Motivation for Variable Length Intervals and Hierarchical Phase Behavior

Jeremy Lau[†] Erez Perelman[†] Greg Hamerly[‡] Timothy Sherwood[*] Brad Calder[†]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Department of Computer Science, Baylor University
[*]Department of Computer Science, University of California, Santa Barbara

Abstract

Most programs are repetitive, where similar behavior can be seen at different execution times. Proposed algorithms automatically group similar portions of a program's execution into phases, where the intervals in each phase have homogeneous behavior and similar resource requirements. These prior techniques focus on fixed length intervals (such as a hundred million instructions) to find phase behavior. Fixed length intervals can make a program's periodic phase behavior difficult to find, because the fixed interval length can be out of sync with the period of the program's actual phase behavior. In addition, a fixed interval length can only express one level of phase behavior.

In this paper, we graphically show that there exists a hierarchy of phase behavior in programs and motivate the need for variable length intervals. We describe the changes applied to SimPoint to support variable length intervals. We finally conclude by providing an initial study into using variable length intervals to guide SimPoint.

1 Introduction

The behavior of a program is not random - as programs execute, they exhibit cyclic behavior patterns. Recent research [1, 7, 8, 27, 28, 29, 24, 9, 18], has shown that it is possible to accurately identify and predict these phases in program execution. Prior research has shown that many programs have wildly different behavior on even the very largest of scales (over the complete execution of the program).

Phase behavior can be exploited for accurate architecture simulation [27, 28], to save energy by dynamically reconfiguring caches and processor width [1, 29, 8, 7], to guide compiler optimizations [20, 2], to guide remote profiling [21], and to choose which core to run a process on in a multi-core architecture [15]. All of these techniques take advantage of the phase behavior that exists in programs, and most of them focus on the phase behavior seen at a specific granularity (fixed length interval length) of execution.

However, to take advantage of time-varying behavior, we must first develop tools to automatically and efficiently analyze program behavior over large sections of execution. To identify phases, we divide a program's execution into non-overlapping intervals [28]. An *interval* is a contiguous portion of execution (a slice in time) of a program. A *phase* is a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency. This means that a phase may appear many times as a program executes. *Phase classification* partitions a set of intervals into phases with similar behavior.

In [28], we detected phase behavior by breaking the program's execution up into *fixed length intervals*, and grouping similar intervals of execution together with clustering techniques from machine learning. The problem with this approach is that the fixed length intervals are frequently out of sync with the actual periodic behavior of the program. This makes it more difficult to automatically find large scale phase behavior. In addition, programs exhibit phase behavior at many different granularities, and focusing on a single fixed interval length limits phase discovery to a single granularity. Some programs exhibit a hierarchy of phase behaviors that can be seen at many different interval lengths.

The goal of this paper is to provide the motivation for why we need to move to variable length intervals, and to describe the changes to SimPoint needed to support this. We present new methods for graphically looking at program behavior, and present an initial study into using variable length intervals to build up a hierarchy of phase behavior.

2 Related Work

Phase behavior has been shown to exist when examining a program's working set [5], and several researchers have recently examined phase behavior in programs.

Balasubramonian *et al.* [1] proposed using hardware counters to collect miss rates, CPI and branch frequency information for every 100K instructions. They use the miss rate and the total number of branches executed for that interval to determine if the program's behavior for the interval was stable. This was used to guide dynamic cache reconfiguration to save energy without sacrificing performance.

Dhodapkar and Smith [7, 8, 6] found a relationship between phases and instruction working sets, and that phase changes occur when the working set changes. They propose that by detecting phases and phase changes, multi-configuration units

can be re-configured in response to these phase changes. They use their working set analysis for instruction cache, data cache and branch predictor re-configuration to save energy [7, 8].

Huang *et al.* [12] examine tracking procedure calls via a call stack, which can be used to dynamically identify phase changes. More recently they examined using procedure call boundaries for creating samples to use for guiding statistical simulation [19].

Isci and Martonosi [13, 14] have shown the ability to dynamically identify the power phase behavior using power vectors. Deusterwald *et al.* [9] recently used hardware counters and other phase prediction architectures to find phase behavior.

In [27, 28], we proposed that periodic phase behavior in programs can be automatically identified by profiling the code executed. We used techniques from machine learning to classify the execution of the program into phases (clusters). We found that intervals of execution grouped into the same phase had similar behavior across all architectural metrics examined. From this analysis, we created a tool called SimPoint [28], which automatically identifies a small set of intervals of execution (simulation points) in a program for detailed architectural simulation. These simulation points provide an accurate and efficient representation of the complete execution of the program. We recently extended this approach to perform hardware phase classification and prediction [29, 18].

The closest work to ours is the work done by Shen et al. [26], where they use wavelets and Sequitur to build a hierarchy of phase information to guide the prediction of data phases of applications. They analyze *data reuse distance* traces, and look for patterns in program behavior with wavelet analysis, whereas our approach is based on code signatures. They take the data reuse distance phases at the finest granularity and use Sequitur to identify hierarchical phase behavior. Their approach works well for simple programs that have very structured data behavior, but their focus on data does not work with complex programs like `vortex` and `gcc` [26]. In comparison, we found that using code signatures finds all of the phase behavior that using data does [17], and is applicable to complex applications, since we base our phase analysis on the structure of the program. Our philosophy is that program behavior is strongly correlated with the code executed - in other words, *you are what you execute*. By examining only code signatures we can detect phase behavior even in complex programs like `vortex` and `gcc`.

Most of the above related work has focused on identifying phase behavior using fine-grain fixed length intervals. The goal of this paper is to automatically build a hierarchy of variable length intervals, and to build a hierarchy of phase classifications.

3 Basic Block Vectors

Basic Block Vectors (BBVs) [27] provide a structure designed to capture information about changes in a program's behavior over time. A basic block is a single-entry, single-exit section of code with no internal control flow. A *Basic Block Vector* (BBV) is a one dimensional array where each element in the array corresponds to one static basic block in the program. We start with a BBV containing all zeroes at the beginning of each interval. During each interval, we count the number of times each basic block in the program has been entered, and we record the count in the BBV. For example, if the 50th basic block is executed 15 times, then bbv[50] = 15. We multiply each count by the number of instructions in the basic block, so basic blocks containing more instructions will have more weight in the BBV.

