RC23058 (W0312-105) December 22, 2003
Computer Science

IBM Research Report

The Phase Shift Detection Problem is Non-Monotonic

Michael Hind, V. T. Rajan, Peter F. Sweeney
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

The Phase Shift Detection Problem is Non-Monotonic

Michael Hind V.T. Rajan Peter F. Sweeney

IBM T.J. Watson Research Center, Hawthorne, NY, 10532
{hindm,vtrajan,pfs}@us.ibm.com

ABSTRACT

Object-oriented languages have enabled the creation of large commercial applications, where high performance is critical. To achieve high performance, dynamic optimization, which is performed at execution time, must be continuously tailored to the application's changing runtime behavior. One important technology to enable this continuous optimization is *phase shift detection*, which allows a dynamic optimization system to react appropriately to improve the system's performance.

However, there has been limited success in exploiting phase shift detection in virtual machines. To help evaluate phase detection algorithms, we attempted to find the *canonical* phase structure of a program's profile. Starting with a simple phase shift detection algorithm specified by three fundamental parameters, we demonstrate with examples and profile data that two of the three parameters are *non-monotonic* with respect to the phase shift detection algorithm's decisions. This result implies that to determine the "best" value for a non-monotonic parameters may require an exhaustive search of all possible values. Furthermore, once the "best" value for a parameter is found for a particular profile, tuning the parameter's value for another profile is no easier than attempting to find the value from scratch.

This fundamental result is important for dynamic optimization systems because it implies that the choice of values for a phase shift detection algorithm's parameters can have a dramatic impact on the quality of the information that is produced, and thus the choices should be carefully studied.

1. INTRODUCTION

By making it easy to create reusable components, object-oriented languages have played an enabling role in the creation of large commercial, web-based applications. A company's business often depends on the availability and performance of these applications, therefore, optimizing the application's behavior is important. Many of these applications are written in dynamic languages, such as Java or C#, that have features that preclude traditional static optimizations. However, the rich runtime environment provided by these languages allows optimizations to be performed at runtime that are tailored to the application's dynamic behavior [10, 6, 16, 11, 9, 2, 23, 26, 19, 30, 3, 27, 14, 15, 8].

Consider an application server for a stock brokerage's web site. One can expect the application server to have different dynamic behavior in the morning than in the evening, on a Monday than on a Friday, and on a weekday than on a weekend. Although there are general trends in application's workload because of different transaction mixes and volume, the behavior of the system on any given day, at any given time cannot be predicted from the general trend because of market factors. Therefore, to achieve peek performance, such systems require continuous optimization [20, 19].

One important technology to enable continuous optimization is phase shift detection; that is, to determine when the application is executing in a phase or is transitioning between phases. Knowing the state of the system allows the runtime to react appropriately to improve the system's performance. For example, Kistler's [18] dynamic optimization system for Oberon automatically performs data-layout of objects based on online profile information and relies on an online phase detection mechanism to trigger this optimization. In addition to performing other forms of specialization when a phase has been entered, some actions could take advantage of the detection of the end of a phase. For example, the runtime system could try to improve the efficiency of garbage collection by optimistically schedule garbage collection [32, 31] at the end of a phase with the hope that many objects would be dead at this time, and thus, improve both the performance and efficiency of the garbage collector. However, heavily optimizing an application when it is in a phase transition may not be a good use of resources, because the program's changing dynamic behavior may render the optimizations obsolete.

Despite the promise, there has been limited success in exploiting phase shift detection in a runtime environment, such as a virtual machine. Motivating by our own mixed success using phase detection to optimistically schedule garbage collection and adaptively tune the aggressiveness of a dynamic optimization system, we decided to look closely at the phase detection problem. To help evaluate different phase detection algorithms, we attempted to find the *canonical* phase structure of a program's profile, such as executed methods, basic blocks, or instructions. The efficacy of a phase detection algorithm could then be determined by comparing it to the canonical phase structure. This paper summarizes our findings. Starting with a simple phase shift detection algorithm specified by three fundamental parameters, we show that two of the parameters are non-monotonic with respect to the phase shift detection algorithm's decisions; that is, as these parameters' values change, the phase shift detection algorithm's decisions are non-monotonic.

This result has an important consequence: when a parameter is non-monotonic, determining the "best" value for that parameter may require an exhaustive search of all possible values; that is, running the phase shift detection algorithm with every possible value for that parameter. Furthermore, once the "best" value for a parameter is found for a particular profile, tuning the parameter's value for another profile is no easier than attempting to find the value from scratch.

Furthermore, this fundamental result is important for dynamic optimization systems because it implies that the choice of values for a phase shift detection algorithm's parameters can have a dramatic impact on the quality of the information that is produced, and thus the choices should be carefully studied.

The remainder of this paper is organized as follows. Section 2 demonstrates that an intuitive notion of a phase is not well-defined, but instead requires the specification of additional parameters. Section 3 presents a simple phase detection algorithm that requires three parameters and the profile of a program's execution. Section 4 illustrates with simple examples that two of the parameters are not monotonic with respect to the phase shift detection algorithm's decisions. Section 5 demonstrates the non-monotonic property with real profile data, a conditional branch trace of Java benchmarks. Section 6 discusses other issues related to phase detection problems. Section 7 summarizes related work. Section 8 discusses conclusions and future work.

2. BACKGROUND

Intuitively, a *phase* occurs when the program's execution behavior is stable, and a *phase transition* occurs when the program's execution behavior changes. We model a program's execution behavior as a string of values (or execution events), denoted as a *profile.* Examples of such events could be executed instructions, basic blocks, or methods, as well as addresses of load instructions or the values loaded. Given a definition of a phase, the general phase shift detection problem, $\mathcal{PSD}(\mathcal{P})$, can be stated as

> Given a string of values, $\mathcal{P}$, produce a set of nonoverlapping substrings of $\mathcal{P}$ such that each substring represents a maximal phase.

To avoid trivial phases, where each value is a phase, it is desirable to have the phase length be maximal; that is, a phase is the largest substring such that it constitutes a phase. There may be regions of $\mathcal{P}$ that are not in any phase and thus, are in a phase transition.

At first glance, solutions to $\mathcal{PSD}$ with an intuitive definition of a phase seem straightforward. For example, if the profile is

$$\mathbf{aaaaaaabbbbbbb} \tag{1}$$

where each symbol, **a** or **b**, represents an execution event, then two phases can be identified: all **a**'s, and all **b**'s. However, the solution is less clear with the following profile:

$$\mathbf{aaaaaabbbaaaaaa} \tag{2}$$

Is there only one phase that includes all the elements, or a phase consisting of **a**'s that repeats and one phase consisting of **b**'s, or are the **b**'s a phase transition? The solution is even less clear when considering the following profile:

$$\mathbf{aababcaabab} \tag{3}$$

In this profile, is **c** a separate phase, a phase transition, or simply an outlying value that should be ignored for purposes of detecting phases? How **c** is interpreted determines the phases in string 3.

The above examples illustrate that a precise definition of a phase is required, and there are two inherent aspects of a phase definition:

- the atomic units of comparison, or *granularity*, and
- how to compute the *similarity* of two strings.

A trivial definition of granularity is to let the size of the unit of comparison, which we will call the *chunk size*, be one; that is, the values of single elements are pair-wise compared, either the values are the same or they are not. With this definition, a phase is the largest substring that contain the same values. For example, if the string is **aabccc**, then there would be three phases: **aa**, **b**, and **ccc**. Alternatively, a larger chunk size would result in computing the *similarity value* of two strings, such that the value ranges from 0.0 to 1.0, where 0.0 represents no similarity, and 1.0 represents perfect similarity between the two strings. The two strings are *similar* if their similarity value is at least some *threshold.*

Let's consider the similarity computation. When the size of the unit of comparison is greater than one, the similarity value of substrings must be computed. An important aspect of this computation is the manner in which the two strings are *modeled.* For example, should order within the strings be considered? If not, should the frequency of string elements (weight) be considered? For example, what is the similarity value for **aaaaab** and **abbbbb**? If the strings are modeled as unweighted sets, both strings are modeled as the set {**a**, **b**} and they have perfect similarity. If the strings are modeled as weighted sets, then string **aaaaab** is modeled as $\{\langle \mathbf{a}, \mathbf{5}\rangle, \langle \mathbf{b}, \mathbf{1}\rangle\}$, where **a** has a weight of five and **b** has a weight of one, and string **abbbbb** is modeled as $\{\langle \mathbf{a}, \mathbf{1}\rangle, \langle \mathbf{b}, \mathbf{5}\rangle\}$, where **a** has a weight of one and **b** has a weight of five. The weighted set similarity value of the two strings is 0.33 because only two out of six elements are the same.

