Discovering and Exploiting Program Phases

In a single second, a modern processor can execute billions of instructions and a program's behavior can change many times. Some programs change behavior drastically, switching between periods of high and low performance, yet system design and optimization typically focus on average system behavior. Instead of assuming average behavior, it is now time to model and optimize phase-based program behavior.

Timothy Sherwood
University of California at Santa Barbara

Erez Perelman
Greg Hamerly
University of California at San Diego

Suleyman Sair
North Carolina State University

Brad Calder
University of California at San Diego

Understanding program behavior is at the foundation of computer architecture and program optimization. Many programs have wildly different behavior on even the largest of scales (that is, over the program's complete execution). During one part of the execution, a program can be completely memory bound; in another, it can repeatedly stall on branch mispredicts. Average statistics gathered about a program might not accurately picture where the real problems lie. This realization has ramifications for many architecture and compiler techniques, from how to best schedule threads on a multithreaded machine, to feedback-directed optimizations, power management, and the simulation and test of architectures. Taking advantage of time-varying behavior requires a set of automated analytic tools and hardware techniques that can discover similarities and changes in program behavior on the largest of time scales.

The challenge in building such tools is that during a program's lifetime it can execute billions or trillions of instructions. How can high-level behavior be extracted from this sea of instructions?

The reality is this: The way a program's execution changes over time is not totally random; in fact, it often falls into repeating behaviors, called *phases*. Automatically identifying this phase behavior is the goal of our research and key to unlocking many new optimizations. We define a phase as a set of intervals (or slices in time) within a program's execution that have similar behavior, regardless of temporal adjacency. Recent research has shown that it is indeed possible to accurately identify and predict these phases in program behavior to capture meaningful phase behavior.[1-8]

The key observation for phase recognition is that any program metric is a direct function of the way a program traverses the code during execution. We can find this phase behavior and classify it by examining only the ratios in which different regions of code are being executed over time. We can simply and quickly collect this information using basic block vector profiles for off-line classification[4,6] or through dynamic branch profiling for online classification.[7] In addition, accurately capturing phase behavior through the computation of a single metric, independent of the underlying architectural details, means that it is pos-

Published by the IEEE Computer Society

sible to use phase information to guide many optimization and policy decisions without duplicating phase detection mechanisms for each optimization.[7]

Phase behavior

We begin the analysis of phases with a demonstration of the time-varying behavior of two different programs from SPEC 2000, gcc and gzip.[9] To characterize the behavior of these programs, we have simulated their execution all the way from start to finish. Each program executes many billions of instructions and gathering these results took several machine-years of simulation time. Figure 1 shows the behavior of each program, measured in terms of various statistics relating to how the program interacts with the underlying architecture over its execution.

Each point on the graph represents the average value for that metric (for example cache misses) taken over 10 million instructions of execution (an *interval*). We draw two important points from these graphs. First, average behavior does not sufficiently characterize a program's behavior. For example, in gzip the instructions per cycle (IPC) varies from 1.2 to 1.7, and the number data cache misses varies by almost an order of magnitude. In this way, gzip's behavior alternates between two phases. Second, not only does the program's behavior change wildly over time, it changes on the largest of time scales. The program can exhibit stable behavior for billions of instructions and then suddenly change. Together, these two points imply that an accurate model of program behavior must account for these long-term changes in the way a program executes.

Although program behavior changes significantly over time, the behavior of all of the metrics tends to change in unison, although not necessarily in the same direction. This implies that the changes are due to something more fundamental. If an automated approach were capable of quantifying these fundamental changes, it would make it possible to extract information about how a program is changing in a way that you could generalize to all hardware metrics.

In addition, such a method should enable the automatic partitioning of a program's execution into a set of phases that will quantify