As in [27, 28], we use BBVs to compare the intervals of the application's execution. The intuition behind this is that the behavior of the program at a given time is directly related to the code executed during that interval. We use the basic block vectors as fingerprints for each interval of execution: each vector tells us what portions of code are executed, and how frequently those portions of code are executed. By comparing the BBVs of two intervals, we can evaluate the similarity of those two intervals. If the distance between the BBVs is small, then the two intervals spend about the same amount of time in roughly the same code, and therefore the performance of those two intervals should be similar. We use BBVs to motivate the need for variable length intervals.

In [17], we propose several alternatives to basic block vectors. One of the alternatives is the loop and procedure vector, where we track the number of times each backward non-interprocedural branch, each procedure call, and each procedure return is executed. We found that loop and procedure vectors were comparable to basic block vectors for phase classification purposes, with fewer dimensions per vector. The variable length interval selection algorithm proposed in this paper uses vectors based on loop and procedure counts instead of basic block vectors because of the benefits of reduced dimensionality.

4 Methodology and Metrics

We performed our analysis for the SPEC2000 programs `ammp`, `bzip`, `galgel`, `gcc`, `gzip`, `mcf`, and `perl`. All programs were run with reference inputs, and `bzip`, `gcc`, and `gzip` were run with multiple inputs. We chose the above programs since they were the most interesting and challenging for phase classification from our prior studies. We collect all of the frequency vector profiles using SimpleScalar [4].

To generate our baseline fixed length interval results, all programs were executed from start to completion using SimpleScalar. The baseline microarchitecture model is detailed in Table 1.

I Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Pred	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 8 operations per cycle, 64 entry re-order buffer
Mem Disambig	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Func Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Mem	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

4.1 Metrics for Evaluating Phase Classification

Since phases are intervals with similar program behavior, we measure the effectiveness of our phase classifications by examining the similarity of program metrics within each phase. We focus on overall performance in terms of Cycles Per Instruction (CPI) within each phase. After classifying a program's intervals into phases, we examine each phase and calculate the average CPI of all intervals in the phase. We then calculate the standard deviation in CPI for each phase, and we divide the standard deviation by the average to get the *Coefficient of Variation* (CoV). CoV measures standard deviation as a fraction of the average. When we compute the average and the standard deviation, we weight the CPI of each interval by the length of the interval, so intervals that represent a larger percentage of the program's execution receive more weight in the calculations.

We use the CoV to compare different phase classification algorithms. Better phase classifications will exhibit lower CoV. If all of the intervals in the same phase have exactly the same CPI, then the CoV will be zero. We calculate an overall CoV metric for a phase classification by taking the CoV of each phase, weighting it by the percentage of execution that the phase accounts for, and then summing up the weighted CoVs. This results in an overall metric we can use to compare different phase classifications for a given program. It represents the average percentage of deviation that a phase classification exhibits.

5 Problems with Fixed Length Intervals

Prior work in phase classification has concentrated on using fixed length intervals. In this section we show that fixed length intervals can result in sub-optimal phase classification. In phase classification, intervals are the building blocks for forming phases and identifying changes in phase behavior. At this level, any noise accumulated in the intervals will have ramifications on the quality of the phases detected. A fixed interval length will generally result in several intervals that attempt to represent a single behavior which can be more concisely represented. Using fixed length intervals can also result in phases that do not accurately represent the behaviors of the program. We present examples of how and why these problems occur.

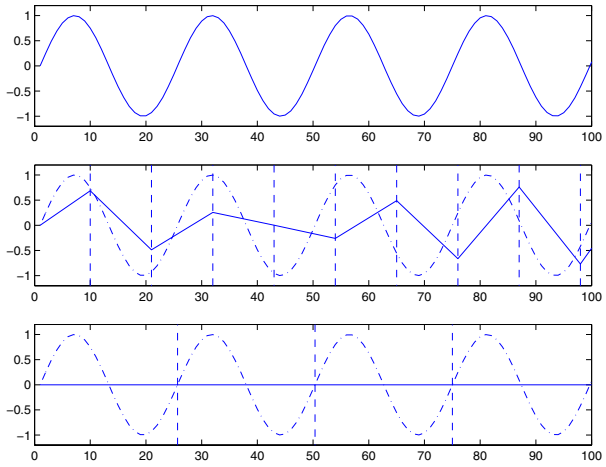


Figure 1: An example of what happens to a signal (top figure) when it is sampled with different interval lengths. The signal in this example is a sinusoid, shown in the top figure, and the intervals it is broken into are drawn vertically in the lower two figures. The average signal for each interval is shown as the straight line within an interval. When the interval is dissonant with the period of the signal, it results in a jagged and unstable characterization as can be seen in the central figure. The optimal interval duration, shown in the bottom figure, captures exactly one cycle of the repetitive behavior, which results in a concise and stable characterization of the signal.

5.1 Interval Dissonance and Harmony

To illustrate the ramifications of interval length on the representation of time-series data, we present a simple example. Figure 1 shows what happens when a simple sinusoid is sampled at different interval durations. The top figure shows the signal, a sine wave with a constant period equal to 25.

We now consider the effects of dissonance between a fixed interval length and the period of the signal. The central figure shows what happens when we split the signal into fixed length intervals of length 11. Here we see the original signal in the background, with vertical dashed lines depicting where the intervals are split. The signal average for each interval is plotted as a point at the end of that interval, and *the solid line in this figure connects these averages to show the interval-based representation of the signal*. In this figure the solid line is very jagged, because the length of the interval is out of sync with the period length of the cyclical signal. Using fixed length intervals may require many intervals to ac-

curately represent the signal. We can quantify the number of intervals required to accurately represent a signal as the ratio of Least Common Multiple between the interval length and the period of the signal, and the length of the interval: $(LCM(|interval|, |signalperiod|)/|interval|)$. In this example, it would require a total of 25 intervals to accurately represent the signal using intervals of fixed length 11.

On the other hand, let us consider harmony between an interval length and the period of the signal. The bottom figure shows an interval length of 25, equal to the period of the signal. The same format is used as in the central figure. Here we see intervals capturing entire cycles of the signal, and the resulting behavior of the intervals is constant. This is the ideal situation, since it would require exactly 1 phase to represent this signal accurately.

In this simple example, fixed intervals with length 25 can accurately represent the signal. But most programs do not exhibit simple fixed-frequency phase behavior. For example, `gzip` exhibits low-frequency phase behavior in its low-IPC phases, and high-frequency phase behavior in its high-IPC phases. It is unlikely that a single fixed interval length can accurately capture phase behavior at both these frequencies. Additionally, there are some benchmarks where the period changes over time. For example, `vpr-route` exhibits behavior patterns corresponding to each routing it tries. As its simulated annealing algorithm converges on a solution, it spends less and less time evaluating each solution.

