In Proceedings of the International Symposium on Code Generation and Optimization (CGO 2006).

Selecting Software Phase Markers with Code Structure Analysis

Jeremy Lau Erez Perelman Brad Calder

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

Abstract

Most programs are repetitive, where similar behavior can be seen at different execution times. Algorithms have been proposed that automatically group similar portions of a program's execution into phases, where samples of execution in the same phase have homogeneous behavior and similar resource requirements. In this paper, we present an automated profiling approach to identify code locations whose executions correlate with phase changes. These "software phase markers" can be used to easily detect phase changes across different inputs to a program without hardware support.

Our approach builds a combined hierarchical procedure call and loop graph to represent a program's execution, where each edge also tracks the max, average, and standard deviation in hierarchical execution variability on paths from that edge. We search this annotated call-loop graph for instructions in the binary that accurately identify the start of unique stable behaviors across different inputs.

We show that our phase markers can be used to accurately partition execution into units of repeating homogeneous behavior by counting execution cycles and data cache hits. We also compare the use of our software markers to prior work on guiding data cache reconfiguration using data-reuse markers. Finally, we show that the phase markers can be used to partition the program's execution at code transitions to pick accurately simulation points for SimPoint. When simulation points are defined in terms of phase markers, they can potentially be re-used across inputs, compiler optimizations, and different instruction set architectures for the same source code.

1 Introduction

The behavior of a program is not random - as programs execute, they exhibit cyclic behavior patterns. Recent research [1, 6, 7, 24, 25, 26, 22, 8], shows that it is possible to accurately identify and predict these phases in program execution. There are many applications of phase behavior - for example phases can be exploited to accelerate architecture simulations [24, 25], to save energy by dynamically reconfiguring caches or processor issue width [1, 26, 7, 6], or to guide compiler optimizations [19, 2].

In prior work [25, 26] we classified a program into phases by first dividing a program's execution into non-overlapping fixed-length intervals of 1, 10, or 100 million instructions. 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 with similar behavior (e.g., IPC, cache miss rates, branch miss rates, etc), regardless of temporal adjacency. This means that intervals that belong to a phase may appear throughout the program's execution. Our prior work uses an off-line clustering algorithm to break a program's execution into phases to perform fast and accurate architecture simulation by simulating a single representative portion of each phase of execution [25, 16, 26, 22]. We also developed an on-line hardware algorithm to dynamically identify phase behavior to guide adaptive architecture reconfiguration [26, 17]. This prior work also relied on fixed length intervals of execution.

The goal of this paper is to find phase transitions that match the procedure call and loop structure of programs, instead of using fixed length intervals. We select a subset of each program's procedures and loop boundaries to serve as interval endpoints. Because each interval is aligned with code boundaries, the program's execution is divided into *Variable Length Intervals* (VLIs) of execution. We use the code at each selected procedure or loop boundary as a *software phase marker* that, when executed, signals a phase change without any hardware support. These software phase markers are selected by analyzing each program's procedure call and loop iteration patterns.

In this paper we present the formation of a Hierarchical Call-Loop graph, which we use to find the software phase markers. The graph contains local and hierarchical execution time for each procedure call and loop, as well as the variance on all paths from each call or loop. We present a simple and fast algorithm to select code structures that serve as software phase markers from the Call-Loop graph. We show that the software phase markers accurately identify phase changes at the binary level, with no hardware support, across different inputs to the program.

The remainder of the paper is laid out as follows. First, prior work in phase analysis is examined in Section 2. The simulation framework used in this work is described in Section 3. Section 4 presents the method for generating the hierarchical call-loop graph, and Section 5 presents the algorithm for selecting phase markers. Section 6 exams applying the code phase markers to data cache reconfiguration and SimPoint. Our findings are summarized in Section 7.

2 Phase Behavior and Related Work

In this section we give a brief overview of recent related work on phase analysis, and provide more detailed descriptions of the two areas of research closest related to ours – procedural phase analysis, distance reuse software phase markers, and

using Sequitur to create variable length intervals.

2.1 Related Phase Analysis Work

Program phase behavior can be detected by examining a program's working set [4], 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 hundred thousand instructions. They use the miss rate and the total number of branches executed for each interval to dynamically evaluate the program's stability. They used their approach to guide dynamic cache reconfiguration to save energy without sacrificing performance.

Dhodapkar and Smith [6, 7, 5] found a relationship between phases and instruction working sets, and that phase changes occur when the working set changes. They proposed dynamic reconfiguration of multi-configuration units in response to phase changes indicated by working set changes. They use working set analysis for reconfiguration of instruction cache, data cache and branch predictor to save energy [6, 7].

Hind et. al. [11] provide a framework for defining and reasoning about program phase classifications, focusing on how to best define granularity and similarity to perform phase analysis.

Sherwood et. al. [24, 25] 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 [25], 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 then extended this approach to perform hardware phase classification and prediction [26, 17]. In [17] we focus on hardware techniques for accurately classifying and predicting phase changes (transitions).

Isci and Martonosi [13, 14] have shown the ability to dynamically identify the power phase behavior using power vectors. Deusterwald et. al. [8] recently used hardware counters and other phase prediction architectures to find phase behavior.

2.2 Basic Block Vectors

Basic Block Vectors (BBVs) [24] 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 use the BBV structure when using our variable length intervals with SimPoint. We start with a BBV containing all zeroes at the beginning of each interval of execution. During each interval, we count the number of times each basic block in the program has been executed, and we record the count in the BBV. For example, if the 50th basic block is executed 15 times in the current interval, 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.

We used BBVs to compare the intervals of the application's execution [24, 25]. 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. Basic block vectors are fingerprints for each interval of execution: each vector indicates what portions of code are executed, and how frequently those portions of code are executed. BBVs are used to evaluate the similarity of their corresponding 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 compare our software phase marker approach to basic block vectors, since it is one of the more accurate techniques for phase classification [5].

2.3 Procedure and Loop Phase Analysis

Our approach is based on analyzing an application's procedure and loop structure and variance on execution paths to partition a program's execution into phases. This approach is motivated by prior work that used fixed length vectors of loop and procedure counts to identify phase behavior, and prior work [12, 18, 9] into finding phase behavior at the procedure and loop level.

Huang et. al. [12] proposed a hardware algorithm for tracking procedure calls via a call stack, which they used with a set of thresholds to break a program's execution into phases at the procedure level. They focus on using a hardware architecture to dynamically find and track these phases. Georges et. al. [9] implemented their approach to perform offline phase analysis of Java programs. They implement Huang's algorithm off-line to provide a workload case study on phase behavior in Java programs. For both of these works, only procedures are considered for splitting a program's execution into phases, and no software marking approach is examined.

