Phase Guided Profiling for Fast Cache Modeling

Andreas Sembrant, David Black-Schaffer and Erik Hagersten
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{andreas.sembrant, david.black-schaffer, eh}@it.uu.se

ABSTRACT
Statistical cache models are powerful tools for understanding application behavior as a function of cache allocation. However, previous techniques have modeled only the average application behavior, which hides the effect of program variations over time. Without detailed time-based information, transient behavior, such as exceeding bandwidth or cache capacity, may be missed. Yet these events, while short, often play a disproportionate role and are critical to understanding program behavior.

In this work we extend earlier techniques to incorporate program phase information when collecting runtime profiling data. This allows us to model an application’s cache miss ratio as a function of its cache allocation over time. To reduce overhead and improve accuracy we use online phase detection and phase-guided profiling. The phase-guided profiling reduces overhead by more intelligently selecting portions of the application to sample, while accuracy is improved by combining samples from different instances of the same phase.

The result is a new technique that accurately models the time-varying behavior of an application’s miss ratio as a function of its cache allocation on modern hardware. By leveraging phase-guided profiling, this work both improves on the accuracy of previous techniques and reduces the overhead.

1. INTRODUCTION
The goal of this work is to develop and explore methods for understanding a program’s cache behavior over time and as a function of its cache allocation. Such information is important for understanding performance [24], resource sharing [15, 9], and scheduling [16]. In particular, the ability to analyze a program’s behavior as a function of its cache allocation is essential for modern systems with shared caches where the cache allocation can change dynamically. This requirement makes it difficult to use data from hardware performance counters, as they only provide information for one particular cache allocation.

The low overhead statistical cache model, StatCache, developed by Berg and Hagersten [5, 6], can estimate the miss ratio for caches of arbitrary size. It answers the question: what is an application’s miss ratio if it receives $x$ amount of cache? StatCache has been used to estimate shared miss ratios for multi-threaded applications [7] and co-scheduled applications [15], and forms the basis of a commercial code optimization product [1].

However, these existing models only report the application’s average miss ratio, which can be misleading. Consider, for example, an application whose miss ratio is high enough to exceed the system bandwidth for a short portion of its execution. In such an application, the average miss ratio would fail to indicate that the application is at all bandwidth bound.

The simplest way to extend these methods to handle program phases is to divide the program execution into many windows and profile each window. This approach has the downside of a significant increase in overhead from having to sample all portions of the application’s execution. To combat this, periodic profiling can be used, wherein only a randomly selected subset of all windows are profiled. Unfortunately, the number of profiled windows must still be high to capture short application phases.

A more intelligent solution is to use phase-guided profiling [31, 27]. In this approach, a phase-detection algorithm is used to select only a small part of each phase to profile, and this data is then used for subsequent instances of the same phase. This minimizes the number of profiled windows by avoiding redundantly sampling windows from the same phase.

For such an approach to be generally applicable, it must have the following properties: 1) It should not require custom hardware support; 2) It should have minimal runtime overhead without loss of accuracy and fidelity; 3) It should be transparent and non-intrusive (e.g., require no recompilation of the analyzed program and work with dynamically generated code); 4) It should be architecturally independent (e.g., not affected by system load and able to model different cache sizes), and, finally; 5) It should be fully automatic (e.g., users should not have to adjust settings for each application).
To accomplish this, we leverage the ScarPhase (Sample-based Classification and Analysis for Runtime Phases) library developed during our previous work with phase classification [31]. This provides us with low-overhead (2%) runtime detection of program phases. We then combine this phase information with the StatCache [5] statistical cache model to accurately model application cache behavior as a function of time and allocated cache.

The main contributions of this paper are:

- A method for accurately modeling cache behavior (miss ratio as a function of cache allocation) over time.
- An efficient method for capturing program cache behavior on modern hardware by integrating the StatCache cache model and the ScarPhase phase detection library.
- A comparison with previous statistical cache modeling methods demonstrating improved accuracy (39%) and efficiency (6×).
- An analysis of the impact of different types of intra-phase variations on phase-guided memory profiling.