Once chunk size and a model have been specified, we take any two strings in the profile and compute a similarity value between 0.0 and 1.0. The final step for determining if these two strings are in the same phase is to compare this value to a *threshold*, which states how similar two strings must be to be considered in the same phase.

Our discussion reveals three fundamental parameters to a simple phase shift detection algorithm that precisely define a phase:

- size of substrings to compare, *chunk size*,
- how similarity between substrings is computed, *model*, and
- a *threshold* to determine if the similarity between the substrings is sufficient.

Given values for chunk size, model, and threshold a phase can be defined as follows:

> A string $\mathcal{S}$ is a *phase* if the length of $\mathcal{S}$ is at least $2 * \textit{chunk size}$, and the similarity of two consecutive chunks contained in $\mathcal{S}$ are greater than or equal to $\mathcal{T}$.

In the following section we present a simple algorithm that takes as input these three fundamental parameters and a profile, and computes the phases in the profile. The purpose of the algorithm is to study the properties of these

```
/* This algorithm assumes $S_1$ is the first element in the profile

$\mathcal{PSD}$ ($\mathcal{P}$, $\mathcal{CS}$, $\mathcal{M}$, $\mathcal{T}$) {
1:  $inPhase = false$            // state of algorithm
2:  $numPhases = 0$              // number of phases
3:  $numValues = \mathcal{CS} + \mathcal{CS}$        // place in profile
4:  $phaseStart = -1$            // beginning of a phase, set below
5:  while ($numValues < |\mathcal{P}|$) {
6:      $\mathcal{S}^1 = \mathcal{P}_{numValues-CS-CS+1} \cdots \mathcal{P}_{numValues-CS}$
7:      $\mathcal{S}^2 = \mathcal{P}_{numValues-CS+1} \cdots \mathcal{P}_{numValues}$
8:      $simValue = similarity\ [\mathcal{M}]\ (\mathcal{S}^1, \mathcal{S}^2)$
9:      if ( $simValue >= \mathcal{T}$ && !$inPhase$ ) {      // start of phase
10:             $inPhase = true$
11:             $phaseStart = numValues - \mathcal{CS} - \mathcal{CS} + 1$
12:             $numPhases = numPhases + 1$
13:     } else if ( $simValue < \mathcal{T}$ && $inPhase$ ) {  // end of phase
14:             $inPhase = false$
15:             $phaseBoundary[numPhases - 1] =$
                    $\langle phaseStart, numValues - \mathcal{CS}\rangle$
16:     }
17:     $numValues = numValues + \mathcal{CS}$
18: }
19: if ( $inPhase$ ) {
20:     $phaseBoundary[numPhases - 1] =$
            $\langle phaseStart, numValues - \mathcal{CS}\rangle$
21: }
22: return $phaseBoundary$
23: }
```

Figure 1: A simple phase shift detection algorithm.

```
$Similarity[weighted]$ ($\mathcal{S}^1$, $\mathcal{S}^2$) {
   $similarity = 0$
   $\mathcal{W}^1 = weightedSet\ (\mathcal{S}^1)$
   $\mathcal{W}^2 = weightedSet\ (\mathcal{S}^2)$
   forall $\langle v, w_2\rangle \in \mathcal{W}^2$ {
      if ( $\langle v, w_1\rangle \in \mathcal{W}^1$ ) {
         $similarity += min(w_1, w_2)$
      }
   }
   return $similarity\ /\ |\mathcal{S}^2|$
}

$Similarity[unweighted]$ ($\mathcal{S}^1$, $\mathcal{S}^2$) {
   return $|(unweightedSet(\mathcal{S}^1) \cap unweightedSet(\mathcal{S}^2))|$ /
      $(|unweightedSet(\mathcal{S}^2)|)$
}
```

Figure 2: Algorithms for computing the similarity value for a *weighted* and *unweighted* models.

parameters, not to propose a new algorithm. Many phase shift detection algorithms have been proposed in the literature that choose one value for each of these parameters [**?**, 24, 19]. Therefore, results we obtain are applicable to these algorithms as well.

3. SIMPLE ALGORITHM

This section presents a simple phase shift detection algorithm that takes as input a profile $\mathcal{P}$ and three parameters:

Chunk Size ($\mathcal{CS}$) is an integer greater than one that specifies how a profile is partitioned into fixed length atomic units of comparison, denoted chunks.

Model ($\mathcal{M}$) is a function that takes two strings, $\mathcal{S}^1$ and $\mathcal{S}^2$, as input, converts each string into an abstract representation, such as a set, weighted set, etc., and computes a *similarity value* between the abstract representations such that the value ranges from 0.0, if there is no similarity, to 1.0, if there is perfect similarity. For purposes of this paper, we consider two models: *weighted*, a weighted set representation; and *unweighted*, an unweighted set representations.

Threshold ($\mathcal{T}$) is a constant ranging from 0.0 to 1.0. Two string are similar if their similarity value is at least $\mathcal{T}$; that is, $\mathcal{M}(\mathcal{S}^1, \mathcal{S}^2) \geq \mathcal{T}$.

The profile input to the algorithm, $\mathcal{P}$, represents the behavior of a program's execution as a string of values. The algorithm outputs a sequence of pairs of integers that identifies the phase structure. Each pair denotes the start and end points of a phase in $\mathcal{P}$. The sequence specify nonoverlapping substrings of $\mathcal{P}$. The largest substrings in the profile that do not overlap any phases are the phase transitions.

Fig. 1 presents our simple phase shift detection algorithm. The $inPhase$ boolean flag tracks the state of the algorithm: $inPhase$ is $true$ when the algorithm has recognized a phase; otherwise, the algorithm is in a phase transition. The variable $numPhases$ identifies the number of phases identified, while the variable $phaseStart$ is used to remember the start point of a phase. The variable $numValues$ identifies the number of elements processed, including the current iteration, as well as the index of the last profile element currently being processed.

The while loop partitions $\mathcal{P}$ into nonoverlapping consecutive chunks of size $\mathcal{CS}$ starting from the leftmost element in $\mathcal{P}$. $\mathcal{S}^1$ and $\mathcal{S}^2$ are the current candidate chunks. The method $similarity$ is called with $\mathcal{S}^1$ and $\mathcal{S}^2$ to compute their similarity value that is stored in the local variable $simValue$. If the candidate chunks are similar, $simValue >= \mathcal{T}$, and $inPhase$ is $false$, then a new phase has begun. If the candidate chunks are not similar, $simValue < \mathcal{T}$, and $inPhase$ is $true$, then the current phase has ended, the start and end of phase is recorded in the $phaseBoundary$ array, and $inPhase$ is set to $false$. After all the values in $\mathcal{P}$ have been examined, if $inPhase$ is $true$ the last phase is added to the $phaseBoundaries$ array.

Fig. 2 illustrates how the *weighted* and *unweighted* models compute the similarity value of two strings. For the *weighted* model, the sum of the minimum weight of each element in the weighted sets is divided by the size of the second string, which is the same as the first string. If an element is not in a set, its weight is zero. For the *unweighted* model, the number of elements that are in both sets is divided by the number of elements in the second set. These definitions of similarity are similar to those used by others [**?**, 24, 19]. Section 7 provides more details.

4. SIMPLE EXAMPLES

This section provides simple examples that illustrate the monotonic property of the phase shift detection algorithm's parameters. For each parameter, we determine if the parameter is monotonic or non-monotonic with respect to the phase shift detection algorithm's decisions. Monotonicity is an important property, because it provides the insight into how the phase shift detection algorithm will respond to parameter value changes.

We demarcate the start and end of a phase using "[" and "]". For example,

[**aaaa**] **bc** [**dddd**]

specifies a *phase structure* for the string **aaaabcdddd** such that there are two phases, **aaaa** and **dddd**, and one phase transition, **bc**.

