Dynamic Phase Analysis for Cycle-Close Trace Generation

Cristiano Pereira Jeremy Lau Brad Calder Rajesh Gupta

Department of Computer Science and Engineering
University of California, San Diego
{cpereira,jl,calder,gupta}@cs.ucsd.edu

Abstract

For embedded system development, several companies provide cross-platform development tools to aid in debugging, prototyping and optimization of programs. These are full system emulation systems that can emulate the final binary to be run on the real board, its operating system and devices. Many of these emulation systems do not provide cycle level information due to the time consuming nature of cycle accurate simulation.

In this paper we propose a method to provide Cycle-Close Traces *of cycle-level statistics for the complete execution of the program in orders of magnitude less time than performing full cycle accurate simulation, with an average error of 3.2%. Our approach uses dynamic phase analysis to generate targeted cycle-close simulation samples. Detailed simulation results for these samples are used to produce fast cycle-close traces during a program's execution, so the user can also watch, pause and debug the currently executing code and its corresponding architecture performance characteristics at any point during execution.*

Categories and Subject Descriptors: B.8.2 Performance Analysis and Design Aids, I.6.7 Simulation Support Systems: Environments

General Terms: Performance, Design, Experimentation

Keywords: Simulation, Tracing, Phase, SimPoint

1. INTRODUCTION

The increasing complexity of embedded system design requires efficient methodologies to speed up debugging, prototyping and performance/power estimation of both software and hardware. To address this, several vendors provide cross platform development tools for embedded environments. For example, with these environments a developer can debug an ARM application for Symbian [17] OS with I/O interfaces for a cell phone using a specific ARM board, all running in an emulated environment on an x86 machine. Existing products that provide this functionality include tools from Virtio [20], Virtutech [21], CoWare [2], Accelerated Technologies [1], Vast [19], and other companies. These tools provide the ability to emulate a full embedded development board along with embedded operating systems and I/O devices. This allows the developer to debug and integrate the actual binaries to be run on the real hardware all in an emulated environment.

To make these full software emulation systems usable, the products available [20, 21, 2, 1] provide near-native speed instruction set emulation. This is accomplished through dynamic binary translation methods such as just-in-time (JIT) compilation and caching of translated code, as done in Embra [22] and DynamoSim [7], or through compiled instruction-set simulation [9, 11]. These techniques provide cross-platform execution at nearly the same execution speed as running the application on an actual embedded processor board. These systems provide the user with the ability to pause execution, and examine the currently executing code and its interaction with devices. However, many of these tools lack cycle accurate power and performance evaluation, because performing cycle accurate simulation during the full program's execution incurs an unreasonable performance overhead [16]. In addition, this overhead will increase by several orders of magnitude as embedded processors become more complex and more difficult to model.

The goal of our work is to quickly provide cycle-close traces (run-time performance/power profiles) in these emulation systems. We use targeted sampling exploiting program repetition to construct a complete performance trace of the whole program's execution without running the entire program through cycle accurate simulation. We therefore call this a *cycle-close trace* rather than a cycle accurate trace. A cycle accurate trace comes from performing cycle accurate simulation of the entire program's execution. In comparison, cycle-close tracing builds a performance trace of the entire program's execution from a handful of intelligently chosen cycle-accurate samples. Cycle-close tracing provides accurate predictions of cycle level metrics such as CPI, power consumption, number of cache hits, branch mispredictions, etc. The fast and accurate cycle-close power and performance estimates can be used to validate and debug the functionality of the software, to guide software optimizations, and to perform fast design space exploration of new architectures.

We use *phase analysis* [14, 4, 15] to provide fast cycle-close traces. *Phase analysis* has been used off-line to select simulation points to significantly reducing simulation time. In this paper, we use phase analysis to dynamically choose which samples to simulate. This provides fast and accurate cycle-close traces of the complete program's execution from a small number of samples. The phase analysis and trace is generated as the program is emulated; no off-line analysis is required.