6 Hierarchical Representation of Program Execution

In this section we examine two representations of a program's execution by only looking at how the code is executed using the fixed length interval basic block vectors. This is used to (a) motivate what could be found if we did use variable length intervals aligned to phase boundaries, and (b) to show that hierarchical phase behavior exists.

6.1 3D Non-Accumulated Representation

The previous section presented a simple example where fixed length interval lengths can have significant impact on the representation of the signal. Here we examine actual program execution data, and see how it is even more susceptible to representation problems when using fixed length intervals.

For the non-accumulated representation, each interval is represented with a basic block vector (BBV) that indicates the number of times each basic block was executed in that interval as described in Section 3. We take this set of basic block vectors and reduce the number of dimensions down to three using random linear projection [28]. Then we plot each 3-dimensional vector as a point in space, and draw lines between the temporally adjacent points to show the execution order. This provides a visual portrait of how the program executes over time, through its code space. An almost constant pattern should show up as a tight cluster in space. In theory, if the boundaries between BBVs always fell perfectly on a phase transition, then we would expect to see a set of interconnected tight clusters with a lot of BBVs placed on top of each other. If the boundaries are not aligned with the periodic program behavior, then we should see *oscillations* as discussed in the prior section. The oscillations should show up as *rings* or *tori* ("donuts") if we plot these paths in 3-d space. Figure 2 shows a 3-dimensional representation of the execution of two benchmarks: `gzip-graphic` and `bzip2-source`. A fixed length interval size of 100 million instructions was used. In both `gzip` and `bzip` there are interesting cycles that appear. For each program we zoom in on one of the cyclic regions.

In SPEC2000, the `gzip` benchmark repeatedly compress and decompresses the data a total of 5 times, at compression levels 1, 3, 5, 7, and 9. At compression levels 1 and 3, a faster version of the `deflate` algorithm is used. This time-varying program structure is clearly visible from the `gzip` graphs shown in Figure 2. For example we see that execution bounces back and forth between `deflate_fast` and `inflate` 3 times (`deflate_fast` → `inflate` → `deflate_fast` → `inflate`), corresponding to compression and decompression at levels 1 and 3. There are 5 bounces between `deflate` and `inflate`, corresponding to compression and decompression at levels 5, 7, and 9. The vectors exhibited by each deflation and inflation phase form a torus. Each cycle around the torus corresponds to the compression or decompression of a block of data. If correctly sized variable length intervals were used, then each cycle around the region should become a single interval, instead of a series of intervals forming a torus. But because the interval length was too small, we have a large number of intervals composing this cyclical behavior. In addition, the fixed length interval is out of sync with the actual period of the phase, because different points in space are sampled on subsequent iterations around the torus.

Similarly, SPEC2000's `bzip` compresses and decompresses the data twice, at compression levels 7 and 9. Thus, execution bounces between compression and decompression three times, as seen in Figure 2. Each iteration around the looping structures corresponds to compression of a block of data, as seen in the `gzip` plots. There are two looping structures within the compression phase - these correspond to compressing blocks with different entropy properties. `bzip` performs run-length encoding on its front end, and more time is spent in the run-length encoder on blocks with more contiguous sequences of repetitive bytes.

As seen from these two examples, the majority of a program's execution is spent in loops. The average number of instructions per loop iteration can change over time. For example, fewer instructions are typically needed to decompress a block of data than to compress a block of data. This means

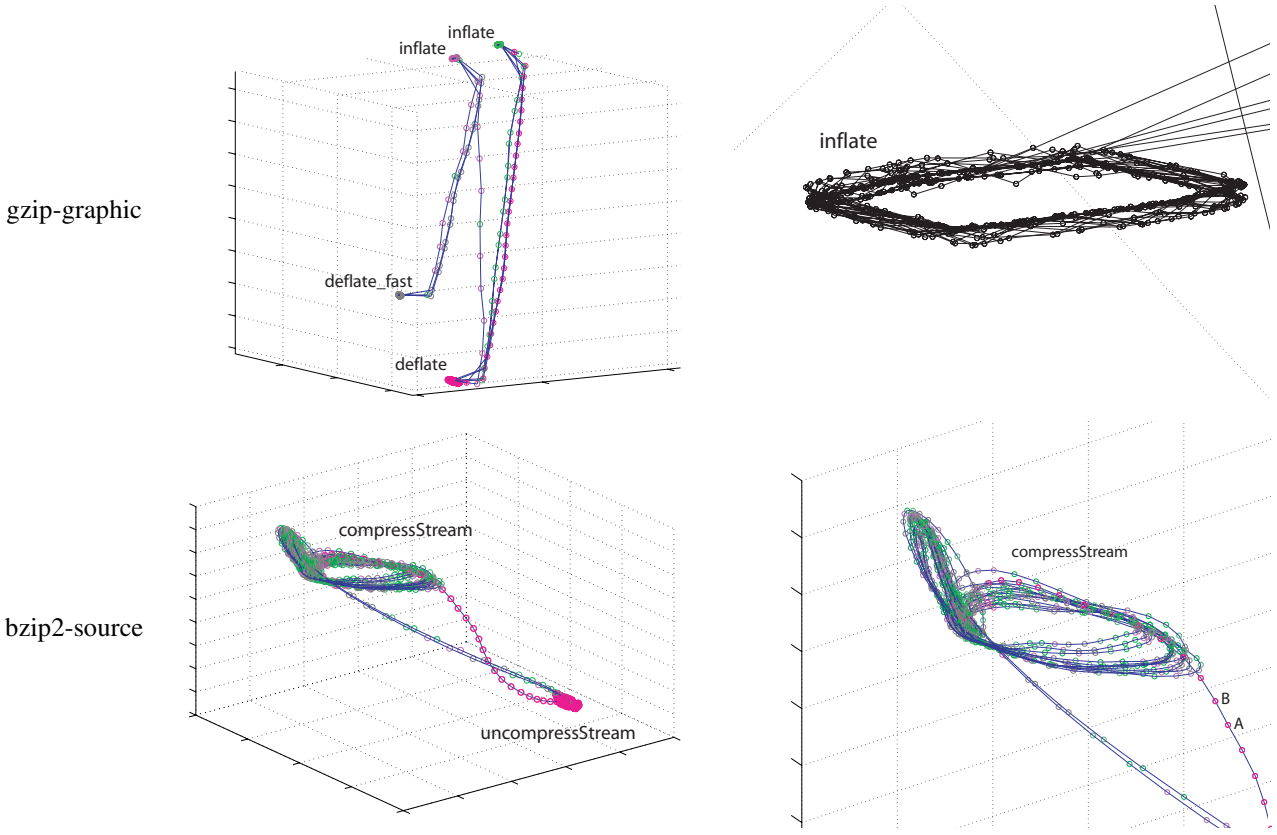


Figure 2: The three dimensional non-accumulated representation of `gzip-graphic` and `bzip2-source`. Each point represents an interval during execution, and the line connecting the points represents the execution order in time. The right figure of `bzip2-source` has two points labeled A and B, which indicate two temporally adjacent intervals of program execution. The figure on the left plots the entire execution, while the figure on the right zooms in on a looping region in the execution. The looping structures are traversed once for each block of data compressed or decompressed.

