Wavelet-Based Phase Classification

Ted Huffmire and Tim Sherwood
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
huffmire@cs.ucsb.edu, sherwood@cs.ucsb.edu

Abstract

Phase analysis has proven to be a useful method of summarizing the time-varying behavior of programs, with uses ranging from reducing simulation time to guiding run-time optimizations. Although phase classification techniques based on basic block vectors have shown impressive accuracies on SPEC benchmarks, commercial programs remain a significant challenge due to their complex behaviors and multiple threads. Some behaviors, such as L2 cache misses, may have less correlation with the code and therefore are much harder to capture with basic block frequency vectors.

Comparing the similarity of two or more intervals requires a good metric, one that is not only fast enough to analyze the full execution of the program, but that is also highly correlated with important performance degrading events (such as L2 misses). We examine the use of many different interval similarity metrics and their uses for program phase analysis across a range of commercial applications and show that there is still significant room for improvement. To address this problem, we introduce a novel wavelet-based phase classification scheme that captures and compares images of memory behavior in two or more dimensions. Over a set of five commercial applications, we show that a wavelet-based scheme can strictly outperform a broad range of prior metrics both in terms of accuracy and overhead.

Categories and Subject Descriptors: C.2 Performance of Systems: Measurement techniques

General Terms: Measurement, Performance

Keywords: Optimization, Phase Analysis, Phase Classification, Phases, Program Behavior, Wavelets

1. INTRODUCTION

Most computer programs, including commercial applications, exhibit reoccurring behavior over time that can be broken down into phases. Knowledge of this behavior can be exploited to improve system performance by activating specific optimizations in response to changes in the current phase. Phases have proven useful in directing compiler optimizations [1], reducing the power consumption of processors [4] [5] [6] [11] [12], reducing the overhead of program profiling [17] [18], and speeding up architectural simulation [25] [8].

Due to the increasing impact of memory latency on overall system performance, it is especially critical that phase behavior in the memory hierarchy be captured efficiently and accurately. One popular phase analysis toolkit, SimPoint [10], performs phase classification by analyzing the number of times each basic block of a computer program is executed during a fixed window of executed instructions called an interval. This technique has been shown to work on a variety of benchmarks and has the significant advantage that it is not tied to a particular architecture configuration allowing it to be used in studies where the architectures are modified.

One of the limitations of current program phase analysis is that they derive both their descriptional power and computational efficiency from carefully crafted metrics. These metrics, such as dynamic branch counts [7], working set signatures [5], basic block vectors [22] [23], or any of the other recently proposed metrics [21] [15] [7] are expected to capture the essence of a program's execution. In other words, a successful metric will reflect any important dynamic change in program behavior with a resulting change in the metric. The problem is that in many domains, such a metric may or may not be known. For SPEC-like programs, the above listed metrics have been proven to be effective, but they can be problematic when analyzing memory bus traces, I/O behavior, network traffic, or other hardware elements which are more loosely correlated with program code execution.

For example, although many phase-based optimization schemes have been proposed and evaluated on SPEC benchmarks, less attention has been paid to commercial applications, which exhibit complex, multi-threaded behavior. Web browsers, image editing software, and word processing applications typically have large working sets and high utilization of the L2 cache. In contrast, SPEC programs, except for mcf, have almost negligible main memory behavior. While existing phase analysis techniques have been very useful, unfortunately there are some hardware metrics that are very difficult to accurately capture just by analyzing code execution [**?**] [14]. For example, the memory bus behavior is not well described by SimPoint because it is not strongly correlated with the code. Since there is a large variance in the number of L2 misses among the intervals grouped together by SimPoint, the metric of basic block vector distance does not do an adequate job in these cases.

PACT'06, September 16–20, 2006, Seattle, Washington, USA.

In the ideal case, a metric could be found that would capture the time varying behavior of the memory bus without requiring detailed cache simulation, and intervals with similar memory bus behavior should be clustered together in this metric space. In evaluating many different structures in terms of their ability to correlate with the memory bus behavior of commercial applications, we discovered a problem with basic block vectors. Since they are simply a list of *execution frequencies*, basic block vectors have no notion of *time within an interval*. While some of the other proposed structures (such as local stride) combine the idea of time (size of stride) and frequency (number of occurrences of a stride), they work by exploiting *a priori* knowledge of common access patterns. Instead of this approach, in this paper we present a novel general-purpose method for classifying program phases that combines both time and frequency information through the use of wavelets.

Our novel wavelet-based phase classification algorithm is inspired by the idea of wavelet-based image query. Wavelets provide a way of efficiently summarizing a matrix of data with hierarchical precision in both the time and frequency domains. In image query systems, wavelet signatures can be used to find images that most closely match a query, and we apply this technique to program phase analysis by considering the similarities between snapshots of the memory behavior. We empirically evaluate the effectiveness of wavelet-based phase classification and show that for the L2 miss classification problem it is both significantly more accurate and even faster than prior techniques.

The primary contributions of this paper include:

- A comparison of the effectiveness of various phase classification metrics at capturing memory bus behavior for a range of commercial applications
- A novel wavelet-based phase classification technique that accurately classifies memory bus behavior through wavelet signature clustering
- We show that wavelet-based phase analysis can classify memory bus behavior 80% more accurately than basic block vectors and over 50% better than the best prior techniques.

2. RELATED WORK

2.1 Basic Block Vectors

Computer programs exhibit repeating behavior during their execution. One well-known method of performing phase classification identifies these phases by analyzing the execution history of the program [22] [23] [24]. This history is summarized in a structure called a basic block vector, which records the number of times that every basic block in the program executes during a fixed execution interval. Two intervals can be compared by computing the Euclidean distance between their basic block vectors. A statistical technique called k-means is employed to group the intervals into clusters of similar behavior. Initially, every interval is assigned to a random cluster, and the center of each cluster is calculated. Next, every interval is assigned to the cluster whose center is closest to it. This process repeats until a stable clustering is found.

