McGill University
School of Computer Science
Sable Research Group

A Survey of Phase Analysis: Techniques, Evaluation and Applications

Sable Technical Report No. 2006-1

Dayong Gu and Clark Verbrugge
{dgu1, clump}@cs.mcgill.ca

March 10, 2006

Abstract

Most programs exhibit significant repetition of behaviour, and detecting program phases is an increasingly important aspect of dynamic, adaptive program optimizations. Phase-based optimization can also reduce profiling overhead, save simulation time, and help in program understanding. Using a novel state space characterization, we give an an overview of state-of-the-art phase detection and prediction techniques. We use this systematic exposition to situate our own long-range phase analysis, and evaluate our approach using a novel phase prediction evaluation metric. Our results show that phases with relatively long periods can be predicted with very high accuracy. This new technique, as well as the unexplored regions in our high level characterization suggest that further improvements and variations on program phase analysis are both possible and potentially quite useful.

1 Introduction

It is well known that the behavior of a program is not random. A typical program performs similar work, loads similar resources, and in general shows stable performance over signifi cant periods of time. Most programs are also quite repetitive, with similar behavior occurring cyclically throughout the whole execution. Detecting the repetitive portions of program execution is the process of *phase detection*. Phase detection technique can be used to capture the beginning of relatively stable executions, and also to identify the repetitive cycles in the whole program execution. Both of the these properties are valuable for doing adaptive optimizations, reducing profi ling and simulation overhead, applying system reconfi gurations, improving program understanding, and so on.

Of course successful application requires a good understanding of the form of phase detection being offered; a number of phase detection approaches exist, based on a variety of different phase properties. Scientifi c and computationally-intensive applications may benefi t more from stable phase prediction techniques than irregular applications based on dynamic data structures. In this paper we explore a systematic classifi cation of phase detection and prediction techniques. This is intended to organize phase analysis works and thus aid appropriate selection, as well as expose gaps in the phase analysis state space exploration. We further develop a quantitative metric for evaluation of repetitive phase prediction. Previous work on evaluating phase detection has concentrated on fi xed-length phase intervals; our metric applies to variable-length, long range phases, and we give experimental evidence of its use on standard Java benchmarks.

Specifi c contribution of this paper include:

- We summarize a variety of popular program phase detection techniques, and present a structured *solution space* to organize the different approaches. Similarities and differences among the techniques provide direction for future work in phase analysis.
- We give a precise defi nition for variable periodic phase length and present a new set of evaluation metrics for long range periodic phase detection.
- Using a new long-range phase detection technique we give experimental evidence of both the existence and the effi cacy of periodic phase detection.

In the next section we present motivation for phase detection, listing the important application areas. Section 3 gives basic defi nitions and defi nes our solution space; section 4 then examines various representative phase detection approaches and positions them in the solution space. This is followed by an exposition of the evaluation metrics in section 5. Finally, we summarize and conclude in section 6.

2 Motivation

Phase detection and prediction techniques focus on discovering the flat, stable portions or the repetitive portions in program execution. The phase information obtained is valuable in various areas including program understanding, debugging, reducing simulation and profiling workload, system reconfiguration and adaptive runtime optimizations. In this section we briefly give descriptions of how phase information can support these goals.

- **Program understanding and debugging**

  Phase detection techniques can determine the boundaries of each sub-portion of the program execution. Such results can be used to analyze the workload of a program at different stages, locating bottlenecks and detecting program defects at a finer granularity than the whole program scope. A. Georges *et al.* [14], associate the major workload of a program with representative methods. By measuring hardware events only for these selected methods, hardware related performance bottlenecks can be located with much less effort. Compile-time data reordering frameworks can also benefit from phase information mapped to static program structures, by focusing optimizations within the actual critical areas [41].

- **Reducing simulation and profiling workload**

  Program simulation, especially on accurate, cycle-level hardware simulators, can be quite time-consuming. It is very worthful to select the crucial simulation periods to model, and thus save a large portion of the total simulation time. Phase detection techniques can be used to help simulators find the interesting points to simulate. Sherwood et al., for example, use phase detection techniques to determine the portion of execution to simulate and to guide computer architecture research [34].

  Similarly, workloads for both offline and online profiling can be reduced by only sampling representative parts selected by phase detectors. This also benefits trace size; many profilers can generate huge traces, and phase detection can also function as a lossy compression solution to the trace files that attempts to preserve the most meaningful information. W. Liu *et al.* demonstrate the use of phases for reducing profile cost by giving a phase-driven simulation mechanism that can obtain acceptable accuracy while only simulating a small portion of the code [24]. In the case of online profiling, reductions in sample content and frequency have been recognized as important; various authors mention that optimizations based on runtime profiling need to be applied judiciously, or the cost will outweigh the benefit in many situations [1, 3, 21, 26].

- **System reconfiguration**

  Embedded or mobile systems often have demanding resource requirements, it can be very valuable to reconfigure the system dynamically to minimize resource consumption. Dhodapkar and Smith, for example, introduce tuning points based on phases; these are selected to save power and improve overall performance by enabling or disabling resources adaptively [10]. Similarly, the phase detection technique introduced by Shen et al. has been shown to be effective in adaptive cache resizing and memory remapping [32]. Trade-off's between speed and energy use of a system based on phase information have also been explored [4, 9, 19].

- **Adaptive optimizations**

  Runtime, adaptive optimization is the most exciting application of phase detection, and many adaptive systems are built on determining and exploiting phases. The SOFTWARE CODE TRACE in Dynamo,

for example, is refreshed based on monitoring the generation rate of new CODE TRACES in recent intervals [20]. In fact, this is a type of phase detection, and most systems that attempt to locate "hot" code based on runtime data can be seen as phase detectors. M. Arnold *et al.* [2] give a survey of adaptive optimization techniques, especially in a virtual machine environment. Many techniques introduced in that work can get benefit from phase information.

3 Definitions

An initial step in phase detection is to come up with criteria for deciding when phases begin and end: what is a phase? To a large extent this can be seen as a choice between different thresholds for similarity when examining program execution; a phase is sequence of similar actions, as seen from a particular sampling perspective. This viewpoint does not capture all of the potential phase designations, as we will argue below, but is sufficient for a general exposition of phases based on stable behaviour.

M. Hind *et al.* give a basic classification [17] of phase detection. Formally, they define $\mathcal{PD}[\tau, \sigma](\mathcal{P})$ to be the abstract definition of a phase detection problem that takes a profile string $\mathcal{P}$ as input. This is then parameterized according to the available thresholds:

- *Granularity*(τ) specifies how a profile is partitioned into fixed-length, atomic units of comparison, denoted *chunks.* Granularity size is also the minimum size of a detectable phase.
- *Similarity*(σ) is a boolean function that, give two strings, determines if the two strings are similar. That is $\sigma_1(S_1, S_2)$ returns *true* if S_1 is similar to S_2, and *false* otherwise. Using continuous output (eg the interval $[0.0, 1.0]$) from such a function can provide detail on relative similarity, although a binary decision must be made at some level.

Using this model, they take two input strings (traces), convert each string into an abstract representation, and compute the similarity between the abstract representations. They then give a generic algorithm based on this model and demonstrate it on a simple alphabet string example.

The above approach, and its specific instantiations, are in fact based on recognizing *stable* phases. A stable phase can be defined as above, or more abstractly as a maximal length sequence of consecutive intervals containing no large performance change. Such definitions are very appropriate for identifying phases in programs in which long sequences of unchanging behaviour occur frequently. Scientific benchmarks, for instance, tend to exhibit such activity, and studies of the SPECcpu95 [39] and SPECcpu2000 suites [38] show the utility of this kind of phase definition [10, 23, 24, 33, 35].

