Online Phase Detection Algorithms

Priya Nagpurkar[†] Michael Hind[‡] Chandra Krintz[†] Peter F. Sweeney[‡] V.T. Rajan[‡]
[†]University of California, Santa Barbara [‡]IBM T.J. Watson Research Center

Abstract

Today's virtual machines (VMs) dynamically optimize an application as it is executing, often employing optimizations that are specialized for the current execution profile. An online phase detector determines when an executing program is in a stable period of program execution (a phase) or is in transition. A VM using an online phase detector can apply specialized optimizations during a phase or reconsider optimization decisions between phases. Unfortunately, extant approaches to detecting phase behavior rely on either offline profiling, hardware support, or are targeted toward a particular optimization.

In this work, we focus on the enabling technology of online phase detection. More specifically, we contribute (a) a novel framework for online phase detection, (b) multiple instantiations of the framework that produce novel online phase detection algorithms, (c) a novel client- and machine-independent baseline methodology for evaluating the accuracy of an online phase detector, (d) a metric to compare online detectors to this baseline, and (e) a detailed empirical evaluation, using Java applications, of the accuracy of the numerous phase detectors.

1 Introduction

Dynamic optimization systems [11, 4, 7, 3] perform optimization while a program is executing. Such systems include modern VMs with dynamic compilers [26, 33, 2], dynamic binary optimizers [4], and reconfigurable hardware [9]. These systems achieve their performance gains by biasing their optimization strategies to the application's current execution behavior. However, such decisions can degrade performance when the underlying execution behavior changes between phases.

An *online* phase detector determines when an executing program is in a stable phase or in a transition between phases. This technology can be used by dynamic optimization systems to perform specializing optimizations when the behavior is stable or it can reconsider optimization decisions when the behavior changes.

Researchers have shown that they can capture, characterize, predict, and visualize program phase behavior [30, 31, 32, 12, 29, 10, 24]. Moreover, existing systems use phase behavior to guide effective hardware reconfiguration [9, 32, 29], hardware-based value profiling [32], program and system analysis [22, 24], remote profiling [25], efficient simulation [30], and cycle-close trace generation [27].

Unfortunately, extant approaches for detection and prediction of phase behavior rely on either *offline* profiling [18, 29, 13, 27, 24], hardware support [30, 31, 32, 12, 5, 23, 9, 25], or are targeted toward a particular optimization (client), e.g., dynamic hardware reconfiguration. Programs that execute on virtual machines, such as programs written in the Java programming language, are compiled dynamically, executed on any hardware for which a VM is available, and optimized in a variety of different ways. As a result, it is desirable for a phase detection solution not to depend on 1) offline profile information, 2) specialized hardware, 3) architecture-specific metrics, or 4) a specific optimization client.

Vital to the efficacy of phase-guided optimizations is the accuracy of online phase detection algorithms [14]. By defining a large class of online phase detectors and evaluating their accuracy, this paper takes a necessary initial step in the understanding of online phase detector accuracy for dynamic optimization systems.

To facilitate the design and implementation of online phase detection algorithms, we define a parameterizable framework; a phase detector is an instantiation of the framework. Section 2 describes this framework, its components, and parameters. To evaluate the accuracy of these algorithms, Section 3 defines a new client- and machine-independent empirical methodology. Sections 4 and 5 employ this methodology to assess the accuracy of our online phase detectors. Section 6 discusses related work, and Section 7 draws conclusions and discusses future work.

2 Our Framework

This section presents our framework for online phase detection algorithms. Figure 1 presents a component view of this framework. The input to the framework is a se-

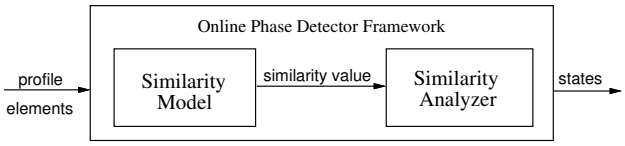


Figure 1: Illustrated view of phase detection framework

quence of profile elements, i.e., an execution profile. The first component, a *similarity model*, consumes the profile elements and transforms them into a sequence of similarity values that represents the degree of similarity between recent profile elements. The model passes the similarity value in an online manner to the second component, the *similarity analyzer*. The analyzer determines whether the similarity is sufficient to signify the execution is in phase, $\mathcal{P}$, or in transition between phases, $\mathcal{T}$. The output from the framework is a series of states, one per input element. From this output, we can identify phase boundaries at points in the output at which there is a $\mathcal{T}$ followed by a $\mathcal{P}$ state or a $\mathcal{P}$ followed by a $\mathcal{T}$ state.

The detector can include optional features, such as a level of confidence in the current state, or whether a detected phase is similar to a previously known phase [32]. Unlike an offline phase detector, our online detectors do not have the complete profile available from which it identifies phases. Because an online detector executes concurrently with the program, it must be efficient in both time and space. Moreover, because the clients of the framework make decisions based on phase boundaries, the algorithms that the framework instantiates must output phase boundaries accurately. This paper focuses on this latter constraint: phase detector accuracy.

The model and analyzer components can be implemented in many ways. For example, the model can differ in how it consumes, internally represents, and computes the similarity of the profile. Many extant phase detection approaches compute similarity using unweighted sets [9] and weighted sets [17, 16, 30, 31, 32]. A simple analyzer reports a $\mathcal{P}$ state when the similarity value exceeds a predetermined fixed threshold [17, 16, 30, 31, 32, 9, 10]. By varying the implementation and parameterization of these components, the framework can be used to investigate, compare, and evaluate both extant and novel algorithms.

In our online phase detectors, a model represents the most recently consumed profile elements with a *current window* (CW), and represents the next most recently consumed profile elements with a *trailing window* (TW). A similarity value captures the similarity of the elements in the two windows. A window policy of the model determines, for example, the CW size, the TW size, and the number of profile elements consumed at a time, which we refer to as *skipFactor*. A significant amount of prior work [17, 16, 30, 31, 32, 9, 10] sets the size of the CW, TW, and *skipFactor* to the same value. We investigate the efficacy of such a parameterization as part of our analysis.