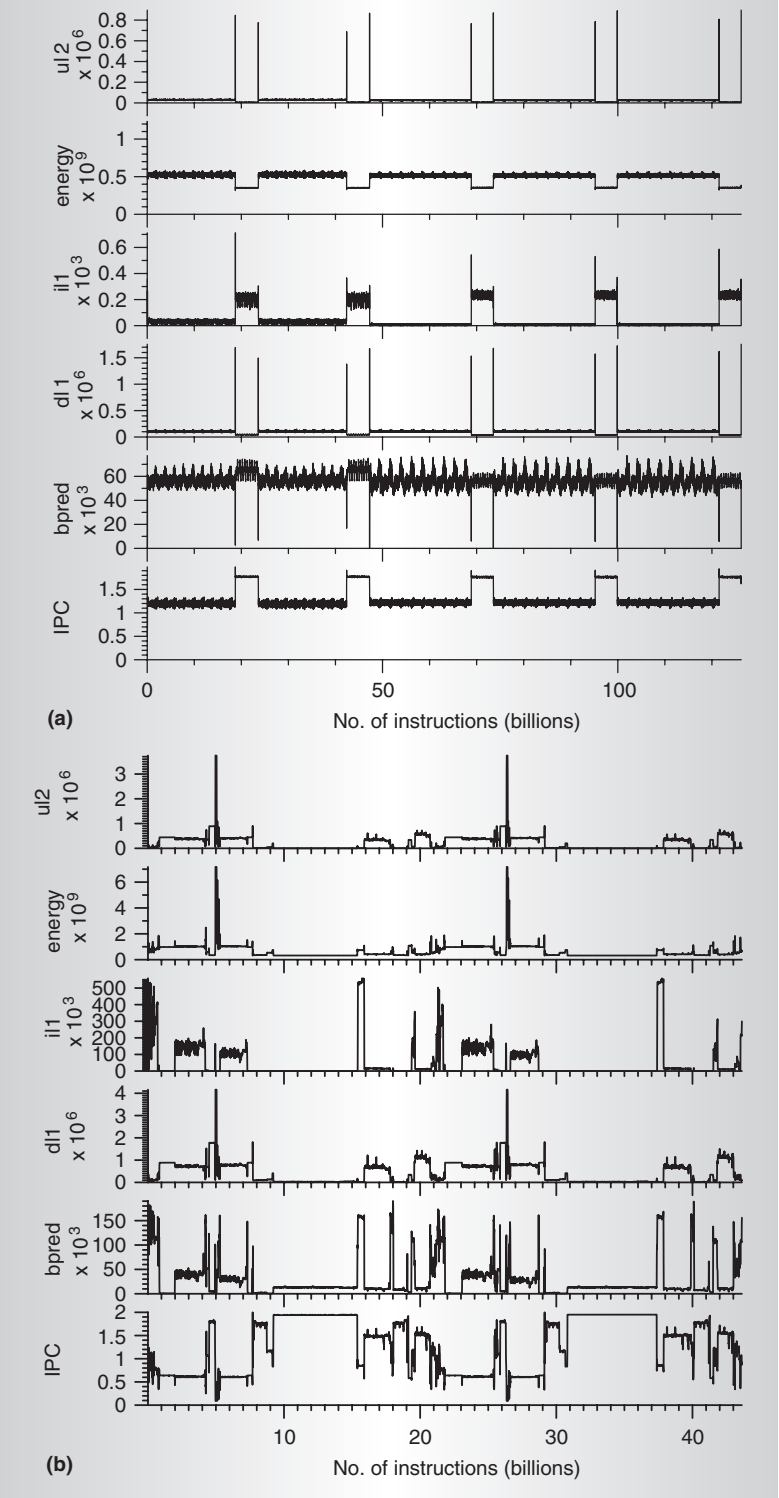

Figure 1.Plot of metrics over billions of instructions executed by the programs gzip with input graphic (a) and gcc with input 166 (b). Each point on the graph is an average over 10 million instructions. These graphs plot the number of unified L2 cache misses (ul2), energy consumed by the execution of the instructions, number of instruction cache misses (il1), number of data cache misses (dl1), number of branch mispredicts (bpred), and the average IPC.

the changing behavior over time. The goal is that after classification, each phase would contain only intervals that have similar behavior. Optimizations can then target individual phases or use phase information to aid in control. To solidify our discussions on classifying phase behavior, the following list contains definitions for both phases and the components of a phase used for analysis and optimization:

- An *interval* is a section of continuous execution (a slice in time) within a program. For the results presented here, we chose intervals of the same size, as measured by the number of instructions executed within an interval (either 1, 10, or 100 million instructions[4]). We are currently exploring the use of variable-sized intervals.
- 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 reoccur multiple times through the program's execution.
- *Phase classification* breaks a program's intervals of execution on a given input into phases with similar behavior. This phase behavior is for a specific program binary running a specific input (a binary-input pair).
- *Similarity* defines how close the behavior of two intervals are as measured across some set of metrics. Well-formed phases should have intervals with similar behavior across architecture metrics (such as IPC, cache misses, and branch mispredicts) and have similar working sets.
- The *similarity metric* is independent of hardware metrics; we use it to judge the similarity of intervals when performing phase classification.
- A *phase change* is a noticeable and sudden change in program behavior (similarity) over time (as if going from one phase to another phase).

Single metric for identifying phases

As discussed earlier, any effective technique for finding phase information requires a notion of how similar two parts of a program's execution are to one another. In creating this similarity metric, it is advantageous not to rely on statistics such as cache miss rates or performance. Doing so would tie the phases to those statistics and then you would need to reanalyze the phases every time some architecture parameter changed, either statically (for example, if the cache size changed) or dynamically (if some policy changed adaptively). Tackling these problems *requires a metric that is independent of any particular hardware-based statistic,* yet still related to the fundamental changes in behavior illustrated in Figure 1.