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

For the programs we examine, our approach is 312 times faster on average, compared to fully simulated performance/power traces, with 3.2% average error.

2. DYNAMIC PHASE CLASSIFICATION

In this section we discuss phase behavior, code signatures, and we explain how the dynamic phase classifier works as the program executes. Our work builds upon program phase analysis techniques presented in [14, 15, 5].

Programs exhibit large scale repeating behavior called *phase behavior* [14]. We use the dynamic clustering approach of Sherwood et al. [15] to track phase behavior through the execution of branches. Our dynamic phase classifier assumes no hardware support, and is implemented within the emulation infrastructure.

To detect phases, a program's execution is first divided into contiguous non-overlapping fixed-length intervals. An *interval* is a continuous portion of execution (a time slice) of a program and 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. In our paper, we evaluate fixed-length interval sizes of {100k,200k,400k,800k,1M,10M} and the trade-offs between simulation speed and trace accuracy.

Dynamic Phase Classification [15] partitions a set of intervals into phases with similar behavior. This partitioning is done by dynamically creating a code signature [3] to represent each interval of execution, and intervals that have similar code signatures are dynamically grouped into the same phase. The key observation is that for a given interval the architectural metrics (CPI, number of branch mispredictions, number of cache misses, power consumption) are a function of the execution path (code signature). While demonstration of the correlation between code signatures and architectural metrics is beyond the scope of this paper, we refer the reader to [14, 3] for more discussion.

To perform dynamic phase classification, we need to create code signatures in the emulation environment as we execute the program. For our approach, we create a code signature, which is a vector with N entries (dimensions). This is the approach used in [15]. At the start of an interval, the vector is zeroed. For each branch PC executed in an interval, we hash the branch PC to an entry in the vector, and we increment that entry by the number of instructions executed since the last branch. Thus, at the end of each interval, the vector contains a signature of the code executed in the interval. Figure 1 shows the structure of the dynamic phase classifier. The vector on the left tracks the number of instructions executed between the branches in the current interval of execution.

At the end of each interval of execution, we need to compare the current code signature with signatures from earlier intervals. If the current signature is very similar to earlier signatures, we can re-use the power/performance statistics from the earlier interval. Otherwise, we need to collect power/performance statistics. We use the *Manhattan* distance between code signature vectors to evaluate their similarity. If the distance between two vectors is smaller than a *phase classification threshold* [15], the two intervals are determined to be similar, and they are classified into the same phase.

To compare the current code signature vector with vectors from previous intervals, we must store previous vectors in a table called the *phase signature table*, as shown in Figure 1. Each table entry stores the signature of the phase and its associated architectural metrics. Each table also contains a unique Phase ID, which is used during phase prediction, as described in the next section. At the beginning of program execution, the phase signature table is empty. At the end of each interval of execution, the current branch vector is compared against all branch vectors in the phase signature table by calculating the Manhattan distances. A branch vector matches a phase table entry if the Manhattan distance between the current branch vector and the vector stored in the phase table entry is less than the *classification threshold*. If the current branch vector does not match any signature in the table, the current interval is a new phase, so the current branch vector code signature is added to the table, and is assigned a new phase ID (table entry). If the current branch vector matches multiple phase signature entries (i.e. the Manhattan distance is less than the classification threshold for multiple entries), we choose the phase signature table entry with the smallest distance.

3. PRODUCING CYCLE-CLOSE TRACES

In this section we explain how we use the dynamic phase classifier to provide fast cycle-close tracing for emulation environments. Note that we assume that these environments can switch between fast emulation and processor cycle-accurate simulation [18, 12].

The goal of our approach is to quickly provide detailed architecture metrics (CPI, cache miss rates, energy, etc) for every interval of the program/system execution. To efficiently generate hardware metrics for all intervals of execution, we need to sample (perform cycle level simulation for) one or more intervals per phase using the dynamic phase classifier described in the previous section. We perform cycle accurate simulation on the samples, and then extrapolate the simulation results to the remaining intervals of execution.