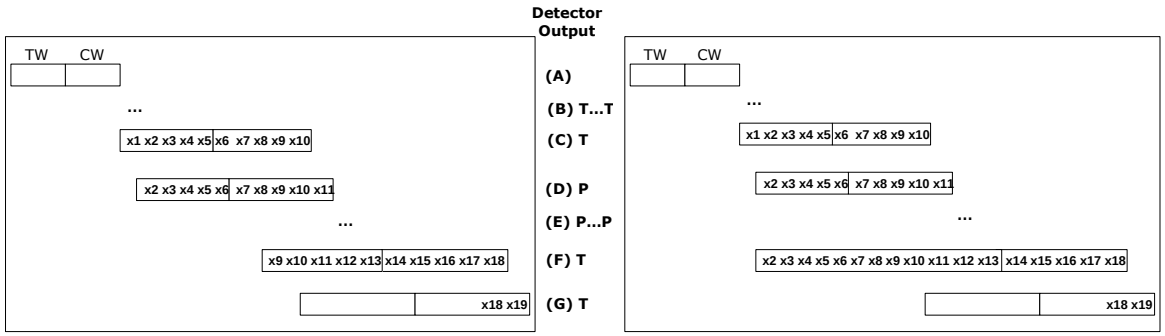
Figure 2 illustrates the basic operation of the framework using two different trailing window policies: Constant (a) and Adaptive (b). Each row illustrates a different point in time as reflected in the contents of the TW and CW. Profile elements are numbered in the order in which they are consumed. Initially, both the CW and TW are empty (row A). As the program executes, the windows fill *skipFactor* profile elements at a time (*skipFactor* equals 1 in this example). Until the windows fill (row B), the detector outputs $\mathcal{T}$. Once the windows are full, the model computes the similarity between the two windows and the analyzer produces a $\mathcal{P}$ or $\mathcal{T}$ state. At row C this computation results in a $\mathcal{T}$ state. The computation at row D results in a new phase being detected, which continues for a series of profile elements in row E.

When the phase ends at row F, we see the difference between the two policies. With the Constant TW, the TW size remains the same (length five in this example). The Adaptive TW policy grows the TW to include all elements in the phase. When the phase ends, the algorithm flushes the TW and initializes the CW with the last *skipFactor* profile elements. Row G illustrates the CW after it consumes the next profile element.

Figure 3 presents a high-level description of our framework's internal process. A detection client invokes `processProfile` with the most recent *skipFactor* profile elements. The model consumes the new profile elements, updates the CW and TW, and computes a similarity value for the updated windows. The analyzer uses this value to determine the new state, $\mathcal{P}$ or $\mathcal{T}$. If the output state begins a new phase, the model can optionally anchor the TW at the start of the phase. While in phase, the analyzer tracks the statistics of the phase. If the output state ends a phase, the model clears the CW and TW and the analyzer can optionally reset any phase-specific statistics. Finally, the framework returns the output state to the detector client.

Our abstract representation of an input allows a wide variety of inputs, such as the methods invoked, basic blocks, branches, addresses loaded, or instructions executed to be considered. This work considers dynamic branch traces. Prior work has shown that such control-flow based profiles can effectively summarize both control- and data-centric execution as well as micro-architectural behavior [32, 20].

In practice, the profile elements may form a hierarchy



(a) (b)

Figure 2: Example of the basic operation of the framework using the Constant TW policy (a) and the Adaptive TW policy (b). Both policies use a *skipFactor* of 1.

```
class PhaseDetector {
  Model model;
  Analyzer analyzer;
  PhaseState state, newState; // initialize to T

  public PhaseState processProfile(profileElements) {
    model.updateWindows(profileElements);
    similarityValue = model.computeSimilarity();
    newState = analyzer.processValue(similarityValue);
    if (state.isTrans() && newState.isPhase()){
      // start phase
      model.anchorTrailingWindow();
      analyzer.resetStats();
    } else if (state.isPhase() && newState.isTrans()) {
      // end phase
      model.clearWindows();
    } else if (state.isPhase()) {  // in phase
      analyzer.updateStats(similarityValue);
    }
    state = newState;
    return state;
  }
}
```

Figure 3: Online Phase Detection Framework

of phases [19], such as what one might expect from a nested-loop structure. Ideally, an online phase detector will find this hierarchy so that the detector's client can exploit it. However, because extant online clients currently do not make use of this phase hierarchy, we present phase detectors that produce flat (not nested) phase structures.

Three orthogonal design choices must be made to instantiate the framework into a concrete online phase detection algorithm. The choices are the window policy, the model policy, and the analyzer policy.

Window Policy The window policy specifies the *skipFactor*, window sizes, and how to manage the TW. The value of *skipFactor* impacts both the overhead of the algorithm and its sensitivity to changes in the profile. A smaller *skipFactor* results in more frequent similarity computations. These comparisons may increase overhead, but result in a more accurate detector.

The size of the CW impacts the granularity at which the algorithm detects phases. A phase that is smaller than the CW may not be detected.

The window policy also dictates the behavior of the TW. Many previous methodologies partition a profile into fixed intervals and then compute the similarity between intervals. In addition to modeling this approach online, i.e., TW size = CW size and computing the similarity between adjacent intervals, we also consider a novel adaptive alternative (that we describe above) for which the TW grows to accommodate the current phase once the algorithm detects that the program is in phase. Because a TW contains a representation of profile elements, such as a set that contains only unique, but not necessarily all, elements, we expect the size of the Adaptive TW to be manageable. As we describe for the example in Figure 2, when a phase ends, the model empties the TW and resets it to its original size.

Model Policy The manner in which a phase detection algorithm models the similarity of profile elements impacts both the accuracy and efficiency of a phase detector [10, 32]. We investigate both unweighted set (also called working set) models and weighted set models.

For the unweighted set model, we consider asymmetric weighting, which computes the percentage of elements in the CW that are also in the TW. This model is biased toward the elements in the CW – which may be effective in combination with the Adaptive TW policy that we describe above. For example, if all elements in the CW are present in the TW, regardless of their frequency, a similarity value of 1.0 results. Likewise, if the CW contains

$\{a, b\}$ and the TW contains $\{a, c\}$, a score of 0.5 results regardless of how often a appears in the two windows.

For the weighted set model, we consider symmetric weighting, which treats both sets equally. It first computes the relative weight of each profile element in each set (TW and CW) independently. The relative weight is the percentage of a window for which a particular element accounts. The model then takes the sum of the minimum of the weights for each element in both windows, producing a number between 0 and 1. For example, assume CW contains $\{(a, 5), (b, 3), (c, 2)\}$ and the TW contains $\{(a, 25), (b, 15), (c, 10), (d, 50)\}$, then a accounts for 25% of TW (50% of CW); b accounts for 15% of TW (30% of CW); c accounts for 10% of TW (20% of CW); and d accounts for 50% of TW (0% of CW). By summing the minimum of these values across windows TW and CW, we produce a similarity value of 0.5 ($= .25 + .15 + .10$).

Analyzer Policy Given a similarity value, the analyzer determines whether this value represents sufficient similarity to indicate a $\mathcal{P}$ state. In addition to exploring a wide range of fixed thresholds, as other researchers have done, we explore analyzers that adapt their threshold based on past similarity values in this phase. The *average* analyzer computes a running average of the similarity values for the current phase, and uses a threshold that is a delta below this average. For example, if the running average of the similarity values of the current phase is 0.88 and the delta parameter is 0.02, the analyzer reports a $\mathcal{P}$ state for values of 0.86 or higher.