This led us to analyze the behavior of programs in terms of the code executed over time. There is a strong correlation between the executed set of paths in a program and the observed time-varying behavior. The intuition behind this is simple: What the code is doing at a particular time determines program behavior. With this idea in hand, it is possible to find the phases in programs using only a metric related to how the code is being exercised—that is, what code was the processor running and how often. It is important to understand that using this approach finds the phase behavior in Figure 1 by examining only the frequency in which the code (really the basic blocks) execute over time.

Basic block vector

To provide this metric, we developed the basic block vector (BBV)[5] to concisely capture information about how a program changes its behavior over time. A basic block is a section of code executed from start to finish with one entry and one exit. We use the frequencies with which basic blocks execute as the metric for comparing sections of the application's execution. The intuition behind this is that program behavior at a given time directly relates to the code executing during that interval, and basic block distributions provide us with this information. A-program, when run for any interval of time, will execute each basic block a certain number of times. Knowing this information provides a fingerprint for that interval of execution and shows where the application is spending its time in the code. The basic idea is that the basic block distributions for two intervals are fingerprints that indicate the similarity between the intervals. If the fingerprints are similar, then the two intervals spend about the same amount of time in the same code, and the performance of those two intervals should be similar.

More formally, a BBV is a one-dimensional array with one element in the array for each static basic block in the program. During each interval, the number of times program execution enters each basic block is counted and recorded in the BBV (weighed by the number of instructions in the basic block). Therefore, each element in the array is this count of entry into a basic block multiplied by the number of instructions in that basic block. The BBV is then normalized by dividing each element by the sum of all the elements in the vector that occurred during that interval.

BBV difference

To find patterns in the program, we must first have some way of comparing the similarity of two BBVs. The operation for this comparison takes as input two BBVs and outputs a single number showing how close these BBVs are. There are several ways of comparing two vectors, such as taking the dot product or finding the Euclidean distance (straight line between two points) or Manhattan distance (distance if movement can only be parallel to the axes). The tradeoffs between these are explained more fully elsewhere,[6] and we use a mix of the both techniques depending on the situation.

Besides BBVs, other methods are also acceptably accurate means of gathering phase information. These methods include creating conditional branch working set bit vectors,[10] sampling instruction PCs using VTune ,[11] and tracking the frequencies of loop (backwards conditional) branches as well as other architecture-independent constructs.[12] Only profiling the number of times each loop branch executes in a given interval yields phase classifications with accuracies similar to those obtained with BBVs.[12]

Basic block similarity matrix

We use a *basic block similarity matrix* to visually inspect the effectiveness of using BBVs in determining the similarities among intervals. The similarity matrix is the upper triangular of an $N \times N$ matrix, where N is the number of intervals in the program's execution. An entry at (x, y) in the matrix represents the Manhattan distance (similarity) between the BBVs at intervals x and y. Figure 2 shows the similarity matrices for the two example programs, gzip and gcc. The matrix's diagonal represents the program's execution over time from start to completion.

To interpret the graph, consider points along the diagonal axis. Each point is perfectly similar to itself, so all the points on the diagonal are dark. Starting from a given point on the diagonal, you can compare how that point relates to its neighbors forward and backward in execution by tracing horizontally or vertically. To compare given interval x with interval $x + n$, simply start at point (x, x) on the graph and trace horizontally to the right to $(x, x + n)$. In the similarity matrices for gcc and gzip, you can see large dark blocks, which indicate repeating behaviors in the program. Large triangular blocks that run along the diagonal indicate *stable regions* where program behavior is not changing over time. Rectangular dark blocks that occur off the diagonal axis indicate *reoccurring* behaviors, where a behavior that occurs later in execution has also occurred sometime in the past. When compared with the metrics shown in Figure 1, it's possible to see that examining just the executed code suffices to capture the program's repeating nature. This motivates the development of a technique to capture these patterns automatically.

Offline phase classification

BBVs provide a compact and representative summary of the program's behavior for each interval of execution. By examining the similarity between them, it is clear that there exists a high-level pattern within each program's execution. To use this behavior, it is necessary to have an automated way of extracting the phase information from programs. Clustering algorithms have proven useful in breaking the complete program execution into smaller groups (phases) that have similar BBVs.[6] Because BBVs relate to the program's overall performance, BBV-based grouping results in phases that are similar not only in their basic block distributions but also in every other metric measured, including overall performance. In addition, you can gather BBVs quickly because they require only the counting of basic block execution frequencies.