Phases, however, are not completely characterized by this definition, and fixed-length interval approaches have been recognized as potentially inadequate for many programs. If the length of the interval is not appropriately set the intervals can easily step "out of sync" with the intrinsic periodic behaviour of a given program. Lau *et al.* [22] illustrate this problem graphically and give basic motivation for considering *variable length* intervals in program execution.

Fixed interval approaches also suffer from reduced accuracy in the case of larger granularities. When studying the program behaviour at a large scale, fewer long, flat execution sequences will appear—adjacent large intervals are likely to be quite different. Many programs, however, do have significant periodic and repetitive portions, even at coarser scales. To capture this form of phase we augment the above *stable* phase definitions with a definition of *long-range periodic* phases:

Definition 1 *A* long range periodic phase *is a tuple* $\mathcal{P} < \alpha, \theta, \delta >$ *where,*

- α *is a set of segments* $S_1, ..., S_n$ *appear in program execution and n* > 1*.*
- θ *is a function computes the correlation between each two items in* α*.*
- δ *is a threshold. For arbitary two items* S_x, S_y *in* α *where* $1 \leq x, y \leq n$*, the following inequation must be hold:* $\theta(S_x, S_y) > \delta$

More simply, periodic phases are repetitive patterns in program execution. This approach can be applied recursively, allowing multiple level repetitive program behaviours to be addressed.

Solution Space

In order to more fully understand phase detection approaches, fixed-interval, variable interval, long-range and others, we here provide a general *solution space* that characterizes different strategies. This three-dimensional space allows us to position a selection of representative techniques in order to make their similarities and differences more obvious. The three axes of our phase solution space are defined as follows:

- **Dimension 1: Phase scale**

  This dimension represents the kinds of basic units a detection technique uses to identify phases. These units can include *static program structure* (**SP**), *Fixed intervals* (**FI**), *Variable intervals* (**VI**), and *Long range* (**LR**) phases.

- **Dimension 2: Data type**

  This dimension reflects the type of data that a detection technique uses to analyze the program behaviour. Discrete points in this dimension are associated with *high level software data* (**H-Sf**), *Low level software data* (**L-Sf**), *mixed software/simulated hardware data* (**Sf+S.Hw**), *simulated hardware data* (**S.Hw**), and finally actual *hardware data* (**Hw**).

  High level software data refers to large scale software structures such as loops, procedures or basic blocks. Low level software data refers to the smaller scale software structures such as instructions.

- **Dimension 3: Application time**

  A phase analysis can be applied to a program data acquired at different stages of execution. In this dimension we represent whether the phase detection is based on *static analysis* (**St**), pure *offline* (**Off**) trace data, *offline/online mixed* profiles(**Off+On**), *simulated online* data (**S.On**), or *online* from actual executions (**On**).

The above dimensions form a discrete 3-D space of different phase detection approaches, as shown in Figure 1 . To further separate different approaches we also represent pure phase determination techniques using a ball in our graph, and phase *prediction* techniques using a circle.

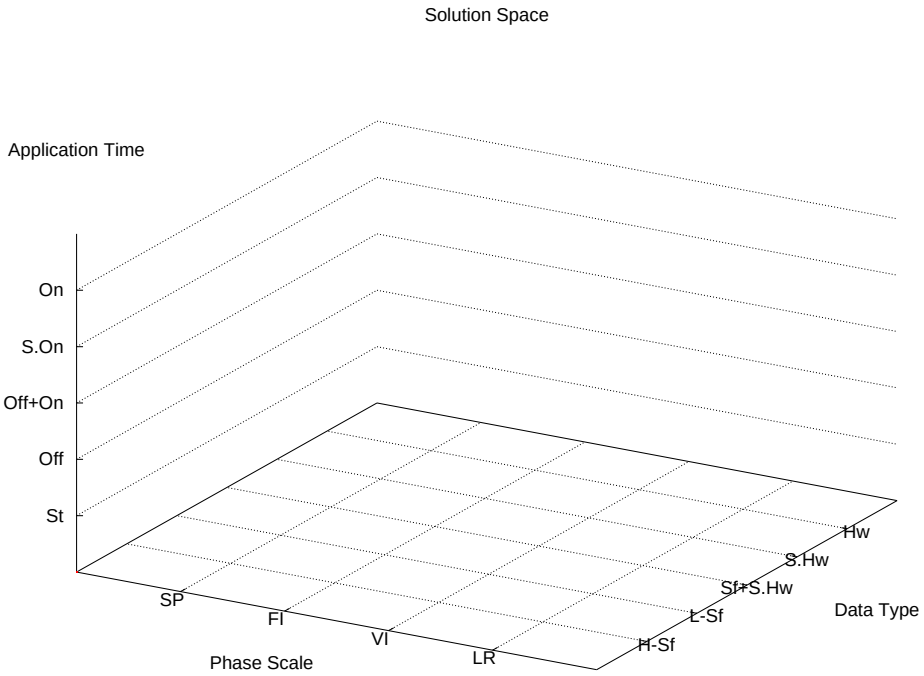

Figure 1: The empty *solution space* for phase analysis categorization.

4 Phase detection and prediction techniques

In this section we introduce a set of representative phase detection techniques, positioning each in the solution space defined in the previous section. We follow the "phase scale" axis, incrementally describing techniques based on static program structure, fixed length interval, variable length interval and long range periodic phase, as they intersect with our other axes. We will partially fill the empty space shown in Figure 1, ultimately producing Figure 11. For reference the coordinate values of the solution space detailed in the last section are summarized in table 1.

4.1 Static program structure based

Phase, inherently, is a dynamic issue. Due to the complexity of the program runtime behaviour it is hard to discover runtime phases from static program analysis. Nevertheless, for some types of program, such as scientific code with quite regular and easily predictable behaviour, it is possible to form a usable analytical model of the runtime program performance.

Fraguela *et al.* [13], for instance, present a compiler tool to predict the memory hierarchy performance of scientific programs, based on an analytical model. The performance of set-associative caches with a particular least recently used *(LRU)* replacement policy is modelled and predicted under loop-oriented workloads. *Probabilistic Miss Equations* (**PME**s) are used to estimate the cache penalty of each memory reference in each loop. Their model can be used to guide compiler optimizations, and they validate their work with trace-driven simulations, on different hardware platforms using the SPECfp95 [39] benchmark suite.

PHASE SCALE	
SP	Phase composited of static program structure
FI	Phase composited of fixed length intervals
VI	Phase composited of variable length intervals
LR	Phase composited of long range periodic portions
DATA TYPE	
H-Sf	Based on high level software data
L-Sf	Based on low level software data
Sf+S.Hw	Based on mixed software data and simulated hardware data
S.Hw	Based on simulated hardware data
Hw	Based on actual hardware data
STATIC/OFFLINE/ONLINE	
St	Purely static analysis solution
Off	Offline solution
Off+On	Offline and online mixed solution
S.On	Simulated solution aim at online application
On	Online runtime solution

Table 1: The position coordinate values of *Solution Space*

There are various similar approaches based on static analysis [7, 15, 43, 44]. These works all model, and thus in some sense detect repetitive program behaviour analytically. Precisely speaking, these are not typical phase detection technique, focusing more on controlling and generating stable phase behaviour. In the solution space we thus show it as a "cup", as opposed to a "ball" for a typical phase detection work or a "circle" for work aimed at phase prediction.

With respect to the solution space axes, these efforts all build phases based on pure static program structure. They combine data from the software structure and a simulated model of hardware performance, and are purely static analysis works. Using **PME** as a representative name, the collective coordinate of these techniques is [**SP** (static program structure), **Sf/S.HW** (software and simulated hardware data), **St**(static analysis)].

