RC-22887 (W0308-006) August 4, 2003
Computer Science

IBM Research Report

Phase Shift Detection: A Problem Classification

Michael J. Hind, Vadakkedathu T. Rajan, Peter F. Sweeney
IBM Research Division
Thomas J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center ,
P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Phase Shift Detection: A Problem Classification

Michael Hind V.T. Rajan Peter F. Sweeney

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

ABSTRACT

Dynamic analysis of a program's execution is an increasingly popular approach to characterizing the semantics of a program. One important aspect of dynamic analysis is the detection of phases, which can help improve program understanding, offer specialization opportunities in a dynamic optimization system, reduce simulation time, and dynamically adapt multi-configuration hardware to program behavior. Although both online and offline phase shift detection algorithms exist, most work has focused on the efficacy of the client, without a detailed study of the various dimensions of the underlying phase shift detection problem.

The goal of this paper is to better understand the fundamental problem of phase shift detection. Specifically, we 1) demonstrate that an intuitive notion of a phase is not well-defined; 2) define an abstract problem with two parameters that captures the essence of the phase shift detection problem; 3) show that existing phase shift detection algorithms are instances of this abstract problem; 4) define an intuitive concrete instance of the abstract problem and demonstrate that the concrete problem has a unique solution; and 5) demonstrate for the concrete problem that two intuitive parameters that define a phase are not "well behaved" such that changes in these parameter's value may result in significantly different phases being identified. This last property illustrates the importance of clients choosing appropriate parameter values when employing a phase detection algorithm.

1. INTRODUCTION

Dynamic analysis of a program's execution [6] is an increasingly popular approach to characterize the semantics of a program for the purpose of program understanding [7, 2, 25, 6, 17, 11, 23], dynamic optimization [13, 18, 5, 24, 30, 3, 12, 10, 4, 20], reducing simulation time [26, 27, 28], and dynamically adapting multi-configuration hardware to program behavior [14, 8]. A dynamic analysis offers an advantage over a static analysis (an analysis of the source program prior to its execution) in that it considers only the events that occur during the execution; whereas a static analysis considers all possible events. However, information collected by a dynamic analysis may only be valid for that particular program execution, i.e., information for a particular input may or may not generalize to other inputs. Therefore, a dynamic analysis is typically implemented online (during program execution), adding time and space overhead to a program's execution.

An important aspect of a dynamic analysis is phase shift detection, which characterizes the profile of a program's execution into phases. For example, consider a program's execution where two mutually exclusive events, A_1 and A_2, are profiled and each event occurs 50% of execution time. Characterizing behavior in this manner provides one level of information. However, if there are distinct phases, i.e., periods of executions where one of A_1 or A_2 is dominant, this additional information can provide an extra level of detail that can be useful [29, 15]. For example, a dynamic optimization system can specialize a program's execution for one event when that event dominates, and later switch to a different specialized execution when the other event dominates [5, 10, 4, 20]. However, in this example, for such a strategy to be effective an online phase detection algorithm must be employed.

Some recent work in the feedback-directed optimization community has described online phase shift detection algorithms. However, what is describe is either indirect, where a phase shift is detected by an increase in compilation activity rather than a direct change in the underlying profile [5], or expensive such that its online use is restricted [20].

The architecture community has also found the problem of phase shift detection of interest in both online [14, 8, 28] and offline (after program execution) [26, 27] contexts. Online phase detection has been used to dynamically adapt multi-configuration hardware to program behavior [14], to tailor code sequences to frequently executed phases [8], and to track and predict phases [28]. Offline phase detection has been used to find a phase of a profile that is representative of the complete program's behavior [26, 27]. This shorter profile is then used for detailed simulation analysis, resulting in greatly reduced simulation time compared to using the full profile.

Although both online and offline algorithms for phase shift detection have been defined, most of this work has focused on the efficacy of the client, without a detailed study of the various dimensions of the underlying phase shift detection problem.[1] The goal of this paper is to better understand the fundamental problem of phase shift detection. This paper demonstrates that even small changes to the definition of a phase can lead to radically different phases being identified. In particular, this paper demonstrates that a number of intuitive parameters that may be used to precisely define a phase are not "well behaved" and therefore changes in the value of these parameters may lead to significantly

[1]One exception is the work by Sherwood et al.[27], which discusses a subset of the dimensions of the problem we describe in this work.

different solutions. This realization is important for two reasons. First, a client of the phase shift detection problem must be careful about what concrete problem is used and what parameter values are passed to the problem's solution. In particular, the problem and parameter values that are chosen should depend on the properties of a program's execution that are important to the client. Second, the lack of "well behaved" parameters demonstrate, in general, that there is no canonical phase structure for a program's execution and therefore, different concrete phase shift detection problems cannot be evaluated independently to determine which solution is best.

The contributions of this paper are

- Demonstrate that an intuitive notion of a phase is not well-defined; that is, it may not have a unique solution.
- Define an abstract problem with two parameters that captures the essence of the phase shift detection problem.
- Show that phase shift detection algorithms in the literature [22, 14, 26, 27, 20] are instances of this abstract problem.
- Define an intuitive concrete instance of the abstract problem and demonstrate that the concrete problem has a unique solution: that is, any algorithmic solution to the concrete problem will produce the same result for the same input.
- Demonstrate for the concrete problem that two intuitive parameters that define a phase are not "well behaved" such that changes in these parameter's value may result in significantly different phases being identified in a program's execution. We show the effects of changing these parameters' values both with examples and with the profiles from a set of Java benchmarks.

The remainder of this paper is organized as follows. Section 2 demonstrates that an intuitive notion of a phase is not well-defined and introduces an abstract problem statement that requires two parameters, granularity and similarity, that captures the essence of the phase shift detection problem. Section 3 instantiates these two parameters, leading to an intuitive concrete phase detection problem and a generic algorithm for solving the problem, and proves that the concrete problem has a unique solution. Section 4 illustrates with examples that the identification of a phase can vary significantly when the value of one of the concrete problem's parameters changes. Section 5 uses profile data, a conditional branch trace of Java benchmarks, to illustrate that the identification of a phase can vary significantly when the value of one of the concrete problem's parameters changes. Section 6 discusses alternative concrete instances of the abstract problem by discussing additional parameter values and additional parameters. Section 7 summarizes related work, and where appropriate, states how previous phase shift detection algorithms are an instance of the abstract problem. Section 8 discusses future work and concludes.