Using clustering for phase classification

The goal of clustering is to divide a set of points into groups such that points within

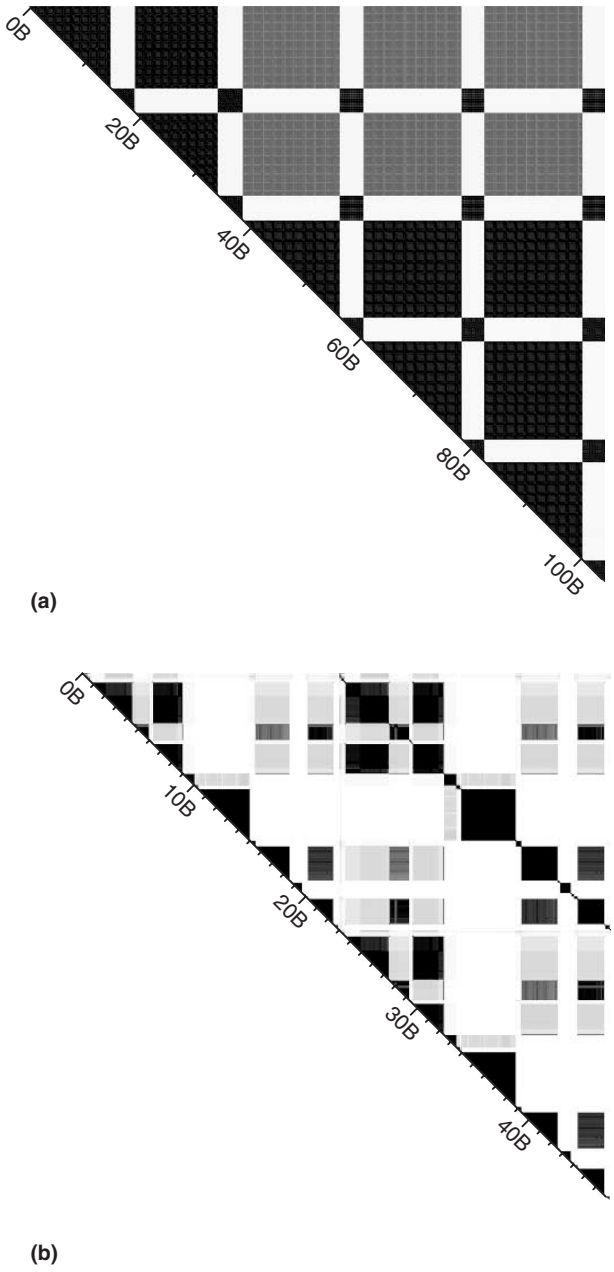

Figure 2. Basic block similarity matrices for gzip-graphic (a) and gcc-166 (b). The matrix diagonal represents a program's execution to completion with units in billions of instructions. The darker the points, the more similar the intervals (the Manhattan distance is closer to 0); the lighter the points, the more different the intervals (the Manhattan distance is closer to 2).

each group are similar (by some metric, often distance) and the points in different groups are dissimilar. A well-known clustering algorithm, *k*-means,[13] can accurately break program behavior into phases. Random linear projection[14] reduces the dimensionality of the input data without disturbing the underlying similarity information; it is a useful technique for speeding up the execution of *k*-means. One serious drawback of the *k*-means algorithm is that it requires the value *k*—the number of clusters—as input. To address this problem, we run the algorithm for several *k* values and use a score to guide our final choice for *k*. The following steps summarize our algorithm at a high level; our earlier work gives a detailed description of each step:[6]

1. Profile the basic blocks executed in each program, breaking the program up into contiguous intervals of size *N* (for example, 1 million, 10 million, or 100 million instructions). Generate and normalize a BBV for every interval.
2. Reduce the dimensionality of the BBV data to *P* dimensions (for example, 15) using random linear projection. The advantage of performing clustering on projected data is that it significantly accelerates the *k*-means algorithm and reduces the memory requirements by several orders of magnitude.
3. Run the *k*-means algorithm on the reduced-dimension data with values of *k* from 1 to *M*, where *M* is the maximum number of phases to use. Each run of *k* means produces a clustering, which is a partition of the data into *k* phases/clusters. During this clustering *k* means compares the similarity of intervals, grouping them together into phases. Each run of *k* means begins with a random initialization step, which requires a random seed.
4. To compare and evaluate the clusters formed for different *k*, we use the Bayesian information criterion (BIC)[15] as a measure of the goodness of fit of a clustering within a dataset. More formally, BIC is an approximation to the probability of the clustering given the data that has been clustered. Thus, the larger the BIC score, the higher the probability that the clustering is a good fit to the data. For each clustering (*k* from 1 to *M*), score the clustering's fitness using the