In [16], we examined different structures for off-line phase classification, including code signature vectors where each dimension represents static procedure calls, returns and loop branches in the binary. These code signatures were used for phase analysis with fixed length intervals. From this analysis, we found that tracking procedures alone resulted in more intra-phase performance variation compared to tracking both loops and procedures.

In our work we found that it is important to use loops in addition to procedures, because loops give us more detailed information about the program's behavior patterns. Also, the utility of procedures for phase classification depends on the programmer's ability to abstract their code into useful and meaningful subroutines. As an extreme example, procedure-based analysis is very limited if the programmer writes all their code in `main`.

Huang et. al. [18] recently considered procedures and loops to partition a program's execution. The partitioning de-

termined where and when statistical samples should be taken during architecture simulation. Their analysis broke up a program's execution at static call sites, and if a procedure executed for too long, they divided the procedure's execution into its major loops. To determine the sample rate, they examine the variability of several architecture metrics for each program region. Both [16, 18] focused on dividing a program's execution trace into intervals of execution based on procedure calls and loop branches to guide architecture simulation. In comparison, our current work focuses on building a procedure call-loop graph to divide a program's execution into phases of repeating homogeneous behavior, and we use this graph to select code constructs that serve as software phase markers that indicate phase changes when executed. In addition, we examine an architecture metric independent method for modeling variance to determine if a call-loop site has too much hierarchical variance when picking software phase markers.

2.4 Software Phase Marking

The closest prior work to ours is the work by Shen et. al. [23], where they use Wavelets [3] and Sequitur [21] to build a hierarchy of phase information to represent the program's behavior patterns. Their approach is very different from ours, since they perform their phase analysis using *data reuse distance* and identify phases with advanced analysis techniques (wavelets and Sequitur), while our approach is based on a program's code structure, represented with our call-loop graph, which can be analyzed very quickly with a simple graph algorithm.

Shen et. al. [23] take the data reuse distance phases at the finest granularity and use Sequitur to find patterns in the data reuse trace, then express each pattern as a regular expression. They then select software phase markers that indicate the beginning of each data reuse pattern by finding basic blocks whose execution patterns are highly correlated with the data reuse patterns. Because they select their phase markers with reuse distance, they are able to find phase markers for programs with stable periodic behavior, but they found it difficult to find structure in more complex programs like `gcc` and `vortex`. We show that our approach can find phase behavior in all programs we examine including `gcc` and `vortex`, and we compare our approach to the approach of Shen et. al. [23] for guiding data cache reconfiguration. The goal of our paper is to (a) run our analysis in a matter of minutes, (b) create phase markers that can be inserted into the binary, and (c) apply our phase analysis to architecture reconfiguration and SimPoint.

3 Methodology and Metrics

For all of the results examined we perform our phase analysis on a subset of the SPECINT2000 benchmark suite. We chose programs that were used in prior phase analysis papers, and that were shown to be more challenging to perform accurate phase analysis for [15]. In particular, we show results for `gcc` and `vortex` since the data phase marker approach of Shen et. al. [23], could not be used to find phase behavior due to the irregular data behavior in these two programs.

The baseline architecture modeled is the same as in prior work [25]. Each of the SPEC programs were simulated to completion to collect the baseline results. We also provide results for data cache reconfiguration, which were simulated using a modified version of the ATOM [27] cache simulator used in [23]. For the data cache reconfiguration results we compare against the reuse distance-based software phase marking approach of Shen et. al. in [23]. Shen was very gracious to provide us with the exact binaries he used, the reuse distance phase markers he selected, and their ATOM-based Cheetah simulator, which allowed us to do a fair comparison to their approach. For the comparison, we used the same binaries they examined, which consists of `tomcatv`, `swim`, `compress95`, `mesh`, `applu`.

3.1 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. After classifying a program's intervals into phases, we examine each phase and calculate the average of some metric (CPI, for example) over all intervals in the phase. We then calculate the standard deviation of the metric 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 each interval by the number of instructions in the interval, so intervals that represent a larger percentage of the program's execution receive more weight in the CoV calculations. By averaging the per-phase CoVs across all phases, we have an overall CoV that measures the homogeneity of a phase classification. Better phase classifications will exhibit lower overall CoV. If all of the intervals in the same phase have exactly the same CPI, then the overall CoV will be zero.

Unfortunately, CoV is not a perfect metric - if a program with N intervals is classified into N phases, the CoV will be zero. For this reason, we also examine the number of intervals and phases for each classification.

4 Representing Program Behavior with a Procedure and Loop Graph

In this section we describe the hierarchical call-loop graph which guides the placement of software phase markers. The call-loop graph is a call graph extended with nodes for loops, where each node and edge is annotated with hierarchical instruction counts and standard deviation in hierarchical instruction count.

4.1 Procedure, Loops, and Phase Behavior

Huang et al [12] found that program behavior tends to be fairly homogeneous across different invocations of the same procedure. We find that this result extends to loops as well: program behavior across loop iterations and across different executions of the same loop nest are fairly homogeneous. We show this in the next section by showing that the Coefficient of Variation is not significantly affected by dividing

a program's execution into procedures and loops, compared to only procedures. Placing phase markers on loops allows for smaller interval sizes compared to procedures alone.

While procedures are sufficient for detecting phase behavior in many programs, like Java applications with small object oriented routines [9], we believe that it is also important to examine a program's loop structure because procedures rely on the programmer to divide their code into meaningful subroutines. In prior studies with SimPoint on the SPEC 2000 benchmark suite, we found tracking loops and procedures to be an effective way to detect phase behavior, while tracking only procedures was not as effective [16].

The call-loop graph has nodes for both procedures and loops. Each node and edge in the call-loop graph carries the call count, total local and hierarchical dynamic instruction count, as well as the average and standard deviation of the hierarchical dynamic instruction count.

4.2 Creating a Call-Loop Graph for Finding Phase Behavior

For our analysis we create a hierarchical call-loop graph where edges are annotated with the hierarchical dynamic instruction counts. We build the call-loop graph by analyzing binaries with ATOM [27]. Procedures are detected by ATOM, and we identify loop back edges by looking for non-interprocedural backwards branches. A loop is the static code region from the backwards branch to its target. There are nodes for both procedures and loops in the call-loop graph.