2. ABSTRACT PROBLEM STATEMENT

This section discusses the phase shift detection problem and demonstrates that an intuitive notion of a phase is not well-defined; that is, it may not have a unique solution. It then specifies an abstract problem that captures the essence of the phase shift detection problem with two parameters: granularity and similarity. A client of the abstract phase shift detection problem needs to specify both granularity and similarity. Section 3 shows that at least one definition of these two parameters provides a precise definition of phase.

Intuitively, a *phase* occurs when the program's execution behavior is similar, 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*. Given a definition of a phase, the general phase shift detection problem, $\mathcal{PSD}(\mathcal{P})$, can be stated as

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

To avoid trivial phases, where each string element 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.

Phases need not be consecutive, that is, the union of the identified phases is not necessarily $\mathcal{P}$. Thus, there may be regions of $\mathcal{P}$ that are not in any phase, but 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

$$\mathbf{aaaaaaabbbbbbb} \quad (1)$$

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

$$\mathbf{aaaaaabbbaaaaaa} \quad (2)$$

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:

$$\mathbf{aababcaabab} \quad (3)$$

In this profile, should **c** be considered 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 and phase transitions in string (3).

To reason about phases in a profile, we need a precise definition of a phase. In general, an intuitive definition of a phase is not well-defined because it will not result in a unique solution. In particular, the following aspects of a phase are undefined:

- the atomic units of comparison, or granularity, and
- how the similarity of two strings is computed

How should granularity be defined? The trivial definition is that the size of the unit of comparison is 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**, with two phase transitions. Alternatively, a larger granularity 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.

How should similarity be computed? When the size of the unit of comparison is greater than one, the similarity value of substrings must be computed. How should two strings be modeled to compute their similarity value? Should order be considered? If not, should frequency of occurrence (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 unweighted set similarity value of the two strings is 0.33 because only two out of six elements are similar.

Let $\mathcal{PSD}\ [\tau, \sigma]\ (\mathcal{P})$ be the abstract definition of the phase shift detection problem that takes as input a profile $\mathcal{P}$ and the parameters:

Granularity (τ) specifies how a profile is partitioned into fixed length atomic units of comparison, denoted chunks, and specifies the minimum size of a phase.

Similarity (σ) is a boolean function that, given two strings, computes if the two strings are similar. That is, $\sigma\ (\mathcal{S}^1, \mathcal{S}^2)$ returns true if $\mathcal{S}^1$ is similar to $\mathcal{S}^2$; otherwise σ returns false.

A phase can be defined with respect to granularity and similarity:

> A string $\mathcal{S}$ is a *phase* if the length of $\mathcal{S}$ is at least the minimum size of a phase, $\mathcal{S}$ is maximal, and the chunks contained in $\mathcal{S}$ are similar.

Although the abstract problem's parameters do not provide a precise definition of phase, these parameters provide an abstraction with which we can classify phase shift detection algorithms with respect to how they define the parameters.

3. CONCRETE PROBLEM STATEMENT

This section presents a concrete instance of the abstract problem, $\mathcal{PSD}\ [\tau, \sigma]$, presented in Section 2 by providing suitable definitions for the parameters τ and σ. The concrete instance defines the phase shift detection problem, denoted the *concrete problem*, such that it has a unique solution; that is, every algorithm that solves the concrete problem will produce the same result for a particular input. A concrete problem that does not have a unique solution makes reasoning about algorithms that solve the problem difficult, if not, impossible.

A concrete problem specifies a set of parameters that define τ and σ and specifies the set of values that are valid for each parameter. The set of values may be defined as a range of continuous or discrete values, or as an enumeration (a set of categorical values). To simplify the discussion, we have purposefully chosen a concrete problem that has a small set of intuitive parameters and for some of the parameters only one value is valid. We prove that our concrete problem has a unique solution. In addition, we demonstrate with this concrete problem that the identification of phases in a profile may be sensitive to the values of the problem's parameters. Section 6 explores some alternative parameter values and parameters.

The concrete problem's first three parameters specify the abstract problem's granularity parameter, τ.

Chunk Size $(\mathcal{G})$ is a positive integer that specifies the length of a chunk. For online algorithms, chunk size impacts responsiveness because $\mathcal{G}$ elements are collected before checking similarity. So there could be a delay of up to $\mathcal{G}$ elements before a phase transition is detected. In addition, a large $\mathcal{G}$ is insensitive to phase transitions that are contained within the chunk.

Chunking $(\mathcal{C})$ specifies how to divide a profile into chunks. In particular, $\mathcal{C}$ is defined as $leftmost$: starting from the first element in profile $\mathcal{P}$ and creating consecutive nonoverlapping chunks.

Factor $(\mathcal{F})$ is a positive integer that is a multiplicative factor of $\mathcal{G}$ such that the length of a phase must be at least $\mathcal{F} * \mathcal{G}$. In particular, $\mathcal{F}$ is defined to be two.

The concrete problem's next three parameters specify the abstract problem's similarity parameter, σ.

Model $(\mathcal{M})$ is a function that takes two strings as input, converts each string into an abstract representation, 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. We assume that string values are atomic; that is, there is no structure to a value that can be exploited, and that there is no stronger relationship between a discrete value n and a discrete value $n + k$ for any nonzero value of k.[2] In particular, two abstract representations are considered:

- **weighted set** the abstract representation of a string is a weighted set, or
- **unweighted set** the abstract representation of a string is an unweighted set.

Breakup $(\mathcal{B})$ is a function that takes two strings as input and specifies how substrings of the strings have their similarity values computed. In particular, $\mathcal{B}$ is defined as *consecutive*: the two strings are consecutive and the similarity value is computed from the last chunk of the first string and the first chunk of the second string.

Threshold $(\mathcal{T})$ is a constant that ranges from 0.0 to 1.0. Two strings are similar if their similarity value is at least $\mathcal{T}$.

Let $\mathcal{PSD}\ [\mathcal{G}, \mathcal{C}, \mathcal{F}, \mathcal{M}, \mathcal{B}, \mathcal{T}](\mathcal{P})$ specify the concrete problem where $\mathcal{P}$ is a profile of the program's execution behavior and is represented a string of values. Given this concrete problem the precise definition of a phase is

[2]Although these two assumptions are not necessary, they simplify the presentation.

> A string $\mathcal{S}$ is a phase in profile $\mathcal{P}$ if the length of $\mathcal{S}$ is at least $\mathcal{F}*\mathcal{G}$, the first chunk in $\mathcal{S}$ is not similar with the preceding chunk (if any) in $\mathcal{P}$, the last chunk in $\mathcal{S}$ is not similar with the succeeding chunk (if any) in $\mathcal{P}$, and any two consecutive chunks contained in $\mathcal{S}$ are similar.