4.1 Threshold

This section demonstrates how the threshold parameter is monotonic with respect to the phase shift detection algorithm's decisions. Assume that the parameter values are $\mathcal{CS} = 3$, $\mathcal{M} = weighted$, and threshold may vary: $\mathcal{T} = 0.6$ or $\mathcal{T} = 0.3$. Consider the following example string that has six chunks:[1]

aab abb bcc bcb ddd dee (4)

When $\mathcal{T} = 0.6$, the phase structure for string (4) is

[**aab abb**] [**bcc bcb**] **ddd dee**

The first phase, **aababb**, is a result of the similarity value of chunks **aab** and **abb** being 0.66 (> 0.6). The second phase, **bccbcb**, is a result of the similarity value of chunks **bcc** and **bcb** being 0.66 (> 0.6). A phase transition containing no chunks occurs between the two phases because the similarity value of chunks **abb** and **bcc** is only 0.33 (< 0.6). The second phase transition is **ddddee** because the similarity between **bcb** and **ddd** is 0.0 and the similarity between **ddd** and **dee** is 0.33 (< 0.6) , both of which are less than 0.6.

When $\mathcal{T}$ is 0.3, the phase structure for string (4) is

[**aab abb bcc bcb**] [**ddd dee**]

The first phase is a combination of two phases that occurred when $\mathcal{T} = 0.6$ because the phase transition that occurred between those two phases is eliminated. In particular, the similarity value of chunks **abb** and **bcc** is 0.33, which is above the threshold when $\mathcal{T} = 0.3$, but below the threshold when $\mathcal{T} = 0.6$. The second phase is a combination of two chunks that were in a phase transition when $\mathcal{T} = 0.6$ because the similarity value of chunk **abb** and chunk **bcc** is 0.33, which is above $\mathcal{T}$ when $\mathcal{T} = 0.3$.

This example demonstrates that the threshold parameter is monotonic with respect to the phase shift detection algorithm's decisions: as threshold's value decreases, a boundary between two chunks may either stay in the same state or change from *inTransition* to *inPhase* and as a threshold's value increases, a boundary between two chunks may either stay in the same state or change from *inPhase* to *inTransition*.

Even though threshold is monotonic with respect to the phase shift detection algorithm's decisions, it may not be monotonic with other aspects of the profile's phase structure. For example, threshold is not monotonic with respect to the number of phases detected: when the threshold is decreased, the number of phases identified might either increase, decrease, or stay the same. The number of phases can decrease because adjacent phases are combined, or increase because phase transitions become phases. Section 5.2 provides more insight into this phenomena.

[1]When appropriate, how a string is divided into chunks is illustrated by placing spaces between chunk boundaries.

4.2 Model

This section illustrates how the choice of a model affects the identification of phases. Assume that the parameter values are $\mathcal{CS} = 6$, $\mathcal{T} = 0.6$, and we vary the model between $\mathcal{M} = unweighted$ or $\mathcal{M} = weighted$. Consider the following example string:

abbbbb aaaaab aaaaac accccc (5)

When $\mathcal{M} = unweighted$, the phase structure for string (5) is

[**abbbbb aaaaab**] [**aaaaac accccc**]

Two phases and one phase transition are identified. The phase transition occurs between chunk **aaaaab**, which has an unweighted set representation of $\{a, b\}$, and chunk **aaaaac**, which has an unweighted set representation of $\{a, c\}$, because their unweighted set representations have a similarity value (see Fig. 2) of 0.5 (< 0.6). The first phase is **abbbb-baaaaab** because the unweighted set representation of the chunks **abbbbb** and **aaaaab** are the same, $\{a, b\}$, and results in a perfect similarity value. Similar reasoning explains the second phase is **aaaaacaccccc**.

When $\mathcal{M} = weighted$, the phase structure for string (5) is

abbbbb [**aaaaab aaaaac**] **accccc**

One phase and two phase transitions are identified. The phase is **aaaaabaaaaac** because the similarity value for the weighted model (Fig. 2) of **aaaaab** and **aaaaac** is 0.83 (> 0.6). The first phase transition exists because the similarity value of **abbbbb** and **aaaaab** is 0.33 (< 0.6). The second phase transition exists because the similarity value of **accccc** and **aaaaac** is 0.33 (< 0.6).

This example demonstrates that the model parameter is not monotonic with respect to the phase shift detection algorithm's decisions: a boundary between two chunks may change from *inPhase* to *inTransition* and from *inTransition* to *inPhase* when the model is changed from *unweighted* to *weighted*.

4.3 Chunk Size

This section demonstrates the that chunk size parameter is non-monotonic with respect to the phase shift detection algorithm's decisions. Assume that the parameter values are $\mathcal{M} = weighted$ and $\mathcal{T} = 0.6$, while chunk size may vary: $\mathcal{CS} = 3$, $\mathcal{CS} = 6$, or $\mathcal{CS} = 12$. Consider the following string:

aaa bbb bbb aaa aaa ccc ccc acc (6)

When $\mathcal{CS} = 3$, the phase structure for string (6) is

aaa [**bbb bbb**] [**aaa aaa**] [**ccc ccc acc**]

Three phases and three phase transitions are identified. Because the similarity value of a string and itself is perfect, the chunks **aaa**, **bbb**, **ccc** are perfectly similar with themselves; however, each of these chunks has no similarity with one of the other chunks. Finally, the similarity value of chunk **ccc** and chunk **acc** is 0.66 (> 0.6). One might argue that $\mathcal{CS} = 3$ is optimal for string (6) because, other than the last chunk, every chunk contains elements that have the same value.

When $\mathcal{CS} = 6$, the phase structure for string (6) is

[**aaabbb bbbaaa**] [**aaaccc cccacc**]

Two phases and one phase transition are identified. The phase transition is identified between chunk **bbbaaa** and chunk **aaaccc** because their similarity value is 0.5, which is below $\mathcal{T}$. The first phase is **aaabbbbbbaaa** because the similarity value of the chunks **aaabbb** and **bbbaaa** is perfect. The second phase is **aaaccccccacc** because the similarity value of the chunks **aaaccc** and **cccacc** is 0.66, which is above $\mathcal{T}$. Surprisingly, a phase transition is identified between a string of six **a**'s that were identified as a phase when $\mathcal{CS} = 3$

This example demonstrates that the chunk size parameter is non-monotonic with respect to the phase shift detection algorithm's decisions: a boundary between two chunks may change from *inPhase* to $inTransition$ or from $inTransition$ to *inPhase* when the chunk size is increased.

When $\mathcal{CS} = 12$, the phase structure for string (6) is

aaabbbbbbaaa aaaccccccacc

No phases are identified because the similarity value of chunk **aaabbbbbbaaa** and chunk **aaaccccccacc** is 0.33, which is below $\mathcal{T}$. $\mathcal{CS} = 12$ demonstrates that a large chunk size may hide phase transitions and hide recurring phases.

5. EMPIRICAL DATA

The section presents the application of the phase shift detection algorithm, presented in Section 3, on conditional branch profiles of Java benchmarks to illustrate the monotonic property of the algorithm's parameters. Section 4 demonstrated, at least for contrived examples, the monotonic properties of the algorithm's parameters. This section demonstrates the monotonic properties also occur in the profile of real programs.

5.1 Experimental Methodology

We gather a profile representing an exhaustive conditional branch trace of the seven Java applications from the SPECjvm98 benchmark suite [29] and the SPECjbb2000 [28] benchmark. We used the size 10 inputs for SPECjvm98. For SPECjbb2000 we ran with a 5-second ramp up followed by a 10-second measurement period. The trace was collected during the measurement period. We generated the profile by modifying the Jikes RVM [1, 17] baseline compiler to trace conditional branches and table jumping (lookupswitch and tableswitch) bytecodes. This conditional branch trace models the bytecode semantics of an application. Each traced conditional branch is encoded as a 32-bit word that consists of a numeric representation of the method in the first 16 bits, the bytecode offset in the next 15 bits and whether a branch was taken or not as the last bit. Each encoded word is written to a trace file.

We use a similarity graph to help illustrate our point in the subsequent subsections. A similarity graph's vertical axis is the range of similarity values and the horizontal axis represents time where each conditional branch is a clock tick. Each point in the graph is the similarity value for two consecutive chunks. The higher the value in the vertical axis, the more similar the two consecutive chunks. Each line segment between two points represents a chunk. The first chunk, that is, the first line segment, is not included in the graphs. The similarity values in the graph are the values assigned to the $simValue$ variable at line 8 of the algorithm given in Fig. 1.