Of course not all static approaches are analytical. A. Georges *et al.*, for example, have developed an analysis for detecting "method level phase behavior in Java" [14]. The authors develop an offline analysis technique for Java workload composed of three steps. In the first step, the execution time is measured for each method invocation. Using an offline tool they then analyze the dynamic call graph to identify phases corresponding to method executions. Methods that take a large portion of the whole execution time but which have a less frequent invocation count are then candidates for major method level phases. Figure 2 show a simple example of a method level phase result based on the data shown in table 2. All the orange (shaded) nodes are identified as phases.

In Georges *et al.*'s implementation, runtime performance for each of the selected phases is measured using hardware event counters. This, however, is less of a combination of high level program structure with microprocessor-level and more of a simple mechanism for accurate timing; phases are also computed offline. The coordinate of this work is then [**SP**, **H-Sf** (high level software data), **Off** (offline)]. In the solution space we denote this work by **MLPJ**.

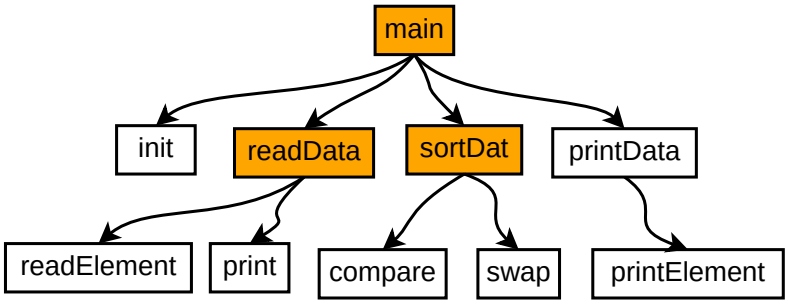

Figure 2: Method level phases determined by measuring execution time and invocation frequency (modified from Figure 3 in [14]).

Method name	information		
	total time	time/call	calls
main	1800	1800	1
init	30	30	1
readData	300	200	1
readElement	200	4	50
print	30	30	1
sortData	1300	1300	1
compare	600	2	300
swap	500	2	250
printData	170	170	1
printElement	150	3	50

Table 2: Method execution data from [14]

4.2 Fixed length Interval detection and prediction

A large number of phase detection techniques are based on fixed length interval data. They share a common style:

- The program execution is divided into fixed length intervals by some means.
- Specific profiling data is measured in each intervals.
- If the difference between the profiling data of two consecutive intervals is larger than a predefined *threshold* a phase transition point is detected.

The fundamental detection logic for these works is more or less similar. The differences among them are in how the profiling data is selected, how the data is organized, how the threshold is set, and what kind of comparison algorithm is used. Below we address fixed length interval approaches by dividing them into two major approaches, pure *detection* works, and techniques that aim at phase *prediction*.

4.2.1 Detection

Phase detection efforts are built on a variety of different basic structures and data sources. High and low level events of different forms have been considered, and both online and offline techniques have been developed.

Sherwood *et al.* make use of moderately high level program structure by using *Basic Block Vectors* (BBVs) to detect phase changes [33]. A BBV is an array with an entry for each static basic block in the program. BBVs are used to track the execution frequency of individual basic blocks; the value of an array entry is simply the number of times that a given basic block has been executed during a given interval. Phase changes are detected when the *Manhattan distance*

$$\Delta_{i,i-1} = \sum_{j=0}^{\infty} |BBV_i[j] - BBV_{i-1}[j]|$$

between consecutive intervals i and $i-1$ exceeds a predefined threshold Δ_{th}. In the published paper this technique is applied to the problem of selecting crucial simulation points.

Following a more low level perspective, A. Dhodapkar and J. Smith use the *Instruction Working Set* to detect phase transitions [10]. This allows the computation of a *relative working set distance*

$$\delta = \frac{|W(t_i,\tau) \bigcup W(t_j,\tau)| - |W(t_i,\tau) \bigcap W(t_j,\tau)|}{|W(t_i,\tau) \bigcup W(t_j,\tau)|}$$

where a *working set* $W(t_i,\tau)$ for i=1,2,..., is a set of distinct *segments* $s_1, s_2, ..., s_\omega$ touched over the i^{th} *window* of size τ. "Segments" here are memory regions of fixed size (*e.g.*, pages). The instruction working set is hashed into an n-bit vector, the *working set signature*. Combined with a suitable threshold, the distance between working set signatures over time is then the basis for a fixed interval phase analysis.

Another low level data choice is provided by Balasubramonian *et al.* who use *conditional branch counts* as the monitoring data [4]. They use a counter to measure the number of dynamic conditional branches executed over a fixed execution interval. In their scheme, no fixed threshold is set; instead the detection algorithm dynamically varies the threshold throughout the execution of the program. This work is based on a cycle-level hardware simulator, SimpleScalar [5], interacting with the phase detection scheme. Phase

analysis is used to determine whether the current state is *stable* or *unstable*. If in the latter case, simulated reconfiguration hardware adaptively adjusts to the new state of the program.

All three of these works are based on fixed length intervals and so fall on the same point of the phase scale axis in our solution space. BBV uses pure high level software data and offline computation, while the second two are online techniques and use low level data—software in the case of *instruction working set*, and (simulated) hardware in the case of the *conditional branch count* approach. Coordinates of the three techniques thus are:

- Basic block distance analysis (**BBDA**): [**FI** (fixed interval), **H-Sf**, **Off** (offline)]
- Instruction working set (**InstWS**): [**FI**, **L-Sf** (low level software), **S.On** (simulated online)]
- Conditional branch counting (**CBrC**): [**FI**, **S.Hw** (simulated hardware data), **S.On**]

4.2.2 Prediction

Detection techniques work in a *reactive* manner; program behaviour changes are observed only after the occurrence of the change. This delay is minimally one interval long, often much more in order to achieve good confidence of stable behaviour. However, if the behaviour changes can be *predicted* the delay between observation and reaction can be reduced.

Prediction techniques can be roughly divided into three types:

Analytic model predictors. The analytic techniques discussed in section 4.1 use a static model to represent the actions of execution kernels, and thus can also give a prediction of future behaviour. This type of predictor can work well on programs where the behaviour can be easily modularized, such as scientific computations. For general purpose programs, however, it is not practical to build an accurate analytic model,

Statistical predictors. Simple statistical predictors can be used to estimate of future behaviour based on historical behaviour [12]. Many statistical predictors have been developed, including (among many others):

- *Last value* predictors assume the next value of a memory location or computation is the same as the last. This approach works well within a stable phase, but not in phase transitions or more complex phase behaviour.
- *Average(N)* predictors uses the average of the last N intervals as the predicated value for the next interval.
- *Freq(N)* predictors chooses the most frequent value in the last N intervals as the prediction for the next interval.
- Exponentially weighted moving average *(EWMA(N))* predictors place more emphasis on the most recent history, weighting a historical value's contribution to a predicted value by an exponential function of age.

Statistical prediction strategies have been widely used in optimizations based on *(return) value prediction* [6,16,28,29]. Hu *et al.*, for instance, present a parameterized *stride predictor* and give return value prediction

data for SPECjvm98 benchmarks on simulated hardware [18]. In general a variety of strategies can be applied to estimate single values from related historical data; most are based on exploiting stable phases, but stride, context and a few other predictors can provide small scale "phase" detection for individual variables.