2. CACHE BEHAVIOR OVER TIME

An application’s cache behavior, in this case its miss ratio, varies due to both program phases and changes in cache allocation. To illustrate this, Figures 1 and 2 plot the miss ratio (intensity) over time (x-axis) as a function of cache size (y-axis), for the complete execution of the gcc/166 and bzip2/chicken benchmarks, respectively. The darker the points, the higher the miss ratio. The y-axis (cache size) indicates how the application’s miss ratio is affected by its cache allocation. The x-axis (time) shows the intrinsic phase behavior of the application. The vertical bar marked “Average” on the right shows the application’s overall average miss ratio as a function of cache size. And, finally, the bars above the graph indicate the phases detected by ScarPhase, with smaller phases grouped together in white for clarity.

The top figures (1a and 2a) show reference results from a simulation using the Pin [8] instrumentation toolkit and the
Figure 2: Miss ratio (intensity) as a function of time (x-axis) and cache allocation (y-axis) for the whole execution of bzip2/chicken on a Nehalem machine. The average miss ratio for the whole execution is shown on the right. The detected execution phases are shown above, with shorter phases shown in white for clarity. The top figure (2a) shows results from a reference simulation and the bottom (2b) from online profiling. The boxed area from 0 to 80B instructions highlights the importance of using hardware-independent information for determining phases: when run with 1MB or more of cache allocation, bzip2/chicken appears to have a single phase up to 80B instructions based on cache miss ratio. However, at lower allocations, or using hardware-independent metrics, distinct phases can be clearly seen. The benefits of time-based information are clearly visible in Figure 1. While the graph shows that there are two periods in gcc/166’s execution with a very high miss ratio at 2MB of allocated cache (phase E at 32 and 70 billion instructions), the overall average miss ratio appears far less severe. With the more fine-grained phase information, the correct portion of the application can be targeted for optimization.

From this information we can also see the limitations of defining phases based on hardware-specific information, such as performance counters. For example, if miss ratio was used to define phases, a machine with 2MB or more of cache would group the first 80 billion instructions of bzip2/chicken into one phase as the miss ratio is constant. (See the red box in Figure 2b.) However, if the application’s cache allocation were decreased, due to resource sharing, for example, its behavior would change, thereby revealing different phases. This demonstrates the importance of finding phases that are architecturally independent properties of the application.

These examples show how important it is to consider both the intrinsic program phase behavior as well as the impact of program cache allocation when examining application behavior. With this more detailed information we can analyze how various runtime optimizations [10, 19] and scheduling decisions [16, 36] will affect the cache performance. For example, migrating gcc/166 to a core with a smaller cache for phase D could potentially save energy without sacrificing performance. However, trying to migrate bzip2/chicken between a large-cache core for phase B and a small-cache core for phases C and D would entail many more relocations and might not be beneficial.
3. STATISTICAL CACHE MODELING

StatCache [5, 6] is a low overhead statistical cache model. It can estimate the miss ratio of random replacement caches of arbitrary sizes. In this section we give an overview of the model and discuss how program phase behavior affects its accuracy.

3.1 Reuse Distance

The input to the StatCache model is cache line reuse distance data. A reuse distance is defined to be the number of memory accesses between two accesses to the same cache line. For example, if the processor first accesses cache line A, then B and C, and finally A again, the reuse distance of the second access to A would be two. It is important to note that reuse distance counts all memory accesses. This is different from stack distance [26] where only the number of unique memory accesses are counted. As a result, measuring reuse distance requires far less bookkeeping.

3.2 The StatCache Cache Model

The reuse distance distribution can be transformed into a miss ratio distribution using the StatCache [5, 6] cache model. StatCache first sorts the reuse distances of the memory accesses into buckets, hi, where hi is the number of memory accesses with a reuse distance of i. Then, the following equation is solved for the miss ratio R:

\[ R \cdot N = h_1 f(R) + h_2 f(2R) + h_3 f(3R) + \cdots \]  

where \( N \) is the number of reuse distance samples, i.e., \( N = h_1 + h_2 + h_3 + \cdots \), and \( f(n) \) is a function that gives the probability that a cache line has been evicted from the cache if we know that it was in the cache \( n \) cache misses ago. With random replacement the function \( f(n) \) is:

\[ f(n) = 1 - (1 - 1/L)^n \]  

where \( L \) is the number of cache lines in the cache. The cache size is \( L \) times the cache line size. We can then model caches of arbitrary sizes by changing the \( L \). The StatCache model can be readily extended to model LRU caches (StatStack [14]) without changing the input data.

3.3 Program Phases

The StatCache model works very well for single phase applications and its accuracy improves with the number of samples. Indeed, Equation 1 assumes a constant miss ratio across the reuse distance samples. If the behavior is constant for the application, a phase oblivious overall miss ratio can be determined by simply applying the model to all samples at once.