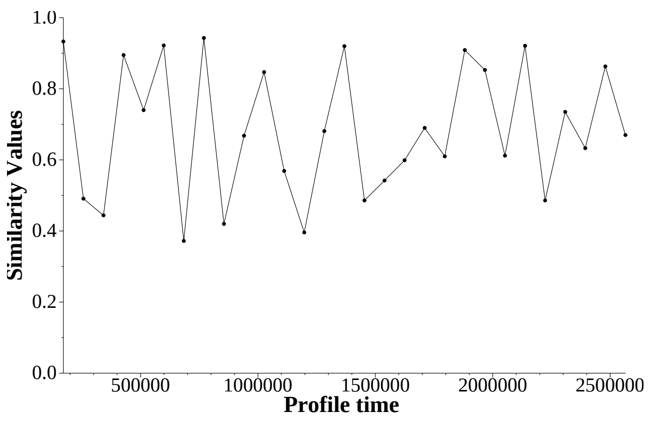
5.2 Threshold

This section demonstrates that the threshold parameter is monotonic with respect to the phase shift detection algorithm's decisions when the profile is from a real program's execution. Fig. 3(a) presents the similarity graph that is computed from a conditional branch trace for the SPECjvm98 `javac` benchmark. For this example, the model and chunk size parameter values are fixed: model is $\mathcal{M} = weighted$, and chunk size is $\mathcal{CS} = 85,504$, chosen to partition the profile into 30 chunks. For a given threshold $\mathcal{T}$, the algorithm detects that the program's execution is in a phase when the line is above $\mathcal{T}$ and is in a phase transition when the line is below $\mathcal{T}$. Because the line segment that drops below $\mathcal{T}$ represents the end of a phase and the line segment that rises above $\mathcal{T}$ represents the start of a phase, the length of a phase is the number of consecutive points above $\mathcal{T}$ plus one and the length of a phase transition is the number of consecutive points below $\mathcal{T}$ minus one.

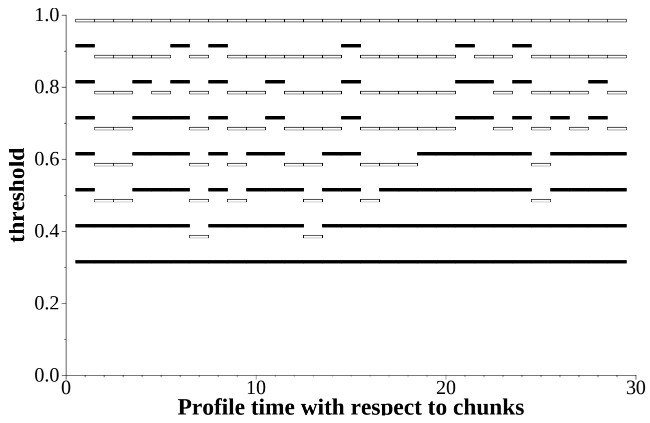
Fig. 3(b) illustrates the decisions that are generated by the simple phase shift detection algorithm, presented in Fig. 1, for different values of the threshold parameter, but the same `javac` profile and parameter values for chunk size and model that were used by Fig. 3(a). The y-axis specifies threshold values, and the x-axis is profile time. Each decision is identified in the figure by a rectangle: black rectangles indicate that the program is in a phase ($inPhase == true$), and white rectangles indicate that the program is in a transition ($inPhase == false$). The black rectangles are raised slightly and the white rectangle lowered to help the eye pick out the pattern. Each decision is the value of the *inPhase* variable at the end of the while loop (line 18) in the phase shift detection algorithm presented in Fig. 1. When the threshold is high, $\mathcal{T} = 1.0$, no phases as detected; resulting in the horizontal tiled white line across the top. When the threshold is low, $\mathcal{T} = 0.3$ for this example, all the chunks are in one phase, resulting in the horizontal black line at $\mathcal{T} = 0.3$. As the threshold is varied between these two extremes, the number of phases varies; however, for every point, there is only one transition between states going from *inPhase* to $inTransition$ as the threshold is increased. Thus, the threshold parameter is monotonic with respect to the phase shift detection algorithm's decisions: when the threshold increases, chunks that were in a phase can become in a transition; and when the threshold decreases, chunks that are in a transition can become in a phase.

In addition, Fig. 3(b) illustrates that the threshold parameter is not monotonic with respect to the number of phases: as the threshold value increases, the number of phases can increase because multiple phase transitions form a new phase, or the number of phases can decrease because a multiple phases form a single phase. For example, when $\mathcal{T} = 0.8$ there are nine phases; however, when $\mathcal{T} = 0.9$ there are six phases, and when $\mathcal{T} = 0.6$ there are seven phases.

Finally, because threshold is monotonic with respect to the phase shift detection algorithm decisions, picking a reasonable value for $\mathcal{T}$ when exploring the other parameters is appropriate. In particular, in the next two sections we choose the average of the similarity value to be the threshold when the model, profile, and chunk size parameter values are fixed.



(a)



(b)

Figure 3: Similarity graph computed from a conditional branch trace of `javac` and phases detected for different threshold values.

5.3 Model

This section demonstrates that the model parameter is non-monotonic with respect to the phase shift detection algorithm when the profile is from a real program's execution. Fig. 4 shows the similarity graphs for the benchmark suite. Each graph shows the similarity values for both the *unweighted* (solid line) and *weighted* (dashed line) models for a particular benchmark. The chunk size is fixed for each benchmark so that 30 intervals were created, and the chunk size is a multiple of 128 to enable the varying chunk size experiments described in Section 5.4. Fig. 3 is replicated in Fig. 4 as the unweighted model's similarity graph in `javac`. Because each point in the graph represents the similarity of two consecutive intervals, each graph contain 29 points. The second and third columns of Table 1 give the number of values in each profile, and the chunk size used to produce 30 intervals.

Although the similarity graphs for all of the benchmarks, except `jbb`, illustrate significant variance between the two models, we use phase correlation [?] to quantify the differences.[2] In particular, we fix the profile input and chunk size and model parameter values. The threshold for a particular similarity graph is the graph's average similarity value. The last two columns of Table 1 provide the thresholds that are used. The phase correlation for a benchmark between one model's similarity graph and threshold and the other model's similarity graph and threshold is the percentage of profile points that the phase shift detection algorithm makes the same decision.

Fig. 5 presents the phase correlation between the *weighted* and *unweighted* models for each of the benchmarks. Only two of the benchmarks (`db` and `mtrt`) have a phase correlation above 82%. Five of the other six of the benchmarks have a phase correlation between 72 and 76%, and `jbb` has a phase correlation below 50%. The high phase correlations for `db` and `mtrt` are expected because the benchmark's similarity graphs for each model follow the same general trend of moving up and down.

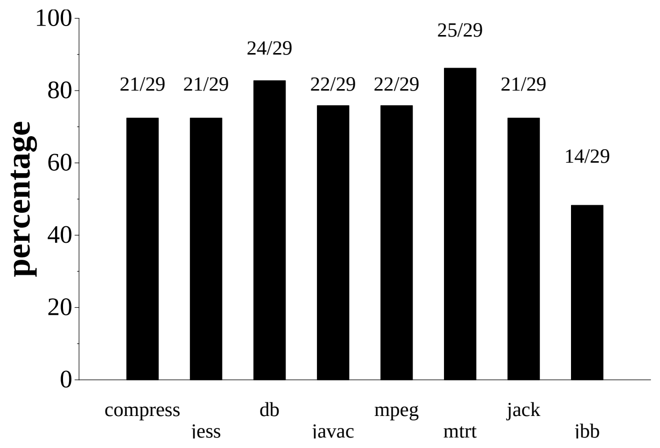


Figure 5: Phase correlation between models. Chunk size is chosen so that 30 chunks are created. Threshold is chosen using the average similarity value. Thus, the height of each bar is the percentage of comparison points (out of 29) where the models agree on the phase shift detection algorithm's decision.

[2]Although Dhodapkar and Smith [?] use the term "correlation", we prefer the term "phase correlation" to prevent any confusion with the mathematical concept of correlation.