3.1 Phase Prediction

Whenever a new phase is detected, we need a cycle accurate simulation sample for that phase. The problem which arises, however, is that the dynamic phase classifier only classifies an interval into a phase *a*fter executing the interval, because we must wait until the end of the current interval for a complete code signature. Therefore, when we detect a new phase, we cannot perform cycle accurate simulation for the new phase, because it is too late. Instead, we must wait until another interval in that phase appears. To determine when to perform cycle accurate simulation for an interval, we use *Phase Prediction*, as in [15].

At the end of each interval we predict the phase of the next interval to determine if we should emulate or perform cycle accurate simulation for that interval. Predicting which phase is next is very important because it determines the number of intervals which are sampled and therefore directly influences the speed at which our tracing mechanism works. The predictor accuracy also influences the number of intervals that are left unsampled.

We experimented with two types of predictors. The first is *Last Value Prediction* and it predicts that the next phase will always be the same as the phase of the last interval executed. The second is a *Run-Length Encoding Markov Predictor*, which encodes the run length of the n recent phases in the prediction (RLE-n). The RLE-n predictor encodes the history of the previous n phase IDs executed along with the run length l of each of the last n phases, which is the number of intervals each phase occurred in a row. Each phase ID and length are hashed together to create an index into the next phase prediction table. Each table entry in the RLE-n

predictor is tagged with this hash, so the RLE-n prediction is only used if there is a tag hit. If there is a tag miss then last value phase ID prediction is used. This approach is used to predict the phase ID (phase signature table entry) for the next interval of execution. We examined several values for n, and from our experiments we concluded that RLE-2 provided the best results. See [15, 5] for more details on the predictor.

3.2 Prediction Guided Sampling

We use phase prediction to determine when to collect a cycle accurate simulation sample. Before each interval is executed, we predict which phase the interval will belong to. To guide sampling, we keep track of which phase IDs already have a representative sample with a flag in the phase signature table. When a new phase is inserted into the phase signature table, it is marked as not having a sample. If we predict that a phase ID without a sample is coming next, then we switch to cycle accurate simulation for that interval. If we predict that a phase ID with a sample is next, then no sample will be taken for that interval and the interval is executed in fast emulation mode. When an interval has finished execution it performs a lookup of its branch vector in the phase signature table. Based on this and the prediction's correctness, we classify each interval into one of the following three categories:

Unsampled - These are intervals, which were emulated based on the next phase prediction, but after performing the phase table lookup there is no sample to be found. When performing the phase table match, the interval's branch vector does not match any phase signature table entry, or there is a match, but that entry does not yet have a sample. To provide architecture results for these intervals, we examine two approaches in the results section. The first is to use the architecture metrics given to the preceding interval. The second is to use the architecture metrics from the interval in the phase ID table that has the *closest match* to the current intervals branch vector.

Sample from Simulation - The phase predictor predicted that these intervals require a sample (the predicted phase ID did not have a recorded sample); cycle accurate simulation is performed for these intervals. Regardless of the correctness of the prediction, the results of the simulation are used for the interval. If the interval matches a current table entry (within the classification threshold), the simulation sample is used to update the table with the simulation results and the signature for that phase ID is also updated. If it does not match a phase table entry, then a new entry is created with the phase signature and hardware metrics stored there. This allows other intervals that are matched with this new phase ID to use that sample for its architecture metrics.

Sample from Match - These are the intervals that were emulated, based on the prediction, and in looking up the branch vector in the phase signature table a saved sample of architecture metrics was found.

3.3 Creating Detailed Cycle-Close Traces

We now put the dynamic phase classifier, phase predictor and sampling approach together to explain the *cycle-close* trace generation process. The emulation system emulates the state of the program's execution, creating branch vectors for each interval and performing next phase prediction. The next phase prediction will predict a phase ID, and in looking up the phase ID in the phase signature table we will decide if the next interval should be sampled or not. Only if the predicted phase ID has not yet been sampled do we decide to sample the next interval. When cycle accurate simulation is chosen, a warmup trace is used to bring the architectural structures to a warm state, and the interval is sampled with cycle accurate simulation.