However, as we observed in the previous section, the miss ratio can vary significantly over time. As Berg and Hagersten [5, 6] had no means to detect phases, they instead gathered samples in short bursts, where each burst was short enough for the miss ratio to remain approximately constant. They then applied the model to each burst separately and averaged the model output to determine the overall miss ratio. This method improves accuracy by ensuring that the miss ratio is approximately constant across the samples given to the model.

The work presented here is phase aware, and groups samples within the same phase together. The model is then applied to all samples from each phase separately. The application’s overall miss ratio is then the weighted average of the phases. This approach improves accuracy as the miss ratio is far more constant within phases than across them, and by combining samples across phases, the model has more samples to work with at each time.

Figure 3 compares these three methods for gcc/166. Each method uses the same reuse distance samples. The phase oblivious approach applies the model to the most samples (all of them together), but incorrectly assumes that the underlying miss ratio is constant across all samples. As a result it has the worst accuracy. Both burst and phase aware apply the model to groups of samples taken from periods with a reasonably constant miss ratio, but the phase aware approach is able to group more samples together for the model, and thereby produce more accurate results.

3.4 Sampler Implementation and Overhead

We have implemented a reuse distance sampler on an Intel Xeon E5620 (Nehalem) machine to provide data to the StatCache model. To minimize the overhead, we use hardware performance counters [18] and page protection to sample and monitor reuses.

For the StatCache model to work, it is important that all memory accesses have the same probability of being sampled. We therefore use the executed loads and stores counters to interrupt the program execution at random (exponentially distributed) intervals. However, when these interrupts occur, a context-dependent number of extra instructions are executed. To avoid biasing the results with this “skid” [2], the counters are set up to interrupt before the target access. After the interrupt, the execution is single-stepped to the desired access. At this point the loads and stores counters are
recorded for the access’s pending reuse, and page protection is turned on for the access’s page.

Execution then continues until a page fault occurs. If the memory access that caused the page fault belongs to a pending reuse, the loads and stores counters are read and the resulting reuse distance recorded. Otherwise, it was a false positive, i.e., the page protection was turned on because of another cache line that resides on the same page. In the latter case, the page protection is temporarily turned off and the execution is single-stepped past the access, before turning the page protection on again.

In this reuse sampler there are two parts to the overhead. First, the application must be single-stepped to the target access. Depending on the length of the skid, this can entail many context switches. Second, it can be equally time consuming to handle page faults, especially when the number of false positives are high. As both of these overheads are directly related to the number of samples required, it is clearly important to intelligently choose when to sample.

4. PHASE GUIDED PROFILING

Phase-guided profiling is a method to reduce the overhead of profiling without sacrificing accuracy, by taking advantage of the (nominally) uniform behavior of each program phase. The idea is to only profile a small part of each phase, and then use that profile for all instances of the same phase. There are two benefits to this approach. First, it removes redundant profiling as only a minimum part of each phase is profiled. Second, it automatically adapts to the application’s characteristics, thereby eliminating the need to adjust profiling parameters for each application and data set.

4.1 Detecting Program Phases

We use the ScarPhase [31] library to detect and classify phases. ScarPhase is an execution history based, low-overhead (less than 2%), online phase detection library. Because it is based on the application’s execution history, it detects hardware independent phases [34, 29]. Such phases can be readily missed by performance counter based phase detection, as shown in Figure 2b.

To detect phases, ScarPhase monitors executed code, based on the observation that changes in executed code reflect changes in many different metrics [33, 34, 11, 35, 20]. To accomplish this, execution is divided into non-overlapping windows. During each window, hardware performance counters are used to sample conditional branches using Intel PEBS [25, 18]. The address of each branch is hashed into a vector of counters called a conditional branch vector (CBRV), similar to a basic block vector (BBV) [33] but with only conditional branches. Each entry in the vector shows how many times its corresponding conditional branches were sampled during the window.

The vectors are then used to determine phases by clustering them together using an online clustering algorithm, such as leader-follower [12]. Windows with similar vectors are then grouped into the same cluster and considered to belong to the same phase.

4.2 Phase Guided Profiling

The simplest approach to phase-guided profiling is to only profile one window from each phase, and to use that profile for all other instances of the same phase. This way, only a small part of each phase is profiled, thereby lowering overhead, and, if the behavior within the phase is uniform, the accuracy will not suffer. As a result, the overhead will be proportional to the number of phases in the application, and the profiling will automatically adapt to the application’s requirements.