Because of the specification of *leftmost* chunking and *consecutive* breakup, each boundary of $\mathcal{S}$ falls on a boundary of a chunk contained in $\mathcal{S}$.

THEOREM 1. *This concrete problem has a unique solution for any given input $\mathcal{P}$.*

PROOF. Let $\mathcal{S}_1, \cdots, \mathcal{S}_n$ be a sequence of chunks representing a partition of $\mathcal{P}$. From the definition of *leftmost* chunking, this sequence must be unique. From the definition of *consecutive* breakup, two adjacent chunks, $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$, that are similar are in the same phase. Furthermore, if $\mathcal{S}_{i-1}$ and $\mathcal{S}_i$ are also similar, and therefore, in the same phase, then $\mathcal{S}_{i-1}$ and $\mathcal{S}_{i+1}$ are also in the same phase. Therefore, "in the same phase" is a transitive relationship. Thus, we can define an equivalence relation between pairs of chunks: two chunks are "in the same phase" if they are adjacent and are similar, or they are not adjacent and there is a sequence of chunks between them such that every adjacent pair in the sequence is similar. This equivalence relation partitions the set of chunks into equivalence classes. Every class that contains at least two chunks (because $\mathcal{F} = \in$) defines a phase, and every class that contains only one chunk is part of a phase transition. A phase transition is a sequence of chunks such that each chunk is in its own class. Because an equivalence class defines a unique partitioning, the solution to this concrete problem for a given input is uniquely defined. □

Note that even our simple concrete problem requires six parameters to be well-defined.

3.1 Generic Algorithm

This section presents a generic algorithm that solves our concrete problem $\mathcal{PSD}$ [$\mathcal{G}$, $\mathcal{C}$, $\mathcal{F}$, $\mathcal{M}$, $\mathcal{B}$, $\mathcal{T}$] when the values of the parameters are constrained as specified above. The input to the algorithm is a profile, $\mathcal{P}$, that represents the behavior of a program's execution as a string of values. The algorithm outputs a vector of pairs of integers that identifies the phases. The length of the vector is the number of phases. Each pair of integers represents a phase: the first integer of each pair specifies the element in $\mathcal{P}$ that is the start of that phase, and the second integer of each pair specifies the element in $\mathcal{P}$ that is the end of that phase. Given any two pairs, $\langle f^i, l^i\rangle$ and $\langle f^j, l^j\rangle$, in the vector, if $f^j > f^i$ then $f^j > l^i$, and if $l^i < l^j$ then $l^i < f^j$. That is, the vector of pairs specify nonoverlapping substrings of $\mathcal{P}$. The largest substrings in the profile that do not overlap any phases are the phase transitions.

Figure 1 illustrates how phases are identified. Because only three ($\mathcal{G}$, $\mathcal{M}$, and $\mathcal{T}$) of the six concrete problem's parameters have multiple values, only these three parameters are explicitly passed to the generic algorithm. The *inTransition* boolean flag keeps track of the state of the algorithm: *inTransition* is *true* when the algorithm has recognized a phase transition; otherwise, the algorithm has recognized a phase. The algorithm starts in a phase transition with *inTransition* set to *true*. The variable n identifies the number of phases identified so far by the algorithm. The

```
/*
* Phase shift detection algorithm that solves the concrete problem:
* PSD [G, C, F, M, B, T] where chunking C is leftmost,
* factor F is two, and breakup B is consecutive.
*/
PSD [G, M, T] (P) {
  inTransition = true                   // state of algorithm
  n = 0                                 // number of phases
  b = 0                                 // start of phase
  i = G - 1                             // end of phase
  while (i + G) < |P|) {
    S^1 = P_b ··· P_i                   // current or candidate phase
    S^2 = P_(i+1) ··· P_(i+G)           // next chunk
    if ( similarity [M, T] (S^1, S^2) ) {
      if ( inTransition ) {
        inTransition = false            // start of phase
      }
    } else {                            // ! similarity(...)
      if (! inTransition ) {            // end of current phase
        phaseBoundary[n] =< b, i >
        n = n + 1
        b = i + 1                       // start of candidate phase
        inTransition = true             // in phase transition
      } else {
        b = b + G                       // start of candidate phase
      }
    }
    i = i + G
  }
  if (! inTransition ) {
    phaseBoundary[n] =< b, i >
    n = n + 1
  }
  return phaseBoundary
}
```

Figure 1: A generic algorithm that solve the concrete phase shift detection problem.

```
similarity [M, T] (S^1, S^2) {          // B is consecutive
  S^x = S^1_(|S^1|-G) ··· S^1_(|S^1|-1) // last chunk
  return M (S^x, S^2) >= T
}
```

Figure 2: Compute for *consecutive* breakup the similarity value of two strings where $\mathcal{S}^1$ is the candidate phase and $\mathcal{S}^2$ is the next chunk.

```
M_weighted (S^1, S^2) {
  similarity = 0
  W^1 = weightedSet (S^1)
  W^2 = weightedSet (S^2)
  forall ⟨v, w_2⟩ ∈ W^2 {
    if ( ⟨v, w_1⟩ ∈ W^1 ) {
      similarity += min(w_1, w_2)
    }
  }
  return similarity / |S^2|
}
```

Figure 3: Computes the similarity value for a *weighted* abstract representation.

```
M_unweighted (S^1, S^2) {
  return (unweightedSet(S^1) ∩ unweightedSet(S^2))/
    (|unweightedSet(S^2)|)
}
```

Figure 4: Computes the similarity value for an *unweighted* abstract representation.

variable b identifies the element in $\mathcal{P}$ that is the start of a candidate phase.The variable i identifies the element in $\mathcal{P}$ that is the end of the candidate phase.