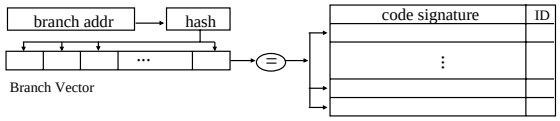

Figure 1: Dynamic phase classifier.

The interval is executed either with or without sampling based on the prediction. At the end of the interval's execution, we now have a valid code signature (branch vector) for that interval. We then look in the phase signature table for a match. If there is no match, we insert a new phase ID into the table. If we sampled this interval, we use the sample to represent this interval's architecture metrics and we update the phase signature table with the sampled architecture metrics and the interval's code signature. If the interval was only emulated and there is a match that contains a sample in the phase table, then that phase table entry is used to provide the architecture metrics for that interval. If the interval was emulated, and if there is a match without a sample or there was no match in the phase signature table, then we use one of the above techniques listed for *Unsampled* to provide the estimated architecture metrics for that interval.

Since we are using small samples (e.g $100K$), before sampling we need to first warmup the architectural structures such as caches and the branch predictor to make sure the performance metrics will be accurately gathered. To accomplish this, while executing the code in the emulation environment, we propose to maintain a circular queue of the last M memory references and the last B branch outcomes. We use these traces to warmup the caches and branch predictor before starting cycle accurate simulation. This requires the profiling of memory addresses and branch outcomes to maintain this circular queue during both fast emulation and cycle accurate simulation. Through experimental results we found that setting both M and B to 50000 provides accurate warmup of the structures for the programs we examined. This resulted in marginal (less than 0.1%) error in the estimated performance metrics as well as a little effect on emulation and simulation time.

4. METHODOLOGY

To evaluate the performance of our approach, we examine MiBench [6] and SPEC benchmarks. The selection criterion was to choose programs which would be typically run on an embedded system platform such as a cell-phone or a PDA (although the technique can be applied to other types of programs as well). We selected 5 programs from MiBench benchmarks: *gs*, *mpeg decode*, *mpeg encode*, *cjpeg* and *djpeg* (the smallest instruction count with half a billion instructions), and 2 programs from SPEC 2000 benchmarks: *bzip* and *gzip*.

Each program was compiled for the Alpha ISA with one input each, for a total of 7 programs. For these benchmarks, SimpleScalar sim-outorder was used for full cycle accurate simulation of each program and collection of code signatures. At every interval of execution, architecture statistics are printed along with a code signature. For the cycle level simulation results in this paper, we simulate a configuration that is similar

I Cache	16k 32-associative, 32b blocks, 1 cycle latency
D Cache	16k 32-associative, 32b blocks, 1 cycle latency
Memory	64 cycle round trip access
In-order Issue	in-order issue of up to 1 operation per cycle
Func Units	1 integer ALU, 1-FP adder, 1 integer and 1 FP MULT/DIV

Table 1: Baseline Simulation Model.

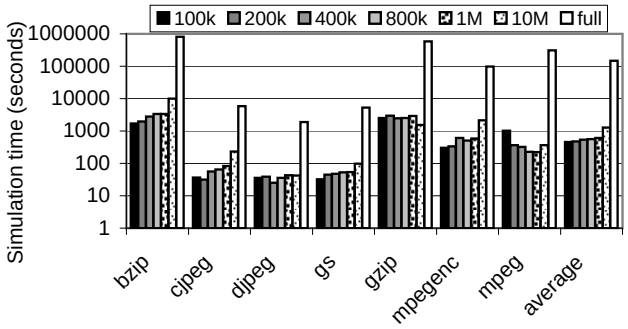

Figure 2: Time in seconds comparing our cycle-close scheme with 25% phase classification threshold for different interval sizes.

to a StrongARM microprocessor with a first level cache using SimpleScalar as shown in Table 1. For the phase classification, we used branch vectors with $N = 32$ dimensions (as in [15]) and a signature table with 1024 entries, which was large enough to capture all of the phase signatures for the programs we examined.