2.2 Alternative Classification Structures

One of the primary benefits of using basic block vectors is that they are independent of the underlying architectural details. Running the same program on two different processor configurations will yield the same result, as long as they share the same ISA. Unfortunately, basic block vectors do not capture all micro-architectural behaviors successfully, such as L2 cache misses, which are important for optimizing the performance of systems. Since basic block vectors are simply a list of execution frequencies, they have no notion of time within an interval. This limitation leads us to look for alternative structures besides basic block vectors. Lau et al. propose several different ways of performing phase classification besides traditional basic block vectors, such as local stride and global stride [15]. Since we will compare our wavelet-based technique against these structures, we describe them in detail in Section 4.1. Although these techniques are also independent of the micro-architecture, they rely on *a priori* knowledge of common access patterns. We do not analyze structures that are dependent on the micro-architecture, such as the instruction mix, branch prediction accuracy, cache miss rate, and IPC [7].

2.3 Applications of Phase Analysis

Phase classification is useful in dynamic optimization. For example, Das et al. have developed a dynamic optimization technique of dividing a program into regions, performing phase detection on each region, and combining the results [3]. Phase detection can also direct compiler optimizations [1] and reduce the power consumption of processors [4]. Phase classification is also helpful in making program profiling more efficient and accurate. A cycle-accurate trace of a long-running program requires enormous space and time resources. Phase analysis can guide the sampling of this data to produce useful trace files with much lower overhead [17] [18]. Phase classification can also make architecture simulation more efficient by guiding the selection of a sample of the execution to simulate [25] [8].

2.4 Prior Wavelet-Based Techniques

We are not the first to apply wavelets to the problem of phase classification. Shen et al. use wavelets to predict the locality phases of a program [21]. Their scheme uses single-level analysis and training runs to identify behavior changes to accurately determine the best place for *phase markers*, but wavelets are never used as a method of comparing similarity - only as a time-frequency analysis method. Their approach is useful for dealing with variable length intervals and reducing software instrumentation overhead, neither of which is related to the problem we are solving in this paper. Our approach requires no software analysis or training (which is good for off-chip analysis) , and we show that the wavelet coefficients themselves hold significant potential as a similarity metric in their own right.

Wavelet transforms of images have been the basis for k-means clustering for the purpose of text segmentation [9] [20], and a similar approach has been used for the analysis of mammograms [2], but fuzzy c-means (FCM) was used rather than k-means. In order to make k-means work more accurately for time series data, Vlachos et al. perform a k-means clustering on the coarse wavelet coefficients and then use the results of this clustering to start a finer clustering [27].

3. FINDING PHASES WITH WAVELETS

Now that we have explained the limitations of current phase classification techniques, we now turn to a discussion of wavelets and how they are useful in overcoming these limitations. Wavelets are mathematical functions that are useful in a variety of scientific applications from digital signal processing to image processing. Unlike the Fourier transform, which captures frequency information only, wavelets encode both frequency *and* spatial information. This feature of wavelets makes them more effective at capturing some behaviors than basic block vectors, which have no notion of time within an interval because they are simply a list of execution frequencies.

We use an idea inspired by wavelet image query to analyze generic traces of data that could be network traces or I/O traces. However, for the purposes of exploring and evaluating our idea, we consider only memory bus accesses because we can more closely compare with past work in this area. To find phases with wavelets, we need to first gather the trace and summarize it in a 2D matrix which is in essence a "picture" of the trace. An example of this picture can be seen in the third step of Figure 1, which shows a grayscale plot of the memory accesses. Next, we divide this large matrix into a sequence of smaller matrices, one vertical "slice" for each interval of execution (1M instructions). Each "slice" is scaled down to an even smaller square matrix so that a Haar wavelet transform can be applied, resulting in a wavelet signature. This signature is a matrix of coefficients that is used by the k-means clustering algorithm to perform the phase classification, and the number of dimensions is the number of wavelet coefficients.

Our technique predicts the L2 miss phases by analyzing wavelet signatures of all L1 accesses (with no knowledge of which of those accesses will miss in either the L1 or L2 caches). We are *not* using L2 misses to predict L2 misses. We have posted the source code of our technique as well as instructions on compiling and running our code at the following URL: http://www.cs.ucsb.edu/∼arch/wavelet.

3.1 An Example of Wavelet Phase Detection

We now describe an example of wavelet phase detection as applied to L2 miss analysis. Our goal is to solve the problem of predicting main memory access behavior by analyzing the raw address stream. This allows us to compare directly to other techniques. We will discuss or algorithm parameters in Section 3.3. Figure 1 illustrates the steps of our design flow:

Trace File Generation – We first use the binary instrumentation utility Pin [16] to generate trace files for real commercial applications. A user of Pin writes a program called a pintool which runs in the same address space as the instrumented process, making it possible to inspect the values of every load. Both the application and all shared libraries needed by the program are instrumented.

Matrix Representation of the Trace – We next generate a 2D matrix of the memory accesses. The columns are execution time, and the rows correspond to the address of the load modulo M, which is the modulo size. Every twenty columns represent 1M instructions, and addresses are mapped to the y-axis by a function that we will describe in Section 3.3.

Dividing the Matrix into Slices – We next divide the matrix of the trace file into smaller matrices that each represent 1M instructions. Each of these "slices" is twenty columns wide.

Resizing the Slices – We next scale each "slice" down to a small square of size 16×16 because the wavelet transform we apply in the next step needs the dimensions of the image to be a square whose sides have length of a power of two. We found that this size provides the optimal trade-off between performance and accuracy.

Performing the Wavelet Transform – We next perform a 2D wavelet transform on each scaled matrix. We describe the details of this transform in Section 3.2. The result is a set of wavelet coefficients that we can use to perform k-means clustering. Unlike wavelet image query, we do not discard any of the coefficients.

Performing K-Means Clustering – We next perform k-means clustering on the wavelet coefficients. Since the size of the transform is 16×16, our data have 256 dimensions. The number of points to be clustered is the number of intervals in the program. The distance between two points is the Euclidean distance between their wavelet coefficients, but we first apply tuning weights to the coefficients using a scheme similar to [13] because the importance of a coefficient is affected by its 2D position. The result of k-means is that every interval in the program has been assigned to one of ten clusters.