The call-loop graph tracks the *hierarchical instruction counts* for each edge. For a call, this is the total number of instructions executed between call and return. For each edge we keep track of (a) for procedures, the number of times the procedure is called, and for loops, the number of times the loop iterates, (b) the maximum number of instructions executed on a single traversal of the edge, (c) the average number of instructions executed on each edge, and (d) the standard deviation in the number of instructions executed on each edge.

In the call-loop graph, each procedure and loop is represented with two nodes to handle recursion and iteration. Each procedure and loop is represented with a head node and a body node. Every head node always has exactly one child, which is its corresponding body node.

For loops, the loop head node keeps track of how much time elapses between loop entry and exit, while the loop body node keeps track of how much time elapses in each loop iteration. If a loop head is selected as a phase marker, then entry to the loop are marked; if a loop body is selected as a phase marker, then each loop iteration is marked.

For procedures, the head nodes keep track of elapsed time for recursive procedures. Procedure body nodes keep track of statistics for each recursive iteration, similar to the loop-body nodes. For non-recursive procedures, the head and body nodes carry identical information.

By representing each procedure and loop with head and body nodes, we can identify more stable program behaviors. For example, we can tell if a loop does similar amounts of work on each iteration, and we can also tell if a loop does similar amounts of work between each entry and exit.

Figure 1 shows an example piece of code and Figure 2 shows a simplified call-loop graph corresponding to this code. Because there is no recursion in this example, we only show the procedure head nodes in the call-loop graph. To save space in the example, the maximum count of instructions is not shown for the edges. Procedure `foo` contains a loop and the edge between `foo` and the `loop-head` contains the hierarchical instruction count from loop entry to loop exit. In comparison, the `loop-head` to `loop-body` edge contains the hierarchical instruction count for each loop iteration. Two nodes are used to represent the loop, and the weight of the edge from `foo` to `loop-head` indicates the number of times the loop is entered, and the weight of the edge from `loop-head` to `loop-body` indicates the number of times the loop iterates.

As shown in Figure 2, each edge in the call-loop graph tracks three values: the number of times the edge was traversed (C), the average number of instructions executed each time the edge is traversed (A), and the standard deviation of the number of instructions across invocations, which is represented in Figure 2 as the Coefficient of Variation (CoV). CoV is just the standard deviation divided by the average. We use the CoV to find edges with low variance, which become candidates for phase marker selection, as described in the next section.

5 Selecting Software Phase Markers

Software phase markers are points in the binary that can be instrumented (branches, procedure calls, returns, loop entries, the start of a procedure, etc) to reliably indicate the beginning of a interval of repeating program behavior when executed. These software phase markers can be used to easily and accurately predict program phase changes at run-time with no hardware support. In addition, software phase markers can be used to predict phase changes across different inputs to a program, and across different compilations of the same source code.

In this section, we discuss our software phase marker selection algorithm. Given a call-loop graph as described in the prior section, our algorithm selects phase markers that can be inserted into a program with a static or dynamic compiler or binary instrumentation.

5.1 Selecting Markers from the Call-Loop Graph

Many programs exhibit repeating behavior at different time scales. When selecting software phase markers, the selection algorithm needs to know whether the user is interested in large or small scale behaviors. For example, optimizations with high overheads will be more interested in large-scale phase behavior, since they require more time between optimizations to recoup the cost of applying each optimization. Our call-graph can be used to find both large and small scale phase behaviors, and in this paper we focus on finding phase behavior by starting the search at small granularities and moving upward to larger granularities to identify marker

```
Proc foo()
   ...
   loop
      ...
      if (cond) call X;
      else call Y;
      ...
   end loop
   ...
   call X;
   ...
end proc

Proc X()
   ...
   call Z;
   ...
end proc
```

Figure 1: Code example

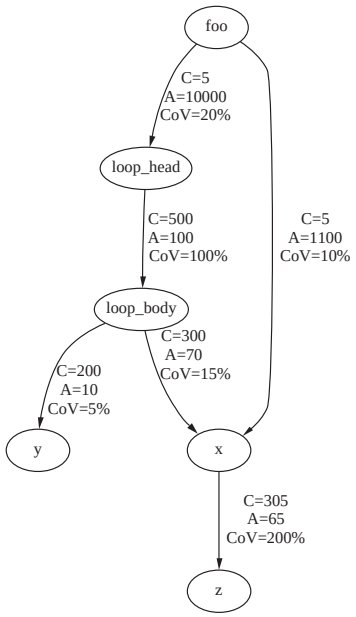

Figure 2: Call-Loop graph for code example: C is the number of times each edge is traversed, A is the average number of hierarchical instructions executed each time the edge is traversed, and CoV is the hierarchical instruction count coefficient of variation. For procedures, only the procedure-head nodes are shown, since the example does not have recursion. Maximum instruction counts are also not shown to save space.

points with low variation in hierarchical instruction count at the desired granularity.

The call-loop graph in Figure 2 shows why we use the hierarchical instruction count CoV to guide the selection of software phase markers. Each edge in the Figure shows `C` - the number of times the edge was traversed, `A` - the average hierarchical number of instructions executed each time the edge was traversed, and `CoV` - the hierarchical instruction count coefficient of variation (standard deviation divided by average). The example in Figure 2 shows that the edge from `X to Z` has a CoV of 200%. In the example, each time `Z` is executed from the path coming from the `loop-body` edge `Z` executes 50 instructions on average. In comparison, each time `Z` is executed coming from the `foo to X to Z` path, `Z` executes 1000 instructions on average. Therefore, looking at the edge from `X to Z` we see a significant CoV in hierarchical instruction count per edge traversal. But if we look at the incoming edges to `X` the additional path differentiation can separate the hierarchical instruction edge counts per traversal into more homogeneous behavior. In this example the edges from `loop-body to X` and `foo to X` both have a low CoV. This means that each time the edge was traversed the number of instructions executed hierarchically was about the same, so these two edges are good locations for software phase markers.

We can also use this example to demonstrate the overall algorithm we use to pick the software phase markers. Our algorithm takes as input a threshold for the minimum average interval size (hierarchical instruction count). For this example, assume that this instruction count threshold is set to 90, so we must have `A` >= 90. The edges `foo to loop-head`, `foo to X` and `loop-head to loop-body` would then be the only edges that would be considered as potential software phase markers. In addition, our algorithm also uses a CoV threshold, which is set on a per profile basis which we describe below. If, for the example, we assume this is set to 50%, then the edge `loop-head to loop-body` would not be considered, since its CoV is 100%. This means that there is too much variation in hierarchical instruction count for each iteration of the loop. In comparison, each time the loop was entered and exited (`foo to loop-head` edge) the CoV was low because the overall execution count for all of the iterations came out to be similar. Therefore, a better place to put the software marker is at the edge `foo to loop-head`. The end result would be the placement of the software markers at edges `foo to loop-head` and `foo to X`.