5. CYCLE-CLOSE TRACING RESULTS

Figure 2 shows the total time in minutes it took to perform the combined emulation and sampling for the set of benchmarks we experimented using a 25% classification threshold. Results are shown for a range of interval sizes, from 100,000 instructions to 10 million instructions, with the last column being full simulation. The Figure is log-scale and shows the execution time in seconds. For these results we used warmup traces of size 50,000 events based on experimentation. The time for full cycle-accurate simulation is obtained by running each benchmark to completion on `sim-outorder`. The cycle-close time was obtained by executing a modified Simplescalar to perform the cycle-close tracing described in the previous section.

Figure 2 shows that the average execution time for full cycle-accurate simulation is 72 hours, with `bzip` being the maximum time taking 224 hours, and `djpeg` the minimum time taking 32 minutes. Using our cycle-close tracing for an interval size of 200k instructions with a 25% classification threshold provides an average simulation time of 8 minutes, with `gzip` taking 50 minutes in the worst case, and `djpeg` taking less than one minute in the best case. We achieve orders of magnitude improvement (312 times faster on the average) in comparison to full performance/power traces with 3.2% error rates.

It is important to note that these results are taken when running SimpleScalar, which is a simulator that is built to model wrong path and out-of-order execution. This adds complexity to the simulator, which is the reason it takes so long to perform cycle accurate simulation of a StrongARM for the complete execution of the program. If one was to build a StrongARM simulator from scratch, it would only have to model a simple in-order, small number of stages, pipeline,

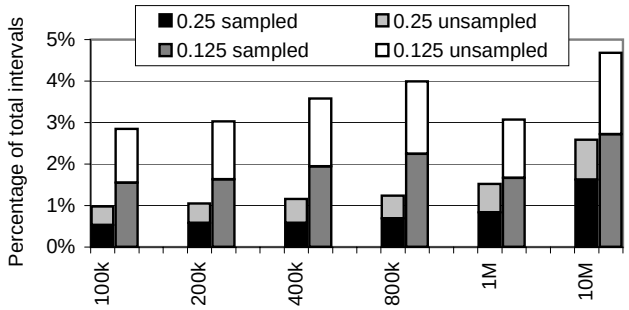

Figure 3: Percentage of intervals sampled and unsampled using RLE-2 prediction with 25% and 12.5% classification thresholds and for different interval sizes.

which will be significantly faster. Even for a tailored in-order simulation model, our cycle-close tracing approach should see similar reductions in simulation time. In addition, our results are more indicative of the cycle accurate simulation time that will be needed as the complexity of embedded processors grows, especially for the cell phone market.

In Figure 2 we can see that there is a trend to increased execution time as the size of the interval increases. This is somewhat counter-intuitive because the number of intervals increases by a lot from 10M to 100k, which should result in more phases and more sampling. However, we found that the number of phases does not increase proportionally to the number of intervals in the program. It increases much slower for most of the programs. Since the amount of sampling is closely related with the number of phases, bigger intervals end up sampling more instructions, which leads directly to longer execution times. For some combinations of programs and interval sizes though, it is the case where increasing the interval size decreases the number phases more dramatically. This has to do with how well the phase behavior of the program aligns with the boundaries of the fixed intervals. Some examples are `mpeg/100k`, `gzip/400k` and `mpegenc/800k`. In general, however, the execution times decrease as interval size decreases.