The while loop breaks $\mathcal{P}$ up into nonoverlapping consecutive chunks of size $\mathcal{G}$ starting from the leftmost element in $\mathcal{P}$, satisfying the *leftmost* chunking parameter. $\mathcal{S}^1$ is a string of elements that represents the candidate (or current) phase, and $\mathcal{S}^2$ is a string of $\mathcal{G}$ elements that represents the next chunk. The method *similarity* is called with $\mathcal{S}^1$ and $\mathcal{S}^2$ to determine if they are similar. If similar and *inTransition* is *true*, a new phase is started by changing *inTransition* to *false*. If $\mathcal{S}^1$ and $\mathcal{S}^2$ are not similar and $\mathcal{S}^1$ is a current phase (because *inTransition* is *false*), then the end of phase is identified as i; the phase is recorded in *phaseBoundary* array; the number of phases, n, is incremented; the start element, b, of the candidate phase is set to the next element after the end of phase just identified; and *inTransition* is set to *false*. Otherwise, we are in a phase transition, and the start element, b, of the next candidate phase is set to the beginning of the next chunk by incrementing b by $\mathcal{G}$. At the end of each iteration, i is incremented by $\mathcal{G}$ to point to the end element of the candidate phase. After all the values in $\mathcal{P}$ have been examined, if *inTransition* is *false* a phase boundary has been found and the phase is added to the *phaseBoundaries* array.

The generic algorithm is guaranteed to identify phases whose size is at least $2 * \mathcal{G}$, because *inTransition* is set to *false* only after two chunks are combined into a phase. The way that phase boundaries are computed guarantees that the first chunk of a phase is not similar to the preceding chunk (if any) and the last chunk of a phase is not similar to the succeeding chunk (if any). Furthermore, two consecutive chunks in a phase are similar because a phase boundary is not identified. In addition, a phase transitions has a size of either zero chunks because after *inTransition* is set to *true*, it may be set to *false* on the next iteration if a new phase is found, or one or more chunks because *inTransition* stays *true* for any number of iterations. Finally, each of a phase's boundaries falls on a boundary of a chunk contained in the phase because the generic algorithm only computes similarity values for consecutive chunks.

Figure 2 illustrates how the similarity of two strings, $\mathcal{S}^1$ and $\mathcal{S}^2$, is computed where $\mathcal{S}^1$ represents the candidate phase and $\mathcal{S}^2$ represents the next chunk. The breakup parameter, $\mathcal{B}$, determines what substrings of $\mathcal{S}^1$ and what substrings of $\mathcal{S}^2$ have their similarity values computed. Because $\mathcal{B}$ is always *consecutive* it is not passed as a parameter, and the last chunk of $\mathcal{S}^1$ is passed to σ with the first chunk of $\mathcal{S}^2$, which, in this case, is all of $\mathcal{S}^2$.

Figure 3 illustrates how the *weighted* model computes the similarity value of two strings: the sum of the minimum weight of each element in the weighted sets is divided by the size of second string, which for *consecutive* breakup is the same as the first string. If an element is not in a set, its weight is zero.

Figure 4 illustrates how the *unweighted* model computes the similarity value of two strings: the number of elements that are in both sets divided by the number of elements in the second set.

The *weighted* model uses a symmetrical similarity computation because the sizes of both strings are the same by definition of *consecutive* breakup. The *unweighted* model uses an asymmetrical similarity computation because the denominator is relative to the size of the second set, which may be different then the size of the first set. (Recall that the unweighted model works on sets rather than strings.) Other similarity computations are possible, including those that deal with strings of different sizes, as is discussed in Section 6.

4. EXAMPLES

This section provides small intuitive examples to illustrate how different values for the concrete problem's parameters affects the identification of phases. Before beginning, the following notation is used to specify the phase structure of a string. A left square bracket, [, demarcates the start of a phase, and a right square bracket,], demarcates the end of a phase. For example,

[aaaa] bc [dddd]

specifies a *phase structure* for the string **aaaabcdddd** such that there are two phases and one phase transition. The first phase is the substring **aaaa**, the second phase is the substring **dddd**, and the phase transition is the substring **bc**. Any concrete instance of the abstract problem, $\mathcal{PSD}$ $[\tau, \sigma]$, generates as output the phase structure of its input $\mathcal{P}$.

4.1 Threshold

This section illustrates how the choice of values for threshold affects the identification of phases. Assume that the parameter values are $\mathcal{G} = 3$, $\mathcal{M} = weighted$, and threshold may vary: $\mathcal{T} = 0.6$ or $\mathcal{T} = 0.3$. Consider the following example string that has six chunks:[3]

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

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

Two phases and two phase transitions are identified. The first phase is **aababb** because the similarity value of chunks **aab** and **abb** is 0.66, which is above $\mathcal{T}$. The second phase is **bccbcb** because the similarity value of chunks **bcc** and **bcb** is 0.66, which is above $\mathcal{T}$. A phase transition containing no chunks occurs between the two phases because the similarity value of chunks **abb** and **bcc** is only 0.33, which is below $\mathcal{T}$. The second phase transition is **ddddee** because the similarity value of chunks **ddd** and **dee** is 0.33, which is below $\mathcal{T}$, and their is no similarity between chunks **bcb** and **ddd**.

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

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

Two phases and one phase transition are identified. 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 phase structure may vary significantly with different thresholds. In particular, when the threshold is reduced, the number of phases identified might either increase, decrease, or stay the same. The number of phases can decrease because adjacent phases are combined. The number of phases can increase because phase transitions become phases. Nevertheless, the phase structures for different threshold values do not diverge: phase boundaries are eliminated only when $\mathcal{T}$ is decreased, never when $\mathcal{T}$ is increased.

4.2 Model

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

$$\textbf{abbbbb aaaaab aaaaac accccc} \quad (5)$$

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 as $\{a, b\}$, and chunk **aaaaac**, which has an unweighted set representation as $\{a, c\}$, because their unweighted set representations have a similarity value of 0.5, which is below $\mathcal{T}$. The first phase is **abbbbbaaaaab** because the unweighted set representation of the chunks **abbbbb** and **aaaaab** are the same, $\{a, b\}$, and results in a perfect similarity value. The second phase is **aaaaacaccccc** because the unweighted set representation of the chunks **aaaaac** and **accccc** are the same, $\{a, c\}$, and results in a perfect similarity value.

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

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

One phase and two phase transitions are identified. The first phase transition contains of the chunk **abbbbb** because the similarity value of it and chunk **aaaaab** is 0.33, which is below $\mathcal{T}$. The second phase transition contains of the chunk **accccc** because the similarity value of it and chunk **aaaaac** is 0.33, which is below $\mathcal{T}$. The phase is **aaaaabaaaaac** because the similarity value of chunks **aaaaab** and **aaaaac** is 0.83, which is above $\mathcal{T}$.

This example demonstrates that the phase structure may vary significantly with different models. In particular, the phase structures for different models diverge: each phase for a model eliminates a phase boundary in the other model.