Table-based predictors. Extending the basic idea of *statistical predictors*, *table-based predictors* predict values using information other than just the most recent history. This approach has been applied to create a memoization predictor for return value prediction [29], but can also be applied to predict phases. In general these techniques encode a current state as well as history as the index into a prediction table. The prediction of the future is stored in the table and can be updated when large behaviour changes are identified. The differences between individual implementation can be:

- What type of data is used to build the prediction
- What is the detailed construction and organization strategy of the historical data
- What algorithm is used to create the index into the prediction table
- What kind of a mechanism is used to update the predicted value in accordance with the most recent measurement

E. Duesterwald *et al.* give a general study on predicting program behaviour [12]. A set of predictor models of both statistical and table-based types on fixed size intervals are introduced and compared. Their experimental results show that table-based predictors can cope with program behaviour variability better than statistical predictors. This work uses hardware data from Power3 and Power4 architecture on SPECcpu2000 [38] benchmarks. We denote this work by *GenPred* and assign it a coordinate in the solution space of [**FI**, **Hw** (hardware data), **S.On**].

T. Sherwood and S. Sair [35] present a *Run length encoding phase prediction* (**RLEP**) technique using low level branch data. First, a phase ID is built for each interval based on its *footprint* for the executed branches. As shown in Figure 3, the PC of a branch is hashed into an index of the accumulator table, and the number of instructions executed are added into the corresponding entry. After the execution of an interval the most significant parts of the accumulator entries are combined to construct the *footprint* of this interval. If the *footprint* is "unique" enough according to their definition a new phase ID is assigned to this interval.

In a subsequent step the phase ID of the current interval and the number of consecutive repetitions of the phase are hashed into the prediction table to find the phase for the next interval. This process is shown in figure 4.

Similar general strategies have been followed in other work [23], and as a representative the coordinate of the **RLEP** predictor is [**FI**, **S.Hw**, **S.On**].

4.3 Variable intervals

In the previous section we introduced a collection of representative, state-of-the-art phase detection techniques. These all share the same property of splitting the execution into fixed length intervals, and detecting behavioural differences by observing noticeable variations according to predefined measurement metrics between consecutive intervals. All those techniques are able to perform satisfiably in a certain set of situations.

In a recent paper, Lau *et al.* point out that fixed lengths can become "out-of-sync" with the intrinsic period of the program [22]. This problem can make a program's periodic phase behaviour difficult to detect using

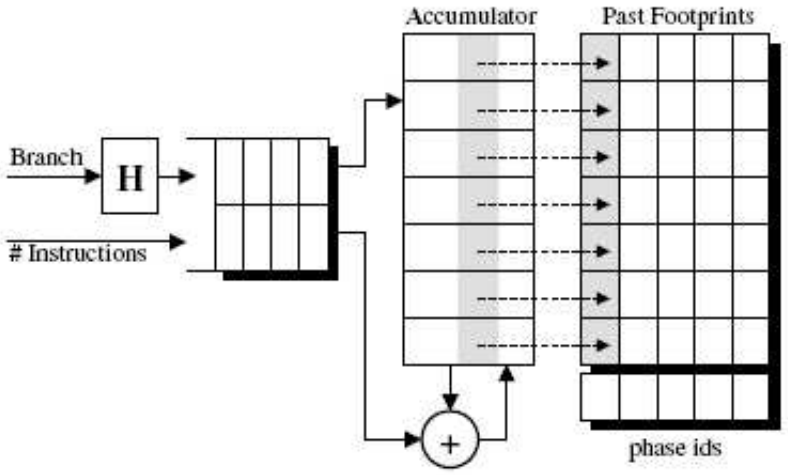

Figure 3: RLEP: Building the phase ID from the branch footprint in [35].

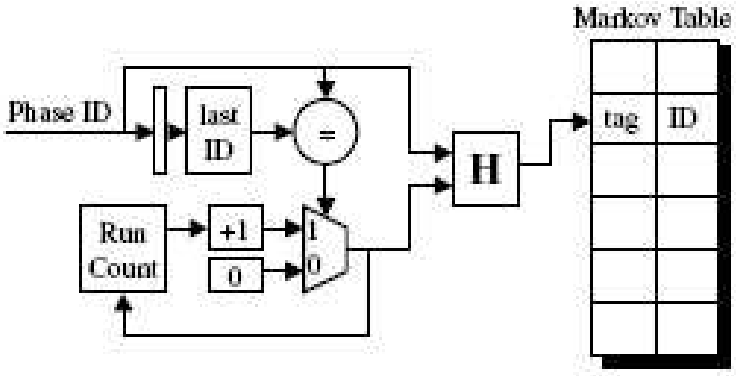

Figure 4: RLEP: Using phase ID and the number of repetitions to predict the next phase in [35].

fi xed length interval solutions, and they graphically show that variable length intervals are necessary in some situations. Figure 5 from [22] show a simple example of how the fi xed length interval solutions can fail in capturing the actual phase because of asychronization. A sinusoid signal is shown in the top fi gure. Two unsuitable fi xed interval division are provided in the lower two fi gures. The average value of the intervals are shown by the solid lines. It is clear that no obvious repetitive features of the input sinusoidal curve are preserved in the lower two fi gures.

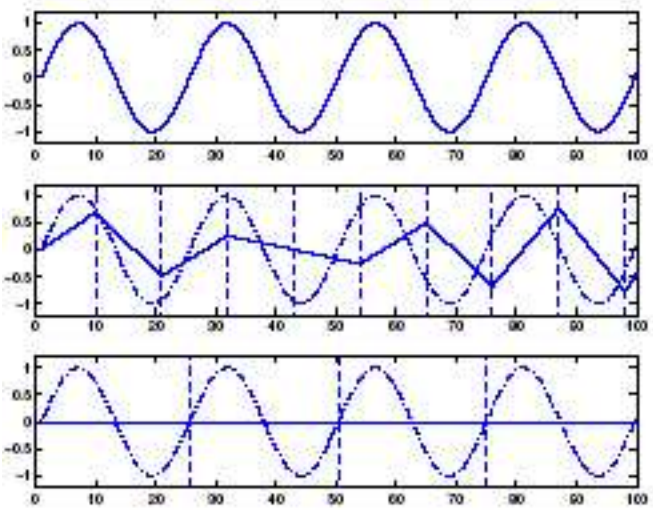

Figure 5: Demonstrating the synchronization problem due to using fi xed length intervals [22].

Lau *et al.* also graphically demonstrate that there are multiple levels of phases in programs that current fi xed length interval techniques cannot handle at all. This motivates an initial study on variable interval phase detection using the SimPoint simulator [37]. Programs are instrumented with ATOM [36] to generate traces of each loop branch, method call and method return. Based on these traces, they construct a hierarchy of variable length interval phases using *SEQUITUR*, a linear-time, context free grammar algorithm that infers a hierarchical structure from a sequence of discrete symbols [27]. *SEQUITUR* recursively replaces repetitive sequences with a grammatical rule that can generate the sequence. This result is a hierarchical representation of the original sequence that can offers insights into its lexical structure. An example is shown in Figure 6.

- **S := BBA**c
- **B :=** A**b**
- **A :=** aa

Figure 6: Grammar generated for the input “aabaabaac” by *SEQUITUR* from [22].

Although their work is still at an early stage the main contribution of this work is important. They show that programs have a hierarchy of phase behaviour at many different levels of granularity, and point out limitations of traditional fi xed length interval solutions. Lau *et al.*’s work is an offline work based on high level software data; we denote it in the solution space as **MotVL** and assign it the coordinate [**VI**(variable interval), **H-Sf**, **Off**]. This work defi nes a point in a largely empty sub-space of the whole solution space.

4.4 Periodic long range phases

The synchronization problems that motivate variable length intervals can theoretically be handled by choosing an appropriately short interval length, of course with a inversely proportionate increase in overhead costs. This makes variable length interval approaches more of an optimization to existing stable phase detection techniques, and does not change the basic kinds of phases detected.