that the period lengths are not stable over time. An ideal fixed interval length for one section of execution (compression) may be dissonant with another portion of the program's execution (decompression). Since no single interval length will do a good job representing the program, we need variable length interval lengths that adjust to the period of the program's current behavior pattern.

Figure 2 shows that it is possible to see periodic behavior in programs by looking at a non-accumulative representation of the program's code space usage over time: the periodic behavior of programs results in cyclic patterns in these graphs. Both these programs also exhibit hierarchical behavior: there is a high-level behavior pattern between compression and decompression, and within each compression and decompression phase, there is a low-level behavior pattern corresponding to each block of data compressed or decompressed.

6.2 2D Accumulated Representation

Another way to examine the program's execution to detect phase changes is with an accumulative representation. With this approach, each interval of execution is represented with a basic block vector that tracks the total number of times each basic block is executed from the beginning of execution to the current interval. A fixed length interval size of 1 million instructions was used. Figure 3 shows a two-dimensional projection of accumulated basic block vector data for `bzip2-source`. Each point represents the accumulated basic block vector from the start of execution up to a fixed length interval boundary, and the lines connecting the points indicate the order in which the intervals were executed.

This graph shows that it is easy to find stable program behavior by looking at an accumulated representation, because stable behavior results in a straight line. If the program is executing the same distribution of basic blocks, the accumulated representation will show a straight line, because the same dimensions of the accumulated vector will be increased by the same quantities. Whenever the line bends, the program is executing a different distribution of basic blocks, and is therefore exhibiting a different behavior pattern.

SPEC2000's `bzip2-source` benchmark fills a 58MB buffer with back-to-back copies of a tarfile containing source

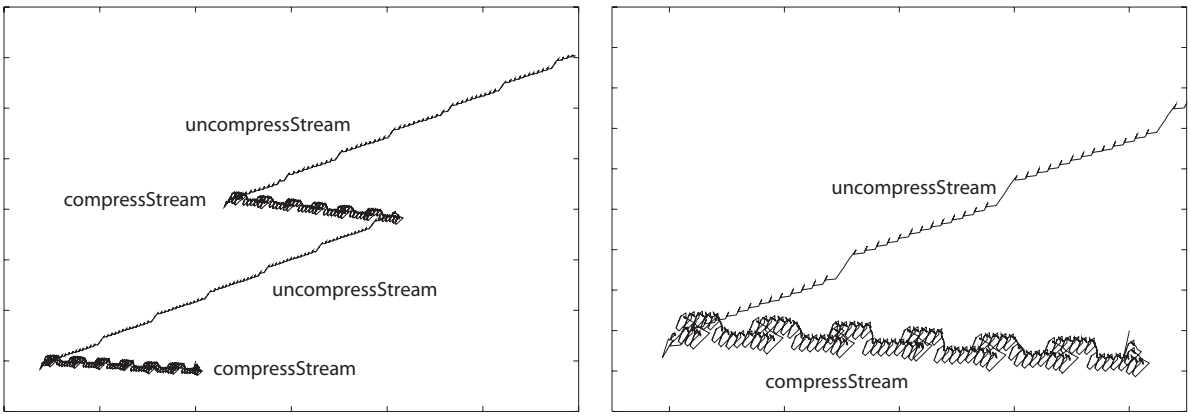


Figure 3: Two dimensional accumulated representation of `bzip2-source`. The figure on the left was produced by calculating the running sum of the vector data in Figure 2. Each point represents the start of execution up through the interval of execution being plotted. Therefore, each point represents a BBV projected down to 2 dimensions consisting of all of the program's execution from start of execution up through that fixed length interval of execution. The figure on the right shows detail of the bottom left corner of the figure on the left.

for some SPEC benchmarks. The 58MB buffer is large enough to hold 6.4 copies of the source tarfile. The tarfile contains a large number of null bytes at the end. This buffer is first compressed with a block size of 700KB, then decompressed, then compressed again with a block size of 900KB, and finally decompressed.

All these properties are visible in Figure 3. The 6.4 copies of the input file can be seen most easily in the decompression phases. The line shifts upwards every time the end of the original input file is reached, because of the large number of null bytes present at the end of the input file. This changes the entropy of the block, which causes `bzip2` to execute code in different proportions - more time is spent doing RLE decompression, and less time spent inverting the Burrows-Wheeler transform. Each circle during compression corresponds to compressing a block of data, and each spike during the decompression phase occurs when writing a decompressed a block of data. Thus, the block size is not directly visible, but the number of blocks is.

When a block size of 700KB is used, there are 14 blocks per copy of the original input file, and if you look closely, there are 14 little spikes within each "plateau" during the first decompression phase. When a block size of 900KB is used during the second decompression phase, there are 11 blocks per copy of the original input file, and there are 11 little spikes per plateau visible in the second decompression phase. The same patterns can be seen by counting loops in the compression phases.

We experimented with different forms of accumulated frequency data, from accumulating the past N BBVs together to represent each interval of execution, to accumulating the past X and future Y original BBVs together (creating a rolling window) to represent the interval, and we found that they did not provide any advantage over using the accumulative form starting from the beginning to examine the phase behavior.

7 Supporting Variable Length Intervals in SimPoint

The SimPoint algorithm was originally designed for fixed length intervals, so modifications were required to handle variable length intervals. Here we describe the changes which allow SimPoint to handle variable length intervals. The changes came primarily in two areas: handling an increased number of intervals, and dealing with the weights associated with variable length intervals.