3 Evaluating Detection Algorithms

This section presents a new methodology to determine the accuracy of online phase detectors. Extant methodologies evaluate accuracy by using a particular phase detector client, such as a feedback-directed optimization, or by using an architecture-specific metric, such as variance in the number of cycles per instruction (CPI). Our methodology computes the accuracy of phase detection algorithms independent of the phase detection client and independent of any architecture-specific information. The methodology consists of two parts: a *baseline solution* (Section 3.1) and an *accuracy scoring metric* (Section 3.2).

3.1 Baseline Solution

Our baseline solution implements an intuitively "correct" solution to phase boundary identification for a particular program's execution, that can be used to compare online phase detectors. The baseline solution is not an online detection algorithm. Instead, it employs a global view of a program's execution trace and makes multiple passes over the trace to identify periods of repetition. The baseline solution identifies periods of the execution as in phase and all other parts as in transition. We use the baseline solution as an oracle to evaluate online phase detection algorithms.

To identify periods of repetition, we consider two source constructs: loops and repeated method invocations, where repeated method invocations are recursive or temporally adjacent sequential invocations. We record the entrance and exit of each repetition construct with a unique identifier.

To determine the duration of a particular period of repetition, we correlate these events with the "time" of the latest dynamic branch, such as the loop was entered after the k^{th} branch occurred. From the profile elements and source constructs, we construct a dynamic call-loop trace that we use to identify phase boundaries. Our approach is similar to the ones described by Lau et al. [18] to find software phase markers, and by Georges et al. [13] to find method-level phases. Their techniques summarize the execution of these repetitive events in a graph that is tied to the program's static structure and that is augmented with dynamic execution-time profile information. In contrast, our approach tracks individual executions of such events in a trace that allows us to distinguish different executions of a loop body as being in phase or not if their execution lengths differ significantly.

Our baseline solution requires a *minimum phase length* (MPL) parameter, which specifies the minimum length that a period of repetition must be before it will be considered a phase. A client would specify the MPL to ensure that the phases identified have sufficient duration to amortize the client's costs. For example, if a client's phase-based optimization requires an approximate cost of 100,000 branches, then employing this action for a phase that is only 50,000 branches long will result in a net loss. Section 4 shows how the MPL value used in our framework impacts accuracy.

All phases identified by our baseline solution are *complete repetitive instances* (CRI's), i.e., a set of profile elements within an entire loop execution (all iterations) or within a recursive execution. We consider a recursive execution to start upon invocation of a method (which the program later invokes recursively) for which there is no other execution instance on the stack. For example, if a program makes the following method calls without returning: `main` → `foo` → `bar` → `foo`, then the root of the recursive execution starts and ends at the invocation and

return, respectively, of the `foo` instance called by `main`. Although it is possible for different iterations of a loop or different recursive executions of a method to have different branch behavior, we assume that these differences are small, and thus, we consider them part of the same phase.

If a CRI is smaller than the client-specified MPL, we attempt to combine the CRI with temporally adjacent CRI's with the same static identifier (e.g., method name or loop number) into a single phase. We do so if the distance (in terms of number of profile elements) between them is one. This enables us to combine perfectly nested loops and temporally adjacent, repeated invocations of the same method into a single phase.

We view nested loops either as one large phase consisting of the outer loop, or as smaller phases represented by executions of one or more nested loops. We employ MPL to decide between these two choices. If the number of profile elements (dynamic branches in our case) in an execution of a nested loop is at least MPL and there is more than one profile element between executions of the nested loop, we consider this execution of the nested loop a phase. If the number of profile elements in an execution of a nested loop is smaller than MPL or there is only one profile element between executions of the nested loop (as in a perfect loop nest), this execution of the nested loop is not viewed as a phase, and we consider the execution of the next outer loop. We repeat this process until the number of profiling elements exceeds MPL. When this occurs, we select the nest as the representative for the phase.

To validate this approach, we collected branch coverage data (percent of branches that are considered part of some "phase") in the baseline solutions. Our empirical study shows that MPL-based selection enables more control over phase size than specifying a loop nest level. For example, using only outer loops to identify phases results in a very small number of large, coarse-grained phases that cannot be readily subdivided.

Each baseline solution identifies the state ($\mathcal{P}$ or $\mathcal{T}$) of each profile element, from which we can extract the phase boundaries that represent the actual repeated execution of the program. We use the extracted phase boundaries to compare and evaluate online phase detection algorithms. We quantify this comparison using the accuracy scoring metric that we describe in the next subsection.

3.2 Accuracy Scoring Metric

To compare the efficacy of a phase detection algorithm against the baseline solution, we introduce a novel accuracy scoring metric that has two components. The first assesses how well the states identified ($\mathcal{P}$ or $\mathcal{T}$) by the detector match those of the baseline solution. We refer to this property as *correlation* in the spirit of the work by Dhodapkar and Smith [9]. We define correlation as $\frac{\texttt{bothInPhase}+\texttt{bothInTransition}}{\texttt{totalEvents}}$, where `bothInPhase` is the total number of profile elements for which both the detector and the baseline solution output $\mathcal{P}$. Similarly, `bothInTransition` is the number of events that the detector and baseline solution both output $\mathcal{T}$. `totalEvents` is the total number of profile elements. This component of the score measures the extent to which the decisions of the detector and the baseline solution correlate.

The second component of the score measures how often the detected phase boundaries match those of the baseline solution, using two values: *sensitivity* and *false positives*. Sensitivity quantifies how often the detector and the baseline solution agree on phase boundaries. It is defined as $\frac{\texttt{numMatchedBoundaries}}{\texttt{numBaselineBoundaries}}$, where `numMatchedBoundaries` is the number of detected phase boundaries that match the baseline solution and `numBaselineBoundaries` is the number of phase boundaries identified by the baseline solution. The false positives value quantifies how often the detector identifies a phase boundary that the baseline solution does not. It is defined as $\frac{\texttt{numUnmatchedBoundaries}}{\texttt{numDetectedBoundaries}}$, where `numUnmatchedBoundaries` is the number of detected phases boundaries not identified by the baseline solution and `numDetectedBoundaries` is the number of phase boundaries identified by the detector.

Phase boundaries identified by the detector and baseline solution match when the following constraints are satisfied. First, the start of the detected phase must occur at, or after, the start and before the end of the identified phase in the baseline solution. Second, the end of the detected phase must occur at, or after, the end of the current phase and before the start of the next phase in the baseline solution. Third, the closest detected boundary to an identified boundary in the baseline solution that satisfies the first two constraints matches the identified boundary.

