce simulation time. However, a simulation solution that uses the phases detected was not proposed.

6. CONCLUSIONS

We have described how phase analysis may be extended to the context of multithreaded workloads running on CMP systems and how current SMT simulation techniques do not scale well with increasing thread counts. Our proposed technique provides an effective solution for reducing the amount of simulation time by reusing samples from similar phase combinations.

While our experimental results are promising, we believe that our cost function in measuring similarity between phase combinations could be more indicative of inter-phase interaction and is under further investigation. In particular, we are currently investigating how to measure similarity by analyzing the memory access characteristics of the program to develop inter-thread dependencies and correlations.

Lastly, we are investigating how to apply phase guided sampling to embedded systems that contain application specific IPs and multiple processors with different instruction set architectures. Embedded systems have a large potential benefit to gain since power is also important to consider; power modeling requires a greater level of modeling detail further lengthening simulation time. Thus, we see that phase guided sampling and the sample synthesis technique presented here can have wide applicability in performance modeling for embedded applications.

7. REFERENCES

[1] http://www.intel.com/software/products/vtune.

[2] http://www.vastsystems.com.

[3] M. V. Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaneous multithreading simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2006.

[4] M. V. Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[5] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. In *NAS Technical Report NAS-99-011*, October 1999.

[6] J. L. Kihm and D. A. Connors. Statistical simulation of multithreaded architectures. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, September 2005.

[7] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical pahse behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[8] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, February 1966.

[9] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *IEEE/ACM International Symposium on Microarchitecture*, December 2004.

[10] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS the International Conference on Measurement and Modeling of Computer Systems*, June 2003.

[11] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *IEEE International Parallel and Distributed Processing Symposium*, April 2006.

[12] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. In *IEEE Micro*, December 2003.

[13] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, 2003.

[14] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *IEEE International Symposium on High-Performance Computer Architecture*, February 2005.

[15] J. J. Yi and D. J. Lilja. Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations. In *IEEE Transactions on Computers*, March 2006.