To illustrate how phase-guided profiling works, we have zoomed in on a short part of gcc/166’s execution in Figure 4. The black triangles show where phase-guided profiling decides to profile, and the white triangles show the same for periodic profiling. Phase-guided profiling places the samples at the beginning of each phase. Periodic profiling, on the other hand, may profile the same phase more than once.
While the phases detected by ScarPhase have reasonably constant behavior within each phase, some applications exhibit intra-phase variation [33, 32]. To illustrate this, Figure 5 plots the Overall Absolute Miss Ratio Error for four applications as a function of the number of windows profiled in each phase. For example, at 10 on the x-axis, the y-axis shows the error if only the first ten windows of each phase are profiled. If there were no intra-phase variation, the error would be constant. However, this is clearly not the case. Indeed, we observed three different types of intra-phase variation: transition, periodic, and instance. These are illustrated in Figure 6 and discussed below.

### No Variations
Figure 6a shows the miss ratio over time for gcc/166’s phase D in Figure 1. This is a typical phase for gcc/166 with very little intra-phase variation. Only a small part of each phase needs to be profiled, and we see that the error drops rapidly after seven windows in Figure 5.

### Transition Variations
In Figure 6b, mcf can be seen to have a long transition phase where the behavior slowly changes from one phase to another. The Figure shows how the miss ratio slowly increases over time between the two

1. **Most phases span several windows and we only need to profile when we are sure we are in the correct phase. A mis-prediction is thus very uncommon. Furthermore, a mis-prediction is unlikely to affect the accuracy since the phase will be profiled later.

2. The Overall Absolute Miss Ratio Error for Figures 5 and 7 is defined as the absolute error from the reference simulation across all cache sizes, as shown in Figure 3.

### Periodic Variations
Figure 6c shows the miss ratio over time for bzip2/chicken’s phase A in Figure 2. The behavior is highly periodic and changes rapidly. It is actually caused by two sub-phases whose vectors are not different enough to create two separate clusters. This tradeoff between uniformity and the number of phases is a limitation of clustering algorithms. If the settings are too sensitive we will identify more phases that necessary, and if they are too insensitive we combine the wrong sub-phases.

### Instance Variations
In Figure 6d we can see that astar/lakes suffers from a different problem. The two separate instances of the same phase have different average miss ratios. The vectors for both instances are nearly identical but the cache behavior is different. This can happen when two instances of the same phase operate on different input data. This is the reason why so many windows must be profiled for the error to start to decrease in Figure 5: all windows in the first instance of the phase must be profiled before the second instance can be included in the results. There has

Furthermore, the period between the profiles must be short enough to catch all phases. If an application has a mix of short and long phases, the profiling period must be set for the shortest phase to accurately capture the application’s behavior. This results in a high overhead and requires the user to adjust the profiler to the application.

ScarPhase returns the phase ID of the just-executed window and a prediction for the next window [35, 23]. Since the phase ID is only known after the window has been executed, we need to rely on the prediction. If the predicted phase has not been profiled, we start to sample reuse distances, otherwise, we turn off the profiler and do not sample.

## 4.3 Intra-Phase Variations
While the phases detected by ScarPhase have reasonably constant behavior within each phase, some applications exhibit intra-phase variation [33, 32]. To illustrate this, Figure 5 plots the Overall Absolute Miss Ratio Error for four applications as a function of the number of windows profiled in each phase. For example, at 10 on the x-axis, the y-axis shows the error if only the first ten windows of each phase are profiled. If there were no intra-phase variation, the error would be constant. However, this is clearly not the case. Indeed, we observed three different types of intra-phase variation: transition, periodic, and instance. These are illustrated in Figure 6 and discussed below.

### No Variations
Figure 6a shows the miss ratio over time for gcc/166’s phase D in Figure 1. This is a typical phase for gcc/166 with very little intra-phase variation. Only a small part of each phase needs to be profiled, and we see that the error drops rapidly after seven windows in Figure 5.

### Transition Variations
In Figure 6b, mcf can be seen to have a long transition phase where the behavior slowly changes from one phase to another. The Figure shows how the miss ratio slowly increases over time between the two

1. Most phases span several windows and we only need to profile when we are sure we are in the correct phase. A mis-prediction is thus very uncommon. Furthermore, a mis-prediction is unlikely to affect the accuracy since the phase will be profiled later.