4.3 Chunk Size

This section illustrates how the choice of values for chunk size affects the identification of phases. Assume that the parameter values are $\mathcal{M} = weighted$, $\mathcal{T} = 0.6$, and chunk size may vary: $\mathcal{G} = 3$, $\mathcal{G} = 6$, or $\mathcal{G} = 12$. Consider the following string:

$$\textbf{aaa bbb bbb aaa aaa ccc ccc acc} \quad (6)$$

When $\mathcal{G} = 3$, 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, which is above $\mathcal{T}$. One might argue that $\mathcal{G} = 3$ is optimal for string (6) because, other than the last chunk, every chunk contains elements that have the same value.

When $\mathcal{G} = 6$, 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 $\mathcal{G} = 3$

This example demonstrates that the phase structure may vary significantly with different chunk sizes. In particular, the phase structures for different chunk sizes diverge: phase boundaries when $\mathcal{G}$ is one value are eliminated when $\mathcal{G}$ is the other value.

When $\mathcal{G} = 12$, 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 $\mathcal{T}$. $\mathcal{G} = 12$ demonstrates that a large chunk size may hide phase transitions and hide recurring phases.

4.4 Discussion

The examples above demonstrate that the phase structure of a string may be sensitive to a parameter's value. Although the notion of divergence provides intuitive insight into when

the phase structures for two values of a parameter are not well behaved, this section presents a way to quantify the well behaved property of a parameter. More precisely, given a profile and the values of a concrete problem's parameters, the partial output of the concrete problem can be viewed as a sequence of similarity values that are generated as a function of time. The full output, which computes a phase structure, may be computed from the sequence of similarity values by applying the threshold. Therefore, we can define a parameter as being *well behaved* if for every pair of a parameter's values, the corresponding sequences of similarity values have a high positive statistical correlation. A correlation value can range from 1.0 to -1.0, where 0.0 represents no correlation, -1.0 represents perfect negative correlation, and 1.0 represents perfect positive correlation.

If two sequences of similarity values correlate positively, then there is a threshold value for each sequence that results in nontrivial phase structures that are similar. Two phase structures are similar if their phase boundaries are near each other. A phase structure is nontrivial if multiple phases are identified.

Threshold is well behaved by construction. Surprisingly, chunk size and model are not well behaved. The next section demonstrates that these parameters may not be well behaved when the input to the generic algorithm is a trace of conditional branches from the execution of Java programs.

5. EMPIRICAL DATA

The section presents the results of running the algorithm, presented in Section 3.1, on real profiles to determine how different values for the concrete problem's parameters affects the identification of phases. Although Section 4 illustrated that, at least for contrived examples, the value chosen for a parameter may have a significant impact on the phase structure of a string, this section demonstrates that this phenomenon also occurs in the profile of real programs.

5.1 Experimental Methodology

We gather a profile representing an exhaustive conditional branch trace of eight Java programs from the SPECjvm98 benchmark suite [31] and the SPECjbb2000 [32] benchmark. We generated the profile by modifying the Jikes RVM [1, 19] baseline compiler to trace conditional branches and table jumping (lookupswitch and tableswitch) bytecodes. Unconditional branches, such as goto and jump to subroutine (jsr), and method invocation and return are not traced. Our 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 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. The higher the value in the vertical axis the better the similarity. Each point in the graph is the similarity value for two consecutive chunks, where a chunk is $\mathcal{G}$ symbols grouped together. Each line segment between two points represents a chunk. The first chunk, that is the first line segment, is not included in the graphs.

Note that divergence is pictorially illustrated in a similarity graph when either the slope of a point for one value of a parameter is positive and the slope of the corresponding point for another value is negative, or a point for one value is a local minima and the corresponding point for another value is a local maxima. Furthermore, divergence in a similarity graph of two sequences of similarity values identifies divergence for all possible values of $\mathcal{T}$, whereas the divergence in two phase structures is dependent on the value or values of $\mathcal{T}$ that were chosen to generate the phase structures.

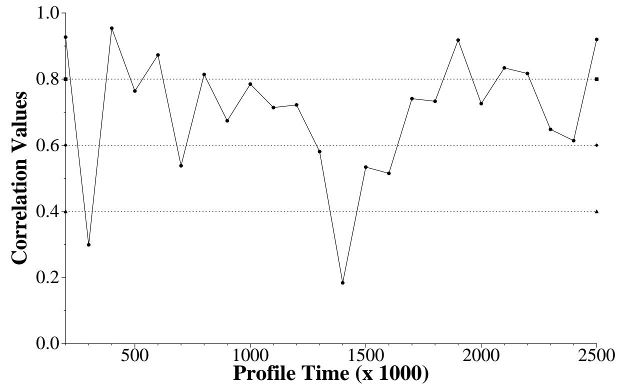

Figure 5: Similarity graph computed from a conditional branch trace of javac with input size 10 when $\mathcal{G} = 100,000$ and $\mathcal{M} = unweighted$.

5.2 Threshold

Figure 5 presents the similarity graph that is computed from a conditional branch trace for SPECjvm98 *javac* benchmark when run with input size 10. The phase shift detection algorithm's parameters are model, $\mathcal{M}$, is *weighted*, and chunk size, $\mathcal{G}$ is 100,000. For a given $\mathcal{T}$, the algorithm detects that the program's execution is in a phase when the line is above $\mathcal{T}$ and the algorithm detects that the program's execution 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.

Figure 5 illustrates that the phase structure varies significantly with the value of $\mathcal{T}$. For example, when $\mathcal{T} = 0.4$ there are three phases, when $\mathcal{T} = 0.6$ there are four phases, and when $\mathcal{T} = 0.8$ there are seven phases. When $\mathcal{T} = 0.4$ the lengths of the three phases from left to right are two, eleven, and twelve, and there are two phase transitions, each with a length of zero. However, when $\mathcal{T} = 0.8$, the lengths of the seven phases are all two, except for the next to last phase, which has a length of three. Of the seven phase transitions, only two have a length greater than zero: one has a length of nine and the other a length of one.

By construction, a similarity graph illustrates that the threshold parameter is well behaved.