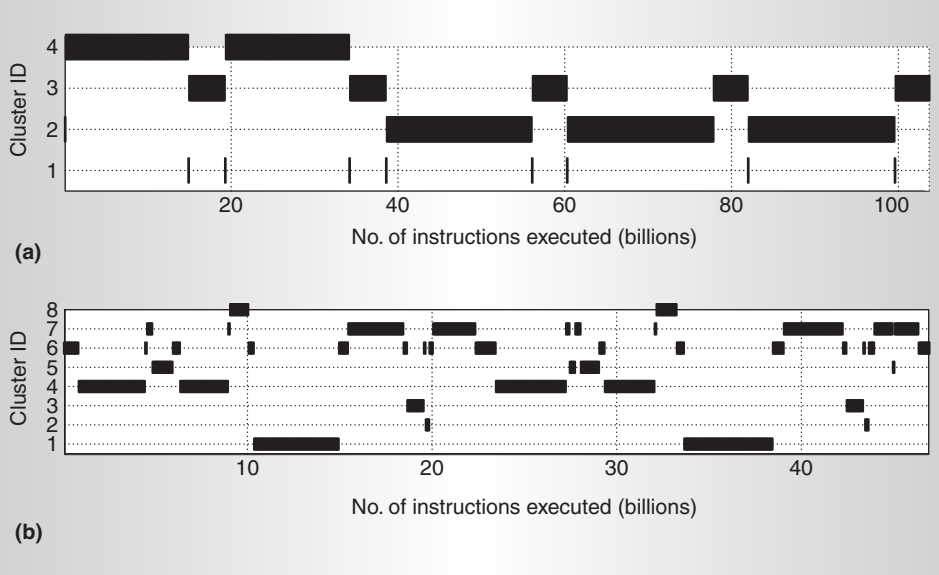

Figure 3. Graphs showing the phase clustering of the execution intervals for gzip-graphic (a) and gcc-166 (b), based on our algorithm. The *x*-axis units are number of executed instructions; the graph shows for each interval of execution (every 100 million instructions) which cluster the interval fell into. Using k-means clustering with the Euclidean distance we partition gzip's full run of the execution into four phases; for gcc, we have eight phases.

formulation given by Pelleg and Moore.[15]

5. Choose the clustering with the smallest k, such that its BIC score is at least X percent of the best score. The clustering k chosen is the final grouping of intervals into phases. For the results presented later, we used an 80 percent threshold and $M = 10$.

These steps provide a grouping of intervals into phases. The *k*-means algorithm groups similar intervals together based on the BBV similarity metric using the Euclidean distance. We then choose a final grouping of phases from the different options based on how well formed the phases are, as measured by the BIC metric.

Clusters and phase behavior

Figure 3 shows the result of running the clustering algorithm on gzip and gcc, using an interval size of 100 million instructions and setting maximum number of phases M to 10. The x axis corresponds to the program's execution in billions of instructions, and the graph indicates to which phase (labeled on the y axis) each interval belongs.

This algorithm partitions gzip's execution into four clusters. Comparing these results with Figure 2a, the cluster behavior captured by the offline algorithm aligns closely with the program's behavior. Clusters 2 and 4 represent the larger sections of similar execution. Cluster 3 captures the smaller phase that lies in between these larger phases. Cluster 1 represents the phase transitions between the three dominant phases. The cluster 1 intervals fall into the same phase because they execute a similar combination of code, which happens to be part of the code behavior in either cluster 2 or 4, and part of the code executed in cluster 3. These transition points in cluster 1 also correspond to the same intervals that have large cache-miss spikes, as shown in the time-varying graphs of Figure 1.

Figure 3b shows how this algorithm partitions gcc into eight clusters. Comparing these results to Figure 2b, we see that it correctly captures even gcc's more complicated behavior. We see that this algorithm accurately groups the intervals corresponding to the dark boxes on the diagonal of the similarity matrix (Figure 2b) into dominant clusters 1, 4, and 7.

SimPoint

Understanding the cycle-level behavior of

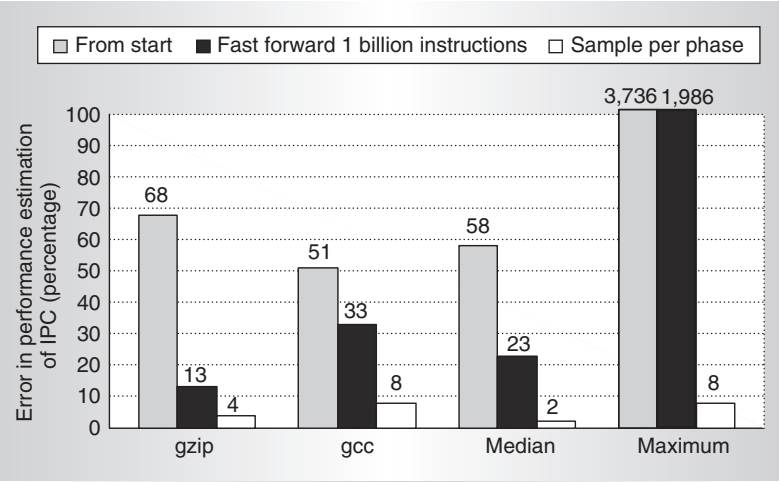

Figure 4. Simulation accuracy for the SPEC 2000 benchmark suite when performing detailed simulation for a few hundred million instructions compared to simulating the entire program. These results cover simulation from the start of the program's execution, for fast-forwarding through 1 billion instructions before simulation, and using SimPoint to choose less than 10 hundred-million-instruction intervals to simulate. The median and the maximum results are for the complete SPEC 2000 benchmark suite.

a processor running an application is crucial to modern computer architecture research. To gain this understanding, architects typically employ detailed cycle-level simulators. Unfortunately, this level of detail comes at the cost of simulation speed, and simulating the full execution of an industry standard benchmark on even the fastest simulator can take weeks or months to complete. Offline phase analysis provides an accurate and efficient work around for this problem.