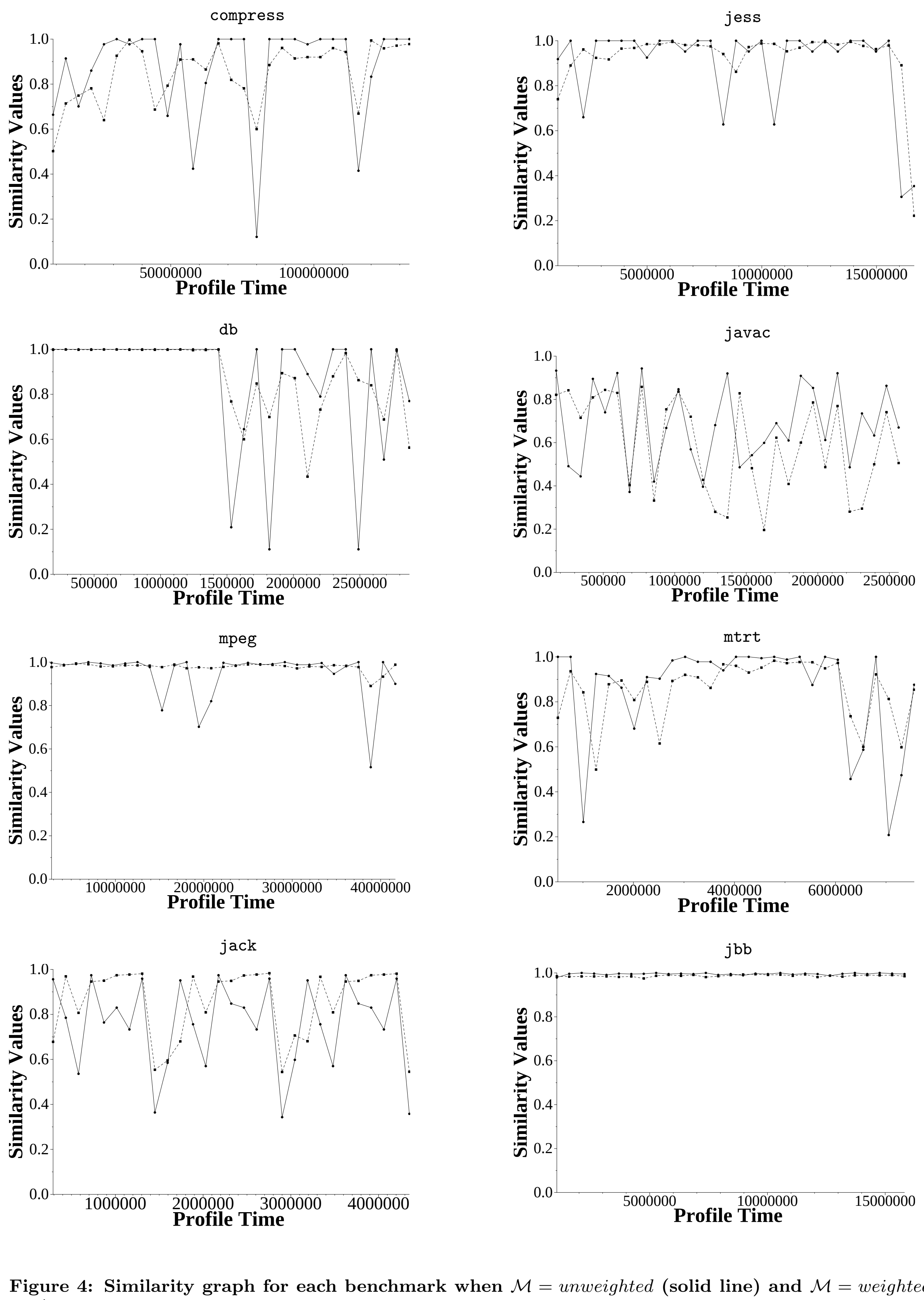


Figure 4: Similarity graph for each benchmark when $\mathcal{M} = unweighted$ (solid line) and $\mathcal{M} = weighted$ (dashed line) and a $\mathcal{CS}$ is chosen for each benchmark so that 30 intervals are created.

Table 1: Profile Characteristics.

Benchmark	Values in Profile	Chunk Size Used	Threshold	
			weighted	unweighted
`compress`	133,359,828	4,445,184	0.8508	0.8726
`jess`	16,643,781	554,752	0.9288	0.9027
`db`	2,876,618	95,744	0.8840	0.8633
`javac`	2,568,401	85,504	0.6041	0.6845
`mpeg`	41,663,308	1,388,672	0.9773	0.9477
`mtrt`	7,559,094	251,904	0.8564	0.8548
`jack`	4,351,129	145,024	0.8550	0.7606
`jbb`	15,821,020	527,360	0.9868	0.9952

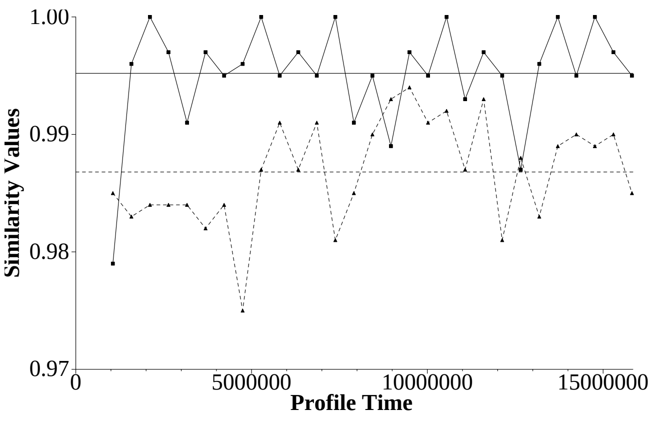


Figure 6: Similarity graph for `jbb` from Fig. 4 when y-axis is expanded, $\mathcal{M} = unweighted$ (solid line), $\mathcal{M} = weighted$ (dashed line), and chunk size is $527,360$. The thresholds are computed as the average of each model's similarity values.

`jbb`'s low phase correlation is surprising because its similarity graphs for the two models are indistinguishable in Fig. 4. Fig. 6 illustrates why `jbb` has poor phase correlation: the y-axis is expanded and the corresponding model specific thresholds are drawn as straight horizontal lines. A point on the x-axis is phase correlated if the corresponding points in each similarity graph are either both above or both below their respective thresholds. For example, the first point in both similarity graphs are phase correlated because they are both below their respective thresholds, however, the second point is not phase correlated because it is out of phase in the *weighted* model and in phase in the *unweighted* model.

5.4 Chunk Size

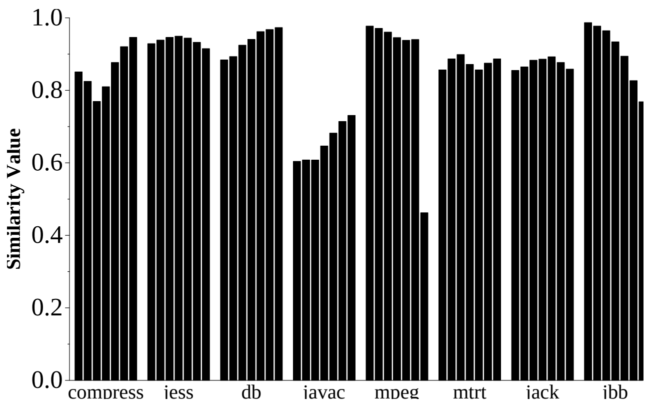
This section demonstrates that the chunk size parameter is non-monotonic with respect to the phase shift detection algorithm's decisions when the profile is from a real program's execution. We study this property for a particular profile and both models. As described in the Section 5.2, choosing an appropriate threshold can significantly impact the number of phases that are detected. For this study, we chose a threshold of the average similarity values for each profile, chunk size, and model. Fig. 7 presents the average similarity values for the *weighted* and *unweighted* model, respectively. The initial chunk size, first bar for each benchmark, was chosen for each benchmark so that it results in 30 intervals, as was done in Section 5.2. The number of intervals is doubled repeatedly from 30 to 1920, resulting in the chunk size being divided by two as bars go to the right.

Interestingly, the average similarity values are relatively similar (within 22%) for each benchmark other than `mpeg`, which drops radically for the smallest chunk size studied. Nevertheless, the average similarity values do vary, sometimes increasing and sometimes decreasing. Using one threshold value for all chunk sizes for a given profile and model would bias the phase shift detection algorithm for some chunk sizes over others. Therefore, we chose the threshold value as the average similarity value for a given chunk size and profile.