Figure 3 shows the average percentage of intervals classified as *Sampled from Simulation* and *Unsampled*. The bottom of each stack bar shows the average percentage of intervals in which cycle accurate simulation was performed. The top of each stack bar the percentage of intervals that were emulated and had no sample available from the phase signature table. We can see for 12.5% and 25% *classification thresholds* an increase in the number of intervals *Sampled* as the size of the interval increases. The same thing happens with the percentage of *Unsampled* intervals. For interval size of 1M instructions the percentage of sampled/unsampled intervals decreases slightly for 12.5% threshold because of a significant decrease in number of phases, since the interval is aligned better with the periodic behavior of the program. For this interval size, an average of 0.8% intervals are sampled for 25% threshold and 1.9% for 12.5% threshold.

The existing full software emulation tools already provide the ability to show the user exactly where in the code the current point of execution is, and provide the ability to pause, debug and set breakpoints to analyze execution. Many of these tools want to provide detailed architecture metrics such as power/energy, but currently do not do so because of how slow this would make the system. We propose that they add our fast cycle-close tracing presented here to display to the user accurate detailed architecture metrics during each inter-

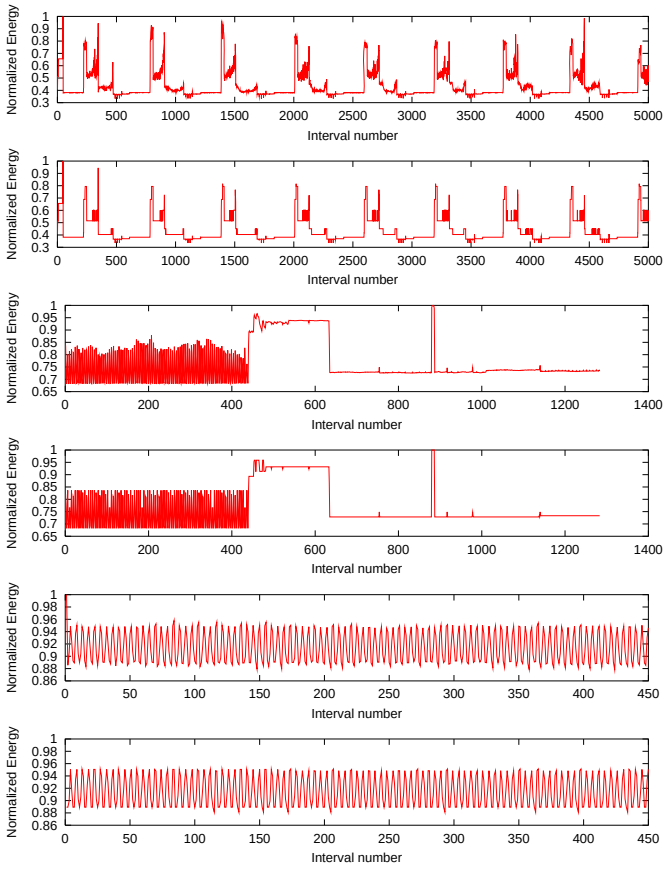

Figure 4: Normalized energy for running bzip for input graphic (top 2 graphs), cjpeg with input gorilla (middle 2 graphs), and djpeg with input gorilla (bottom 2 graphs).

val of execution. They can even display which intervals are sampled from simulation, sampled from phase classification, or were classified as unsampled.

Examples of cycle-close traces are shown in Figure 4 where we use our approach to recreate a trace of normalized energy for the execution of `bzip` (top 2 graphs), `djpeg` (middle 2 graphs) and `cjpeg` (bottom 2 graphs). The energy consumed for each 1 million interval of execution is graphed from the start of execution (0 on x-axis) to the end of execution (far right on x-axis). For each program pair, the top graph shows the actual energy from cycle-close simulation of the complete execution of the program, and the bottom graph shows the energy trace estimated using our approach for 25% phase classification threshold. For `bzip`, we only show a portion (5 billion instructions) of execution to improve readability. These results show that using the phase classification to recreate the detailed trace of the program's execution can achieve a very tight match with full detailed simulation for each interval. We have found similar results for other metrics such as cache hit rates.