7.1 Clustering for Many Intervals

For each level of the variable length interval hierarchy, we use k-means clustering to group similar intervals into clusters. The k-means algorithm is fast: each iteration is linear in the number of clusters, the dimensionality, and the number of intervals clustered. However, since k-means is an iterative algorithm, many iterations may be required to reach convergence. Because of this the algorithm can be slow in practice given a very large number of intervals. In our hierarchical approach, the lowest levels of the hierarchy contain a very large number of intervals; for example, `perl-splitmail` has 2.97 million intervals at the lowest level of its hierarchy.

To speed the execution of SimPoint on very large inputs, we sub-sample the set of intervals that will be clustered, and run k-means on only this sample. We sample the dataset (vectors) using weighted sampling. The number of desired intervals is specified, and then SimPoint chooses that many intervals (without replacement). The probability of each interval being

chosen is directly proportional to the weight of its cluster (the number of dynamically executed instructions it represents).

Sampling is common in clustering for datasets which are too large to fit in main memory [10, 25]. After clustering the dataset sample, we assign Phase ID labels to *all* intervals by locating the nearest cluster center (centroid) to each vector in the entire dataset, and assigning the label corresponding to the nearest cluster. For the experiments reported in this paper, we used a sample size of 100,000 vectors (i.e., no more than 100,000 intervals are used for clustering in SimPoint).

A smaller number of intervals reduces the time required for clustering, as shown in Figure 4. This graph shows the time it takes to run SimPoint for different number of intervals. It shows that if 15% of the intervals are used to create the clustering, then the clustering can be performed in a matter of minutes. Figure 5 shows for these same points the standard deviation (distance from interval to cluster center) among all of the intervals grouped into the final clustering. This also shows that, if at least 15% of the intervals are used to create the sample dataset, which are used to pick the centroids, we achieve about the same deviation as using all of the clusters.

A large number of intervals also leads to more potential phases that can be discovered. This is a property of the granularity at which a program is examined: at coarse granularity, few large intervals represent a high level view of program behavior and complete characterization can be achieved with a few phases; at fine granularity, many small intervals reveal detailed program behavior and more phases are needed for characterization. We use the k-means algorithm to classify code signatures from each interval of execution into k clusters (phases), where k ranges over $1..N$. SimPoint picks a single k from this range that is a good characterization of program behavior. We found a good range for setting N (max K) for the SPEC 2000 benchmarks based on the interval granularity chosen when using a fixed length granularity between 1 to 100 million interval size [24].

When using smaller interval sizes or variable length intervals, it is not necessarily clear what to set N to, especially with a large number of intervals. Therefore, we use a simple heuristic to quantify an upper bound on the number of expected phases. This value is computed as the square root of the total number of intervals: $N = \sqrt{|intervals|}$. Empirically we discovered that as the granularity becomes finer, the number of phases discovered increases at a sub-linear rate. The upper bound defined by this heuristic works well for the SPEC benchmarks. We note, however, if SimPoint continuously picks the number of phases to be N, then this value should be increased sufficiently so it is no longer picked by SimPoint [11]. It is also possible to expedite the search for k by sampling through the range of $1..N$ (e.g., only trying even values for k would reduce the search space by half, or only trying every 5th value for k would reduce the search space by 5 times, etc).

7.2 Consuming Variable Length Intervals

Different variable length intervals can represent different proportions of a program's execution, as opposed to fixed length intervals which each represent the same proportion. Each variable length interval has an associated weight we denote w_i, which represents the percentage of the total program execution for that interval. We have modified several parts of SimPoint so that it handles these weights.

The k-means clustering algorithm has two steps that it repeats: determining which cluster each interval belongs to (called the expectation step), and repositioning each cluster center to the mean of the intervals that it owns (called the maximization step). The expectation step is not changed by variable length intervals. The maximization step handles weights w_i by applying them during the recomputation of the cluster centers. Cluster j is computed as the weighted mean of the variable length intervals x_i that belong to that cluster:

$$c_j = \frac{\sum_{i=1}^{n} w_i x_i m_{ij}}{\sum_{i=1}^{n} w_i m_{ij}}$$

Here $m_{ij} = 1$ if interval i belongs to cluster j, and 0 otherwise, and m_{ij} is determined during the expectation step. The resulting k-means algorithm behaves just like the k-means algorithm for fixed length intervals, but larger intervals have more influence than smaller intervals over the cluster center locations.

The BIC criterion that we use to choose the best clustering also needs modification to handle variable length intervals. The BIC is the log likelihood of the clustering minus a complexity penalty. We adjust the log likelihood, but keep the penalty the same. The likelihood calculation sums a contribution from each interval, so larger intervals should have greater influence. The weighted log likelihood becomes:

$$\mathcal{L} = \frac{n \sum_{i=1}^{n} w_i \log Pr(x_i)}{\sum_{i=1}^{n} w_i}$$

where $Pr(x_i)$ is the probability of interval x_i. In our case, this probability comes from the k-means clustering model of a mixture of spherical Gaussians:

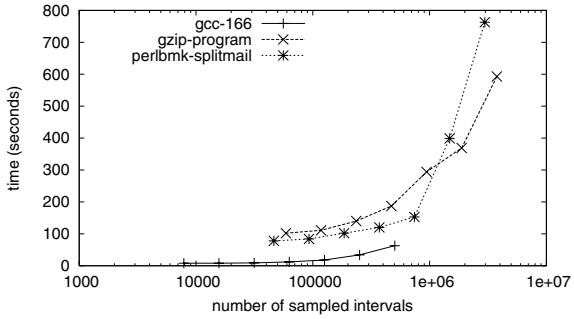
$$Pr(x_i) = \frac{w_{(i)}}{\sum_{i=1}^{n} w_i} \frac{\exp(\frac{1}{2\sigma^2}||x_i - c_{(i)}||^2)}{(2\pi\sigma^2)^{d/2}}$$

Here $c_{(i)}$ is the cluster center that x_i belongs to (the center closest to x_i), and $w_{(i)}$ is the weight associated with cluster $c_{(i)}$ (the sum of the weights of all points belonging to the cluster; this is the denominator in the earlier equation for c_j). The total weighted log likelihood function then simplifies to:

$$\mathcal{L} = \frac{n \sum_{i=1}^{n} w_i \log w_{(i)}}{\sum_{i=1}^{n} w_i} - n \log \sum_{i=1}^{n} w_i - \frac{dn}{2} \log(2\pi e \sigma^2)$$

We must also compute the variance σ^2 of the clustered intervals in a way that accounts for the weights:

$$\sigma^2 = \frac{\sum_{i=1}^{n} w_i ||x_i - c_{(i)}||^2}{d \sum_{i=1}^{n} w_i}$$