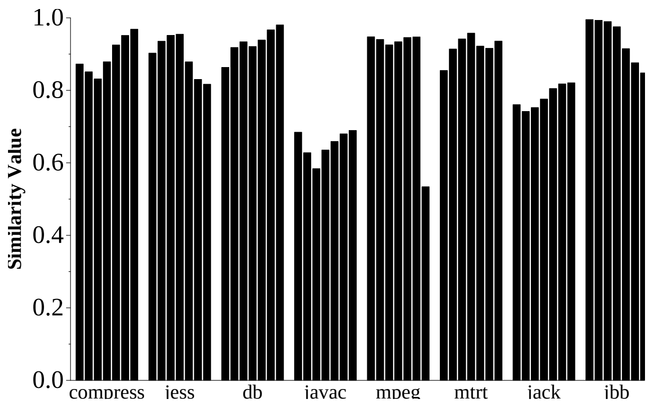
Given a profile, and values for the model and threshold parameters, we specify how different chunk sizes can be phase correlated: different chunk sizes have a different number of decision points. Previously, phase correlation has only been defined when chunk size is fixed [?]. In particular, when the difference in chunk sizes is a factor of two, the smaller chunk size will have twice as many decision points. Half of the decision points, denoted aligned points, coincide with the decision points of the larger chunk size and the other half, denoted intermediate points, do not coincide. For intermediate points we use the decision of the greatest point that is less than the current point. For example, in a profile with 100 events, comparing chunk sizes 5 and 10 will have aligned points at 10, 20, etc., and intermediate points at 5, 15, 25, etc. For each such point $10i + 5$, we use the decision at point $10i$.

The bar charts in Fig. 8 illustrate the phase correlation of the adjacent chunk sizes that we studied in Fig. 7 for each benchmark and model. Each benchmark bar represents a phase correlation between chunk sizes that differ by a factor of two. There are seven chunk sizes and six bars. The first bar compares the largest chunk size studied with the second largest chunk size, which is half the larger chunk size. For most of the benchmarks and models, the phase correlation between adjacent chunk sizes is less than 20%. Two exceptions are `compress`, where there is a sharp 22% decrease from the first and second comparisons in the *weighted* model, and `mpeg`, where the second to last comparison is a local maximum followed by a sharp decline for both models. We suspect that as a chunk size gets smaller the phase correlation between chunk sizes that differ by a factor of two will be less because the chunk sizes will become smaller than the natural size of a phase, and that eventually every benchmark will have a tail if the chunk size gets small enough.

Although the phase correlation between adjacent chunks

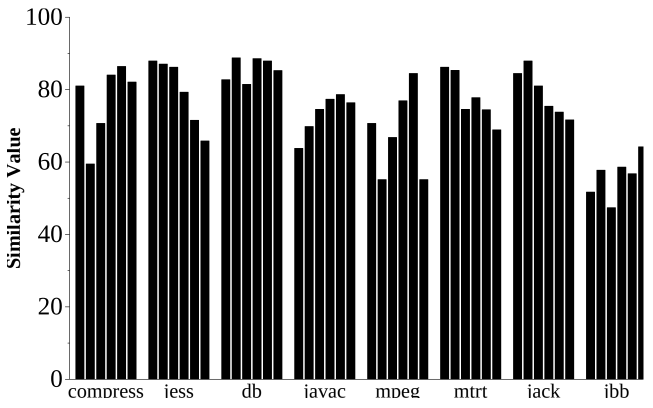


weighted

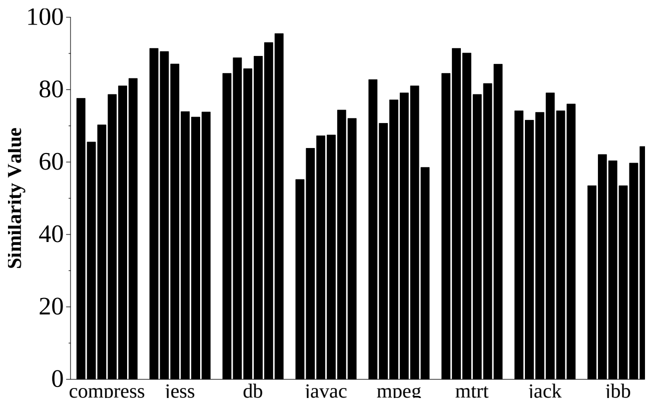


unweighted

Figure 7: The average similarity values for the *weighted* and *unweighted* models, where number of intervals varies from 30, 60, 120, 240, 480, 960, 1920, i.e., within each benchmark the chunk size is reduced by half going from left to right.



weighted



unweighted

Figure 8: Phase correlation of adjacent chunk sizes for weighted and unweighted models.

appears reasonable, usually differing by less than 20%, we now determine if the chunk sizes converge on a canonical phase structure. Before exploring the real data, we use the example data in table below to explain this concept.

Decision Points	Chunk Size			
	120	60	30	15
a	P	P	P	P
a+15	P	P	P	T
a+30	P	P	T	T
a+45	P	P	T	P

This table shows four phase shift detection algorithm's decisions (a, $a+15$, $a+30$, $a+45$) for the four different chunk sizes (120, 60, 30, 15). (A real profile will have many more decision points.) For purposes of this discussion, we assume that a is a multiple of 120, and $a+k$ is a profile execution event that occurs k events after a. For each decision point and chunk size, the decision is recorded in the table, where "P" signifying in a phase, and "T" signifying in a phase transition. The row at decision point a, all chunk sizes agree that the profile is a phase. The row at decision point $a+15$ and the row at decision point $a+30$ each illustrate that there is disagreement, but reading from left to right, only one decision change (from P to T) occurs. However, the row at decision point $a+45$ illustrates that two transitions occur, and thus at this decision point, chunk size is non-monotonic with respect to the phase shift detection algorithm's decisions.

Fig. 9 illustrates that the chunk size parameter is non-monotonic with respect to phase shift detection algorithm's decisions for our benchmark data. The y-axis is the percentage and the x-axis is number of times the decision changes between chunk sizes. The first two sets of bars, buckets 0 and 1, in Fig. 9 indicate how often the chunk size parameter is monotonic with the phase shift detection algorithm's decisions. The first set of bars, bucket 0, represents that the phase shift detection algorithm's decision is consistent: all chunk sizes agree on the same phase shift detection algorithm's decision. The second set of bars, bucket 1, represents that the decision changes only once across all chunk sizes. The third through seventh set of bars, buckets 2 through 6, demonstrate that the chunk size parameter is not monotonic with respect to the phase shift detection algorithm's decisions. For example, the third set of bars, bucket 2, represents when the phase shift detection algorithm's decision change twice across all the chunk sizes: either going from *inPhase* to *inTransition* to *inPhase* again, or from *intransition* to *inPhase* to *intransition* again. Surprisingly, all of the benchmarks have some decisions points that flip flop between all the chunk sizes. The non-monotonic behavior makes tuning the chunk size particularly difficult.

Fig. 10 illustrates in more detail the phase shift algorithm's decision-making process for `db` when $\mathcal{M}$ is *unweighted* and for `jbb` when $\mathcal{M}$ is *weighted* using the same format as Fig. 3(b) in Section 5.2. We picked these two benchmarks and models because they represent the extremes of the set of benchmarks. On the one hand, over 64% of `db`'s decision, the third bar in each bucket, are consistent and over 82% are monotonic across all decision points. On the other hand, less than 3% of `jbb`'s decisions, the last bar in each bucket, are consistent and less than 15% of its decisions are monotonic across all decision points.

Looking at Fig. 10(a), it is evident that the first half of the execution of `db` is monotonic because all the decision points are *inPhase*. However, Fig. 10(b) illustrates that for `jbb` there are few decision points that are monotonic.

Furthermore, these two graphs illustrate that looking only at the phase correlation between adjacent chunk sizes is misleading because one could draw the conclusion that the distance between the chunk sizes may not matter.

6. DISCUSSION

This paper explores the monotonic properties of the phase detection problem's parameters. We chose a simple algorithm for phase detection to illustrate our point. A more sophisticated algorithm might have more parameters, but because this difficulty arises even in a simple algorithm means that it would likely persist in a more complex algorithm. In many cases, the complex algorithm can be reduced to an instance of the simpler algorithm by a particular choice of extra parameters.