As described in the example, our algorithm consists of two phases. The first phase focuses on pruning the graph based on average number of instructions per interval. When a marker is placed on an edge in the call-loop graph, the average number of instructions per interval for that marker will equal the average hierarchical instruction count on that edge (shown as `A` in Figure 2). The second pass of the algorithm searches the remaining nodes in the call-loop graph in reverse depth order, looking for edges with a low average hierarchi-

cal instruction count CoV. This algorithm therefore takes two inputs: a call-loop graph and $ilower$. $ilower$ specifies the minimum allowed interval size, which is the minimum number of instructions allowed per interval. A CoV threshold is used to limit the allowed variability in markers, but it is automatically calculated in the first pass of the algorithm.

Pass 1 - Pruning based on the average hierarchical number of instructions: We first estimate the maximum depth of each node in the call-loop graph. This is done with a modified depth-first search, where a node can be traversed more than once if we later find a longer path to that node. We never re-traverse a node on the current path, to ensure the algorithm terminates if the graph contains a cycle. We place the nodes into a queue, sorted by decreasing estimated maximum depth. This ensures that we will process children before parents. We break ties by sorting by increasing out-degree, so we process leaf nodes before non-leaf nodes. We take each node off the queue and look at each of its incoming edges. We check if the average executed number of instructions for each edge satisfies the $ilower$ requirement. If the requirements are met, we mark the edge as a potential software phase marker. Once all of the incoming edges on the current node are processed, we continue to process the nodes in the queue. After this pass is done we have a list of potential software marker edges, where all of the edges are above the average number of instructions allowed per interval. The next phase will use these edges to calculate a CoV threshold, and select a subset of these edges to use as phase markers.

Pass 2 - Setting and applying the hierarchical instruction count CoV threshold: The first pass of the algorithm prunes away the lower parts of the call-loop graph that represent low-level behavior patterns that are too small to mark according to $ilower$, the minimum average number of instructions allowed per interval. We use the result of the first pass to set a CoV threshold to select software phase markers from the list of potential software phase markers.

We use the potential software phase markers found in the first pass to calculate a CoV threshold independently for each benchmark, because programs inherently have different levels of variability. In general, floating point programs have more stable instruction counts within each loop and procedure, while integer programs are more variable. By tuning our CoV thresholds to the variability found in each program, we can still find stable behavior patterns in highly variable programs.

The base CoV threshold is set to the average CoV (avg(CoV)) across all of the edges in the list of potential phase markers. We also calculate the standard deviation of the CoVs in the list of potential phase markers, and the actual CoV threshold that gets applied to an edge is between avg(CoV) and $avg(CoV) + stddev(CoV)$, scaled linearly with the current edge's average hierarchical instruction count. This encourages the algorithm to pick edges with instruction counts close to $ilower$, by allowing more variability as the average instruction count grows away from $ilower$.

After a $cov\_threshold$ is determined for an edge, we process edges as in the first pass, except that an edge must now satisfy both the $ilower$ minimum instruction count threshold, and the $cov\_threshold$ variability limit to qualify as a phase marker. When the queue is empty we have a set of edges selected as software phase markers that satisfy both the $ilower$ instruction count threshold and whose variation in hierarchical instruction count are below the $cov\_threshold$.

Complexity of the Algorithm: Our algorithm's running time is $O(E+N*log(N))$, where N and E are the number of nodes and edges in the graph. The $N*log(N)$ is due to a sort of all of the nodes to create a total call-loop depth ordering of the nodes during the first part of the algorithm. The algorithm runs in seconds on every call-loop graph we have collected. The approach is faster and less complex than the approach of Shen et. al. [23], where wavelet analysis [3] is applied to reuse distance traces, and Sequitur [21] is applied to the results of the wavelet analysis. It is also significantly faster than the VLI approach in [15] where Sequitur is run on a branch trace to find hierarchical phase behavior.

We later show that the stability of the phases detected by our approach are comparable to the results of Shen et. al. [23], and that we can find predictable phase behavior in irregular programs, which the approach in [23] had some trouble with.

5.2 Limiting Maximum Interval Size for SimPoint

The above algorithm is the default phase marker algorithm we use for our analysis and for finding homogeneous behavior to guide reconfiguration optimizations. This algorithm does not have a limit on the size of the intervals chosen, so in the results in this section we call the above algorithm *no-limit*. Since there is no limit on the phase interval sizes, this algorithm can create phases with large intervals.

We also want to use our approach to pick simulation points for SimPoint [25]. The goal of SimPoint is to pick a set of simulation points, one from each phase of behavior, to guide architectural simulation. These simulation points together provide an accurate representation of the complete execution of the program. To use our approach for SimPoint, we break up the program's execution into *Variable Length Intervals* (VLIs), whenever a new phase marker is seen during execution. When using phase markers with SimPoint, we need to limit the maximum interval size to keep the simulation time reasonable.

We set a limit on maximum interval size with two additional steps in pass 2 of the base algorithm. Both of these steps are used to enforce a maximum interval size, called *max-limit*, when dividing execution into phases.

Maximum Interval Limit: During profiling, we record the maximum hierarchical instruction count on each edge. When looking for phase markers in the call-loop graph, if the maximum hierarchical instruction count on a node's incoming edge exceeds the *max-limit*, we stop searching for a marker on this path (because everything else will be even larger) and we mark the current node's outgoing edges (which must be below the limit).

Merging Loop Iterations: In the second pass, we also try to group together consecutive iterations of a loop if each iteration has a similar average hierarchical instruction count.

If the edge from the loop-head to the loop-body is below the CoV threshold, we group together consecutive loop iterations until they are (a) greater than the minimum interval size, and (b) lower than the *max-limit* threshold. Within these limits, there are many potential groupings of iterations to potentially choose from. For example, we could group two, three, or four consecutive intervals together. We know how many times the loop is traversed on average from our call-loop graph, so we group together N iterations that results in the average number of iterations mod N closest to 0. In other words, we look for a value of N that evenly divides the number of loop iterations per entry to the loop nest that satisfies the above interval size constraints.

Both of the above interval size heuristics are motivated strictly for use with SimPoint to help reduce simulation time. Markers found with these additional constraints can be fairly input specific, so we only advocate this approach with SimPoint. It is not meant to capture behavior across inputs.