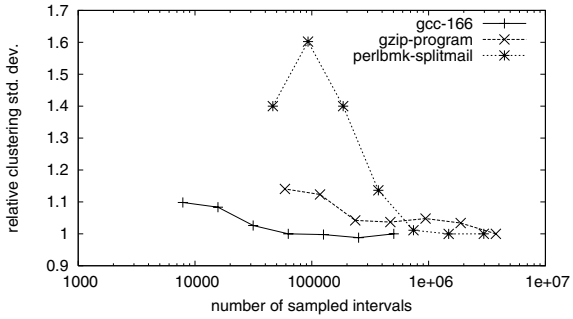


Figure 4: (Left graph) The vertical axis shows the amount of time (in seconds) required to run the SimPoint tool to cluster several program traces with different sample sizes. Sample sizes are shown on the horizontal axis, with the rightmost points being the total trace (no sampling used). Using smaller samples can greatly reduce the time needed to run SimPoint. These times are for one run of SimPoint k-means clustering with k set to 10.

Figure 5: (Right graph) The vertical axis shows the relative clustering standard deviation, and the horizontal axis shows the number of intervals clustered. Clustering only a sample of the vectors reduces time required, but samples which are too small could result in poor clusterings. Here we vary the number of intervals clustered using SimPoint's sampling functionality, from the whole program (rightmost points) to sampling a smaller number of intervals (moving left). The clustering standard deviation is the average distance of each vector from its closest cluster center. This plot shows the standard deviation for a sample divided by the standard deviation based on clustering all the data, giving a relative standard deviation. Lower values are better.

All these general equations can be simplified in the common case that $\sum_{i=1}^{n} w_i = 1$.

These changes do not change the algorithms with respect to fixed length intervals. In other words, setting all $w_i = 1/n$ (as in fixed length intervals) would produce the same results as the former SimPoint algorithm. Thus these changes allow SimPoint to smoothly handle both fixed length and variable length intervals.

8 Initial Study of Using SimPoint with Variable Length Intervals

In this section we present an initial investigation into using SimPoint with variable length intervals. First we describe an initial algorithm for breaking a program's execution into variable length intervals that are better aligned to phases compared to fixed length intervals, creating a hierarchy of these, and then using them with SimPoint.

8.1 Creating a Hierarchy of Variable Length Intervals

We examine an approach similar to Shen *et al.* [26], but based on code instead of data. The approach we examine collects a trace of loop branches, procedure calls, and procedure returns executed by each program. We use Sequitur [16, 22] to find patterns in the loop, call, and return traces, and we use the Sequitur output to generate variable length intervals. Each of these steps will now be described in more detail.

8.1.1 Collecting Traces

We instrument programs with ATOM [30] to collect traces of each loop branch, procedure call, and procedure return executed. In addition, we collect a trace of the number of instructions that execute between tracked events (loop branches, calls, and returns).

8.1.2 Finding Patterns in the Traces

We treat this branch trace as a string, and use Sequitur [16, 22] to find hierarchical patterns in this string. Sequitur takes a string and constructs a grammar with a rule "S" that expands to the input string. It tries to build the most compact grammar possible. For example, given the string "aabaabaac", Sequitur may produce the following grammar:

```
S->112c
1->2b
2->aa
```

The terminology is: "`1->2b`" is a rule, "2b" is a production, "2" is a nonterminal, and "b" is a terminal. A symbol is a terminal or a nonterminal.

Sequitur will generate a rule if the rule can be used more than once in the grammar. Rules that are used only once are eliminated. These two constraints are repeatedly applied to the input string to build the grammar. Sequitur is a linear-time algorithm, and we use it to find patterns in our loop, call and return traces. We only run Sequitur on the trace of loop, call, and return events: Sequitur does not know how many instructions executed between events. This information would

confuse Sequitur, because Sequitur will only generate a rule when it finds an exact match between two productions.

8.1.3 Generating Variable Length Intervals

We next use the results of Sequitur to determine how to break the program's execution (trace) into variable length intervals. Given a Sequitur grammar that represents the entire loop, call, and return trace, we need to determine where each variable length interval will begin and end. The symbols in rule "S" provide convenient endpoints for variable length intervals, because they represent the boundaries of the largest repetition patterns found by Sequitur. The "S" rule often also contains symbols that represent very small portions of the program's execution, so we do some filtering. We count the number of instructions represented by each symbol in "S," then sort the counts, and generate intervals for the symbols in "S," starting from the heaviest-weight symbol, until we have generated intervals for 90% of the program's execution, or we find a symbol that expands to less than 10,000 instructions, whichever comes first.

The remaining symbols are placed into transition intervals, and adjacent transition intervals are combined into larger transition intervals [18]. The result is a partitioning of the program's execution trace into variable length intervals, where no non-transition interval is smaller than 10,000 instructions. Transition intervals are still intervals: our variable length intervals cover 100% of the program's execution. The 10,000 instruction threshold is used as a filter for the Sequitur grammar to ensure that we have intervals for all of a program's large behavior structures, and to keep us from generating many small intervals for insignificant behavior structures.

We take the variable length interval trace described above and build a loop and procedure code vector to represent each variable length interval, which each dimension in the vector represents the number of times the procedure call, return or loop was executed for that interval. These variable length intervals represent the leaf level (the lowest level) in our variable length hierarchy. The next step is to create a hierarchy of the variable length intervals to choose from, which is done using a combination of SimPoint and Sequitur as described in the next section.

8.2 Creating a Hierarchy of Variable Length Intervals

Now that we have a method for partitioning a program's execution into variable length intervals, we build a hierarchy of variable length intervals that correspond to the program's phase behavior.

8.2.1 Constructing the Hierarchy

When we divided each program's execution into variable length intervals, Sequitur conveniently provided us with a hierarchical representation of the patterns in our loop, call, and return trace. But Sequitur relies on exact matches - a single difference in a pair of strings results in the generation of two different rules (or no rules at all). This restriction limits the depth of the hierarchy that Sequitur generates.

Our goal is to build the deepest hierarchy possible, so all the patterns in the program's behavior are visible, even at the largest of scales. To do this, we extend Sequitur's hierarchy by allowing approximate matches. This is done by taking the vectors that correspond to the variable length intervals found in the previous section, and running them through SimPoint to find similar vectors. SimPoint produces a set of labels, which map each vector to a Phase ID. Given a set of Phase ID labels, we map each interval of execution to its Phase ID, producing a trace of Phase IDs, and we run this Phase ID trace through Sequitur. The resulting Sequitur output is the second level of the hierarchy. Finally, we merge the original hierarchy and the second level hierarchy to create the overall variable length interval (VLI) hierarchy structure.