For more irregular programs, however, stable phase behaviour is only visible at extremely fine scales. Figure 8 shows data from the JACK benchmark of SPECjvm98 [40]. Here stable phases are only evident when greatly magnified; there are no significant periods of stable activity. Yet the program clearly does show a large scale *periodic* phase behaviour. This motivates the value of coarse granularity, *long range periodic phase* approaches. Below we first describe our own approach to this problem, and then discuss other related techniques.

Our approach to long range periodic phases involves both detection and prediction, using the hardware event data commonly available in modern processors. Our intention is to apply this online, although our current prototyping results are based on a simulation of online processing.

The algorithm works by capturing and associating the beginning *pattern* of each periodic phase with the phase itself. Thus the initial patterns predict the length of upcoming periodic phases. Several heuristics are used due to the fairly complex behaviour of popular Java benchmarks (SPECjvm98 [40] and DaCapo [30]). These heuristics include:

- We capture variations at multiple levels of granularity
- We measure confidence in our predictor patterns dynamically, adjusting weights and associations automatically at runtime based on the real performance data
- We use *data-rechecking* to confirm or invalidate patterns
- We use a *re-synchronization* scheme to avoid phase shifting problems due to minor variations in periodicity

Figure 7 is a simplified activity diagram showing how these heuristics cooperate to improve overall prediction *confidence*. In this figure we use *P(val, length)* to represent a particular prediction with a validation value *val* and a predicted phase duration *length*. The *R(val, length)* is used to denote the real *val* and *length* in the observed, future phase repetition. A repetition will only be used to give prediction after we identify that it does occur regularly; that is, we have enough confidence in it. The confidence will be strengthened if it is confirmed by later data and decreased otherwise. The length of the repetitive period is adjusted gradually based on the result of a "length rechecking" scheme we use for re-synchronization.

Figure 9 shows a sample result from our work on JACK in predicting the L1 instruction cache miss data. The actual program execution is represented by a red curve, and the simulated results are composed of two parts, a *traced* part and a *predicted* part represented by green and blue curves respectively. Traced results represent the learning period of the algorithm, when confidence in phase prediction is low, whereas the predicted part represents actual estimations of future behaviour. These results demonstrate that our solution performs well after the initial learning period, with most major features quite accurately predicted in the latter half of the program. This result is typical of the Java benchmarks we have investigated.

Our work is designated **PreOnline**. It is a simulated online solution, focusing on long range phases, and is based on real hardware data; this gives it a coordinate of [**LR** (long range), **Hw** (hardware data), **S.On**].

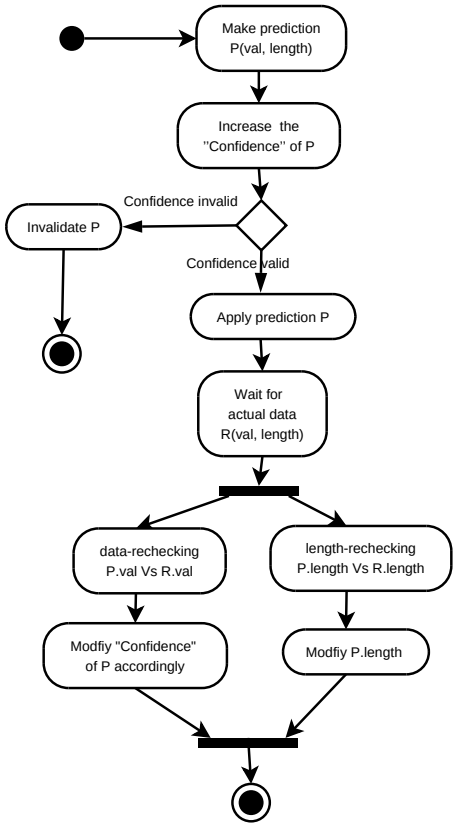

Figure 7: Improving prediction confi dence in long range periodic phase analysis.

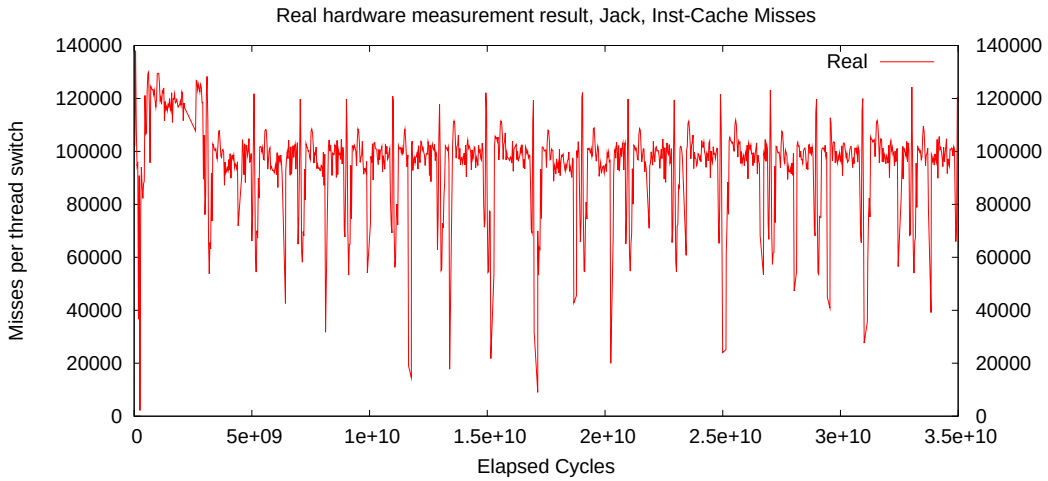

Figure 8: The obvious repetitive behavior of JACK at a coarse granulatiry, L1 instruction cache miss counts are gathered every thread context switch

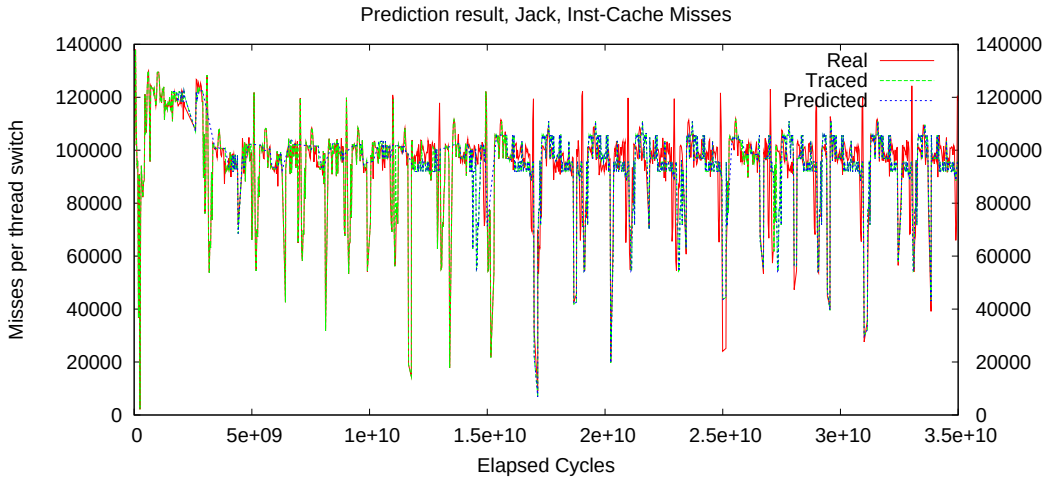

Figure 9: Periodic long range phase prediction results on JACK

Of course long range phases have been considered by others. Shen *et al.*, for example, detect long range phases using somewhat different techniques than discussed so far [32], basing their analysis on a trace of *reuse distance* data from programs. *Reuse distance* is defined as the number of distinct data elements accessed between two consecutive references to the same element. Certainly, reuse distance is not a fixed length interval and can cover a large portion of the program execution. They use a *discrete wavelet transform* [8] as a filter to remove the least significant changes and locate the most important ones. *Phase markers* in the code are then inserted using ATOM [36] to indicate phase transitions determined by this offline processing. As an example, the phase markers made on the TOMCATV benchmark are shown in Figure 10. Shen *et al.* apply their phase analysis to "cache resizing," and test their work on the Cheetah cache simulator [42]. Simulation data suggests this phase analysis can help considerably, reducing cache size up to 40% without significantly increasing the number of cache misses.