For the `limit` phase marker results in this paper we use a minimum interval size of 1 million instructions and max-limit of 200 million instructions. Once phase markers have been chosen with these additional heuristics, we run the program gathering variable length interval basic block vectors, which are then fed through SimPoint to select simulation points.

5.3 Using the Software Phase Markers

We select software phase markers that, when executed, reliably predict the beginning of a repetitive interval of program execution. The most obvious way to use software phase markers is to use them as triggers for dynamic reconfiguration or optimization. This can be done by inserting code into the binary at phase markers to trigger reconfiguration or optimization. This can be done with a binary modification tool such as OM [28] or ALTO [20].

Figure 3 shows the phase markers selected by our algorithm for gzip-graphic. Time is on the X-axis in instructions, and CPI and level 1 data cache miss rates are on the Y-axes. Phase marker locations are indicated with symbols (circles, squares, etc) plotted on top of the CPI or data miss rate. Each phase marker is assigned a unique symbol. In Figure 3, the 2 large sized phases found are between the circles and triangles. In Figure 3, the beginning of each long high miss rate phase is marked with a circle, and the beginning each short low miss rate phase is marked with a triangle.

There are many more phase markers than shown in Figures 3 and 4. To make these graphs readable, we only plot the first occurrence of markers that repeat very frequently. There are actually phase markers at each ridge shown during a long stable region of high miss rate (circle) or low miss rate (triangle).

Figure 4 shows how the phase markers selected by our algorithm for gzip-graphic from an OSF Alpha binary apply to a Linux x86 binary. For this result, we take the markers selected on Alpha and map them back to source code level, using debug line number information. We then use the symbol information from the x86 binary to map the markers to the corresponding assembly instructions in that binary.

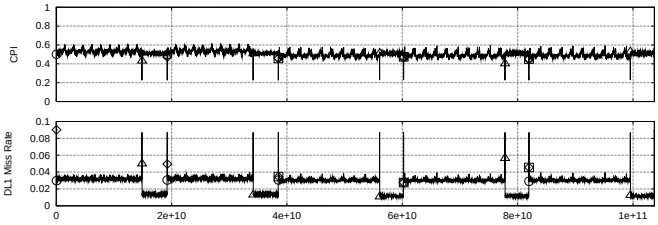

Figure 3: Time varying graphs with phase markers for gzip-graphic for an OSF Alpha executable. Time is on the X-axis, measured in instructions executed. Phase marker locations are indicated with symbols. Each marker is assigned a unique symbol. If a marker occurs many times in a row, we only plot the first instance from each repeating run to make the graph more readable.

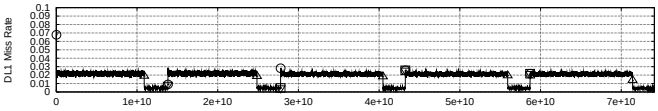

Figure 4: Cross-ISA time varying graph with phase markers for gzip-graphic for Linux x86. The phase markers were selected from the call-loop graph profile from the Alpha binary, were mapped back to source code level, and then used to mark the x86 binary. No call-loop graph was created for the x86 binary. The markers detect the same high-level patterns in the x86 binary.

We applied this matching technique to other benchmarks and found similar results. This result shows the potential of using the phase markers across different compilations of the same source program.

Figures 5 and 6 are visual representation of the complete execution of `bzip2-graphic`. These figures are a 3-dimensional projection of the basic block vectors collected from each execution interval, where each interval is represented with a single point in the figures. Each interval is projected down the 3 dimensions using random linear projection and plotted as a point on the graph. Figure 5 shows fixed length 100 million instruction intervals and Figure 6 shows the software phase marker variable length intervals. In both of these approaches a similar number of intervals were used to represent the entire execution of the program. The same projection matrix was used for Figures 5 and 6, but they are shown at different angles. The angle was chosen for each graph to best show how each set of intervals captures the program's use of its code space.

Bzip2 spends the majority of execution in several code regions, and transitions between these regions just a few times. The dominant code regions can be seen in both figures as dense clouds of points. These code regions are substantially more clearly defined in Figure 6 over Figure 5. The transitions between these code regions can be seen in Figure 5 as a string of points connecting the denser regions. These transitions are not visible in Figure 6 since the entire phase repre-

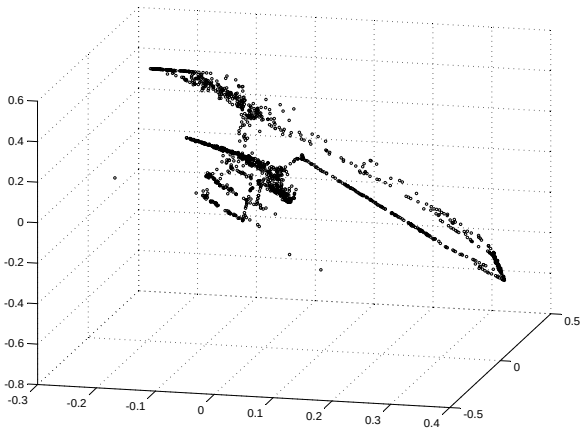

Figure 5: Bzip2 fixed length execution intervals representation. The scattered representation with points spread across the space is a direct consequence of using fixed length intervals across the execution.

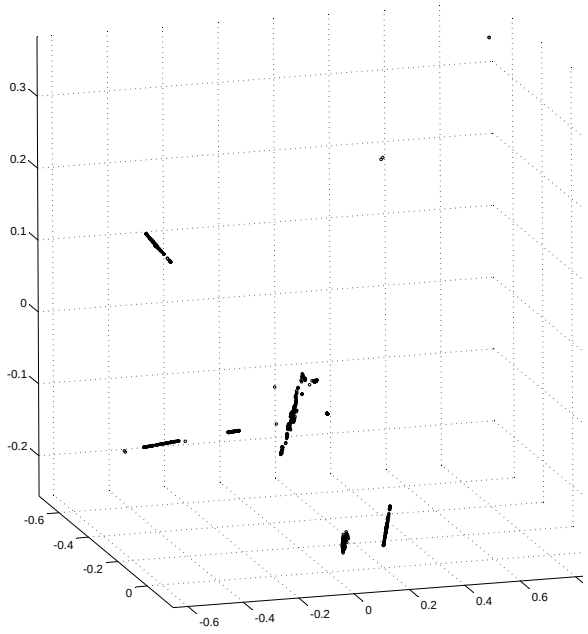

Figure 6: Bzip2 variable length execution intervals representation using our phase markers. The tight clustering of intervals is from marking regions of the hierarchical call-loop graph that have fairly homogeneous behavior each time that edge is traversed during execution.

sentation is synchronized with the program behavior where transitions between dominant regions are encapsulated by unique intervals.