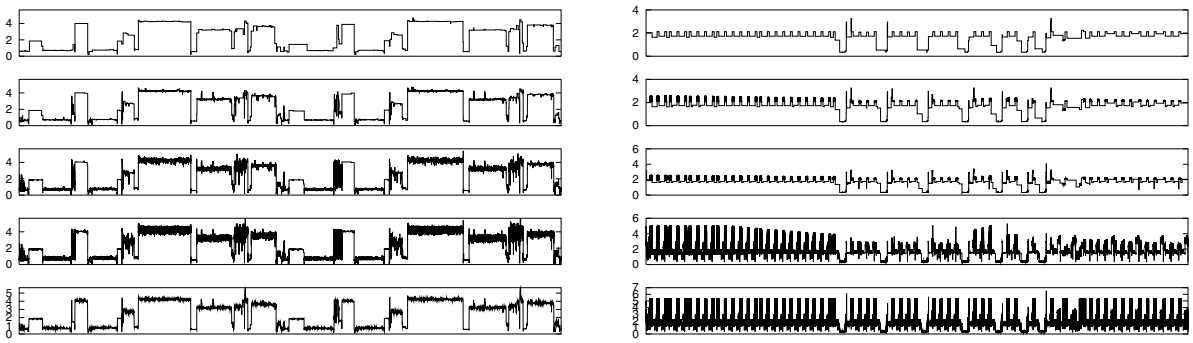
We iterate this process, generating vectors, running SimPoint to produce traces of Phase IDs, and running Sequitur on the Phase IDs, until a small number of symbols remain in Sequitur's "S" rule (< 10).

8.2.2 Hierarchy of Variable Length Intervals

One challenge is showing that the hierarchies we build actually represent important phase transitions at the different levels of the hierarchy. We do this by showing time-varying graphs where we plot the average IPC for the complete execution of each VLI region for a given level of the hierarchy. Figure 6 shows the hierarchy for two of the programs with complex phase behavior, and 5 hierarchy levels are shown for each program. The top graph for each program is the highest level in the hierarchy and the lowest levels are the bottom graphs for each program. IPC is on the y-axis, and the program's execution (time) is on the x-axis. For each variable length interval we plot the average IPC (shown on the Y-axis) for the VLI over that interval's complete execution (corresponding X-axis). These graphs show that at all levels of our hierarchy, the variable length intervals we create reflect the actual behavior of the program at the lowest level. It shows that our algorithm is grouping together the program's execution into variable length intervals that represent phase changes at these different granularities. At the highest levels, the structure of the variable length intervals is visible on these graphs.

8.3 Using the Hierarchy of Variable Length Intervals with SimPoint

Now that we have a hierarchy of intervals, we need to select a set of intervals that meets our needs. The intervals generated at each level of the hierarchy are highly variable in length. Most applications, including SimPoint, require intervals that are more homogeneous in length. We describe an algorithm for interval selection next.



gcc-166 galgel

Figure 6: Time-varying graphs showing average IPC per variable length interval for each level in our hierarchy. The highest levels are at the top of each graph. IPC is on the y-axes, and time is on the x-axes. At the highest levels, the structure of the variable length intervals is visible on these graphs. The bottom-most graphs show the average IPC for fixed length intervals of ten million instructions.

8.3.1 Selecting Intervals from the Hierarchy

We assume our goal is to create a set of variable length intervals that meet a minimum and maximum interval length requirement for simulation purposes. To build a set of intervals that satisfy the length requirements, we perform a postorder depth-first traversal of the hierarchy, starting from "S". For each production, we count the number of instructions represented by each symbol. If each symbol expands to at least the minimum number of instructions, or if accumulating the instruction counts for all symbols in the production results in more instructions than the maximum allowed, we create an interval for each symbol in the production. Otherwise, we place all the instructions in the entire production into one interval. Either way, the interval(s) that result are returned, and they may be further accumulated at a higher level.

It is important to note that the variable length intervals chosen are broken up based upon procedure call and loop boundaries guided by the Sequitur grammar. When working with fixed length intervals, it is certainly possible to choose a "bad" interval length that is dissonant with the actual periodic behavior of the program. But when working with the variable length intervals in our hierarchy, it should not be possible to choose a "bad" interval length, because all the interval lengths that appear in our hierarchy are tuned to the program's periodic behavior found in its control flow.

8.3.2 Initial SimPoint Results

We now evaluate the use of the variable length hierarchy to choose a small number of large representative samples from an application to guide program analysis or simulation with SimPoint. For some applications it is advantageous to pick a small number of reasonable sized samples for this analysis. It was recently shown that phase information can accurately guide SMT simulation [3], but this approach requires a small number of phases to be practical. Using a large interval length will result in a smaller number of phases. In addition, some research groups (e.g., Intel) simulate large samples (on the size of 300 million instructions or more), so that they do not have to deal with warmup issues [23].

The SimPoint algorithm [28] is a phase classification algorithm based on the k-means clustering algorithm. SimPoint groups together *fixed length intervals* of execution with similar behavior. We modified the SimPoint algorithm to group together *variable length intervals* as described in Section 7. We use the algorithm described in Section 8.3.1 to select a set of variable length intervals from our hierarchy, and we run these intervals through our VLI SimPoint (SimPoint modified to support variable length intervals), to produce a phase classification. We first examine the CoV of CPI to make sure that the intervals in each cluster have similar CPI. The CoV calculation is described in Section 4.

Figure 7 compares the results of our variable length approach and standard fixed length SimPoint for large interval lengths. Fixed length interval lengths of 100M, 300M, 500M, and 1000M are compared against using variable length intervals chosen from the VLI hierarchy in the range of 100M-500M and 500M-1000M instructions. The CoV of CPI results show that we are able to achieve a lower CoV of CPI with our variable length intervals compared to fixed length intervals. This means that the clusterings produced by SimPoint are more homogeneous (in terms of CPI) with our variable length intervals, compared to fixed length intervals. This is expected, because we align our variable length intervals with changes in each program's behavior patterns, so our variable length intervals will exhibit more similarity.

Figure 8 shows the percent error in estimated CPI. This graph shows that we are usually able to pick better representatives for each SimPoint cluster with variable length intervals, compared to fixed length intervals. This follows from the CoV of CPI results, because it is easier to pick a good representative for each cluster when the clusters are more homogeneous.