2. The Overall Absolute Miss Ratio Error for Figures 5 and 7 is defined as the absolute error from the reference simulation across all cache sizes, as shown in Figure 3.

### Periodic Variations
Figure 6c shows the miss ratio over time for bzip2/chicken’s phase A in Figure 2. The behavior is highly periodic and changes rapidly. It is actually caused by two sub-phases whose vectors are not different enough to create two separate clusters. This tradeoff between uniformity and the number of phases is a limitation of clustering algorithms. If the settings are too sensitive we will identify more phases that necessary, and if they are too insensitive we combine the wrong sub-phases.

### Instance Variations
In Figure 6d we can see that astar/lakes suffers from a different problem. The two separate instances of the same phase have different average miss ratios. The vectors for both instances are nearly identical but the cache behavior is different. This can happen when two instances of the same phase operate on different input data. This is the reason why so many windows must be profiled for the error to start to decrease in Figure 5: all windows in the first instance of the phase must be profiled before the second instance can be included in the results. There has

---

**Figure 6:** Intra-phase variation in miss ratio for Spec2006. Phases shown with shading. While most phases show little intra-phase variation, as in gcc/166 (6a), the three intra-phase variations identified above explain the improved error characteristics (Figure 7) of randomly selecting windows to profile. Both Transitions (6b) and Periodic (6c) are artifacts of the tradeoff between phase size and the number of phases. The Instance Variations (6d), however, represent data-dependent changes in behavior for the same code path.

**Figure 7:** Choosing windows randomly. Overall Absolute Miss Ratio Error as a function of number of profiled windows in each phase. The windows are randomly selected within each phase.

---

<table>
<thead>
<tr>
<th>Phase</th>
<th>Application</th>
<th>Miss Ratio Error as a function of number of profiled windows in a Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc/166</td>
<td>mcf</td>
<td>0.2</td>
</tr>
<tr>
<td>bzip2/chicken</td>
<td>astar/lakes</td>
<td>0.8</td>
</tr>
<tr>
<td>gcc/166</td>
<td>mcf</td>
<td>0.6</td>
</tr>
<tr>
<td>bzip2/chicken</td>
<td>astar/lakes</td>
<td>1.2</td>
</tr>
</tbody>
</table>

---

180
been a significant amount of research [32, 4, 22] discussing how changes in the code path are correlated to changes in other metrics.

The intra-phase variations identified above have a significant effect on the accuracy of this method. To make an accurate estimate of the behavior of a phase, it is therefore important to consider all instances of the phase. Figure 7 shows the same metric as in Figure 5, but instead of selecting only the first windows, the windows are randomly selected from the whole phase. For example, when \( x \) is ten, the average is calculated from ten randomly selected windows from the phase. If a phase is less than ten windows, the whole phase is profiled. The error starts to decrease rapidly for all applications compared to taking the windows in order. It is therefore important to spread samples throughout a phase.

### 4.4 Phase Sampling Implementation

To handle intra-phase variations, we try to profile several windows spread throughout the phase. However, we do not know the length of the phase in advance. We therefore start with a short period to catch the shorter phases, and increase the period with the number of profiled windows until an upper limit is reached. Specifically, we start sampling windows with an exponential distribution, and increase the period by a factor of two after each window until we reach an upper limit. In this way, we can reliably profile both short and long phases while maintaining a good distribution of samples.

It is worth noting that the runtime overhead is proportional to the number of sampled reuse distances. This is different from traditional profiling and simulation where the overhead comes from number of executed instructions. This has two implications for this work. First, spreading out profiling windows across an application’s execution time does not increase overhead. This is because the overhead is per sample, regardless of when they are taken, and instructions executed between samples run at native speed. Second, because we capture many profile windows, we are less sensitive to selecting optimal windows [34, 27].

For reference, the data collection and modeling for Figures 1b and 2b took minutes to execute, while the simulation to produce the reference results in Figures 1a and 2a took days.

### 5. EVALUATION

In this section we evaluate and compare the accuracy and performance of StatCache with periodic profiling and phase-guided profiling. We implemented periodic profiling by periodically selecting windows to profile. The model was then independently applied to each window. The behavior over time was then approximated by observing how the behavior changes between the profiled windows. Phase-guided profiling used the ScarPhase library as discussed in Section 4. The memory reuse data was captured online using the memory reuse sampler described in Section 3.4. All benchmarks were run from start to completion with their reference input on an Intel Xeon E5620 (Nehalem) system. Because the random nature of the sampling, we average the data from 5 runs.