These figures provide visual evidence that software phase markers are partitioning the execution into naturally occurring intervals that are in the code. This representation is capturing a lot of periodic behavior in the program. On the other hand, the fixed length intervals are ignorant of the underlying program behavior, and consequently are dissonant with periodic behavior. That explains why visually the execution appears more chaotic with fixed length intervals.

5.4 Behavior Characteristics of Software Phase Markers

In this subsection we present behavior results from using the marker selection algorithm. For these results, we set $ilower$ to 1M instructions. We experiment with selecting phase markers from the training input and applying the markers to the ref input (cross-train), as well as selecting and applying markers from the ref input (self-train). All numbers are reported running the ref input.

We compare the results of our phase marking algorithm to SimPoint 2.0 [25], an offline phase analysis tool based on the k-means clustering algorithm from machine learning. For experiments with SimPoint, we collected basic block vectors with ten million instructions per interval, and ran SimPoint on the vectors with a 15 dimension random projection and $k_{max} = 100$. SimPoint classifies the intervals of execution into phases. Note this is an idealized comparison, since the SimPoint analysis cannot be used across inputs. We also show experiments with allowing our phase marking algorithm to only mark procedures. The result is similar to the approach of Huang et. al. [12] and that used by Georges et. al. [9], but we use our call-loop graph, instead of a dynamic call stack approach used in their prior work. This allows us to mark the procedure call edges in the binary.

For our phase marker results, the `no-limit` results represent creating phase markers using the algorithm as stated in Section 5.1 where we do not put an upper bound on the maximum interval length. In comparison, the result called `limit 1-200m` represents using the additional heuristics in Section 5.2 to choose phase markers that constrain the size of the variable length intervals produced to be used by SimPoint. For these results we used a minimum interval size of 1 million instructions and max-limit of 200 million instructions.

Figure 7 shows the average interval length selected by each approach. BBV uses fixed-length 10M instruction intervals. The next two bars show the results using our phase marking approach, but only looking at edges coming into procedure-heads and procedure-bodies in the call-loop graph. The last three bars show the results for choosing any edge in the call-loop graph (marking both procedures and loops). All except the last bar for the phase marker results choose phase markers without specifying a limit on the maximum interval size, whereas the last bar using a maximum limit of 200 million instructions. Bars that might look like they are missing have an average interval size of 1 million instructions. For the self-train results, we examine a call-loop graph of the ref-

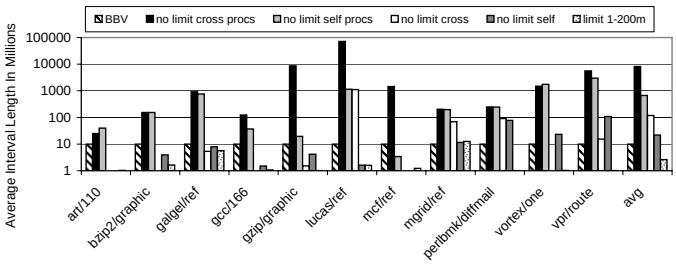

Figure 7: Average instructions per interval. BBV uses fixed 10M instruction intervals. The rest of the results use software phase markers without specifying a limit on the maximum interval size and the last bar using a limit of 200 million instructions. The second and third bar only allow marking of procedures, whereas the last three bars can mark both procedures and loops.

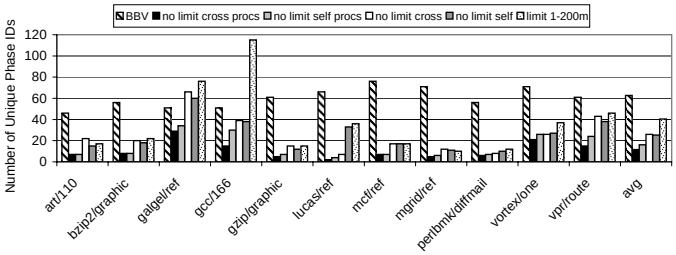

Figure 8: Number of phases detected

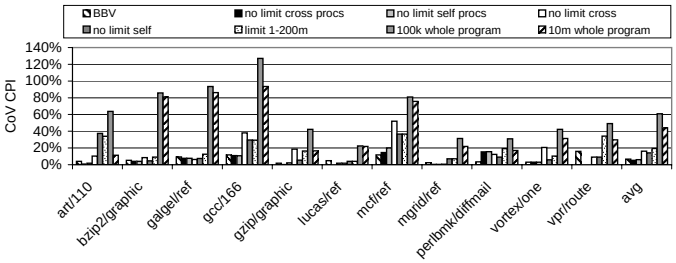

Figure 9: Coefficient of variation of CPI. Whole Program shows each program's variability assuming each interval is classified into a single phase

erence input (self-train), and for (cross-train) we examine a call-loop graph from the training input and apply the markers to the ref input (cross-train). Using only procedures to mark phases results in average interval sizes of 1 billion (self-train) to 10 billion (cross-train) on average. Loops bring the average interval size down to 10-100 million. The last bar shows that putting a limit on the size of the intervals when performing the marking has an average interval size of 3 million instructions.

Figure 8 shows the number of phases detected by each approach. The BBV approach detects the most phases, and it also has the lowest variation in phases as we will see in Figure 9. In most cases, our approach detects half as many phases as the BBV approach. Finally, if we constrain the search space of the call-loop graph by limiting the interval sizes considered for SimPoint, we end up with more phase markers.

Figure 9 shows the average coefficient of variation of CPI per phase. The last two bars in these graphs show the overall program CoV using fixed length intervals of 100,000 instructions and 10 million instructions. These graphs show that both the BBV and our software phase marker approach successfully partition execution into phases of homogeneous behavior. The procedures-only results have a lower CoV CPI for some programs than when using loop and procedures. This occurs because marking procedures detects fewer phases and produces significantly larger intervals compared to procedures and loops. The general trend we have found is that more behavior variability must be tolerated with smaller interval sizes. For example, it is easy to get a CoV of close to zero with a few number of phases: just treat the whole program as one big interval. This is essentially what happens for a few programs (like `vpr`) for the procedure only results. In general, program behavior variability decreases as larger intervals of execution are examined. In all cases, the average behavior variation within each phase is much lower than the program's overall behavior variability.

6 Applications: Data Cache Reconfiguration and SimPoint

In this section we examine applying our code phase markers to data cache reconfiguration as in [23] and to SimPoint as in [15].