To evaluate the accuracy of the traces generated, we compute the average point-wise deviation (APD) for each interval of the program as $\frac{1}{N}\sum_{i=0}^{N}\frac{abs(actual_i - estimated_i)}{actual_i} * 100$, where N is the number of intervals. To use this equation for a given architecture metric, we first gather $actual_i$ for each interval by collecting the actual value of the architecture metric from

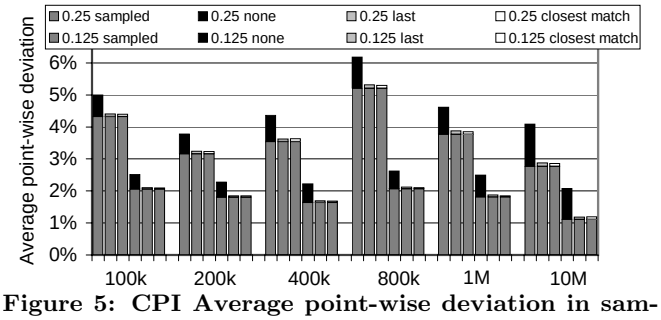

Figure 5: CPI Average point-wise deviation in sampled and unsampled phases.

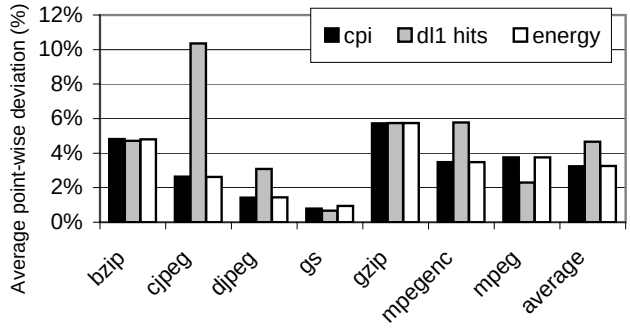

Figure 6: Average point-wise deviation of the cycle-close traces for CPI, energy, and cache hit rates.

full detailed simulation of the program. Remember that each interval also has associated with it a phase ID. Next we gather $estimated_i$ for each interval by using the value of the architecture metric estimated by our cycle-close approach. Finally, we then perform an interval-wise comparison of $actual_i$ and $estimated_i$.

Figure 5 shows the average point-wise deviation for all of the intervals that had a sample (either through simulation or from matching a phase), and the deviation from those that were unsampled. The first three bars show results using a 25% classification threshold and the last three bars are for a 12.5% classification threshold. The three bars show the different options for how to deal with the unsampled intervals. The options are *none*, where no approximation is provided, *last*, where the power/energy statistics from the preceding interval are used, and *closest match*, where we use the power/energy statistics for the previous interval with the closest matching signature. It is clear that providing an approximation is important. The difference between *last* and *closest match* is minimal, implying that *last* should be used since it is a simple technique.

Figure 6 shows the APD error on a per interval basis from using the cycle-close traces for performance, energy, and cache hit rates for a 200k interval size using 25% classification threshold. The results shows that the error for any of these three metrics is less than 10.3% in all cases (in fact less than 6% when excluding `cjpeg/dl1hits`) and approximately 3.7% on the average for all metrics.

6. RELATED WORK

In an effort similar to ours, *Rapaka et al.* [10] implements a scheme to decide when to switch between functional and cycle-accurate simulation. They use *hotspots* to identify frequently executed regions of a program. A hotspot is a region of execution that exhibits temporal locality in terms of the code

being executed. This focuses on reducing simulation time by dynamically identifying these hotspot regions, sampling them, and then reusing those samples to represent their future execution. They define a hotspot as a contiguous repetitive behavior, and they sample this behavior until it converges. Since they can only represent a repetitive region of execution with one converged sample, this means they must find the exact period [13] of the repeating phase behavior (if one exists) and the focus of their paper is how to do that dynamically [10].