3.2 Haar Wavelets

We now describe some background on how wavelets work. Stollnitz et al. provide a much more thorough primer on wavelets and their application to the field of computer graphics [26].

Reasons for Using Haar Wavelets – Many different types of wavelets exist, each with strengths and weaknesses for different applications. The most simple type of wavelet is a square wavelet known as a Haar wavelet [26]. We selected Haar wavelets for our technique because they are simple, fast, and memory-efficient. Haar wavelets have proven effective in determining how similar two images are. We wish to exploit this property for phase classification by determining how similar two intervals are. Two intervals with similar behavior will "look" similar, and Haar wavelets should be able to detect the similarity between the "pictures" of their behavior. The primary disadvantage of Haar wavelets is that they are not continuous. The heart of the Haar wavelet transform is averaging and differencing. Since averaging and differencing works on pairs of array elements, it may miss some high frequency changes that occur between even and odd elements.

1D Haar Transform – The 1D Haar wavelet transform is computed by performing an operation called *averaging and differencing* $O(\log N)$ times on an array of size N. For example, suppose we have an array of integers [8 6 2 4]. We compute the average of the first two elements 8 and 6, which is 7. Then, we compute the average of the second two elements 2 and 4, which is 3. After computing the averages, the next step is to compute the differences, which are known as the detail coefficients. Since 8 is one more than 7 and since 6 is one less than 7, 1 is the first detail coefficient. Similarly, since 2 is one less than 3 and since 4 is one greater than 3, −1 is the second detail coefficient. We store the averages in the first part of the array, followed by the detail coefficients. At this point the array is [7 3 1 − 1]. The next step is to compute the average of 7 and 3, which is 5. The final

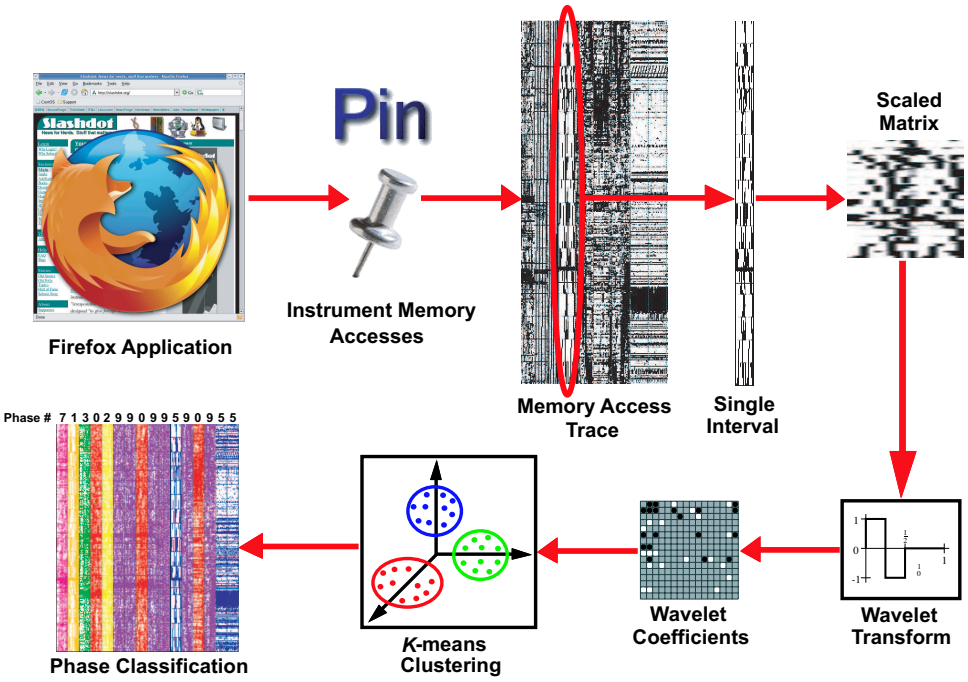

Figure 1: This figure shows the technique of using wavelets for phase classification. First, a trace of memory accesses is generated by instrumenting a real-world application such as Firefox. Next, a matrix of the trace file is generated in which the columns are execution time, and the rows are the addresses of each memory access modulo an integer M, which is the modulo size. This matrix can then be divided into a sequence of smaller matrices, one vertical "slice" for each interval of execution (1M instructions). Each "slice" is then scaled down to a smaller square matrix so that a Haar wavelet transform can be applied. This results in a wavelet signature, which is a matrix of coefficients that is used by the k-means clustering algorithm to perform the phase classification.

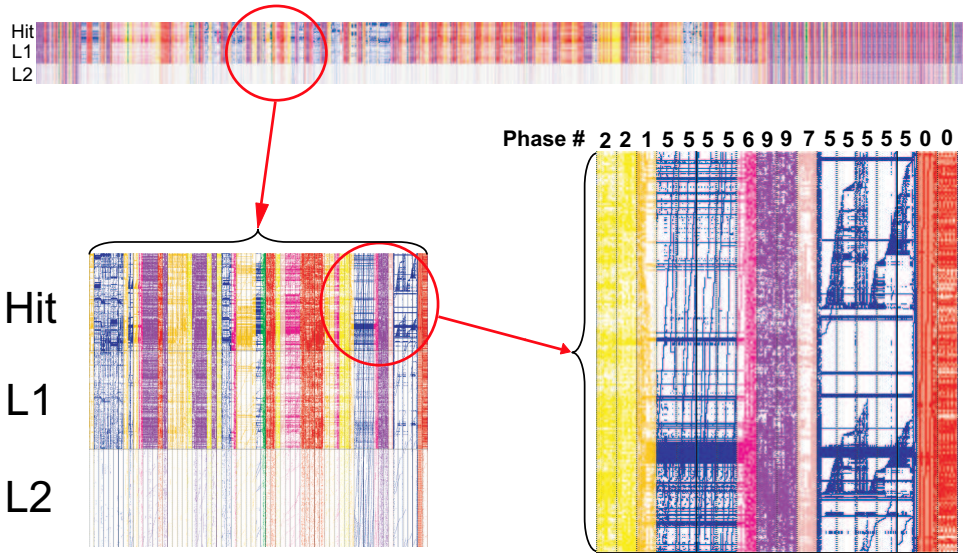

Figure 2: Phase classification of the cache behavior of Firefox using the wavelet technique. The x-axis is time, and each vertical slice represents 1M instructions. The image at the top has three horizontal bands. The top band shows L1 cache hits, the middle band shows L1 misses, and the bottom band shows L2 misses. The y-axis of each band is the address of the memory access modulo an integer M, which is the modulo size. The index above each slice corresponds to the phase. Note that our technique predicts the L2 miss phases by analyzing wavelet signatures of all L1 accesses (with no knowledge of which of those accesses will miss in either the L1 or L2 cache). We are not using L2 misses to predict L2 misses.