Correlation, sensitivity, and false positives are combined into a single weighted sum, called `score`, which we define as $\frac{\texttt{Correlation}}{2} + \left(\frac{\texttt{Sensitivity}}{4} + \frac{(1-\texttt{FalsePositives})}{4}\right)$. We weigh Correlation and matching (Sensitivity and FalsePositives) equally and split the matching weight evenly between Sensitivity and FalsePositives. Thus, Correlation accounts for 50%, Sensitivity accounts for 25%, and FalsePositives accounts for 25% of the score.

Scores fall into the range [0, 1] with higher scores signifying more accurate detectors. Achieving a perfect score in the correlation component, and thus, in the over-

Table 1: Benchmark Characteristics
(a)

Benchmark	Dynamic Branches	Loop Executions	Method Invocations	Recursion Roots
_201_compress	62,808,794	3,980,731	2,407,272	0
_202_jess	15,525,021	140,268	1,558,571	5,984
_205_raytrace	5,801,454	82,556	337,133	6,811
_209_db	3,374,648	317,397	13,621	0
_213_javac	2,770,921	200,121	995,992	10,786
_222_mpegaudio	37,099,265	1,906,483	2,831,987	0
_228_jack	5,926,061	593,135	514,923	4,471
Jlex	2,779,996	146,716	199,868	16

(b)

Benchmark	MPL=1k		MPL=5k		MPL=10k		MPL=25k		MPL=50k		MPL=100k	
	# Phases	% in Phase	# Phases	% in Phase	# Phases	% in Phase	# Phases	% in Phase	# Phases	% in Phase	# Phases	% in Phase
_201_compress	46	33.88	20	34.83	20	34.83	20	34.83	20	34.83	6	99.67
_202_jess	3250	91.44	1092	63.43	473	46.32	134	47.64	88	44.04	30	41.79
_205_raytrace	1448	88.34	198	55.80	84	71.38	41	63.08	25	52.75	17	43.37
_209_db	1152	88.84	303	92.25	147	89.43	51	83.66	13	93.82	5	97.26
_213_javac	665	49.60	149	45.49	76	56.69	29	50.11	15	66.21	9	55.29
_222_mpegaudio	7594	46.70	1968	28.12	894	52.85	894	98.13	22	3.20	2	99.75
_228_jack	1778	53.31	324	48.85	100	43.74	30	36.20	18	29.02	4	13.64
Jlex	102	97.10	53	94.74	49	94.40	39	88.61	32	78.76	2	92.85

all score, would require reporting a change in phase state as soon as it occurred in the baseline solution. This may be impossible for an online detector. For example, in our framework the windows must be full for the algorithm to make an evaluation (compare similarity) and to detect a state change. As a result, the algorithms will always detect a phase after it has started. The degree to which an algorithm is late depends on the window size and is reflected in the correlation portion of the score.

4 Analysis

This section presents the empirical evaluation of instantiations of the framework described in Section 2. After briefly describing our methodology, we present a detailed analysis of different dimensions of the framework.

4.1 Empirical Methodology

Our profile is a conditional branch trace of Java programs, which we obtained by modifying Jikes RVM [15, 1] to produce a profile element for each branch executed. Each profile element represents a unique location in the source code as an integer value that encodes a unique method ID, a bytecode offset in the method where the branch is located, and a bit that represents whether the branch was taken. Our framework, however, is not Java or Jikes RVM specific; it consumes profile elements generated by any toolset for profile extraction, e.g., we can also generate such profiles using the Phoenix instrumentation and compilation framework [28] from Microsoft Research.

We derived baseline solution phase structures from a call-loop trace by instrumenting loop and method entries and exits (both normal and exceptional). We record the unique loop or method identifier and the offset in the profile trace at that point. This allows us to correlate baseline and detected phase boundaries.

We evaluate our phase detection algorithms using eight Java benchmarks, seven from the SPECjvm98 [34] benchmark suite, and JLex [6] (a lexical analyzer generator for Java). We currently consider single-threaded applications only, though the framework can be extended to handle multi-threaded applications. We use input size 10 for the SPEC benchmarks and the default input for JLex. We optimize all application and library methods upon first invocation and extend the optimizing compiler of Jikes RVM to add branch, method, and loop tracing instrumentation.

Table 1(a) lists each benchmark and its dynamic execution characteristics. Column 2 gives the number of dynamic branches in a trace. Column 3 gives the number of loops executed. Column 4 gives the number of method invocations; and column 5 is the number of method invocations that are the root of recursion. Both *Loop Executions* and *Recursion Roots* represent the frequency of code structures that can give rise to repetition of program behavior. Although loop executions dominate, we must also consider recursion when identifying phases.

For our baseline solutions, we consider the following MPL values: 1000, 5000, 10000, 25000, 50000, and 100000 (henceforth abbreviated to 1K, 5K, 10K, 25K, 50K, 100K). Table 1(b) provides information about the phases found by the baseline solution for different MPL values. For a given MPL value, the column to the left lists the number of phases found (*# Phases*) and the col-

Table 2: Window size comparison. (a) shows average percent improvement in best score across all framework parameters when we use a CW size smaller or equal to the MPL as compared to a CW larger than MPL for three TW policies: Adaptive, Constant, and Fixed Interval. (b) is the average of best scores across all benchmarks when the size of CW is smaller than, equal to, and half of MPL.

(a)

	Adaptive TW		Constant TW		Fixed Interval	
Benchmark	Smaller	Equal	Smaller	Equal	Smaller	Equal
_201_compress	28.54	19.96	33.21	22.45	42.71	26.34
_202_jess	13.75	9.31	5.98	5.23	-2.88	-7.91
_205_raytrace	-6.25	-1.25	-0.56	5.26	-2.30	-2.30
_209_db	20.18	10.24	20.21	9.19	13.36	6.35
_213_javac	19.76	15.73	21.78	19.71	25.59	15.14
_222_mpegaudio	12.70	22.61	9.25	17.98	28.86	21.44
_228_jack	22.55	17.25	24.80	20.77	22.25	18.50
Jlex	13.75	9.31	8.93	10.09	3.30	1.72
Average	15.62	12.90	15.45	13.83	16.36	9.91

(b)

	Smaller	Equal	1/2 MPL
Adaptive TW	0.652	0.637	0.664
Constant TW	0.648	0.639	0.664
Fixed Interval	0.601	0.570	0.610

umn to the right shows the percentage of profile elements (dynamic branches in this case) that are in phase (*% in Phase*). The number of phases found varies significantly across benchmarks and across MPL values. For example, with an MPL value of 1K, `compress` has only 46 phases whereas `mpegaudio` has 7,594. However, with an MPL value of 100K, `mpegaudio` has only two phases.