6.1 Data Cache Reconfiguration and Comparison to Data Reuse Markers

Adaptive data cache reconfiguration dynamically reduces the cache size to reduce energy consumption and access time, without increasing the miss rate. In this section we perform the exact same data cache reconfiguration experiment done by Shen et. al. [23] to apply our software phase markers to data cache reconfiguration. To ensure that our results are comparable to prior work, we obtained from Shen the benchmarks and simulation infrastructure used in [23] as discussed in Section 3. We also obtained from them their binaries and their data phase markers, and we simulate the same adaptive cache hardware: 64-byte blocks, 512 sets, 32KB to 256KB cache size. The cache reconfigures by changing associativity from 1 to 8 ways.

In using their approach (Shen et. al. [23]) for adaptive cache analysis, during execution the first two intervals for each phase marker are spent experimenting with the different cache configurations. In the first two intervals, the best cache configuration is determined for the phase. After the first two intervals, when the phase marker is seen again, the best cache configuration is automatically used for the interval. We apply the same algorithm with our software phase markers.

We compare their markers against an ideal SimPoint [25] approach. SimPoint is an offline phase analysis tool based on the k-means clustering algorithm from machine learning.

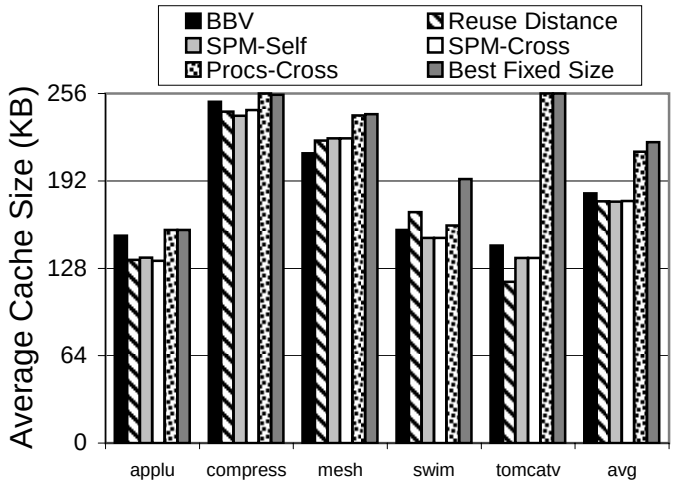

Figure 10: Average cache size with no allowed increase in cache miss rate. BBV is the idealistic SimPoint-based approach, Reuse Distance is the approach of Shen et al (trained with train input), and Software Phase Marker (SPM) is our approach (trained with ref input (self) and train input(cross)). Procs only uses procedures for picking the phase markers (no loops). Best Fixed Size is the smallest fixed cache size with the maximum hit rate out of the cache configurations we examine.

SimPoint classifies the intervals of execution into phases, assigning each 10M instruction interval a $phaseid$. We take the $phaseid$s produced by SimPoint, and we use them to trigger cache reconfiguration at each interval boundary. The ideal SimPoint-based cache reconfiguration has oracular knowledge of the phaseid for each upcoming interval, as determined by the offline k-means algorithm. We find this approach to be a good approximation to the hardware BBV phase classification approach in [26, 17] with perfect next-phase prediction.

Figure 10 shows the average cache size used by each cache reconfiguration approach on the benchmarks used by Shen et. al. [23]. The figure shows the average cache size used over the execution of the program. The Self results show using the same input to both generate the phase markers as gathering the results, and the Cross results show using the training input to generate the phase markers and the reference input for reporting the adaptive cache size results. For these results, we found for our approach that the benchmarks are very regular: the average coefficient of variation of hierarchical instruction count in marked procedures and loops is less than 1% for these programs. This means that it is easy to predict the number of instructions executed within the program's main procedures and loops with very high accuracy. On these benchmarks, our phase marking approach outperform the idealistic BBV approach, indicating that the fixed-length intervals are out of sync with the phase behavior exhibited by these programs. For example, our phase marking approach selects intervals with an average of 4M instructions for applu. Fixed length intervals of 10M instructions should not work well in this case, and we see that with the BBV average cache size being close to the best fixed cache size for applu.

The results also show that our simple software phase marking approach is as effective as the more complicated reuse distance-based approach of Shen et. al. [23] for cache reconfiguration on these regular programs. We also find that selecting markers from the train input is as effective as selecting markers from the ref input, as expected due to the regular behavior patterns exhibited by these programs. Examining only procedures does not work very well for some of these programs because they spend most of their time in loops. We end up marking only a few procedure call edges for some of the programs as described in the prior section. Shen et. al. [23] only provided us with their phase markers and binaries for the programs we provide results for in Figure 10. Therefore, we could not run their reuse-distance analysis to generate results for gcc or vortex for their approach, which is why `gcc` and `vortex` are not shown. For our own binaries we ran this experiment for `gcc` and `vortex`. For `gcc`, we found the best fixed cache to be 256KB and 240KB using software phase markers. For `vortex`, we found the best fixed cache to be 245KB and 200KB using software phase markers.

6.2 Applying Phase Markers to SimPoint

The focus of our approach with SimPoint is not to improve the accuracy or to reduce simulation time over the standard SimPoint technique. Instead, the focus of using phase-marker based simulation points is to create simulation points that can be re-used across inputs, and potentially re-used if the program is recompiled, even on another architecture.

We now examine using our phase marking algorithm to partition a program's execution at phase marker boundaries. We use the additional heuristics described in Section 5.2 to limit the maximum interval size. We set the maximum interval size threshold to 200 million instructions. The average interval length, number of phase IDs and CoV of CPI results for these `limit` phase markers are shown in Figures 7, 8, and 9.

To produce the variable length interval SimPoint results we use the phase markers to partition each program's execution into variable length intervals. Whenever a marker occurs during execution, that is a start of a new interval. As we break the program's execution into VLIs, we collect a basic block vector for each variable-length interval. To pick the simulation points, we pass these variable length basic block vectors through the SimPoint 3.0 VLI [15, 10] algorithm. We had to use this new version of SimPoint, since each VLI represents a different percentage of execution. Standard SimPoint 2.0 assumes each interval represents an equal fraction of program execution.