step is to compute the detail coefficient, which is 2 since 7 is two more than 5 and since 3 is two less than 5. The final transformed array is [5 2 1 −1]. The first element, 5, is the overall average of the entire array. The second element, 2, is a coarser detail coefficient, and 1 and −1 are finer detail coefficients. The inverse process yields the original array.

2D Haar Transform – The 2D Haar transform can be applied to images. An image is a 2D array of intensity values. The 1D transform is applied to every row of the image and then to every column of the result of the row transforms. The upper leftmost element of the resulting matrix contains the overall intensity of the image. If the original image is not square, it is necessary to either scale the image to a square image or to pad the image with blank pixels prior to applying the 2D transform.

Since the importance of a coefficient depends on its position, we apply a set of tuning weights to the coefficients. We restrict ourselves to the Y channel (intensity channel), and we use the tuning weights for "painted" rather than "scanned" query images, which are described in detail in [13]:

$w_y[0] = 4.04$
$w_y[1] = 0.78$
$w_y[2] = 0.46$
$w_y[3] = 0.42$
$w_y[4] = 0.41$
$w_y[5] = 0.32$

A pixel located at position (row,col) is weighted by w_y[bin(row,col)], where:

bin(i,j) = min(max(level(i),level(j)),5)
level(i) = $\lfloor(log_2(\text{i+1}))\rfloor$

3.3 Parameter Choices

We now explain the design choices we encountered when developing our wavelet-based phase classification technique, and we discuss how the parameters we selected satisfy our design goals. The two most important parameters are the scaled matrix size and the modulo size, and we discuss our quantitative analysis of the interaction of these two parameters in Section 5.3.

Matrix Representation of Trace File – We are interested in the behavior of programs as they execute over time, so it makes sense to have time in terms of instructions executed as the x-axis (the columns of the matrix). We use the y-axis (the rows of the matrix) for the memory accesses. An address is mapped to one of the 400 pixels in the y-axis of the large matrix by the function specified in Equation 1:

$$f(address) = ((address\%M) * (400/M)) \quad (1)$$

This function basically makes the matrix a picture of the memory behavior modulo the M, which we will call the *modulo size*. To ensure that the matrix is neither too dense nor too sparse to capture the memory bus behavior, we used 20 columns per million instructions in the x-axis (every 1/20 of the one million instructions is mapped to a column), and the y-axis has 400 rows. However, this dimension parameter is not of primary importance because the large matrix is scaled to a smaller matrix before any analysis is performed.

Modulo Size – The choice of modulo size (M) is important because this parameter affects the appearance of the plots of the memory behavior, which in turn affects our algorithm's ability to capture phase behavior. We chose M to be 16Kb, which is the L1 cache size. The reason for setting M to the L1 cache size is that memory accesses will eventually make it to the cache, and taking the modulo lets us see interesting striding behavior. Since the address of each memory access is mapped to the y-axis by the function $((address\%M) * (400/M))$, where M is the modulo size, a different value of this parameter will change the position along the y-axis to which a given memory access is mapped.

Interval Size – The number of instructions per interval is another design parameter. If we choose too large a granularity, we will not be able to capture behavior that reoccurs at a smaller time scale. However, if we choose too small a granularity, we will have too many intervals to process efficiently. The interaction between interval size and phases has been studied before, and we chose an interval size of one million instructions because it is a good granularity for capturing memory behavior and has been used by many prior works.

Scaled Matrix Size – The choice of size for the smaller matrix is important. If it is too large then the similarity is more a function of the small details (noise) in the trace, but if it is too small then there is no way to really even determine temporal similarity. We chose a small 16x16 matrix because it provides the optimal tradeoff between performance and accuracy, as we describe in Section 5.3. We used the default resolution translation algorithm provided by the Java 2 Platform Standard Edition (J2SE 5.0) for scaling the matrix down to a smaller size. Specifically, we used the method $getScaledInstance()$ from class $java.awt.Image$ with SCALE_DEFAULT as an argument. However, the scaling algorithm is not an important parameter because the choice of resolution translation algorithm has much less impact when scaling a large image down to a smaller size than when scaling a small image up to a larger size. We believe that either bicubic or bilinear interpolation is accurate enough for this purpose, but not nearest neighbor interpolation.

Wavelet Type – We chose the Haar wavelet transform because it is both fast and it worked, so we did not see a need to move to a more complex scheme at this point. We described the 1D and 2D Haar transforms in Section 3.2.

Clustering Algorithm – We selected k-means because it has proven to be very effective for phase classification in prior works. The purpose of clustering is to group intervals with similar behavior together. Initially, every interval is assigned to a random cluster. Each iteration of the algorithm determines the center of the cluster and assigns every point to the nearest cluster center. The algorithm iterates until a stable clustering is found. Since some randomness is involved, each execution of the algorithm may result in a different clustering. For this reason, some implementations such as SimPoint take the average of multiple runs of the algorithm. During each run, SimPoint makes a decision about when to stop iterating the clustering algorithm based on how stable the clustering is. Unlike SimPoint, our technique simply executes a fixed number of iterations of the clustering algorithm, rather than using a stopping condition since we do not yet know of a stopping condition for our technique. We chose a maximum number of phases of K=10 because it captures memory phases well and has been used by many prior works.