To address this problem we created the SimPoint[4,6] tool to choose simulation points intelligently using offline phase classification algorithms. SimPoint calculates phases for a program/input pair, and then chooses a single representative from each phase and estimates the remaining intervals' behaviors by performing a detailed simulation only on that chosen representative. We choose this representative for each phase (which is a cluster of intervals) by finding the interval closest to the cluster's center (centroid). This selected interval for a phase is called a *simulation point* for that phase. We then perform detailed simulation at the simulation points and weight the performance by the size (number of intervals) in its cluster. SimPoint can significantly reducing simulation time and provides an accurate characterization of the full program. Figure 4 shows simulation accuracy results using SimPoint for the SPEC 2000 programs, comparing them to the complete execution of these programs. As described earlier, we choose one simulation point for each cluster, so for gzip, which has 4 clusters, we simulated 400 million instructions. Despite simulating this limited number of instructions, our method had only a 4 percent error for gzip.

For the non-SimPoint results, we ran a simulation for the same number of instructions as the SimPoint data to provide a fair comparison. Figure 4 shows that starting simulation at the program's beginning results in a median error of 58 percent when compared to the full simulation, whereas blindly fast forwarding for 1 billion instructions results in a median 23 percent IPC error. When using the clustering algorithm to create multiple simulation points, we saw a median IPC error of 2 percent, and an average IPC error of 3 percent. In comparison to random-sampling approaches, SimPoint achieves similar error rates but requires significantly less—five times less—fast-forwarding time.[4] In addition, statistical sampling can be combined with SimPoint to create a phase clustering that has a low per-phase variance.[4]

Several researchers in academia and at Intel are using SimPoint to accurately guide their architecture simulation research. We distribute the code to track the basic blocks, perform the analysis and the clustering, and pick the simulation points as part of the SimPoint tool (http://www.cs.ucsd.edu/users/calder/simpoint/).

Online phase classification

Although the offline model provides a powerful way to study and summarize program phases, a way of exploiting phase behavior in programs at runtime requires a slightly different approach. The results of offline analysis show that many programs drastically change behavior over time and often in a very structured way. The offline analysis is useful for discovering and exploiting these phases, but there are potentially more opportunities for phase-specific optimizations if we can detect phases in a program as it runs.

With phase information about a running

program, architects can use this additional information to make intelligent decisions about power management, resource allocation, or thread scheduling. In fact, a runtime system or processor can use phase information to not only adapt its operations to program phase behavior, it could also use it to predict future behavior.[7] An effective online phase detection scheme enables all these possibilities.

Phase classification architecture

An online approach's goal is to create a small, easily implementable (in hardware or software), runtime phase detection scheme.[7] The challenge of designing an online scheme to capture phases are:

- only a small, fixed amount of storage is available for all of the information collected;
- the online scheme learns the program's behavior as it executes, so there can only be one pass through the data; and
- the computation involved should be small enough to not interfere with the system.

These challenges will have some cost, namely that any dynamic technique will be less discriminating in its choice of how to group phases when compared to an offline technique. Figure 5 gives an overview of an architecture for phase classification. Each stage of the hardware implementation is an approximation of one stage of the offline algorithm. There are essentially four stages of the design, our earlier work gives a detailed description.[7]

1. To approximate the tracking of basic blocks in the offline approach, we track the program counter (PC) of every committed branch and the number of instructions (I) committed between the current branch and the last branch. This tuple provides almost exactly the same information as the more careful tracking of basic blocks.
2. To approximate random linear projection, we apply the projection beforehand and store only the projected data, rather than wait to apply this reduction after generating a profile. We approximate the projection by using a hashing function on the branch PCs before we insert them into the accumulators.
3. To build up a set of vectors, we use N accumulators. The hashing function in step 2 generates a hash of the PC that maps it into one of the N accumulator buckets. To approximate the offline algorithm's use of the basic block sizes, we use I instead, adding I to the bucket that maps to the branch PC. Stages 1 through 3 must occur at processor speed, but it only involves a counter, a hash, and an accumulator update, all of which we can pipeline.
4. After updating the accumulator table for some fixed amount of time, we must approximate the clustering algorithm. For classification, we must evaluate the fingerprint with the following two ques-