Figure 11 shows the number of simulated instructions required, and Figure 12 show the CPI error rates for the various flavors of SimPoint. In these graphs, the first three bars show the number of simulated instructions required when using the

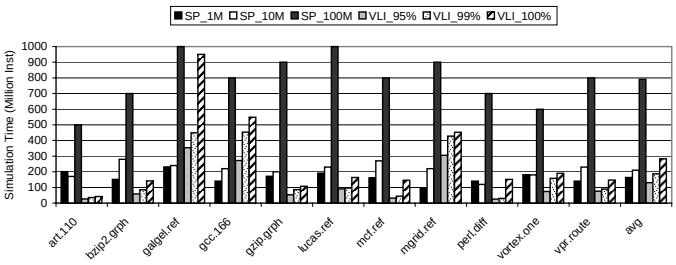

Figure 11: The number of simulated instructions when using fixed length intervals of size 1, 10 and 100 million instructions for SimPoint. This is compared to using phase markers to break the program's execution into VLIs.

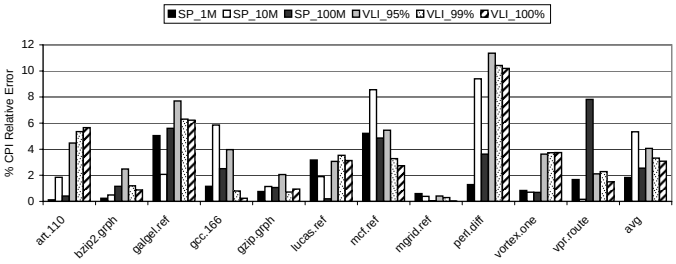

Figure 12: Estimated CPI SimPoint error when using fixed length intervals compared to variable length intervals using phase markers.

existing SimPoint [25] 2.0 approach with fixed length intervals using interval sizes of 1, 10 and 100 million instructions. For the fixed length interval results, k_{max} was set to 300 for the 1 million interval, 30 for 10 million interval, and 10 for 100 million interval as in [22]. In all these experiments, simulated CPI data was used for the chosen intervals with perfect warmup.

The last three bars in these graphs shows results for VLIs created with our phase markers. The last bar (VLI 100%) shows the results when a simulation point is selected from each cluster. We also consider a common optimization applied to SimPoint, where clusters are sorted by their weight, then simulation points are selected from the top N clusters which account for 95% or 99% of execution. This is a common technique which trades simulation time for accuracy [10].

For the VLI results in Figures 11 and 12, the number of clusters and simulation time are highly correlated to the number of phase markers shown in Figure 8. This is because the phase markers delineate different code behaviors, which means the BBVs created for different phase markers should be reasonably different from each other. In Figure 8, `galgel` and `gcc` have a large number of phase markers for the `limit` approach, because we end up marking many small child in the graph due to our restrictions on maximum interval length, which results in more phase markers. The call-loop graphs for these programs have several points where we are forced to make a choice between marking a very large interval, or marking a large number of small intervals. In these cases, to enforce our upper limit on interval size (simulation time) we end up marking a large number of small intervals.

Overall, the results show that the variable length intervals, when using the 99% filter, has about the same simulation time as 10m fixed length SimPoint with a comparable error rate. The results show that our VLI approach does not really provide an improvement over 1m or 100m fixed length SimPoint when just looking at the results, but the results are comparable, which was our goal.

6.2.1 Source Code Simulation Points

The goal of using phase markers with SimPoint is to partition a program's execution into variable length intervals that match the natural phase-based code transitions during execution. This allows us to mark phase transitions at the code level and to potentially pick the same simulation points across different compilations of the same source code when running the same input. This can be useful for using SimPoint to guide architecture studies where there are ISA changes, or to study compiler optimizations (picking phase markers that are not compiled away).

To examine the potential effectiveness of our approach, we examined selecting phase markers to be used across two compilations of each program. We compiled each Alpha binary without optimization and with full peak optimization. We then used our phase marker selection algorithm to choose a single set of simulation points to be used across these two binaries.

To verify that these markers will result in the same phase behavior across the two binaries, we took the selected phase markers and produced a trace of executed phase markers for each binary for a specific input, similar to the trace shown in Figure 3. We then compared these two phase marker traces between the two binary runs to make sure that we see the exact same number of phase markers, and the exact same order of phase markers between the two traces. For the programs we examined, these traces were an identical match. This means that by using these phase markers to select simulation points, we can accurately map the simulation point across the binaries based on these phase marker traces. This will allow us to simulate the exact same part of execution (even though the number of dynamic instructions can vary) for a given simulation point across these different compilations of the same source code. Presenting the details for this approach and flushing out the algorithm is our current and future research.

7 Summary

We presented an automated profiling approach to identify code constructs (branches, procedure calls and returns) that indicate phase changes when executed. We call these code constructs "software phase markers," and they can be used to easily detect phase changes across different inputs to a program without hardware support.

Our approach builds a combined hierarchical procedure

call-loop graph to represent a program's execution, where each edge also tracks the average hierarchical execution variability on paths from that edge. We run a simple graph algorithm on the call-loop graph to identify marker points with low variability that will result in a desired minimum interval size.

We demonstrated that the phases detected by our approach have behavior homogeneity comparable to the phases detected by the offline SimPoint [24, 25] clustering algorithm. We demonstrated that our approach, which has significantly faster analysis time for selecting markers, is as effective as the approach of Shen et al [23] for dynamic data cache reconfiguration, even though we choose our markers by examining only our call-loop graph, and they set their markers by examining data reuse distance traces. In addition, we detect phase behavior in irregular programs such as `gcc` and `vortex`, which [23] says that they have trouble with.

Finally, we show that our phase markers can be used to create variable length intervals to guide SimPoint. The benefit here is not an improvement in SimPoint accuracy or reduction in simulation time. Instead, our goal is to provide a new feature, where simulation points can be mapped to source code, so simulation points can be re-used if the program is recompiled, even on a different instruction set. This is the focus of our current research, which we call cross-binary simulation points.

Acknowledgments

We would like to Xipeng Shen and Chen Ding for providing us with data and their cache simulation infrastructure, we thank Michael Huang for answering our questions about EXPERT, and we 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] A. Cohen and R. D. Ryan. *Wavelets and Multiscale Signal Processing*. Chapman & Hall, 1995.

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

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

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

[7] 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.

[8] 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.

[9] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 2004.

[10] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7, September 2005.

[11] M. Hind, V. Rjan, and P. Sweeney. Phase shift detection: A problem classification. Technical report, IBM, August 2003.

[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] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

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

[17] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, January 2005.

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

[19] 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.

[20] R. Muth, S. Debray, S. Watterson, and K. D. Bosschere. alto : A link-time optimizer for the DEC Alpha. In *Software—Practice and Experience*, pages 31:67–101, January 2001.

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

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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.

[28] A. Srivastava and D. W. Wall. A practical system for intermodule code optimizations at link-time. *Journal of Programming Languages*, March 1993.