3.4 Visualizing the Clustering

Figure 2 shows the result of performing wavelet-based phase classification on a trace file of the memory accesses of Firefox, which was instrumented as it loaded a web page in 902M instructions. The index above each slice corresponds to the phase. There are three bands in the image. The top band corresponds to cache hits, the middle band corresponds to L1 misses, and the lower band corresponds to L2 misses. L1 misses have a lower density than hits, and L2 misses have a lower density than L1 misses. We are concerned with how well the L2 misses are classified. An ideal phase classification will group together those intervals with similar memory bus behavior. Our cache simulator has the following parameters: The L1 cache is a 16K, 2-way associative cache with a block size of 32 bytes. The L2 cache is a 1MB, 4-way associative cache with a block size of 64 bytes.

In order to improve our clustering algorithm, it is very useful to be able to see a picture of the clustering chosen by our algorithm. This would be trivial if we were only working with two or three dimensions, but our data have many more dimensions. Therefore, we project the multidimensional data onto two dimensions. Figure 3 shows the clustering in the form of a 2D plot that is a random projection of the wavelet coefficients of each interval. This plot is generated by multiplying an array containing the wavelet coefficients of dimension $\#Intervals \times \#Coefficients$ by a random matrix of dimension $\#Coefficients \times 2$ resulting in a matrix of dimension $\#Intervals \times 2$. Each row of this matrix is a 2D point. The intensity of each point corresponds to the phase of the interval, and the size of a point corresponds to the number of L2 misses in that interval. The phase numbers are shown for clarity.

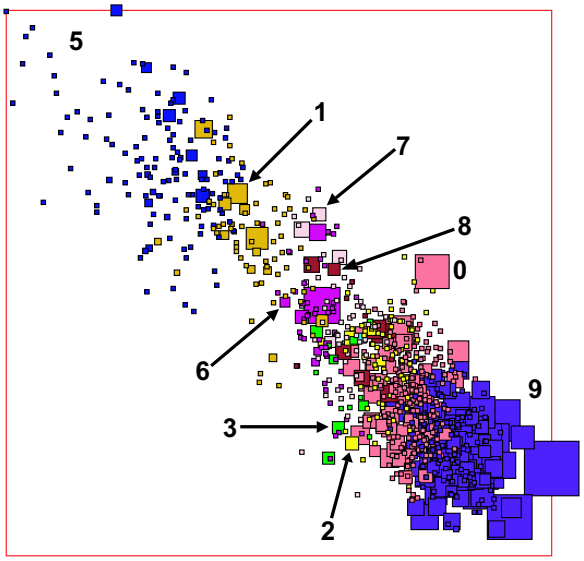

Figure 3: A 2D random projection of the wavelet coefficients for Firefox. There is one point for each interval in the trace file. The size of the point corresponds to the number of L2 misses in that interval, and the intensity of the point corresponds to the phase. The phase numbers are shown for clarity. The wavelet technique successfully clusters intervals with many L2 misses together. The points in the upper left correspond to intervals with the lowest density of cache hits, and the points in the lower right correspond to intervals with the highest density of cache hits.

4. COMPARISON OF STRUCTURES

In this section we describe the different phase classification structures that we evaluated besides our wavelet-based technique. We also describe the metric we used to compare how well a particular metric captures memory bus behavior. Finally, we describe our visualization utility for understanding memory phases.

4.1 Alternatives to Basic Block Vectors

We would like to see how well our wavelet-based phase classification technique performs in comparison to other previously proposed phase classification techniques. The methods of performing phase classification that we consider in this paper are:

- **Basic Block Vector** – This is traditional phase classification using a basic block frequency vector. We describe this technique in detail in Section 2.
- **Local Stride** – The frequency vector holds information about the strides of the memory accesses. The stride is the absolute value of the difference between the memory address accessed by a PC and the address previously accessed by the same PC. The k-th element of the frequency vector stores the frequency of accesses with a stride of k. We used vector sizes of 100 and 10,000. In the version that uses a vector size of 100, all strides that are greater than 100 are ignored. In the version that uses a vector size of 10,000, all strides greater than 10,000 are ignored.
- **Local Stride with PC Hash** – The local stride is XOR-ed with the PC. We used a vector size of 10,000 in this experiment.
- **Global Stride** – The stride is calculated as the absolute value of the difference between adjacent memory accesses. We used a vector size of 10,000.
- **Global Stride with PC Hash** – The global stride is XOR-ed with the PC. We used a vector size of 10,000.
- **Working Set (frequency)** – The working set [5] is the set of all memory addresses accessed by the program during an interval. The frequency vector holds the frequency of accesses to each element of the working set during an interval. To minimize the size of this vector, a hash function is used to determine the index of the vector to increment.
- **Working Set (bits)** – Another version of the working set experiment uses a bit vector, and any nonzero frequency is assigned a value of one.
- **Wavelet Coefficients** – This technique is described in detail in Section 3.

4.2 Metric: Weighted Standard Deviation

In order to compare the different phase classification techniques, we need a metric of how well a given technique captures memory bus behavior. Computer architects have adopted the Coefficient of Variation (CoV) of CPI as a metric for evaluating different phase classification techniques

[15] [14]. The formula for computing CoV is shown in Equation 2:

$$CoV = \sum_{i=1}^{phases} \frac{\frac{\sigma_i}{average_i} intervals_i}{total \ \ intervals} \quad (2)$$

However, since computing the coefficient of variation involves dividing by the average, there will be a problem if a very long phase has almost no L2 misses. This occurs because the average could be less than one, and dividing by a number N, where $0 < N < 1$ may result in a large number. For example, in OpenOffice, there is a phase consisting of 218 intervals, but there is only one L2 miss during this phase. Therefore, the average number of misses per interval for this phase is 1 divided by 218, which is 0.0045872. Dividing the standard deviation for this phase, which is 0.067573, by the average yields an (unweighted) Coefficient of Variation of 14.7308 for this phase, which is an unrealistically large value for a phase with only one L2 miss. For this reason, the metric that we will use is the weighted standard deviation of the L2 misses, which is calculated using the formula in Equation 3:

$$\sigma_{weighted} = \sum_{i=1}^{phases} \frac{\sigma_i intervals_i}{total \ \ intervals} \quad (3)$$

5. EVALUATION

5.1 Weighted Standard Deviation

Figure 4 shows the weighted standard deviation in the L2 misses for a variety of commercial applications. The wavelet technique is the most effective on average, followed by local stride, local hash, global stride, global hash, basic block vector, and working set. For local stride, a smaller vector size (100) is more effective than a larger vector size (10K). Combining local stride with PC hash is not beneficial on average. Global stride is less effective than local stride, and combining global stride with PC hash is not beneficial. The working set frequency vector is more effective on average than the working set bit vector, which demonstrates that there is a cost for the reduced space requirements of the bit vector. Basic block vectors outperform both versions of working set on average.