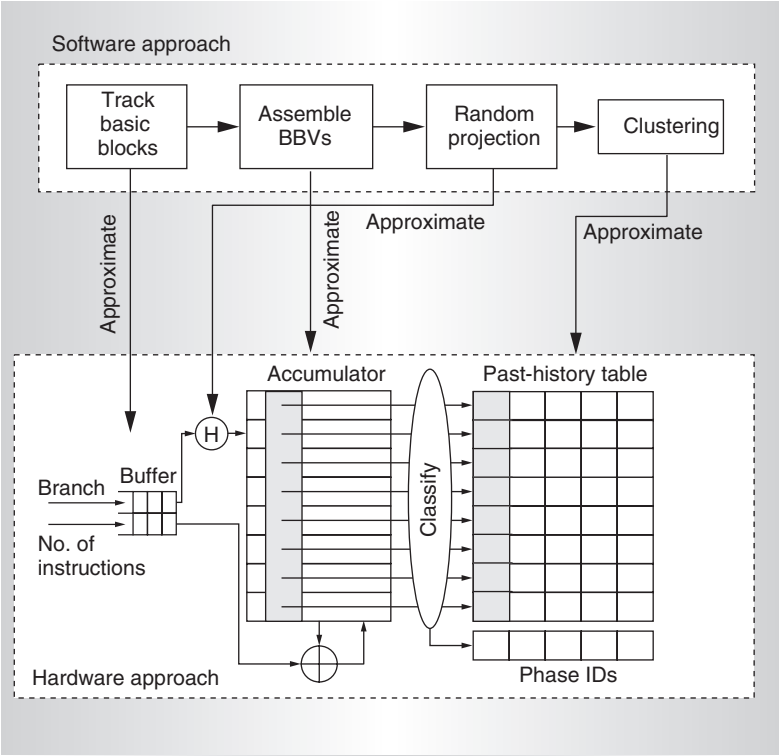

Figure 5. Phase classification architecture. The architecture captures each branch program counter along with the number of instructions from the last branch. It increments the bucket entry, corresponding to a hash of the branch program counter, by the number of instructions. After each profiling interval has completed, the buckets represent a fingerprint that is compared to the fingerprints in the past-history to classify the interval into a phase.

Table 1. Examination of per-phase homogeneity compared to the program as a whole (denoted by full). For the two programs and each of the top five phases in each program, we show the average value of each metric and the standard deviation.

	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gcc	full	1.32	(43.4%)	27741	(135.5%)	445083	(110.7%)	50763	(203.2%)	6.44E+08	(90.0%)	227912	(139.7%)
	18.5%	0.61	(1.6%)	34665	(22.0%)	753382	(5.4%)	125091	(23.2%)	1.03E+09	(1.8%)	395997	(5.3%)
	18.1%	1.95	(0.3%)	13048	(3.9%)	28112	(15.1%)	43	(73.9%)	3.22E+08	(0.2%)	1006	(5.6%)
	7.2%	0.64	(0.2%)	843	(15.1%)	885081	(0.1%)	75	(215.5%)	9.78E+08	(0.3%)	443655	(0.1%)
	4.0%	1.49	(1.2%)	10145	(7.6%)	703554	(6.8%)	15591	(5.2%)	4.20E+08	(1.1%)	354084	(7.0%)
	3.9%	1.76	(1.6%)	2015	(13.6%)	98947	(5.9%)	102	(45.1%)	3.57E+08	(1.6%)	15595	(12.6%)
	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gzip	full	1.33	(16.3%)	56045	(11.1%)	90446	(58.2%)	60	(138.1%)	4.82E+08	(13.5%)	22880	(112.0%)
	17.1%	1.24	(3.4%)	53300	(10.8%)	96960	(10.1%)	12	(44.2%)	5.05E+08	(3.5%)	24218	(8.6%)
	9.4%	1.23	(3.8%)	54973	(11.5%)	99523	(11.3%)	11	(45.5%)	5.09E+08	(3.8%)	24518	(9.3%)
	8.8%	1.76	(0.6%)	56449	(4.8%)	37331	(5.6%)	241	(8.4%)	3.55E+08	(0.6%)	5617	(15.6%)
	8.0%	1.22	(4.3%)	54791	(6.8%)	99671	(11.9%)	40	(25.7%)	5.14E+08	(4.4%)	28153	(11.0%)
	7.4%	1.24	(3.1%)	55215	(11.1%)	96701	(9.6%)	12	(35.4%)	5.04E+08	(3.2%)	23701	(8.4%)

tions in mind: Have we seen this behavior before? And if so, when did we see it? We answer these questions by comparing values in the accumulator table with the set of past behaviors (the past-history table). If an entry in the past-history table is within a certain threshold distance of the accumulator table's entry, then this interval is similar to other intervals previously classified into that phase.

The online algorithm can perform phase classification on programs at runtime with little to no impact on the processor core's design. One goal of phase classification is to divide the program into a fairly homogeneous set of phases. This means that an optimization adapted and applied to a single segment of execution from one phase will apply equally well to the other parts of the phase. To quantify the extent to which the online algorithm achieves this goal, we measured the homogeneity of a variety of architectural statistics on a per-phase basis.