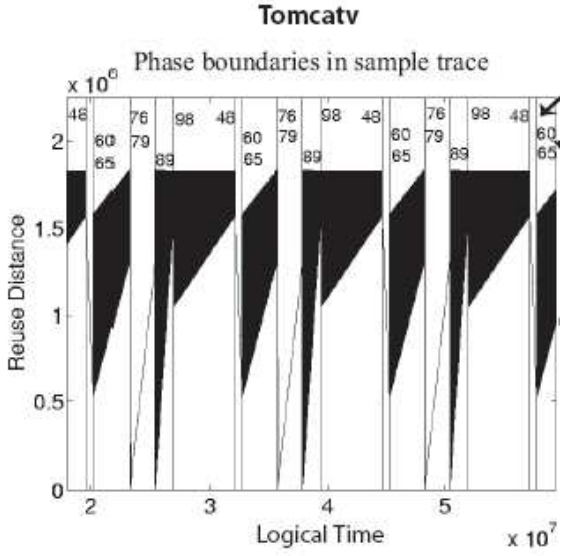

Figure 10: Phase markers based on reuse distance are inserted into TOMCATV [32].

We refer this work as **LocalityPD**. It is an offline analysis (*Discrete Wavelet Transform*) and simulated online (*phase markers*) mixed solution. *Reuse distance* is considered high level software data, and this work aims at long range periodic phases. The coordinate of **LocalityPD** is thus [**LR**, **H-Sf**, **Off+On**].

In fact, the same fundamental methodology employed by Shen *et al.* can be used on various trace data, including hardware events. Recently, they presented an extension of their work to hardware trace data, such as IPC and cache hit rates [31]. This latter effort applies only to a reduced subset of highly input-driven programs, named "service application" by the authors. The technique, however is general, and so we denote this work as **SerApp** and assign it the coordinate [**LR**, **Hw**,**Off+On**].

As a summary, all the phase detection techniques and their positions in the solution space are shown in Figure 11. For clarity we also provide isometric projections onto each of the three orthogonal planes in Figure 12.

From these figures we can observe that people have attempted to detect phase behaviour from different directions. Historically, work has concentrated on the measurement of fixed length intervals. More recent work, however, has attempted to go past the fixed length limitation to variable length periods, which are more adaptive to the inherent period of a program. Software-related data is widely used, but hardware related data is getting more and more attention, primarily for its tighter relation to performance and (typically) lower collection cost. Application time of phase techniques varies significantly; we have static techniques, offline, multi-pass analyses on runtime trace data, and so on. Although real online detection implementations are still lacking, many simulated online works have been developed. These simulated online analysis works only process runtime trace data in one pass; this makes them inherently easier to port into actual online applications.