One might argue that the choice of a fixed chunk size is too restrictive, that the chunk size should be allowed to vary according to the local properties of the profile under consideration. However, determining the appropriate choice of chunk size becomes a problem similar to the phase detection problem. One would like the chunk size within a phase to be uniform and representative of the phase being considered. But that implies that we have either identified the phase, or the appropriate choice of chunk size requires additional parameters to specify some criteria. Our work suggests that some of the additional parameters would be non-monotonic with respect to the chunk size detection algorithm's decisions.

Many practical applications of phase shift detection, such as in a dynamic optimization systems, require an online algorithm; that is, the algorithm cannot look into the future of the profile and will be unable to store a complete history of the profile. The simple algorithm we present is an online algorithm, provided the three parameters are specified. However, choosing the threshold value as the average of the similarity values for the whole profile is clearly not possible in an online environment. One could vary the threshold based on the local similarity values, such as the current average. This would yield a more sophisticated algorithm, but would have the same difficulty, namely, the results may be sensitive to the criteria used to vary the threshold. However, as we saw, the results of phase shift detection is monotonic with the variation in threshold. So it may be possible to vary the threshold in some systematic way and get phases which are more useful and meaningful.

There is a common perception that phases are likely to recur. For example, the stock brokerage web site may have a particular behavior on most mornings or during peak activity periods. If a phase detection algorithm can detect such recurrences, it can use the previous phase behavior to more accurately predict program behavior. Further, speculative optimizations can be further tailored based on the effectiveness of such optimizations performed during the previous occurrence of the phase. Although we did not supply an algorithm for detecting phase recurrence, such an algorithm would share the same underlying monotonic properties of parameters discussed in this paper.

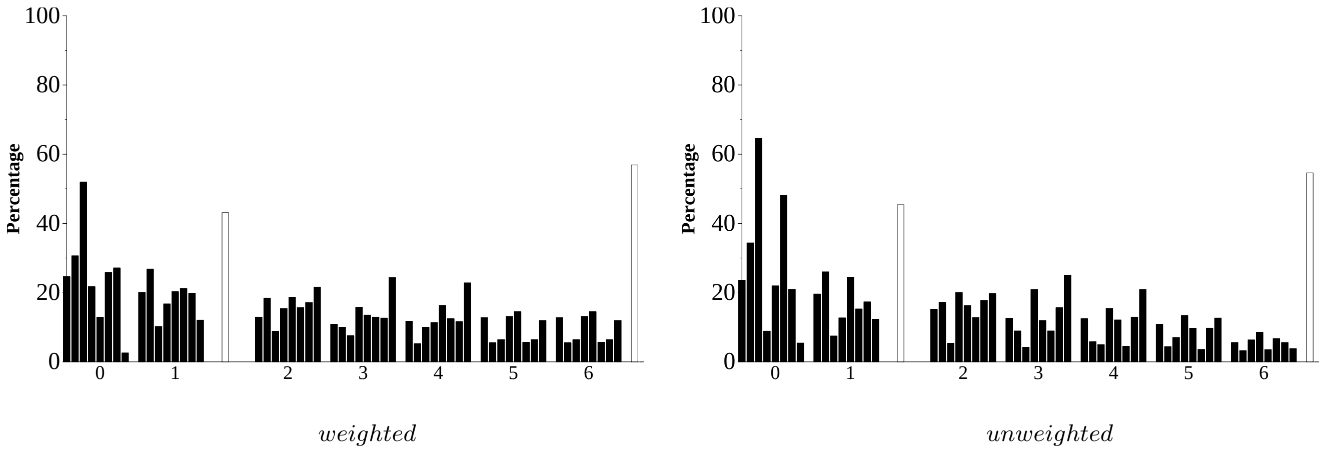


Figure 9: Histogram of monotonicity of phase shift detection algorithm's decisions across seven different chunk sizes. The x-axis of each graph specifies the number of decision changes when chunk size is varied. For each bucket, 0..6, we give the percentage of profile points that are monotonic with respect to phase shift detection algorithm's decisions for each benchmark in the order listed in Table 1: `compress`, `jess`, `db`, `javac`, `mpeg`, `mtrt`, `jack`, `jbb`. For example, the eighth bar in bucket 0 (the smallest one) signifies that in only 2.6% for *weighted* and 5.4% for *unweighted* of the interval comparisons for `jbb` do all chunk sizes give the same phase shift detection algorithm decisions; that is, when all chunk sizes agree that the phase shift detection algorithm's decision is either a phase or a transition. The two white bars give the average number, across all benchmarks, of points from buckets 0 and 1, and greater than 1, respectively. The first white bar shows how often the decisions are monotonic, 43% for *weighted* model and 45% for *unweighted* model.

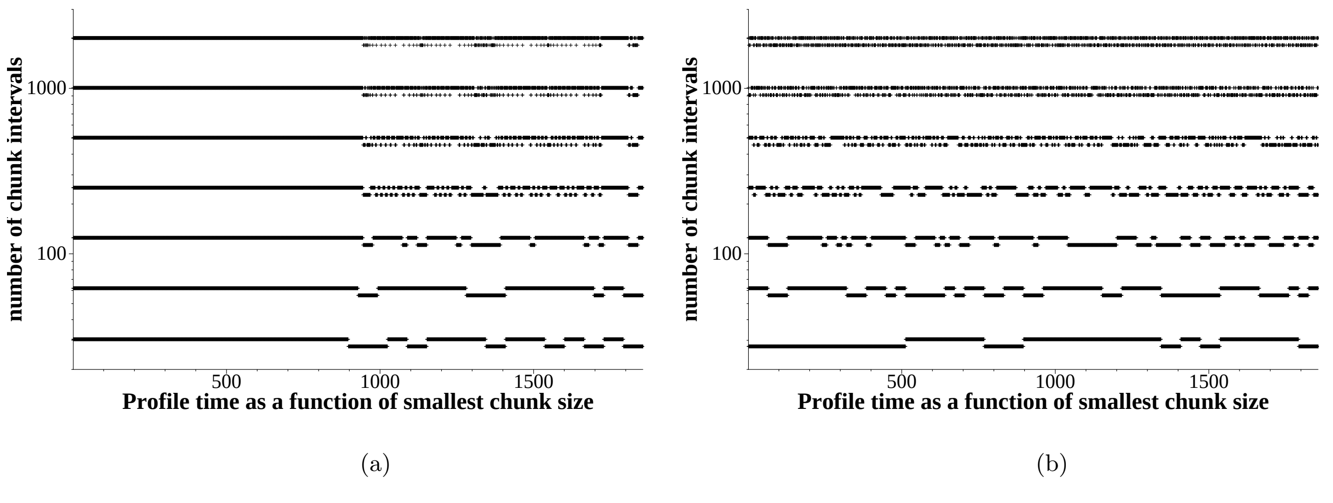


Figure 10: Phase shift detection algorithm's decisions using weighted model for `db` (a) and `jbb` (b).

7. RELATED WORK

Prior work can be categorized into phase shift detection and related algorithms, and phase shift detection in other fields.

7.1 Phase Detection and Related Algorithms

The concept of phases in a program's execution is not new. Madison and Batson [22] in 1976 describe the phase behavior of references to array segments of Algol 60 programs. Their application domain is operating systems. They use an unweighted working set, which corresponds to our *unweighted* model, and described an algorithm that is similar to the one described in this paper. One of the parameters in their algorithm is the size of the working set, or chunk size in our terminology. They consider a new reference to be in the same phase if it is a member of the working set. They are interested in phases that are at least a constant multiple (3 in their figures) of their working set size.

Dhodapkar and Smith [?] study online phase shift detection in the context of multi-configuration hardware, such as instruction caches. They describe algorithms for 1) detecting changes in working sets; 2) identifying recurring working sets; and 3) estimating the number of elements in a working sets. Their technique is an example of the algorithm described in this paper. They employ an unweighted set representation, which corresponds to our *unweighted* model. Because they are interested in large windows (they use a window size of $100,000$ instructions), they present a lossy-compressed representation called *working set signatures*. Their experiments run each benchmark until $20,000$ non-overlapping intervals are collected. They define their similarity threshold value empirically, to be 0.5 to remove most noise and detect only significant phase changes. They report that their experiments indicated that detecting phase changes were relatively insensitive to the threshold value. This is to be contrasted with our results, such as the one shown in Fig. 3, which show that the phases can be sensitive to this parameter.