To create the reference data, we simulated the cache behavior for the whole execution using the Pin [8] instrumentation toolkit and the Dinero [13] cache simulator. Pin was used to divide the execution into windows and extract a memory reference trace that was sent to Dinero. After each window, the miss ratio for the window was extracted from the Dinero simulation.

We chose the eight applications from SPEC 2006 [17] with the most interesting phase behavior (astar/lakes, bzip2/chicken, bwaves, dealii, gcc/166, mcf, perl/splitmail and xalan) and simulated and modeled each for twelve cache sizes from 1KB to 2MB. The cache sizes were chosen to cover the most interesting changes in cache behavior for the benchmarks.

\(^3\)To avoid periodic behavior, the sampled windows are selected at random from an exponential distribution with a fixed period.
The CDF can also give valuable insight into application behavior. For example, if gcc/166 hits the bandwidth limit when it has a miss ratio above 20%, the average would indicate that gcc never hits the limit, while the CDF shows that 7% of the execution would be bandwidth bound.

### 5.2 Sampling Parameters

We chose parameters for the window size and sample rate for periodic and phase-guided profiling to produce similar accuracy on gcc/166. (The exact settings and details on the selection process are found in the appendix.) This benchmark was chosen as the baseline because it has the highest number of phases (most difficult to accurately model) and a short execution (least chance to make up for missed phases).

The results of choosing settings to produce similar accuracy for gcc/166 can be seen in Figure 8a. The shaded areas are the average miss ratio CDF +/- one standard deviation. The data shows the CDF for both the periodic and phase-guided methods, as well as the reference simulation. The smaller the shaded area and the closer it is to the reference the better the accuracy. In this graph both the periodic and the phase-guided methods show similar accuracy (1.34% and 1.29%, respectively), as expected. However, to obtain this degree of accuracy, the periodic method imposes an overhead of 89.3% compared to 32.5% for the phase-guided approach.

### 5.3 Accuracy: Error

Figure 8a is on average off by one percent for phase-guided profiling.

Despite using fewer samples, phase-guided profiling has a better accuracy. There are two reasons for this. First, phase-guided profiling can combine reuse distances from several windows belonging to the same phase which reduces the modeling error. Second, phase-guided profiling is better at distributing the samples over the execution: it forces shorter phases to be profiled which would otherwise have been missed. The profile thus represents a larger portion of the execution.

### 5.4 Performance: Overhead

Figure 10 presents the overhead for the benchmarks. Phase-guided profiling demonstrates significantly better performance than periodic profiling. The overhead is on average six times lower (21% compared to 133%) than periodic profiling.

The overhead for periodic profiling would be constant if the cost of a reuse distance was the same for all applications, and if the number of samples in each window was fixed. This is, however, not the case. First, the cost of a sample depends on the memory behavior, i.e., the number of false positives (page faults). Second, the sample rate is per memory access. Applications with more memory instructions will therefore collect more samples.
This significant improvement is achieved by intelligently deciding when and where to profile. Only the most necessary parts of the execution are chosen, resulting in fewer samples required for similar accuracy. The longer the execution and the fewer phases an application has, the better the phase-guided profiling performance will be compared to periodic.

5.5 Summary
The accuracy and overhead results show that the StatCache cache model can be efficiently combined with phase-guided profiling to estimate the miss ratio over time for different cache sizes. The result is both more accurate and has a better performance than periodic profiling.

6. RELATED WORK
In this section we discuss work related to cache behavior over time and phase detection.

Agaram et al. [3] looked at memory behavior over time by analyzing performance by data structure. They observed that a stable overall miss ratio can hide important changes. For example, the overall miss ratio can appear stable while the miss ratios for individual data structures changes. However, they did not map this behavior to application phases. The ScarPhase phase detection would detect such behavior as separate phases if it was caused by changes in the code path.

Both Agaram et al. and others [33, 21] have observed that phase behavior depends on the size of the sampled windows. Dividing the execution into windows effectively averages the execution: the smaller the windows are, the larger the intra-phase variations will be, and vise versa. This is not a significant issue for this work since we profile several windows from the same phase. The profile for the phase will therefore be much closer to the true behavior than if only a single window was selected.