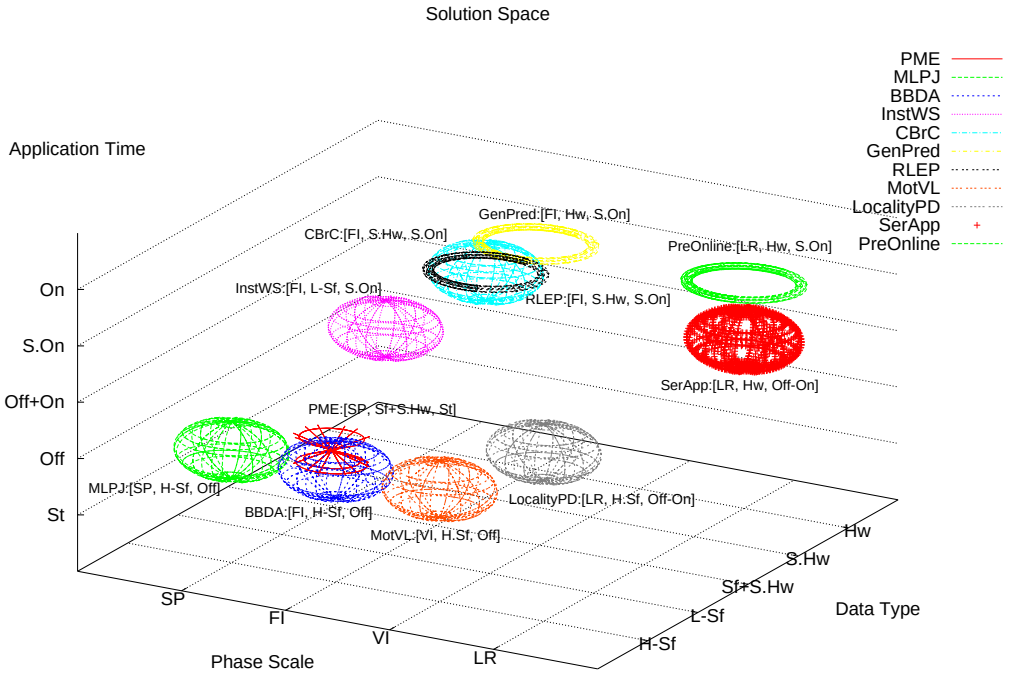

Figure 11: A summary of phase analysis techniques in the *solution space*

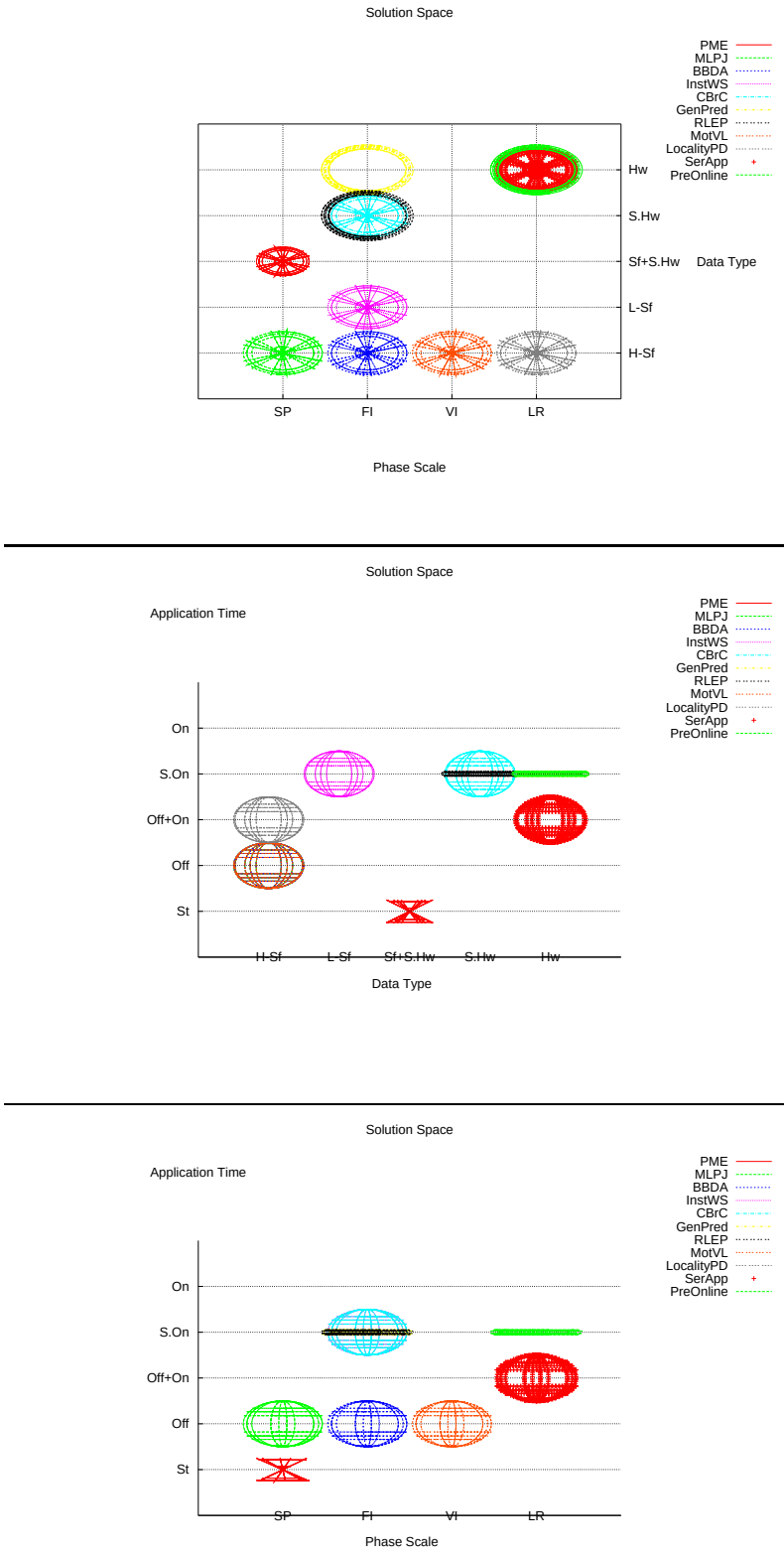

Figure 12: Phase analysis techniques in the *solution space*, projected onto the 3 planes.

5 Evaluation metrics

As well as organizing phase analysis approaches, it is important to develop techniques for evaluating and ultimately comparing them: we need a set of well defined metrics. In this section, we will give an introduction to the widely used evaluation metrics for fixed length interval phase techniques. We also propose a set of metrics aimed at evaluating long range periodic phase works, and use our own algorithm and results as an example of applying and understanding these measurements.

5.1 Fixed length interval phase evaluation metrics

Most phase detection works are based on fixed length intervals, and consequentially a set of comparatively mature metrics has been developed. Below we describe the most widely used phase detection evaluation metrics:

- **Sensitivity and False Positives.** A simple measure of phase detection efficacy is to consider how often an algorithm correctly identifies phases [35].

  Sensitivity measures the ability of a phase detection mechanism to identify a phase change after there is a significant performance change. It is defined as the fraction of intervals showing significant performance changes with respect to the preceding interval over all intervals.

  The *False positive* rate is the fraction of intervals where the performance shows no significant change, but is nevertheless claimed as a phase transition by the detector. Both these measurements are obviously quite dependent on the definition of significant.

- **Transition correlation and accuracy score.** Nagpurkar et al. propose an evaluation strategy based on a theoretical perfect phase detector, giving a "correct" phase boundary solution for a particular program's execution [25]. By comparing the results of the perfect detector and a given, real detector they define the *transition correlation* as

  $$\text{Correlation} = \frac{\text{bothInPhase} + \text{bothInTransition}}{\text{totalEvents}}$$

  where *bothInPhase* is the total number of profile elements for which both detectors agree it is in a stable phase, and similarly *bothInTransition* is the total number of profile elements for which both detectors agree it is in a period of phase transition.

  In combination with *Sensitivity* and *False Positive*, they further introduce a novel accuracy scoring metric, defined as

  $$\text{Score} = \frac{\text{Correlation}}{2} + \frac{\text{Sensitivity}}{4} + \frac{\text{False Positive}}{4}$$

  The *Score* weights correlation equally with the sum of sensitivity and false positive. Obviously other weightings and combinations are possible as well.

- **Stability and Average Phase Length.** A better stable phase detector should logically detect more stable phases than other detectors. *Stability* is thus defined as the fraction of intervals that belong to a detected stable phase; a higher stability means a more complete coverage of the program. Similarly, *Average Phase Length* is defined as the number of intervals that are part of stable phases, divided by the total number of stable phases.

- **Performance variance**

  Shase detection techniques need be able to resolve stable phases in the presence of relatively small performance variations. A small *performance variance* in a stable phase is an indicator that the phase detector has figured out the phase boundary correctly [35]. A poor phase detection result will show a comparatively large performance variance within a phase due to the inclusion of more intervals than is strictly necessary. Of course the concrete definition of this metric must be considered in the context of the whole program variation, and so is highly application-specific.

- **CoV**

  CoV refers to the *Coefficient of Variation*, a statistical measure of standard deviation as a percentage of the average: *CoV = stddev/mean*. *CoV* can be used to compare different phase classification results. For stable phase detection, a lower *CoV* is desired; in an extreme case, all the intervals in a detected phase would have exactly the same value in the measurement data, resulting a *Cov* of zero, or perfect phase identification.

Dhodapkar and Smith apply the first three groups of metrics to compare three fixed interval phase detection techniques, **BBDA**, **InstWS** and **CBrC** [11]. They conclude that **BBDA** can provide higher sensitivity and more stable phases. However, **InstWS** yields 30% longer phases than **BBDA**, albeit with less stability within phases. **CBrC** can provide as good sensitivity as **BBDA**, but is less effective at detecting major performance phase changes.

The **Cov** metric is used in a number of works [22, 23, 35]. Lau *et al.*, for instance, compute the average **Cov** based on CPI (Cycle Per Instruction) for the SPECcpu2000 [38] benchmarks [23] and find results can vary 10%–15% depending on the different parameter settings of their phase detection mechanism.

5.2 Periodic phase detection and prediction metrics

Most of the metrics in section 5.1 are only effective for measurement of stable phases. For long periodic phase detection there are no widely used, relatively standard metrics. Based on attempts to evaluate our own work we therefore developed a set of metrics suitable for this purpose, directed in particular at measuring online prediction utility.

- **Traced Rate**

  In order to predict repetitive sequences some portion of the execution must be initially, and occasionally traced. This provides the basic historical data required to make future predictions, and validates the current predicted behaviour. The *traced rate* is thus defined to measure the tracing overhead in a predictive setting; it is calculated as the percentage of the traced portion of the whole execution (the complement is the *predicted rate*). For two similar prediction results a lower *traced rate* is desireable, indicating more extensive predictor success and less data acquisition overhead.

- **Level Coverage**

  Predicting behaviour to arbitrary accuracy is difficult. Fortunately, in many cases it is not necessary to make a precisely accurate prediction; a prediction within reasonable bounds is sufficient for an optimization decision. An adaptive optimizer, for example, that works on the basis of “hot path” profiling does not need perfectly accurate information—an accompanying predictor need not determine the exact frequency of each execution path as long as it can correctly predict which ones are hot. In

our case we do not need to predict hardware event counts to precise binary equivalence, some number of trailing bits can be ignored while still identifying phase trends and general behaviour.

We thus define *level coverage* as the percentage of predicted values which are binned in the same "level" as the real value. This is an important correctness metric for the prediction work, although the number and size of levels is necessarily application specific.