The table illustrates the trend that as MPL values increase the number of phases decreases. This is expected, since as the MPL value increases, our baseline solution identifies larger loops (and recursive chains) as phases.

Counter to intuition, the percentage of profile elements in phase does not correlate with the MPL value. This is an artifact of how the baseline solution selects which loop in a loop nest to identify as a phase (Section 3.1). With a small MPL value, an inner loop may be considered a phase while the containing loop is not. When the MPL is increased, the nested loop may no longer be bigger than the new MPL value, but the containing loop will be large enough to be a phase. When the containing loop becomes a phase, all the profile elements of the inner loop and containing loop are now part of the phase, and thus, increase the percentage in phase value compared to just the profile elements from the inner loop.

However, the percentage in phase value can also decrease when the MPL value is increased. For example, consider a simple loop that has sufficient profile elements to satisfy the MPL value, and thus, is identified as a phase. However, if the number of profile elements is not enough for a larger MPL value, none of these profile elements will be consider in phase with this larger MPL value.

We used our framework to instantiate a large number of phase detection algorithms. Given the various combinations of parameterizations possible (*skipFactor*, curent window size, trailing window policy, model policy, and analyzer policy), we generated over 10,000 different algorithms, which we then compared against our baseline solutions. We computed a score for each detector using our accuracy scoring metric from Section 3.2. In the subsections that follow, we summarize and analyze this data in a way that indicates the general trends in accuracy. In particular, we use the data to investigate the various framework parameters discussed in Section 2.

4.2 Window Policy

We first evaluate the impact of the current window (CW) size on detector accuracy. Intuitively, CW size should be related to the MPL parameter that is used by the baseline solution to find the actual phases in a program.

For our phase detection algorithms, we considered CW sizes of 500, 1K, 5K, 10K, 25K, 50K, and 100K. We computed scores for each CW size and MPL value combination, across all other parameters that we considered: skip factor, TW size, and model and analyzer policies. We then extracted the best score across all combinations and evaluated, for each benchmark, the average when the CW size was smaller, equal to, and larger than the MPL value. Table 2(a) shows the results. We present three sets of data (pairs of columns) for each benchmark. The first set is data for detectors that use the Adaptive TW policy and a skip factor of 1.[1] The second set is data for the Constant

[1]The other Adaptive TW policy parameters that we used for this data set and those that follow include an anchor policy of RN (rightmost noisy + 1) and the sliding window resizing policy. We define and support these choices empirically in Section 5.

TW policy and a skip factor of 1. The final set, *Fixed Interval*, is data for a Constant TW policy with a skip factor equal to the CW size. This last policy is the one most commonly used by extant approaches to phase detection, in which the *skipFactor*, TW size, and CW size are all the same value [17, 16, 30, 31, 32, 9, 10, 21, 8].

Each pair of columns under each policy is the percent improvement in score when we use a CW that is smaller than (first column) or equal to (second column) the baseline's MPL, over using a CW size that is larger than the MPL. We cannot compare this data across sets (Adaptive, Constant, and Fixed Interval), because each column is relative to the base case of that set, i.e., the score when CW size is larger than the MPL value. We compare these configurations using other data in the next subsection.

The data shows that, on average, the highest accuracy occurs with detectors that employ a CW size that is smaller than the MPL value. Although using a CW size that is equal to the MPL value also enables higher accuracy than using one that is larger, the improvement is not as great as for a CW size smaller than the MPL. One reason for this is that the detectors employ two windows, the size of which totals at least twice the CW; this total size is similar to MPL.

Table 2(b) shows the average of best scores across all benchmarks and MPL values, for each TW policy (Adaptive, Constant, and Fixed Interval). We show data for a CW size that is smaller than an MPL value (column 2) for a CW size that is equal to an MPL value (column 3), and for a CW size that is 1/2 the MPL value or smaller (column 4).

The scores that result from using a CW size smaller than MPL are similar to those for a CW of 1/2 the MPL. If the CW is 1/2 MPL or smaller, then the size of TW and CW together is at least MPL; thus, the detector is able to accurately identify the same phases as our baseline solution (for that MPL value). The data also shows that a CW smaller than MPL in some cases outperforms a CW of 1/2 the MPL. The reason for this is that the particular CW size that produces the best score for the smaller CW case varies across benchmarks. There is no single CW size smaller than 1/2 MPL that outperforms a CW of 1/2 MPL across all benchmarks on average. We therefore, use 1/2 the MPL as our CW size for the remainder of the paper in an effort to focus our analysis of the remaining dimensions of our algorithms.

We next evaluate the impact of skip factor on detector accuracy for the three TW policies (Adaptive, Constant, and Fixed Interval) and consider all other parameterizations of the model and analyzer policies. We again consider the best score across these configurations.

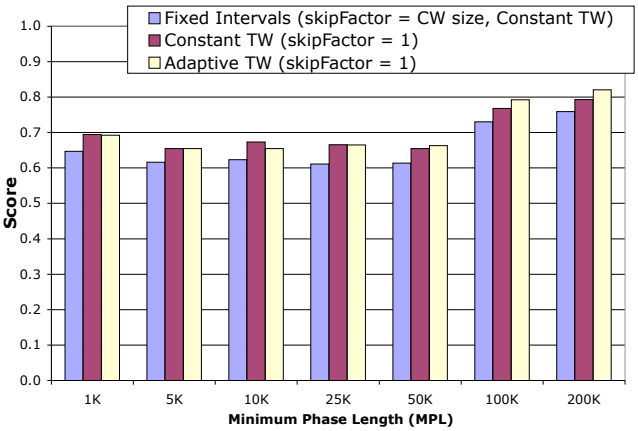


Figure 4: Evaluation of skip factor and Fixed versus Adaptive windowing. The data is the average of best scores across all benchmarks, models, and analyzers. The CW size is less than 1/2 the MPL.

Figure 4 compares the three TW policies. The x-axis is MPL and the y-axis is the average of best scores across all configurations and benchmarks. A higher score is better. We consider two values of skip factor: one and CW size. The former enables high responsiveness by the detector to detect fine-grain changes in phase behavior. We evaluate the accuracy enabled by two different skipFactor values by comparing the Fixed Interval bars (*skipFactor* = CW size) against the remaining two (skipFactor = 1). The data shows that on average, the approach commonly used in existing systems (*skipFactor* = CW size) is significantly less accurate than both the Constant TW and Adaptive TW policies when *skipFactor* is one. Thus, the remaining evaluations use a skipFactor = 1.

When we compare Constant TW and Adaptive TW, the results are less clear. In general, our experiments show that for small MPLs, Constant TW does somewhat better than the Adaptive TW. However, this is not the case for all benchmarks when we consider them individually. For larger MPLs, Adaptive TW is consistently more accurate than a Constant TW. We added MPL 200K to this data set to evaluate whether the trend continues, and it does. For larger MPLs, some of the shorter running benchmarks exhibited a very small number (1 or 2) of very large phases, which were not useful or fair to include in a comparison (all detectors achieve very high scores since there are so few phases to match against).