5.3 Model

Figure 6 is the same similarity graph presented in Figure 5 with an additional dashed line for the similarity values that are computed from the *weighted* model. This graph illus-

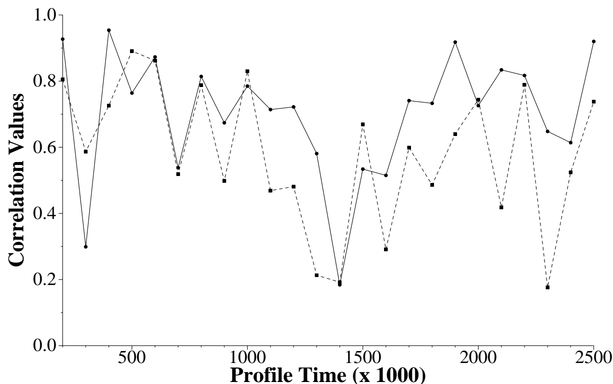

Figure 6: Similarity graph computed from a conditional branch trace of javac with input size 10 when $\mathcal{G} = 100,000$ and $\mathcal{M}$ is both *unweighted* (solid line) and *weighted* (dashed line).

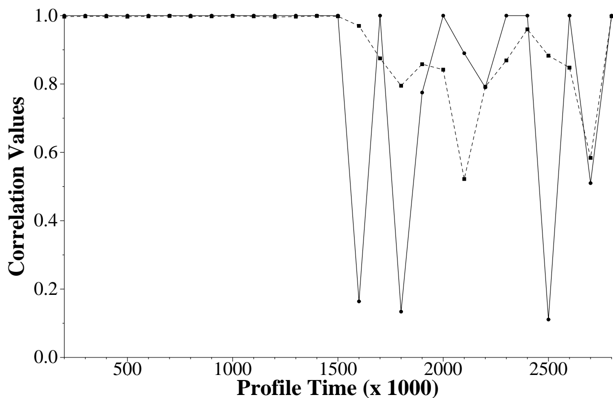

Figure 7: Similarity graph computed from a conditional branch trace of db with input size 10 when $\mathcal{G} = 100,000$ and $\mathcal{M}$ is both *unweighted* (solid line) and *weighted* (dashed line).

trates that the phase structure varies with the value of $\mathcal{M}$. Although there is similarity between the two lines, there is also some variation. Pictorially, of the twenty four pairs of points, six pairs diverge. For example, point 500 is the highest point of its two neighbors for the *weighted* model while the corresponding point is the lowest of its two neighbors for the *unweighted* model. Quantifying the property of well behaved, the correlation of the two sequences of similarity values is 0.6, which corresponds approximately with our intuitive interpretation.

Figure 7 presents the similarity graph that is computed from a conditional branch trace for SPECjvm98 *db* benchmark when run with input size 10. The similarity values for both the *unweighted* (solid line) and *weighted* (dashed line) models are graphed when $\mathcal{G} = 100,000$. There are two interesting phenomena to point out. In the first half of the graph, both models have the same perfect similarity values. The perfect similarity value means that the program has very consistent and similar behavior over time. One may conclude with a perfect similarity, the model is unimportant. The second half of the graph, the lines for the different models diverge, illustrating that when the program does not have consistent and similar behavior, the model that is used may have a significant affect on what phases are identified. The correlation of the two lines is 0.361, which states that the models have a poor correlation for this input. The second half of the profile has a correlation 0.040.

5.4 Chunk Size

Figures 8 presents two similarity graphs that are computed from a conditional branch trace for SPECjvm98 *jess* benchmark when run with input size 10. The phase shift detection algorithm's parameters are model, $\mathcal{M}$, is *weighted*, and chunk size is varied: $\mathcal{G}$ is 10,000 (the graph on the left) or $\mathcal{G}$ is 100,000 (the graph on the right). The graph on the left has ten points for every point for the graph on the left.[4] When $\mathcal{G} = 10,000$, the points in the graph on the left are bimodal: a cluster above the similarity value of 0.8, and a cluster of points between the similarity values 0.6 and 0.4. When chunk size is increased to $\mathcal{G} = 100,000$, few points are below similarity value 0.6. As seen in the example in Section 4.3, a larger chunk size may hide phase transitions. Because the number of points in the two graphs differ by a factor of ten, we computed the correlation in two ways: we computed the average of every 10 points when $\mathcal{G} = 10,000$ and correlated the average with the points when $\mathcal{G} = 100,000$, and we replicated every point when $\mathcal{G} = 100,000$ nine times and correlated the replication with the points when $\mathcal{G} = 10,000$. For both cases, the correlation was very close to 0.0, 0.03 and 0.007, respectively.

This graph illustrates that the phase structure will vary significantly with the value of $\mathcal{G}$. With a large $\mathcal{G}$ there is high similarity between consecutive chunks, and with a smaller $\mathcal{G}$, the similarity values are chaotic: the values bounce frequently between being nearly perfect, to being only 50% similar, and occasionally being very dissimilar. This illustrates that larger $\mathcal{G}$ can have a smoothing effect on similarity values that results in near perfect similarity, and smaller $\mathcal{G}$ values can reveal significant variation in similarity values.

6. ALTERNATIVES

This section explores some natural ways to extend the concrete problem presented in Section 3 with alternative parameter values and with alternative parameters.

Consecutive breakup may result in the concrete problem identifying as a phase a string that changes over time such that there may exist two chunks that are identified as being in the phase, are not consecutive, and are not similar. This scenario can be prevented by defining an alternative breakup parameter value, *complete*, which computes the similarity value as the minimum similarity value of every nonoverlapping chunk in one string with every nonoverlapping chunk in the other string [26]. This guarantees that when a string changes over time the change is not dissimilar with any chunk already in the phase.

Nevertheless, *complete* breakup, unlike *consecutive* breakup, is not transitive: the order that strings are combined matters. Consider three strings $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$, such that the similarity value of $\mathcal{S}_1$ and $\mathcal{S}_2$ is $\mathcal{V}_1$, the similarity value of $\mathcal{S}_2$ and $\mathcal{S}_3$ is $\mathcal{V}_2$, and both $\mathcal{V}_1$ and $\mathcal{V}_2$ are above some threshold $\mathcal{T}$; nonetheless, the similarity value of $\mathcal{S}_1$ and $\mathcal{S}_3$ is not guaranteed to be above $\mathcal{T}$.[5] Therefore, although $\mathcal{S}_1||\mathcal{S}_2$ is a

[4] Because of the increase in the number of points, we did not draw a line between the points because the line would hide the details.