Bounding the Weighted Standard Deviation – We have also included in this graph the result of applying the wavelet technique to the L2 miss stream. Although this takes the data we are trying to estimate as input, it serves as a soft bound on how well any scheme would perform. It is unlikely that any technique that analyzes the L1 access stream could possibly have a lower weighted standard deviation than a technique that is allowed to cluster the L2 misses. Our results show that on average, the accuracy of our technique is within 55% of the optimal.

5.2 Execution Time

Figure 5 shows the execution time per instruction. This was calculated by measuring the time to perform the phase classification and dividing this value by the total number of instructions in the trace file. We performed the timing experiments on a 2.2GHz Intel Celeron processor with 1Gb of RAM running Linux 2.6.9. On average, global hash has the worst performance, and our wavelet-based technique has the best performance (9ns per instruction on average). Since we implemented our technique in Java, it is likely that its performance would be faster if it were implemented in C. Note that SimPoint's implementation could also be more efficient because it processes the input multiple times.

5.3 Parameter Analysis

Since the scaled matrix size and the modulo size are the two most important parameters, we analyzed a matrix of combinations of the two parameters. Figure 6 shows a contour plot of the average weighted standard deviation of all the benchmarks, and Figure 7 shows the average execution time of all the benchmarks. Since Figure 7 is a perfect gradient from left to right, the execution time is dependent on the scaled matrix size, but the modulo size has no impact on the performance. However, Figure 6 shows that both parameters impact the accuracy of the phase classification, as measured by the weighted standard deviation. These contour plots show that the optimal combination of parameters is a scaled matrix size of 16x16 and a modulo size of 64Kb. A smaller modulo size will not provide any additional accuracy, and a larger scaled matrix size will have worse accuracy and greater cost. Our choices of a scaled matrix size of 16x16 and a modulo size of 32KB are very close to optimal. If we had chosen a modulo size of 64KB, our results would likely be slightly better.

In a separate experiment, we varied the L1 cache size and calculated the total number of misses in each interval for all the benchmarks. While keeping the size of the L2 cache constant at 1MB, we varied the L1 size at power of two intervals from 1K to 64K. We found that the impact of the L1 cache size on L2 misses is negligible.

5.4 Storage Requirements

Implementing our phase classification technique in hardware rather than in software will result in much better performance, as will optimizations to the phase classification algorithm. While we have characterized the performance impact of our scheme, we have not directly measured the memory requirements. However, each interval requires the storage of a 16x16 matrix instead of a Basic Block Vector or a Stride Array, so the storage should be comparable to other techniques. A hardware widget capable of performing online phase classification would not be difficult to design. It would consist of a buffer of memory of size 8K, which is the number of pixels (20 by 400) in one "slice." For each interval, this slice is scaled down to 256 bytes (16 by 16), and the 2D transform is performed in place on this memory region. The 2D transform is extremely fast in hardware.

6. VISUALIZING MEMORY PHASES

We have developed an interactive visualization tool to help us understand intuitively the complex phase behaviors of commercial applications. Figure 8 shows our visualization tool. First, the user scrolls to the desired location of the image representing the trace file. Clicking on an interval highlights that interval, and the source code corresponding to that interval appears in the source code window, with the relevant line highlighted. Of course, viewing the source code is not always possible for shared libraries or closed-source commercial software. A pull-down menu allows the user to select from the lines of source code that result in the most misses during the interval. A histogram of the PCs

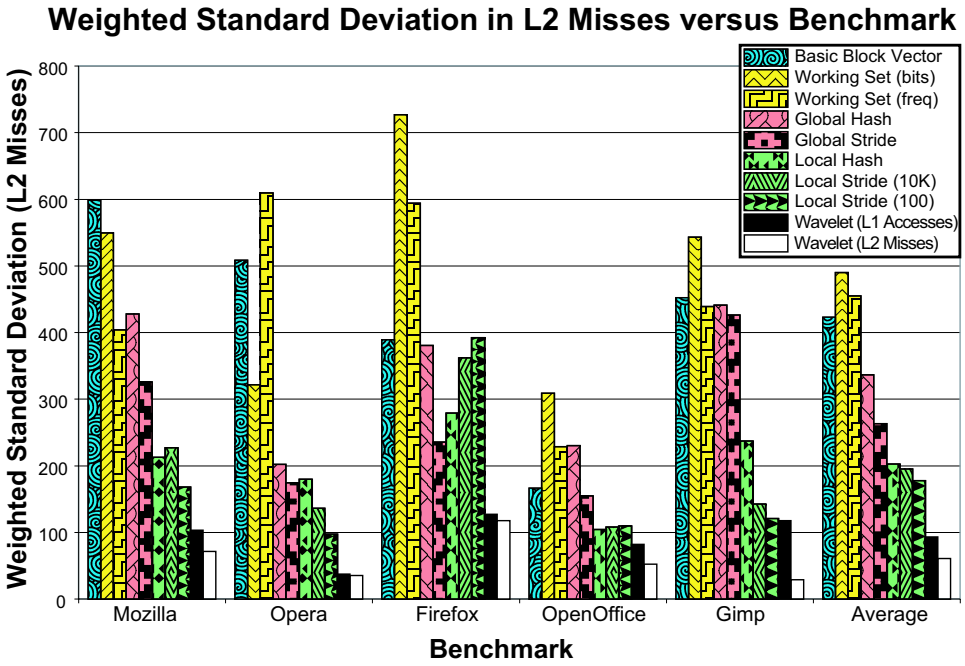

Figure 4: This chart shows the weighted standard deviation of the L2 misses for several commercial applications. On average, the wavelet technique applied to the L1 access stream captures the L2 misses the best. We have also included in this graph the result of applying the wavelet technique to the L2 miss stream. Although this takes the data we are trying to estimate as input, it serves as a soft bound on how well any scheme would perform.