The remainder of the paper presents results for MPL values of size 1K, 10K, 50K, and 100K. We continue to include data for both Constant TW and the Adaptive TW policies in our subsequent comparisons.

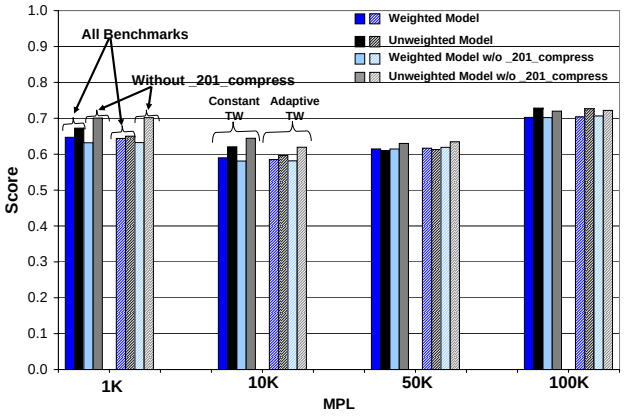


Figure 5: The average of best score across all benchmarks for two models. There are two sets of bars per MPL group for the Constant TW and Adaptive TW policies. For each policy, weighted and unweighted model scores are shown with and without the compress benchmark.

4.3 Model Policy

Figures 5 presents an empirical comparison between two models described in Section 2: weighted and unweighted. The x-axis shows MPL values and the y-axis shows the average of best score across all benchmarks. For each MPL, there are two groups, each with four bars. The first group is for the Constant TW policy and the second group is for the Adaptive TW policy. Within each group, there are two pairs of bars. In each pair, the left bar is the weighted model results and the right bar is the unweighted model results.

The first pair of bars in each group shows the average of best score across all benchmarks. These results show that the unweighted model is more accurate than the weighted model in all but the 50K MPL case. When we consider the individual benchmark data, however, unweighted is significantly more accurate in a majority of cases for all benchmarks except one: _201_compress (compress hereafter). For this benchmark, the detectors that employ the weighted model are almost 50% better in many cases (we omit this data due to space constraints).

To show the average accuracy of detectors without considering the compress benchmark, we include a second pair of bars in each group of four in the graph. This pair shows the average of best scores for detectors that employ the weighted and unweighted models, respectively, on average across all of our benchmarks, except compress. From this data, we can conclude that, in general, the unweighted model is more accurate than weighted model for all MPLs and trailing window policies. As a result, we consider only the unweighted model for our analysis of similarity analyzers in the next section.

4.4 Analyzer Policy

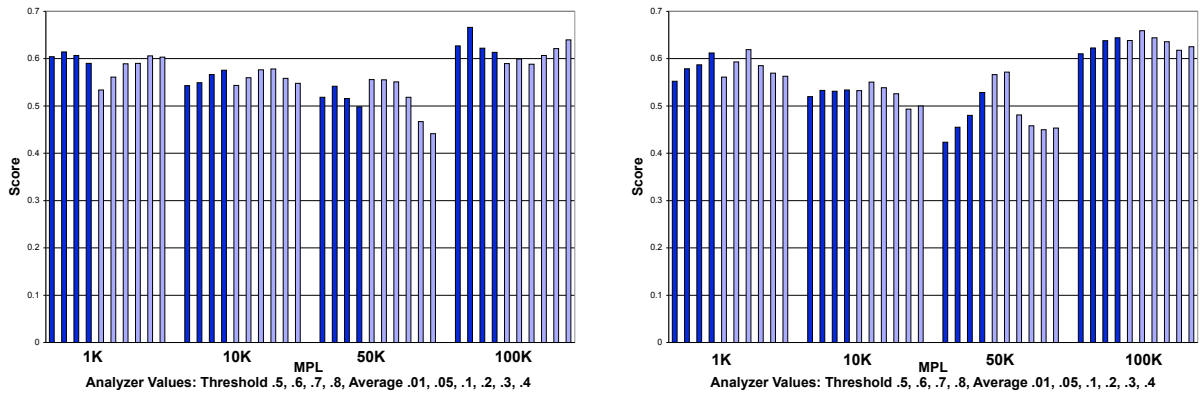
Figure 6 shows a comparison between two categories of analyzers: Threshold and Average, each with different parameters. The figure contains two subgraphs. The left graph (a) presents the data for the Constant TW policy and the right graph (b) presents the data for the Adaptive TW policy. In each graph, the x-axis presents MPL values and the y-axis presents the average of best scores across all benchmarks. For each MPL, there are ten bars. Within the ten bars, the first four bars, which are darker, represent the Threshold analyzers with values of 0.5, 0.6, 0.7, and 0.8; and the last six bars represent the Average analyzers with values 0.01, 0.05, 0.1, 0.2, 0.3 and 0.4.

The data presents mixed results. Neither the Threshold nor the Average analyzers are clear winners for all MPL values and all benchmarks. However, if one were to pick a particular analyzer, certain values seem to be a better choice for a specific trailing window policy. In particular, if the Threshold analyzer is chosen, a threshold value of 0.6 wins in three out of four of the MPL values for the Constant TW policy, whereas the threshold value of 0.8 wins in three out of four of the MPL values for the Adaptive TW policy. If the Average analyzer is chosen, there is not a clear trend for the Constant TW policy; however, a value of 0.05 wins for three out of four of the MPL values for the Adaptive TW policy. A more comprehensive analysis of the data is required to better understand these trends.

5 Additional Analysis

This section analyzes other parameters of our phase detection framework. The first parameter specifies how window resizing and anchoring is performed when an algorithm using the Adaptive TW policy detects the start of a phase. This parameter impacts the detection of phase-start boundaries, and therefore, can produce a more accurate representation of the phase. It is also important for an Adaptive TW policy because it serves as a signature of the entire phase.

Before discussing this parameter fully, we discuss other properties that can also impact the accurate identification of phase start boundaries. First, as mentioned in Section 3, an online algorithm will have a delay in profile elements before it can detect the beginning of a phase. Second, phase boundaries may not always align with skipFactor values. Third, phases often exhibit startup periods where



(a) Constant TW

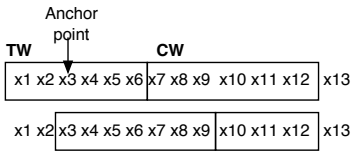
(b) Adaptive TW

Figure 6: The average of best scores across all benchmarks for the Constant TW (a) and the Adaptive TW (b). Each chart is grouped into four sets of bars, one for each MPL value. Each MPL category has ten analyzers corresponding to (from left to right) the four Threshold analyzers (darker bars) with increasing threshold (0.5, 0.6, 0.7, 0.8) and the six Average analyzers (lighter bars) with increasing deltas (0.01, 0.05, 0.1, 0.2, 0.3, 0.4).