[5] In the special case where similarity values $\mathcal{V}_1$ and $\mathcal{V}_2$ are 1.0,

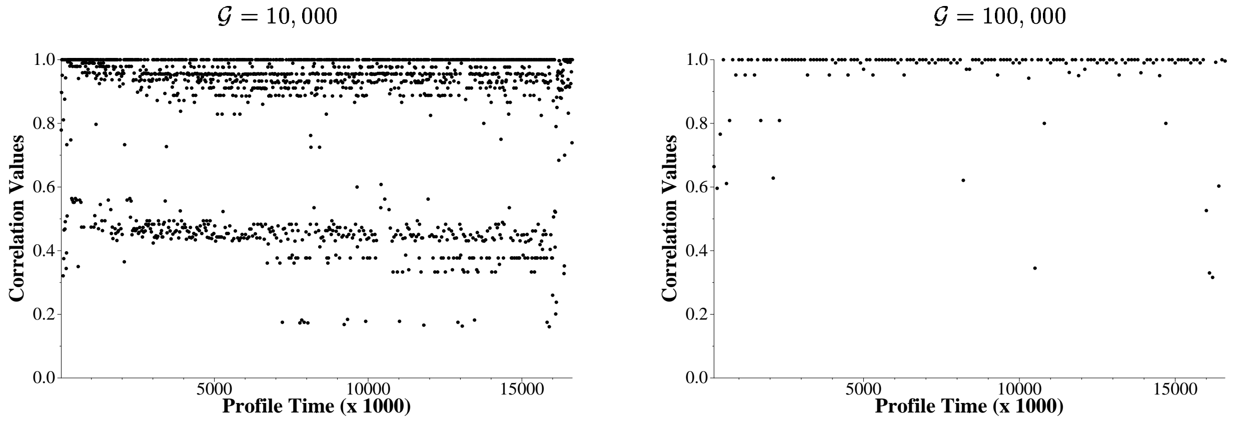

Figure 8: Two Similarity graphs computed from a conditional branch trace of jess with input size 10 when $\mathcal{M}$ = *unweighted* and chunk size varies: $\mathcal{G} = 10,000$ (left) or $\mathcal{G} = 100,000$ (right).

```
M_weightedNormalized (S^1, S^2) {
  similarity = 0
  W^1 = weightedSet (S^1)
  W^2 = weightedSet (S^2)
  forall ⟨v, w2⟩ ∈ W^2 {
    if ( ⟨v, w1⟩ ∈ W^1 ) {
      similarity += min(w1/|S^1|, w2/|S^2|)
    }
  }
  return similarity
}
```

Figure 9: Computes the similarity value for a normalized *weighted* abstract representation.

valid phase, and $\mathcal{S}_2||\mathcal{S}_3$ is a valid phase, $\mathcal{S}_1||\mathcal{S}_2||\mathcal{S}_3$ may not be a valid phase, where || is string concatenation.

In particular, if $\mathcal{S}_1$ is **aa**, $\mathcal{S}_2$ is **ab**, $\mathcal{S}_3$ is **bb**, and $\mathcal{T} = 0.5$, then $\mathcal{S}_1||\mathcal{S}_2||\mathcal{S}_3$ would be a valid phase when breakup is *consecutive*; however $\mathcal{S}_1||\mathcal{S}_2||\mathcal{S}_3$ would not be a valid phase when breakup is *complete* because there is no similarity between **aa** and **bb**. For a concrete problem to support *complete* breakup and still guarantee a unique solution, an additional parameter, orientation ($\mathcal{O}$), can be introduced that specifies which substring to combine first when there is a choice for combining substrings.[6]

Complete breakup will identify a phase transition even if only one chunk in a phase has a poor similarity value with the next chunk that is a candidate to be added to the phase. Spurious phase transitions may be prevented by defining an alternative breakup parameter value, *average*, which computes the similarity value between the two complete strings; that is, compute the similarity value of the current phase and a new chunk where the lengths of the strings may not be the same. Therefore, the algorithm $\mathcal{M}_{weighted}$ that is presented in Figure 3 should not be used to compute *average* breakup because the algorithm, used by *consecutive* breakup, assumes that both strings are of equal length, $\mathcal{G}$. To compute the similarity value of strings of different lengths an additional parameter, scaling ($\mathcal{R}$), can be introduced that specifies how a string is normalized. Figure 9 presents an algorithm that normalizes the weight of each element in a set with the size of that set and that then computes a similarity value from the normalized weights.

As an example, assume that $\mathcal{G} = 3$ and consider the following input string with five chunks:

$$\textbf{aab bab bbc aab bab} \qquad (7)$$

When $\mathcal{T} = 0.4$, $\mathcal{M}$ = *weighted*, and $\mathcal{B}$ = *complete*, the phase structure for string (7) is:

[**aab bab**] **bbc** [**aab bab**]

One phase that repeats is identified because the similarity value of chunks **aab** and **bab** is 0.6, which is above $\mathcal{T}$, and one phase transition, **bbc**, is identified because the similarity value of chunk **aab** and chunk **bbc** is 0.33, which is below $\mathcal{T}$.

When $\mathcal{B}$ = *average* and the algorithm in Figure 9 is used to compute a normalized similarity value, the phase structure for string (7) is:

[**aab bab bbc aab bab**]

One phase is identified because the similarity value of string **aabbab** and chunk **bbc** is computed as 0.5, which is above $\mathcal{T}$, the similarity value of string **aabbabbbc** and chunk **bab** is 0.66, which is above $\mathcal{T}$, and the similarity value of string **aabbabbbcaab** and chunk **bab** is 0.83, which is above $\mathcal{T}$.

The chunking parameter, $\mathcal{C}$, assumes consecutive nonoverlapping chunks. Because phase boundaries are located at chunk boundaries, part of a phase might be included in a chunk that is identified with another phase or with a phase transition.

As an example, assume $\mathcal{G} = 5$ and consider the following input string with five chunks:

$$\textbf{aaaaa aaaaa aabbb bbbbb bbbbb} \qquad (8)$$

When $\mathcal{T} = 0.61$, $\mathcal{M}$ = *weighted*, and $\mathcal{B}$ = *consecutive*, the phase structure for string (8) is:

[**aaaaa aaaaa**] **aabbb** [**bbbbb bbbbb**]

---

the transitive property holds trivially because the strings $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_2$ have perfect similarity.

[6]The parameters orientation and chunking specify different properties. Chunking specifies how the input is broken up into chunks, and orientation specifies the order in which chunks have their similarity values computed. When breakup is *consecutive*, orientation is not necessary because *consecutive* is transitive.