One important feature of this work not found in these others is that we model arbitrary cache sizes. Focusing on just one cache size can ignore important phase distinctions at other cache sizes, as seen in Figure 2b.

ThreadSpotter [1] is a commercial tool that can detect memory bottlenecks. It leverages the work with reuse distances and statical cache models from [5, 6] to find memory bottlenecks and provide developers with information on how to improve performance. ThreadSpotter uses exponential back-off to reduce overhead by increasing the time between samples for long-running applications. This allows profiling of both short- and long-running applications. Unfortunately, the method is best suited for average miss ratios. Consider gcc in Figure 1. Exponential back-off might detect phase B, but it would start to merge it with A, C or D in later instances.

Nagpurkar et al. [28] used phase-guided profiling for distributed profiling in embedded devices, where each phase was profiled separately on a different device. The results showed that phase-guided profiling can reduce communication, computation and energy costs. Their implementation used custom hardware [35] to collect basic block vectors in order to detect phases, and assumed perfect prediction.

In this work we use code-based phases to guide reuse distance sampling. Shen et al. [32] turned this approach around, and instead used stack distances [26] to define phases. They argue that their phases are better at predicting memory behavior. While they do not report any overhead numbers, it is clear that the cost of using hardware performance counters to sample code execution paths is much cheaper than sampling reuse distance or stack distances.

7. CONCLUSIONS
In this paper we have shown the importance of considering both program phases and the effects of cache allocation in understanding application behavior. Phase-aware analysis is required to identify important transient behavior in applications (see Figure 1b), which is obscured by average metrics. The effects of different cache allocations can also have a significant impact on program behavior (see Figure 2b), and ignoring them can lead to incorrect phase classification.

We have also shown the benefits of integrating online phase detection and statistical cache modeling to produce a phase-guided statistical cache analysis tool. By doing so we have improved both performance and accuracy over previous techniques, while also providing more valuable data in the form of time-dependent cache behavior. To further improve the accuracy we investigated different sources of intra-phase variation and described a sampling technique to overcome them.

The resulting method has better accuracy than previous statistical cache modeling methods, requires no custom hardware or application modifications, and has an overhead six times lower than previous methods.

Appendix: Selecting Sampling Parameters
One of the goals of this work has been to find methods that are automatic and do not require custom settings for every benchmark or data set. This is important since it allows the user to seamlessly work with different input data and applications without having to adjust the tools for each change. Phase-guided profiling makes most parts of the profiling automatic by adapting to the number and length of phases in the application. Configuring periodic profiling, however, is more difficult as the sampler does not adapt to the application’s behavior. In general, a good sampler setting should be able to collect information from all phases of an application regardless of the input. In this appendix we show how we selected the settings used in the evaluation to achieve this.

We chose to base our setting on gcc/166 as it is short and has many phases. This makes it a particularly tricky application to profile accurately. Therefore, settings with good accuracy for gcc should produce good results for other applications, but may do so at the cost of higher overhead than necessary. This tradeoff between overhead and accuracy is a problem with fixed sampling strategies in general. To evaluate the impact of changing the sampling settings on the periodic and phase-guided sampling, we ran the samplers five times and varied the number of samples in each window and the period between the windows. The more samples and the shorter period, the higher the overhead.
For phase-guided profiling, the number of samples from several windows belonging to the same phase before processing them. Since the phase-guided method is able to combine samples in each window is reduced to one per 1M memory accesses. For periodic profiling, every eighth window is profiled with one sample for every 400K memory access in the instructions. For periodic and phase-guided profiling. The error bars indicate the standard deviation in error across the different runs. As expected, the accuracy tends to improve with the overhead. However, the phase-guided profiling is both more accurate and has a lower overhead across the full range. We can also see that the variance is lower for phase-guided profiling, since it is less sensitive to different settings. However, the phase-guided profiling is both more accurate and has a lower overhead across the full range. We can also see that the variance is lower for phase-guided profiling, since it is less sensitive to different settings.

For the evaluation we chose the two settings indicated in Figure 11 with roughly the same accuracy on gcc/166. In both cases the application is divided up into windows of 100M instructions. For periodic profiling, every eighth window is profiled with one sample for every 400K memory access in the window. For phase-guided profiling, the number of samples in each window is reduced to one per 1M memory accesses. However, the number of samples in each phase is still higher since the phase-guided method is able to combine samples from several windows belonging to the same phase before processing them.

**References**