In comparison, our approach does not have to align exactly with the periodic behavior of the hotspots. This is because we represent each interval by a unique code signature. This allows us to automatically break what they would refer to as a hotspot up into several unique re-occurring behaviors when using fixed length intervals. Therefore, we do not have to search for the exact periodic boundary as in [10], which can be difficult to find for complex programs. The handful of unique fixed length code signatures we find to represent a hotspot is used to create an accurate cycle-close trace of execution.

Nagpurkar et al. [8] uses phase analysis to guide collection of application profiles (basic block counts, hot path execution, etc) from remote applications running on energy and performance constrained devices. They also use dynamic phase classification to determine when to gather a profiling sample from each phase. They then communicate these samples back to a server where they can be aggregated together, even potentially across multiple users. Our work extends their approach by examining how to use phase prediction to decide if an interval should be sampled or not before the interval is executed. Their work [8] does not use phase prediction, instead they profile the metrics of interest for every interval on the device. Then after an interval is executed, they use dynamic phase classification to examine the code signature to see whether to communicate back (sample) the results for that interval to the aggregation center. Another difference is that our work focuses on dynamically creating a cycle-close trace of the complete execution (every interval) of the program, whereas their work focuses on creating an approximate summary of the overall program behavior at the trace aggregation center.

7. SUMMARY

For embedded system development, several companies are providing cross-platform development tools to aid in debugging, prototyping and optimization of embedded programs and their designs. These are full system emulation systems that can emulate the final binary to be run on the real board, its operating system and the devices. In this paper we provide the ability to produce cycle-close traces of architecture metrics (CPI, power, cache hit rates, etc) of a program's dynamic execution for such systems. Our approach uses dynamic phase classification to generate targeted cycle-close simulation samples. For the programs we examined, cycle-close tracing is 312 times faster on average than a cycle accurate trace of the full program's execution, with only 3.2% error.

The ability to generate and display fast cycle-close traces in tandem, as a program executes with full system emulation, provides a whole new level of analysis for these types of emulation systems. The user will be able to see how detailed architecture characteristics change as the program executes. For the future, we plan on applying similar techniques to multi-processor environments such as CMP (Chip Multiprocessors) and SMT (Simultaneous Multithreading). In this case, resource sharing makes the sampling decision more difficult because interactions between threads can result in many more different behaviors throughout the execution of the system.

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

8. REFERENCES

[1] Accelerated technology. http://www.acceleratedtechnology.com/.
[2] Coware - http://www.coware.com/.
[3] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.
[4] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.
[5] J. Lau, S. Schoenmackers, and Brad Calder. Transition phase classification and prediction. In *In the 11th International Symposium on High Performance Computer Architecture*, February 2005.
[6] Mibench. http://www.eecs.umich.edu/mibench/.
[7] W. Mong and J. Zhu. Dynamosim: A trace-based dynamic compiled instruction set simulator. In *Proceedings of the International Conference on Computer Aided Design*, pages 131–136, November 2004.
[8] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*, March 2005.
[9] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 41st Design Automation Conference*, June 2002.
[10] V. Rapaka and D. Marculescu. Pre-characterization free, efficient power/performance analysis of embedded and general purpose software applications. In *Proceedings of DATE Conference*, March 2003.
[11] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *Proceedings of the 40th Design Automation Conference*, June 2003.
[12] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge. Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification. In *2005 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.
[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*, September 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*, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.
[15] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.
[16] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 867–872, June 1999.
[17] Symbian OS - http://www.symbian.com/.
[18] P.K. Szwed, D. Marques, R.M. Buels, S.A. McKee, and M. Schulz. Simsnap: Fast-forwarding via native execution and application-level checkpointing. In *Interact-8: Workshop on the Interaction between Compilers and Computer Architectures*, February 2004.
[19] Vast system technologies. http://www.vastsystems.com/.
[20] Virtio's Virtual Platform. http://www.virtio.com/.
[21] Virtutech. http://www.virtutech.com/.
[22] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *SIGMETRICS'96*, pages 68–79, May 1996.