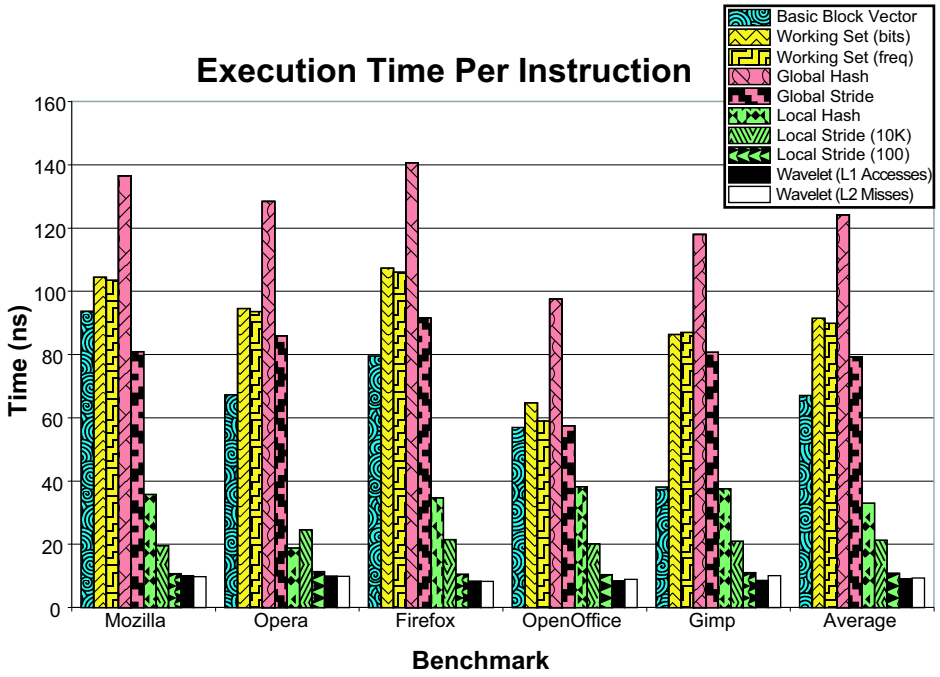

Figure 5: This chart shows the execution time of our technique per instruction in nanoseconds. Global hash has the worst performance (124.2ns on average), and our wavelet-based technique has the best performance (8.7ns on average).

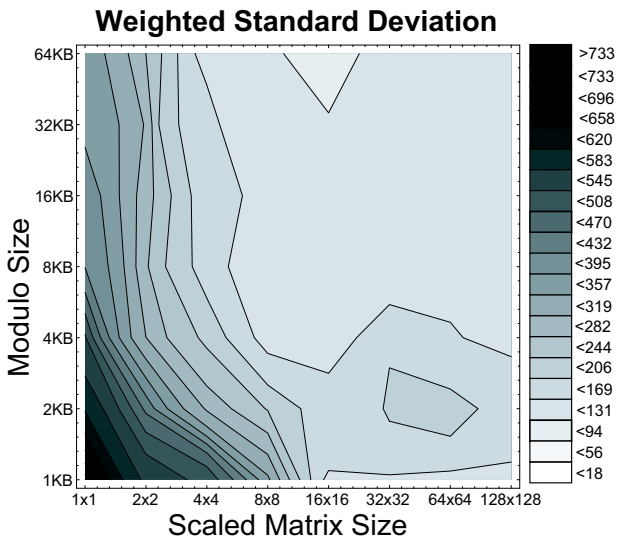

Figure 6: Contour plot of weighted standard deviation. This contour plot shows the average weighted standard deviation calculated using our technique over all the benchmarks. The x-axis is the scaled matrix size, and the y-axis is the modulo size. Darker shades of gray indicate a higher weighted standard deviation (less accuracy).

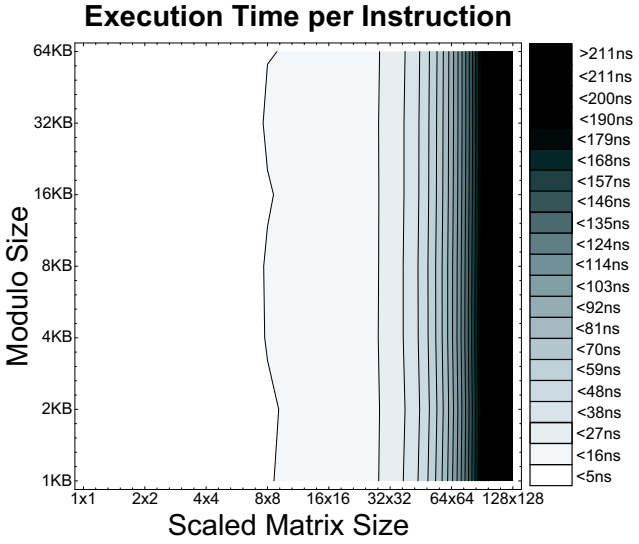

Figure 7: Contour plot of execution time. This contour plot shows the average execution time per instruction of our technique for all the benchmarks. The x-axis is the scaled matrix size, and the y-axis is the modulo size. Darker shades of gray indicate longer execution time (worse performance).

that have the greatest number of L2 misses is shown in the lower left in decreasing order. We can also show histograms for other statistics, such as the distribution of local stride. Another version of this utility shows the intervals that most closely match the selected interval using the wavelet technique.

This tool is useful for understanding memory phases because it allows a user to see the relationship between a specific phase and statistics about that phase. System designers can use this utility to gain insights into the memory phases in order to make hardware and software work together more efficiently. Software developers can identify problematic lines of code that result in many cache misses. Since application performance is heavily impacted by the shared libraries, programmers can identify problematic modules with our utility and take corrective action.

We are not the first to develop a utility for viewing program phases. Reiss et al. propose a visualization tool called JIVE that dynamically identifies and displays the phases of a Java program as it executes [19]. JIVE instruments the program, slowing it down by a factor of two. JIVE displays what classes are executing, the number of allocations of each class, and the state of each thread. Since our utility is geared towards understanding the memory phase behavior of commercial applications, it presents a different set of statistics and visual information than JIVE.