Sherwood et al. [24] present a technique for finding a portion of a profile that is representative of the entire profile where the profile represents the execution behavior of a program. The smaller profile can then be studied intensively using simulation techniques that would be not be possible with the much larger profile created by the full execution of the program. Their technique is offline; they partition the full profile into equal-size intervals and then use techniques to compare these intervals to all intervals. They employ a weighted set representation, which corresponds to our *weigthed* model, and take the component-wise difference of two equal-size intervals, that is, the Manhattan distance between two vectors representing weighted sets at their dissimilarity metric. This is related to the similarity metric given in Fig. 2 as their dissimilarity metric plus twice our similarity metric is always equal to 2.

Sherwood et al. [25] present an online version of the algorithm in [24]. They used the algorithm to detect phases, and then capture a signature of the phase, which they use to detect repeated occurrences of the same phase. This allows them to reuse the optimization from the previous occurrence of that phase. They introduced the idea of phase prediction, which allows them to anticipate the next phase in the execution of the program.

Dhodapkar and Smith [?] compare three phase shift detection techniques previously published in the literature against a set of metrics. Each technique uses a different form of profiling, as well as a different phase shift detection algorithm. They studied (1) *conditional branch counts* [5], which track the number of conditional branches executed in a given interval of executed instructions; (2) *instruction working sets* [?], the technique of Dhodapkar and Smith described above; and (3) *basic block vectors* [24], the technique of Sherwood et al., also described above. Their metric of phase quality is how uniform the number cycles per instruction (CPI) is within the phase. Their techniques found regions of program execution where the CPI varies only 2%. In contrast to our work, they do not explore varying the chunk size with respect to their metrics, and when they vary the model, they also vary the profile, which makes interpreting their results more difficult because multiple parameter values are varied simultaneously. Finally, they not draw any conclusions about the underlying phase shift detection problem.

Kistler and Franz [19] propose an online phase detection technique that captures the number of occurrences of an event, such as basic block counter, during the two most recent nonoverlapping time intervals. They employ a weighted set representation, which corresponds to our *weigthed* model, and represent their information as a vector. Their similarity metric is the sum of the geometric angle between two vectors and a term that reflects the absolute size of the change. This gives a value between 0.0 and 1.0, where 1.0 is perfect similarity. They use a threshold of 0.95. They conclude that the expense of their approach is ameliorated by computing it only once every five minutes.

Duesterwald et al. [12] study the behavior of programs using metrics derived from hardware counters. They show that programs exhibit significant behavior variation that can be exploited using online statistical and table-based predictors. They also introduce a cross-metric predictor that uses one metric to predict another, and show that table-based predictors outperform statistical predictors by up to 69%.

Dynamo [4] is a transparent dynamic optimizer that performs optimizations at runtime on a native binary based on the execution frequency of hot traces of instructions. It maintains a code cache of optimized versions of these traces by detecting hot code fragments, optimizing them, and storing them in the code cache. Changes in the code cache's working set are observed by tracking the creation rate of optimized code fragments to determine when a phase shift occurs.

Diniz and Rinard [11] describe *dynamic feedback*, a compiler optimization technique that produces multiple optimized versions of the program with different optimization strategies. At runtime they execute each version briefly and determine which gives the best performance. They periodically re-execute each version to account for program phase shifts.

Chilimbi and Hirzel [8] describe an online optimization system for dynamic prefetching based on data references profiles. To account for potential phase shifts, they periodically re-gather the profile. The trigger mechanism is based on a fixed duration, not on the profile data.

Arnold et al. [3] describe an online optimization system that gathers control flow edge execution frequencies for selected methods to drive optimizations in the Jikes RVM [1]. The profile is gathered only after the method is sufficiently executed, as determined by a cost/benefit model. The authors discuss the possible need to re-gather a profile when a phase transition occurs, but do not provide an algorithm for detecting phase transitions.

7.2 Related Problems

The problem of detecting phases in a sequence of data occurs in many domains other than the ones we have discussed in this paper.

One example is phase detection in biological data. In this domain they use techniques such as hidden Markov models [13]. This technique has more parameters, and therefore, requires more profile values to train and to detect phases.

The problem of phase detection is called change-point detection in the statistics literature [21]. The bulk of work in this area deals with continuous variables, rather than discrete symbols as in the domain addressed in this paper.

One example of phase detection or change point detection is the detecting the change of speakers in broadcast news. For example, Chen et al. [7] use the maximum likelihood approach to change detection via the Bayesian information criterion, to detect change of speakers in broadcast news.

8. CONCLUSIONS AND FUTURE WORK

Object-oriented languages have enabled the creation of large, long-running commercial applications where high performance is critical. To achieve high performance, dynamic optimization, which is performed at execution time, must be continuously tailored to the application's changing runtime behavior. One important technology to enable continuous optimization is phase shift detection, which allows a dynamic optimization system to react appropriately to improve the system's performance.

Starting with a simple phase shift detection algorithm specified by three fundamental parameters, we demonstrate with examples and profile data that two of the three parameters are *non-monotonic* with respect to the phase shift detection algorithm's decisions. That is, as the parameter's value changes monotonically, the output of the algorithm is non-monotonic. Our results implies that to determine the "best" value for a non-monotonic parameters may require an exhaustive search of all possible values. Furthermore, once the "best" value for a parameter is found for a particular profile, tuning the parameter's value for another profile is no easier than attempting to find the value from scratch.

This fundamental result is important for dynamic optimization systems because it implies that the choice of values for a phase shift detection algorithm's parameters can have a dramatic impact on the quality of the information that is produced, and thus the choice should be carefully studied.

We are interested in pursuing the following future work:

1. determine, by brute force, if a program's execution has a canonical phase structure,
2. determine how best to choose values for the fundamental phase shift detection parameters,
3. study how the various choices of parameter values impact different phase detection clients, and
4. study the effectiveness of different types of profiles.

9. ACKNOWLEDGEMENTS

The authors thank David Grove for his help in designing the conditional branch tracing in Jikes RVM, and Kartik Agaram, Amer Diwan, Martin Hirzel, Tim Sherwood, and Laureen Treacy for feedback on earlier drafts of this work.

10. 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, February 2000.

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, October 2000.

[3] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices*, 37(11):111–129, November 2002.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, May 2000.

[5] Rajeev Balasubramonian, David H. Albonesi, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *the 33th International Symposium on Microarchitecture*, pages 245–257, December 2000.

[6] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, November 1991.

[7] Michael Chen and Kunle Olukotun. Targeting dynamic compilation for embedded environments. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'02)*, pages 151–164, August 2002.

[8] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. *ACM SIGPLAN Notices*, 37(5):199–209, May 2002.

[9] Michał Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. *ACM SIGPLAN Notices*, 35(5):13–26, May 2000.

[10] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[11] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: An effective technique for adaptive computing. *ACM SIGPLAN Notices*, 32(5):71–84, May 1997. Published as part of the Proceedings of PLDI'97.

[12] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *International*

Conference on Parallel Achitecture and Compilation Techniques, September 2003.

[13] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[14] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization*, pages 241–252. IEEE Computer Society, 2003.

[15] Kim Hazelwood and David Grove. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization*, pages 253–264. IEEE Computer Society, 2003.

[16] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.

[17] Jikes Research Virtual Machine (RVM). http://www.ibm.com/developerworks/oss/jikesrvm.

[18] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, 2000.

[19] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. 25(4):500–548, July 2003.

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

[21] D. C. MacEnany. *Bayesian Change Detection*. PhD thesis, University of Maryland, The Insititute for Systems Research, 1991.

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

[23] Michael Paleczny, Charistopher Vick, and Cliff Click. The Java Hotspot server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, April 2001.

[24] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Prog. Lang. and Oper. Sys. (ASPLOS 2002)*, October 2002.

[25] Timothy Sherwood, Seleyman Sair, and Brad Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, pages 336–349, June 2003.

[26] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 36(11):180–195, November 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[27] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 38(5):312–323, May 2003.

[28] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000, 2000.

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

[30] John Whaley. Partial method compilation using dynamic profile information. *ACM SIGPLAN Notices*, 36(11):166–179, November 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[31] P. R. Wilson and T. G. Moher. Design of the opportunistic garbage collector. *ACM SIGPLAN Notices*, 24(10):23–35, October 1989.

[32] Paul R. Wilson. Opportunistic garbage collection. *ACM SIGPLAN Notices*, 23(12):98–102, December 1988.