Figure 9 shows the number of instructions used to repre-

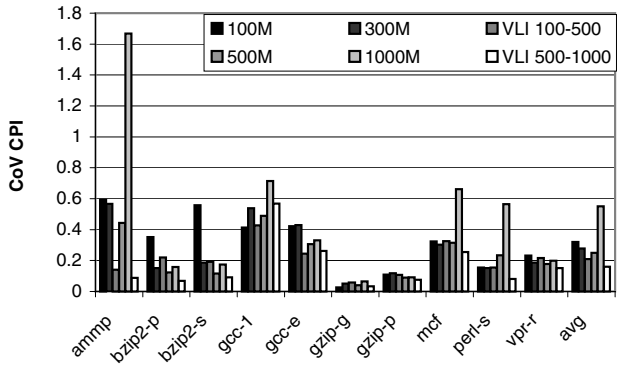


Figure 7: Coefficient of Variation of CPI. 100M, 300M, 500M, and 1000M are results for standard fixed length SimPoint run at the corresponding granularity, while VLI 100-500 is our variable length approach given a minimum interval length of 100M and a maximum of 500M, and VLI 500-1000 corresponds to a minimum of 500M and a maximum of 1000M.

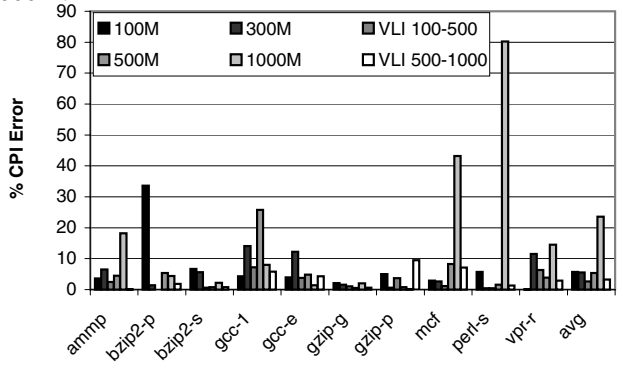


Figure 8: Percent error in SimPoint estimated CPI for the vectors in Figure 7

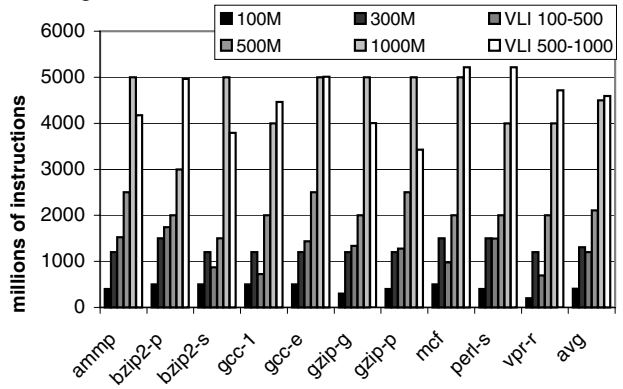


Figure 9: Number of instructions used to represent each program for CPI estimation. The same vectors from Figure 7 are shown.

sent the program. Our 100M-500M variable length intervals require about as many instructions to represent the program as fixed length 300M, and variable 500M-1000M requires about as many instructions as fixed length 1000M.

The CoV of CPI and percent CPI error results show that variable length intervals are more important at large granularities. At these coarser granularities, significant portions of a program's phase behavior can lie in a single interval. At high granularities, even a slight interval misalignment can result in substantial accuracy loss. The variable length intervals accurately encapsulate phases at all levels and provide accurate representation of a workload at any granularity. Overall, the results show that variable length intervals provide more consistent results than fixed length intervals.

9 Summary

The goal of this paper is to motivate the need to move towards variable length intervals aligned to a program's actual phase boundaries and to describe the changes made to SimPoint to support variable length intervals.

Prior work on automated phase classification [28] focused on using fixed length intervals for identifying phase behavior in programs. But when interval lengths are out of sync with the phase behavior exhibited by the program, many more intervals are required to represent each behavior pattern, compared to an appropriately sized variable length interval. By examining how programs use their code space with 3D non-accumulative graphs and 2D accumulative graphs, we can see how fixed length intervals split up the execution space, and that better representations can be made if intervals are aligned to a program's actual phase boundaries. More importantly, these graphs showed that programs have a hierarchy of phase behavior at many different granularities, which is more difficult to discover with fixed length intervals.

Two changes to SimPoint were needed to support this work. The first was the ability to support many more intervals than the previous design. This was accomplished by performing the clustering on a randomly selected subset of intervals, and then classifying the remaining intervals using that clustering. The other major change was supporting non-uniform weights in k-means and the BIC criterion. These changes were described in Section 7.

Finally, we presented initial results using SimPoint and Sequitur with variable length intervals for creating a hierarchy of variable length intervals.

Acknowledgments

We would like to thank the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by NSF grant No. CCR-0311710, NSF grant No. ACR-0342522, UC MICRO grant No. 03-010, and a grant from Intel and Microsoft.

References

[1] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, pages 245–257, 2000.

[2] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *35th International Symposium on Microarchitecture*, December 2002.

[3] 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.

[4] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[5] P.J. Denning and S. C. Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, March 1972.

[6] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. In *36th International Symposium on Microarchitecture*, December 2003.

[7] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, February 2002.

[8] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[9] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Architectures and Compilation Techniques*, October 2003.

[10] F. Farnstrom, J. Lewis, and C. Elkan. Scalability for clustering algorithms revisited. *SIGKDD Explorations*, 2(1):51–57, 2000.

[11] G. Hamerly, E. Perelman, and B. Calder. How to use simpoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review*, 31(4), March 2004.

[12] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[13] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *Workshop on Workload Characterization*, September 2003.

[14] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *36th International Symposium on Microarchitecture*, December 2003.

[15] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Processor power reduction via single-ISA heterogeneous multi-core architectures. *Computer Architecture Letters*, 2, April 2003.

[16] J. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[17] J. Lau., S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[18] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *11th International Symposium on High Performance Computer Architecture*, February 2005.

[19] W. Liu and M. Huang. Expert: Expedited simulation exploiting program behavior repetition. In *International Conference on Supercomputing*, June 2004.

[20] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[21] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *International Symposium on Code Generation and Optimization*, March 2005.

[22] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. In *The Computer Journal vol. 40*, 1997.

[23] 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 *Sumbmitted to MICRO*, 2004.

[24] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[25] F. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, 3(2):131–169, 1999.

[26] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[27] 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.

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[29] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[30] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.