Two phases are identified: one phase consists of all **a**'s and the other phase consists of al **b**'s. One phase transition, **aabbb**, is identified because the similarity value of chunk **aaaaa** and chunk **aabbb** is 0.4, which is below $\mathcal{T}$, and the similarity value of chunk **aabbb** and chunk **bbbbb** is 0.6, which is below $\mathcal{T}$. Nevertheless, the two **a**'s in the phase transition intuitively belong to the first phase and the three **b**'s in the phase transition intuitively belong to the second phase.

Phase boundary identification can be improved with an additional parameter, *Overlap Factor* ($\mathcal{K}$), that specifies how many elements overlap between the current phase and the next chunk, where $0 \leq \mathcal{K} < \mathcal{G} - 1$. If $0 < \mathcal{K}$ the candidate chunk overlaps with the elements in the phase that is being constructed, where the number of overlapped elements is $\mathcal{K}$. If $\mathcal{K} = 0$ the candidate chunk does not overlap with the elements in the phase that is being constructed. *Consecutive* breakup implicitly has $\mathcal{K} = 0$ because chunks are nonoverlapping.

When $\mathcal{K} = 3$, then the phase structure for string (8) is what would be expected:

[**aaaaa aaaaa aa**] [**bbb bbbbb**]

In general, $\mathcal{T}$ must be greater than $\mathcal{K}/\mathcal{G}$; otherwise no phase transitions will be detected because at least $\mathcal{K}$ elements in the next chunk are in the candidate phase by definition of overlap factor.

Notice that if $\mathcal{K} = 2$, then the phase structure for string (8) is:

[**aaaaa aaaaa a**] [**abbb bbbbb**]

7. RELATED WORK

Prior work can be categorized into phase shift detection algorithms [22, 14, 27, 28, 20, 15, 5], phase shift detection in other fields [16, 21, 9], and potential clients [10, 4].

7.1 Phase Detection and Related Algorithms

There are several examples of phase detection algorithm mentioned in the literature and they all form concrete instances of the general problem we discuss in this paper.

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 and described an algorithm that is similar to the one described in this paper. One of the parameters in their model is the size of the working set. 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 [14] 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 framework described in this paper. They employ an unweighted set 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 nonoverlapping 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 Figure 5, which show that the phases can be sensitive to the parameters.

Sherwood et al. [27] 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 partion the full profile in equal-sized intervals and then use techniques to compare these intervals to all intervals. They employ a weighted set approach and take the component-wise difference of two equalized sized 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 Figure 3 as their dissimilarity metric plus twice our similarity metric is always equal to 2.

Sherwood et al. [28] present an online version of the algorithm in [27]. 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.

Kistler and Franz [20] 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. This weighted set of information is viewed 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 perform similarity. They use a threshold of 0.95 to trigger. They conclude that the expense of their approach is ameliorated by computing it only once every five minutes.

Duesterwald et al. [15] 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 [5] 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.

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 [16]. 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. [9] use the maximum likelihood approach to change detection via the Bayesian information criterion, to detect change of speakers in broadcast news.

7.3 Dynamic Optimization Systems

Recent work has identified the need for phase shift detection in dynamic analysis.

Chilimbi and Hirzel [10] describe an online optimization system for dynamic prefetching based on data references profiles. To account for potential phase shifts, they periodically regather the profile. The trigger mechanism is based on a fixed duration, not on the profile data.

Arnold et al. [4] 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 regather a profile when a phase transition occurs, but do not provide an algorithm for detecting phase transitions.

8. CONCLUSIONS AND FUTURE WORK

This paper studies the fundamental problem of phase shift detection. It demonstrates that the problem requires the specification of two parameters (granularity and similarity) that define a phase, and shows how existing phase shift detection algorithms are instances of this problem. The paper also uses an intuitive instance of the problem to demonstrate that any algorithmic solution to this instance will produce the same result for the same input. Finally, the paper demonstrates that for the concrete problem that two intuitive parameters that define a phase are not "well behaved" and thus, changes in a parameter's value may result in significantly different phases that are identified in the program's execution with both examples and profiles from a set of Java benchmarks. This illustrates the importance of clients choosing appropriate parameter values when employing a phase detection algorithm.

Future work includes computing the correlation of chunk size when the values chosen are closer together than in Figure 8 to better understand the behavior of this parameter, identifying what concrete instance of the abstract problem should be used for various phase shift detection clients, and what parameter values are appropriate. Another topic is to identify other parameter properties, besides well behaved, that provide the ability to predict how the phase structure of strings may change when the parameter's value changes. Other topics include detecting recurring phases [28], studying the properties of other types of profiles, and exploring the tradeoffs between offline and online phase detection algorithms.

9. ACKNOWLEDGEMENTS

The authors thank David Grove for his help in designing the conditional branch tracing in Jikes RVM. We thank Kartik Agaram for suggesting a better presentation of the paper. We thank Laureen Treacy for proofreading 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] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, Las Vegas, Nevada, 15–18 June 1997. *SIGPLAN Notices* 32(5), May 1997.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[4] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 2002.

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[6] Thomas Ball. The concept of dynamic analysis. In *ACM SIGSOFT 1999 Symposium on the Foundations of Software Engineering*, pages 216–234, September 1999. Jointly held with 7th European Software Engineering Conference.

[7] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[8] Ronald D. Barnes, Erik M. Nystrom, Matthew C. Merton, and Wen mei W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *the 35th International Symposium on Microarchitecture*, pages 233–244, November 2002.

[9] S.S. Chen, E. Eide, M.J.F. Gaels, R.A. Gopinath, D. Kanvesky, and P. Olsen. Automatic transcription of broadcast news. *Speech Communication*, 47:69–87, 2002.

[10] Trishul Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 199–209,

June 2002.

[11] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 258–269, June 2002.

[12] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[13] 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.

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

[15] 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.

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

[17] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[18] 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.

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

[20] Thomas Kistler and Michael Franz. Continous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[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] Darko Marinov and Robert O'Callahan. Object equality profiling. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2003.

[24] M. Paleczny, C. Vic, and C Click. The Java Hotspot(TM) server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, April 2001.

[25] Thomas Reps, Thomas Ball, Manuvis Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ACM SIGSOFT 1997 Symposium on the Foundations of Software Engineering*, pages 432–449, September 1997. Jointly held with 5th European Software Engineering Conference.

[26] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

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

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

[29] Kevin Skadron, Margaret Martonosi, David I. August, Mark D. Hill, David J. Lilja, and Vijay S. Pai. Challenges in computer architecture evaluation. *IEEE Computer*, 26(8):30–36, August 2003.

[30] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 180–195, October 2001.

[31] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

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