Table 1 shows the results of performing this analysis on the phases determined at runtime for gcc and gzip, using an interval of 10 million instructions. For both programs, this table shows a set of statistics. The first phase (listed as "full") is the result of treating the entire program as a single phase. In addition to the average value, Table 1 also shows the standard deviation for each statistic. If the phase-tracking hardware is successful in classifying the phases, the standard deviations for the various metrics should be low for a given phase identification.

Looking at gcc's energy consumption, we observe that it swings radically (a standard deviation of 90 percent) over the program's complete execution. Figure 1 shows this graphically, plotting the energy usage versus instructions executed. However, after dividing the program into phases, we see that each phase has very little variation within itself; all have less than a 5 percent standard deviation. By analyzing gcc, we also see that the phase partitioning does a very good job across all of the measured statistics, even though partitioning used only one metric (tracking the executed branches).

Dynamically using the phase information can lead to new compiler optimizations with code tailored to different phases of execution, phase prediction, multithreaded architecture scheduling, power management, and other resource distribution problems controlled by software, hardware, or the operating system. We have evaluated our phase classification architecture through simulation by examining the effectiveness of phase tracking and prediction for per-phase value profiling, and for reconfigurable data caches and processor widths.[7]

Phase classification research will open the door for a new class of program analysis techniques, and runtime and hardware optimizations targeted toward fine-tuning the behavior of a program or system. Significant research lies ahead to better understand phase behavior and to tune phase classification algorithms, thresholds, and interval sizes to many different uses.

References

1. R. Balasubramonian et al., "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," *Proc. 33th Annual Int'l Symp. Microarchitecture*, IEEE CS Press, 2000, pp. 245-257.
2. A. Dhodapkar and J.E. Smith, "Dynamic Microarchitecture Adaptation Via Co-Designed Virtual Machines," *Proc. Int'l Solid State Circuits Conf.*, IEEE Press, 2002, pp. 154-155, 444.
3. A. Dhodapkar and J.E. Smith, "Managing Multi-Configuration Hardware Via Dynamic Working Set Analysis," *Proc. 29th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 2002, pp. 233-246.
4. E. Perelman, G. Hamerly, and B. Calder, "Picking Statistically Valid and Early Simulation Points," *Proc. 12th Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2003, pp. 244-255.
5. T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE CS Press, 2001, pp 3-14.
6. T. Sherwood et al., "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, 2002, pp. 45-57.
7. T. Sherwood, S. Sair, and B. Calder, "Phase Tracking and Prediction," *Proc. 30th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 2003, pp. 336-349.
8. M. Van Biesbrouck, T. Sherwood, and B. Calder, "A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation," to be published in *Proc. Int'l Symp. Performance Analysis of Systems and Software*, 2004.
9. T. Sherwood and B. Calder, *Time Varying Behavior of Programs*, tech. report UCSD-CS99-630, Univ. of Calif., San Diego, Aug. 1999.
10. A. Dhodapkar and J.E. Smith, "Comparing Program Phase Detection Techniques," *Proc. 36th Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, 2003, pp. 217-227.
11. B. Davies et al., *Ipart: An Automated Phase Analysis and Recognition Tool*, tech. report, Microprocessor Research Labs, Intel Corp., Nov. 2003.
12. J. Lau, S. Schoenmackers, and B. Calder, *Structures for Phase Classification*, tech. report UCSD-CS2003-0772, Univ. of Calif., San Diego, Oct. 2003.
13. J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," L.M. LeCam and J. Neyman, eds., *Proc. Fifth Berkeley Symp. on Mathematical Statistics and Probability*, vol. 1, Univ. of Calif. Press, 1967, pp. 281-297.
14. S. Dasgupta, "Experiments with Random Projection," *Uncertainty in Artificial Intelligence: Proc. Sixteenth Conf.*, Morgan Kaufmann Publishers, 2000, pp. 143-151.
15. D. Pelleg and A. Moore, "X-Means: Extending K-Means with Efficient Estimation of the Number of Clusters," *Proc. 17th Int'l Conf. on Machine Learning*, Morgan Kaufmann Publishers, 2000, pp. 727-734.

Timothy Sherwood is an assistant professor in the Computer Science Department at the University of California at Santa Barbara. He has a PhD in computer science from the University of California at San Diego. **Erez Perelman** is a PhD candidate in the Computer Science Department at UCSD. He has a BS in computer science from UCSD. **Greg Hamerly** is a post-doctoral student at the Katholieke Universiteit Leuven. He has a PhD in computer science from the University of California at San Diego. **Suleyman Sair** is an assistant professor of electrical and computer engineering at North Carolina State University, and a member of the Center for Embedded Systems He has a PhD in computer science from the University of California at San Diego. **Brad Calder** is an associate professor of computer science and engineering at UCSD. He has a PhD in computer science from the University of Colorado, Boulder.

Direct questions or comments about this article to Brad Calder, University of California at San Diego; Department of Computer Science and Engineering; 9500 Gilman Drive, Dept 0114; La Jolla, CA 92093-0114; calder@cs.ucsd.edu.