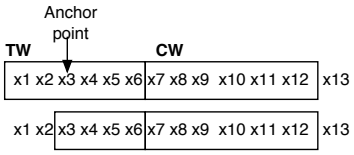
the behavior is less stable (but is not considered a transition) than the steady state of a phase [25].

The anchor point is the position in the TW at which a new phase starts. We explore two options to determine where to place the anchor point. The first option places the anchor point one element to the right of the rightmost noisy element in the window (RN). Noisy elements are those that are in the TW and not in the CW. The second option places the anchor point at the leftmost non-noisy element (LNN). Both techniques attempt to eliminate instability during the start of a phase to enable more accurate detection of $\mathcal{T}$ or $\mathcal{P}$ states thereafter. RN is more aggressive at doing so. For example, if the TW contains elements a, b, and, c and the CW contains a, a, and c, then b is a noisy element. The RN policy selects the position of c in the TW and the LNN policy selects the position of a in the TW, as the start of the phase. Both policies attempt to eliminate profile elements that are part of the warm-up period [25] of the phase that may not be as stable as the steady state of the phase.

Once we identify the starting position of the new phase, we have two options for window resizing. We can *slide* the TW right, so that the left boundary of the TW is at the anchor point, thus reducing the size of the CW:



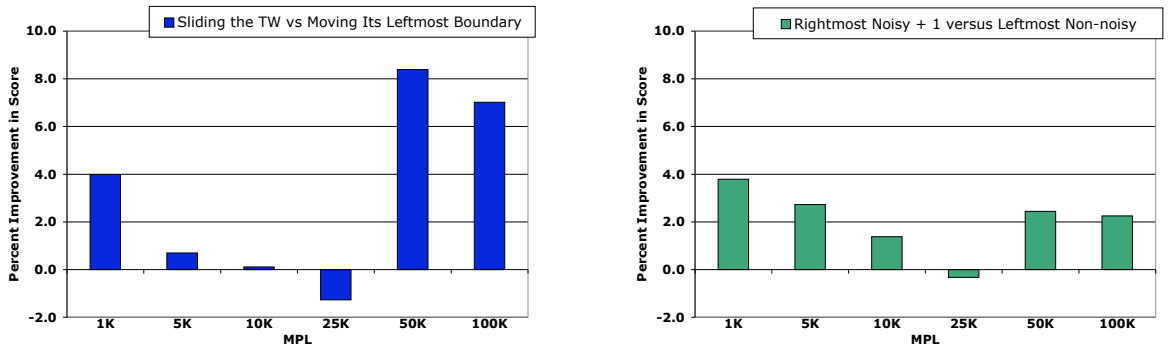
Alternatively, we can *move* the left boundary of the TW to the right and leave the CW unaffected:



By sliding, we reduce the size of the CW; however, we continue to compare the two windows for similarity while the CW fills in this case. This enables the TW to hold as much of the phase as possible (our original goal with the trailing window policy). By moving the TW, we shrink its size as opposed to the CW.

Figure 7 evaluates these two policies across benchmarks for each of the MPLs (x-axis). Graph (a) shows percent improvement in score for Sliding over Moving of the TW (we use the RN anchoring strategy here). Graph (b) shows the percent improvement in score due to the use of RN over LNN to select an anchor point (for the Sliding resizing policy). On average, Sliding is more accurate than Moving and as such, is a better resizing policy. It also seems intuitively correct for an Adaptive TW to include most of the recently detected phase before evaluating subsequent profile elements. In addition, RN is more accurate than LNN, on average. We use the Sliding and RN policies for the results in the previous sections and below.

Our last set of results compares the best scores for the Adaptive and Constant TW policies using a modified technique for finding the beginning of a phase. As discussed previously, an online algorithm detects a phase after some initial part of the phase has been seen. However, once a phase is detected, such an algorithm can identify where the phase began using the anchoring policy discussed



(a) (b)

Figure 7: Percent improvement in score for Slide over Move resizing, using the RN anchoring strategy (a). Percent improvement in score for RN over LNN anchoring, using the Sliding resizing policy, (b).

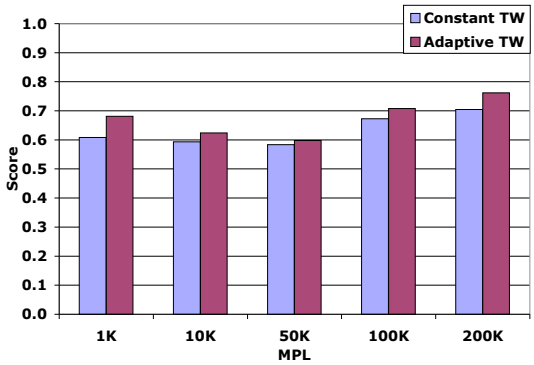


Figure 8: Average of best scores across all benchmarks, models, and analyzers for the Constant and Adpative TW policies using the anchoring policy for detecting the beginning of a phase.

above. This information can be used to accurately identify phase signatures [18, 29] and their repetition online. Figure 8 compare these new phase boundaries against those of the baseline solution. The results indicate that for every MPL, the Adaptive TW policy is significantly more accurate than the Constant TW policy, showing promise for its use in online detection of recurring phases.

6 Related Work

The framework and algorithms that we describe herein enable detection of phase behavior. This is in contrast to the majority of related work that focuses on prediction of future phase behavior [32, 12, 25, 23, 29, 13, 18] and on characterization of periods of program execution into phases after they execute or as part of a prior run of the program [19, 29, 13, 32]. Our algorithms execute concurrently with the program and detect phase behavior after a phase begins by computing and evaluating the similarity between two windows of profile elements. Moreover, our algorithms are online; they detect phases while they occur. Our algorithms do not distinguish phases for future use (i.e., identify temporally disjoint, repeating phases), although we are investigating such extensions. In addition, we introduce a baseline solution and scoring metric that enable us to compare the phase boundaries detected by any detection algorithm against the natural, dynamic, repetition of program structures.

Dhodapkar and Smith [9] study online phase detection in the context of multi-configuration hardware (e.g., reconfigurable instruction caches). The authors describe algorithms for detecting changes in working sets, identifying recurring working sets, and estimating the number of elements in a working set. They employ an unweighted set model, a fixed window size of $100,000$ instructions, and a *skipFactor* that is equal to the window size. They define their similarity threshold value empirically to be 0.5 to remove most noise and to detect only significant phase changes. The authors report that their experiments indicate that detection of phase changes is relatively insensitive to the threshold value. The algorithm in [9] can be viewed as one instantiation of our framework. We use our framework to investigate the efficacy of this detection algorithm and to compare it to a wide range of other algorithms that employ alternative parameterizations of the similarity and threshold components. Our results indicate that a *skipFactor* that is equal to the window size is significantly less accurate than a *skipFactor* of one. In addition, we show that an adaptive windowing strategy is more accurate than a fixed one for large MPLs (Section 4.2) and for capturing phase boundaries (Section 5).