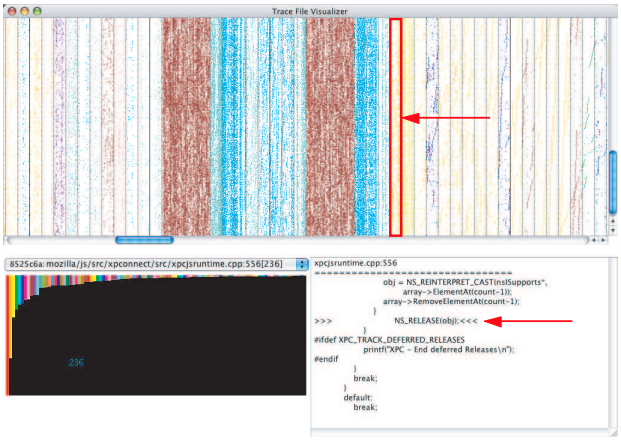

Figure 8: A memory phase visualization utility. This java applet allows the user to scroll around the trace file and select an interesting interval. Clicking on an interval highlights it and displays a histogram of the L2 misses in the selected interval in the lower left. A pull-down menu contains the lines of source code that result in the most L2 misses in the selected interval. Selecting one of these lines displays the source code in the lower right, and the selected line is highlighted.

7. CONCLUSIONS

Understanding the memory behavior of commercial applications is challenging because of their complex behavior. We have attacked this challenge by devising a new technique for performing phase classification that is based on wavelets. Our technique can accurately capture the memory bus behavior of real web browsers, productivity programs, and image editing software. We have compared our technique against several other well-known phase classification structures using the metric of weighted standard deviation in the number of L2 misses as the basis for comparison. We found that our technique captures the memory bus behavior significantly more accurately than prior techniques, and with less overhead. We have shown that our method is within 55% of optimal on average. We have also presented a visualization utility that makes it easier to understand the memory behavior of commercial applications, facilitating the design of more efficient systems.

Since traditional phase classification uses basic block vectors, it is necessary to have an executing program with a program counter on a von Neumann style architecture. This makes it unsuitable for unconventional computer architectures, FPGAs, and embedded systems. Since our wavelet-based technique does not have this limitation, we have opened up the possibility of studying the phase behavior of systems for which this has not been possible before. For example, we would like to use our wavelet-based phase classification technique to identify phases in the switching behavior of a circuit in order to perform accurate power analysis.

Acknowledgments

We would like to thank Chen Ding for his insightful comments on the paper. We would also like to thank the anonymous reviewers for their for their helpful feedback. We would also like to thank Robert S. Cohn at Intel for his assistance with using Pin. This research was funded in part by National Science Foundation Grant CNS-0524771 and NSF Career Grant CCF-0448654.

8. REFERENCES

[1] 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.

[2] C.H. Chen and G.G. Lee. Image segmentation using multiresolution wavelet analysis and expectation-maximization (EM) algorithm for digital mammography. *International Journal of Imaging Systems and Technology*, 8(5):491–504, 1997.

[3] Abhinav Das, Jiwei Lu, and Wei-Chung Hsu. Region monitoring for local phase detection in dynamic optimization systems. In *The Fourth Annual International Symposium on Code Generation and Optimization (CGO)*, New York, NY, USA, March 2006.

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

[5] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture (ISCA'02)*, Anchorage, AK, USA, May 2002.

[6] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *11th International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[7] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, New Orleans, LA, USA, September 27–October 1 2003.

[8] Lieven Eeckhout, John Sampson, and Brad Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IEEE International Symposium on Workload Characterization (IISWC'05)*, Austin, TX, USA, October 6–8 2005.

[9] Julinda Gllavata, Ralph Ewerth, and Bernd Freisleben. Text detection in images based on unsupervised classification of high-frequency wavelet coefficients. In *17th International Conference on Pattern Recognition*, Cambridge, UK, August 2004.

[10] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. SimPoint 3.0: Faster and more flexible program analysis. In *Workshop on Modeling, Benchmarking, and Simulation*, June 2005.

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

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

[13] Charles E. Jacobs, Adam Finkelstein, and David H. Salesin. Fast multiresolution image querying. In *SIGGRAPH 1995*, Los Angeles, CA, August 1995.

[14] Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly, and Brad Calder. The strong correlation between code signatures and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[15] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowner, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, Chicago IL, June 2005.

[17] Priya Nagpurkar, Chandra Krintz, and Timothy Sherwood. Phase-aware remote profiling. In *International Symposium on Code Generation and Optimization (CGO'05)*, San Jose, CA, USA, March 2005.

[18] Cristiano Pereira, Jeremy Lau, Brad Calder, and Rajesh Gupta. Dynamic phase analysis for cycle-close trace generation. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*, New York, NY, USA, September 2005.

[19] Steven P. Reiss. Dynamic detection and visualization of software phases. In *Workshop on Dynamic Analysis (WODA'05)*, St. Louis, MO, USA, May 2005.

[20] E. Salari and Z. Ling. Texture segmentation using hierarchical wavelet decomposition. *Pattern Recognition*, 28:1819–1824, Dec 1995.

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

[22] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large-scale program behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, San Jose, CA, October 2002.

[23] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, Nov–Dec 2003.

[24] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *30th International Symposium on Computer Architecture (ISCA'03)*, San Diego, CA, USA, June 9–11 2003.

[25] Ram Srinivasan, Jeanine Cook, and Shaun Cooper. Fast, accurate microarchitecture simulation using statistical phase detection. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, Austin, TX, USA, March 2005.

[26] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. Wavelets for computer graphics: A primer, part 1. *IEEE Computer Graphics and Applications*, 15(3):76–84, May 1995.

[27] Michail Vlachos, Jessica Lin, Eamonn Keogh, and Dimitrios Gunopulos. A wavelet-based anytime algorithm for k-means clustering of time series. In *Workshop on Clustering High Dimensional Data and its Applications*, San Francisco, CA, May 2003.