- **Correlation**

  Standard statistical measures between predicted and actual data streams of course can be applied. Here we use the *Pearson* correlation function, computed from:

$$Correlation = \frac{\Sigma XY - \frac{\Sigma X \Sigma Y}{N}}{\sqrt{(\Sigma X^2 - \frac{(\Sigma X)^2}{N})(\Sigma Y^2 - \frac{(\Sigma Y)^2}{N})}} \quad (1)$$

  Level coverage measures the absolute differences between two set of data, whereas *correlation* measures the similarity between the main trends in the two sets of data. Correlation does not indicate how successful each individual prediction is, but does give a sense of overall agreement.

- **CorScore**

  The *CorScore* is defined to combine the traced rate and correlation metrics; this allows us to measure success with respect to overhead cost. It is computed as *Correlation*(1-Traced Rate)*100*. A higher *CorScore* means a comparatively higher correlation is obtained with a comparatively lower traced rate.

Metric results for the predictions made by our **PreOnline** periodic phase detection technique are shown in Table 3.

In general our algorithm can detect and predict the behaviour of most benchmarks with high accuracy. This does vary according to the characteristics of the the specific benchmark. In programs with intrinsic, regular, repetitive patterns, such as JACK and JESS in the SPECjvm98 benchmark suite and PS in the Dacapo suite, our algorithm can obtain quite good prediction results by tracing only a small portion of the execution. Thus, we have very high correlation values while maintaining the trace rate below 20%. In the most extreme case, PS, we trace no more than 1% of the execution and give nearly perfect prediction with a correlation of 0.944.

Of course for other benchmarks with less intrinsic repetition we may need to trace nearly half of the execution in order to obtain a correlation more than 0.8, e.g., as in PMD and BLOAT. Their much lower CorScores indicate the corresponding reduced return for reward. Not all cases correlate well either; ANTLR, does not even reach 0.6. Nevertheless, on average we achieve a CorScore of 62.97, a high enough value to suggest further investigation of our technique.

6 Conclusions and future work

Phase detection and prediction have received more and more attention in the program analysis and optimization community; extracting the internal phases of a program is important to a variety of applications. At the same time, the mechanism of phase detection also vary widely; an understanding of how current techniques are related is important to future research.

bench	Correl.	CorScore	LevCov(%)	Traced Rate(%)
compress	0.983	90.95	62.66	7.51
db	0.701	50.12	71.79	28.55
jack	0.845	69.84	94.69	17.30
javac	0.805	57.53	89.96	28.49
jess	0.931	81.82	97.68	12.06
raytrace	0.912	84.73	99.52	7.78
antlr	0.597	32.36	80.68	45.79
bloat	0.915	50.27	86.32	45.07
fop	0.749	35.44	82.66	52.71
pmd	0.807	46.10	79.13	42.93
ps	0.944	93.51	99.96	0.95
Average	0.835	62.97	85.91	26.29

Table 3: Metric analysis of prediction results on Java benchmarks. High Correl, CorScore and LevCov values and low Traced Rate values are desireable.

In this paper, we cover a large set of phase detection techniques and define a general solution space that aims to highlight common features as well as differences between approaches. Our solution space provides researchers with a clear understanding of the whole spectrum of phase techniques, and further shows the empty spaces and directions for further research.

Our phase space incorporates both stable and "periodic" phase approaches; periodic phases have not been previously clearly defined as far as we know. We motivated and introduced an initial work on long range periodic phase, and provided experimental results showing its efficacy on non-trivial Java benchmarks with inherit periodic behaviors. To quantitatively evaluate our approach we have also defined a new set of metrics that evaluates periodic phase prediction results in an online measurement context.

Our future work will focus on developing our periodic phase prediction approach and demonstrating its application in a realistic, online setting. An initial port of this algorithm to JikesRVM is in progress. We will then focus on reducing profiling overhead and optimizing GC points based on predicted phase information.

References

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, Nov. 1997.

[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[3] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 111–129, New York, NY, USA, 2002. ACM Press.

[4] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general purpose architectures. In *the 33rd Annual Intl. Sym. on Microarchitecture*, pages 245–257, Dec. 2000.

[5] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-1997-1342, 1997.

[6] M. Burtscher. An improved index function for (D)FCM predictors. *Computer Architecture News*, 30(3):19–24, June 2002.

[7] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 286–297, New York, NY, USA, 2001. ACM Press.

[8] I. Daubechies. *Ten lectures on wavelets*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[9] A. Dhodapkar and J. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines, 2002.

[10] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244. IEEE Computer Society, 2002.

[11] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 217. IEEE Computer Society, 2003.

[12] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 220. IEEE Computer Society, Sep. 2003.

[13] B. B. Fraguela, R. Doallo, J. Touriño, and E. L. Zapata. A compiler tool to predict memory hierarchy performance of scientific codes. *Parallel Comput.*, 30(2):225–248, 2004.

[14] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in Java workloads. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 270–287, Oct. 2004.

[15] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, 1999.

[16] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 207–216. IEEE Computer Society, Jan. 2001.

[17] M. J. Hind, V. T. Rajan, and P. F. Sweeney. Phase shift detection: A problem classification, 2003.

[18] S. Hu, R. Bhargava, and L. K. John. The role of return value prediction in exploiting speculative method-level parallelism. 5:1–21, Nov. 2003.

[19] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 157–168, New York, NY, USA, 2003. ACM Press.

[20] H.-S. Kim and J. E. Smith. Dynamic software trace caching. In *the 30th International Symposium on Computer Architecture (ISCA 2003)*, 2003.

[21] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.

[22] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals to find hierarchical phase behavior. In *2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, March 2005.

[23] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *HPCA*, pages 278–289, 2005.

[24] W. Liu and M. C. Huang. Expert: expedited simulation exploiting program behavior repetition. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 126–135, New York, NY, USA, 2004. ACM Press.

[25] P. Nagpurkar, M. Hind, C. Krintz, P. Sweeney, and V. Rajan. Online phase detection algorithms. In *CGO '06: Proceedings of the international symposium on Code generation and optimization*, Washington, DC, USA, March 2006. IEEE Computer Society.

[26] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 191–202, Washington, DC, USA, 2005. IEEE Computer Society.

[27] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm, 1997.

[28] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. pages 303–313, Oct. 1999.

[29] C. J. F. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. In *Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop (VPW2)*, pages 40–47, Oct. 2004.

[30] D. Project. The DaCapo benchmark suite (beta050224). `http://www-ali.cs.umass.edu/DaCapo/index.html`, 2003.

[31] X. Shen, C. Ding, S. Dwarkadas, and M. Scott. Characterizing phases in service-oriented applications. http://www.cs.rochester.edu/u/xshen/Abstracts/tr848.html, 2005.

[32] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. *SIGPLAN Not.*, 39(11):165–176, 2004.

[33] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior, 2002. In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[35] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349, 2003.

[36] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[37] A. Srivastava and A. Eustace. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[38] Standard Performance Evaluation Corporation. SPECCPU2000 benchmarks. `http://www.spec.org/cpu2000/`.

[39] Standard Performance Evaluation Corporation. SPECCPU95 benchmarks. `http://www.spec.org/cpu95/`.

[40] Standard Performance Evaluation Corporation. SPECjvm98 benchmarks. `http://www.spec.org/osg/jvm98`.

[41] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. *SIGPLAN Not.*, 38(5):91–102, 2003.

[42] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Measurement and Modeling of Computer Systems*, pages 24–35, 1993.

[43] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 261–271, New York, NY, USA, 1994. ACM Press.

[44] X. Vera and J. Xue. Let's study whole program cache behavior analytical. In *International Symposium on High-Performance Computer Architecture (HPCA 8) (IEEE)*, pages 175–186, Feb. 2002.