Lu et al. [21] employ online phase detection to drive

data cache prefetching in a dynamic binary optimization system. Specifically, they compare the average PC address from the most recent 4K samples to a range of values, which is created from the average and standard deviation of the previous seven 4K samples of the PC address. If the new average is sufficiently outside this interval for two consecutive 4K sample windows, a phase has ended. This algorithm can also be modeled in our framework, where the model computes the averages of the 4K samples, and the analyzer does the interval bound test. Das et al. [8] build on this work by advocating a local phase detection for events in each region of the program. This work uses Pearson's co-efficient of correlation between the current set of samples and the target set for the region. They compare this value to a fixed threshold.

Phase prediction systems must first observe and characterize program behavior prior to forecasting the recurrence of future phases. These prior works perform these tasks during an initial run of the program [29, 18, 24, 31] and wait until an interval of execution completes before reporting whether the interval is in phase [32, 17, 16]. Our baseline solution is similar to these extant approaches in that it exploits a global view of the execution trace. Our baseline solution, however, is novel in that it provides a methodology for comparing and evaluating phase detection algorithms. It captures all common, source-level, looping constructs that execute and combines them into phases according to the minimum phase length provided by an optimization client (phase profile consumer).

In addition, we use our framework to evaluate the efficacy of delaying detection until interval completion by using a *skipFactor* equal to the window size (Subsection 4.2). We show that by using a *skipFactor* of one, detection algorithms are more sensitive and responsive to changes within an interval and are thus more accurate in capturing the phase behavior of the program. These results confirm the findings of Lau et al. [19], who show that fixed-size interval boundaries can miss phases due to misalignment. In this prior work, the authors employ variable-length intervals to guide selection of program simulation points. However, their technique requires a prior run of the program; i.e., unlike our system, it is not online.

7 Conclusions and Future Work

This paper presents a novel framework for developing a wide range of online phase detection algorithms. We also describe a novel client- and machine-independent methodology for evaluating the accuracy of these algorithms, and perform a detailed empirical study of numerous algorithms using this methodology. Our conclusions are that the current window size should be smaller than the desired minimum phase length and that a skip factor of 1 has better accuracy than the standard practice of setting the skip factor to the current window size. We also find that an adaptive trailing window policy can be more accurate than a constant trailing window policy. Finally, our results for models tend to favor the unweighted model, although the results for analyzers are mixed.

In addition to investigating further other algorithms for phase detection, we plan to explore three primary directions in future work. First, we will extend our framework to instantiate algorithms that detect phases that repeat themselves. Such an enhancement would allow a dynamic optimization system to record the efficacy of a phase-based optimization at the end of the phase and determine whether to employ the same optimization when the phase reoccurs. Second, we plan to investigate and optimize the overhead of accurate phase detection. There are three sources of overhead in a phase-aware optimization system: profile collection, phase detection, and phase consumption and use by the client. Finally, we plan to investigate phase-aware dynamic optimizations and how they are impacted by phase detector accuracy and overhead. As part of this research, we will identify how to set the MPL for a particular client and whether it is effective to adapt the MPL over time.

Acknowledgments

We thank Matt Arnold, Brad Calder, Jeremy Lau, Martin Hirzel, Laureen Treacy, and the anonymous reviewers for feedback on this work. This research was funded in part by NSF grant Nos. ST-HEC-0444412, ITR/CCF-0205712, and CNF-0423336, and by DARPA contract No. NBCH30390004.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, Feb. 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, Oct. 2000. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[3] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices*, 37(11):111–129, Nov. 2002. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, May 2000. In *Conference on Programming Language Design and Implementation (PLDI)*.

[5] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. mei W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *the 35th International Symposium on Microarchitecture*, pages 233–244, Nov. 2002.

[6] E. Berk. Jlex: A lexical analyzer generator for Java. www.cs.princeton.edu/ appel/modern/java/JLex.

[7] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. *ACM SIGPLAN Notices*, 37(5):199–209, May 2002. In *Conference on Programming Language Design and Implementation (PLDI)*.

[8] A. Das, J. Lu, and W.-C. Hsu. Region monitoring for local phase detection in dynamic optimization systems. In *ACM Conference on Code Generation and Optimization*, Mar. 2006.

[9] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, pages 233–244, May 2002.

[10] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *the 36th International Symposium on Microarchitecture*, pages 217–227, Dec. 2003.

[11] P. C. Diniz and M. C. Rinard. Dynamic feedback: An effective technique for adaptive computing. *ACM SIGPLAN Notices*, 32(5):71–84, May 1997. In *Conference on Programming Language Design and Implementation (PLDI)*.

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

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

[14] M. Hind, V. Rajan, and P. F. Sweeney. Phase detection: A problem classification. Technical Report 22887, IBM Research, Aug. 2003.

[15] Jikes Research Virtual Machine (RVM). http://jikesrvm.sourceforge.net.

[16] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[17] T. P. Kistler. *Continuous Program Optimization*. PhD thesis, University of California, Irvine, 1999.

[18] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *ACM Conference on Code Generation and Optimization*, Mar. 2006.

[19] 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*, Mar. 2005.

[20] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2005.

[21] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweighted dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6, 2004.

[22] A. Madison and A. P. Batson. Characteristics of program localities. *Communications of the ACM*, 19(5):285–294, May 1976.

[23] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *International Symposium on Computer Architecture*, June 1999.

[24] P. Nagpurkar and C. Krintz. Visualization and analysis of phased behavior in Java programs. In *ACM Principles and Practices of Programming in Java*, June 2004.

[25] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *ACM Conference on Code Generation and Optimization*, Mar. 2005.

[26] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM)*, pages 1–12, Apr. 2001.

[27] C. Pereira, J. Lau, B. Calder, and R. Gupta. Dynamic phase analysis for cycle-close trace generation. In *International Conference on Hardware/Software Codesign and System Synthesis*, Sept. 2005.

[28] The phoenix framework from microsoft research. http://research.microsoft.com/Phoenix/technical.aspx.

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

[30] 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*, Sept. 2001.

[31] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[32] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, pages 336–349, June 2003.

[33] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 36(11):180–195, Nov. 2001. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[34] The Standard Performance Evaluation Corporation. SPEC JVM 1998. http://www.spec.org/osg/jvm98, 2000.