Real-Time Program-Specific Phase Change Detection for Java Programs

Meng-Chieh Chiu Benjamin Marlin Eliot Moss

College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA 01003 USA
{joechiu,marlin,moss}@cs.umass.edu

ABSTRACT

It is well-known that programs tend to have multiple phases in their execution. Because phases have impact on micro-architectural features such as caches and branch predictors, they are relevant to program performance Xian et al. [2007], Roh et al. [2009], Gu and Verbrugge [2008] and energy consumption. They are also relevant to detecting whether a program is executing as expected or is encountering unusual or exceptional conditions, a software engineering and program monitoring concern Peleg and Mendelson [2007], Singer and Kirkham [2008], Pirzadeh et al. [2011], Benomar et al. [2014]. We offer here a method for real-time phase change detection in Java programs. After applying a training protocol to a program of interest, our method can detect phase changes at run time for that program with good precision and recall (compared with a "ground truth" definition of phases) and with small performance impact (average less than 2%). We also offer improved methodology for evaluating phase change detection mechanisms. In sum, our approach offers the first known implementation of real-time phase detection for Java programs.

CCS Concepts

•Computer systems organization → Real-time systems; •Computing methodologies → *Machine learning;* **•Software and its engineering** → *Runtime environments;*

Keywords

Program Phases; Software Engineering; Real-time Systems; Machine Learning

1. INTRODUCTION

Programs exhibit different phases of execution. For example, a compiler's phases might be: initialization, parsing, semantic analysis, optimization, and code generation, with additional smaller phases in each. The phases might iterate as the compiler processes each function in the input, etc. But why are phases relevant? Software engineers are interested in phases as part of program analysis and understanding Peleg and Mendelson [2007], Singer and Kirkham [2008], Pirzadeh et al. [2011], Benomar et al. [2014]. They are further interested in phases as a program runs in order to determine whether the program is operating as expected. Computer architects, and indeed even ordinary users, are interested in phases because of their impact on micro-architectural components such as caches and branch predictors, and thus on performance Xian et al. [2007], Roh et al. [2009], Gu and Verbrugge [2008], in terms of both time used and energy consumed. Another way of putting this is that program phases are an important aspect of dynamic program behavior.

Most current program phase analysis methods are offline. They may provide rich and useful results Georges et al. [2004], Wang et al. [2012], Nagpurkar and Krintz [2004], Watanabe et al. [2008], but only after the fact. Some methods have "online" in their names Nagpurkar et al. [2006], Otte and Richardson [2007], but that means that they work in a single forward pass over a stream of information about the program, such as the branch instructions that the program executes. These techniques do not operate in real time. Even if they did, the cost of analyzing each branch as it occurs would likely slow program execution considerably.

Two practical applications of *real-time* phase change detection (undoubtedly there are others as well) are saving energy and controlling garbage collection. Phase information can indicate whether lowering clock frequency or voltage, or both, might save energy with little impact on performance. This is true, for example, of phases where the CPU is mostly idle waiting on main memory accesses. In the case of garbage collection, collections can be scheduled on particular phase boundaries so as to recover more "dead" bytes with less effort ke Chen et al. [2006], Zhang et al. [2006]. Real-time program phase information may also help with scheduling concurrent tasks on a multiprocessor.

We offer here a method for real-time phase change detection in Java programs. After applying a training protocol to a program of interest, our method can detect phase changes at run time for that program with good precision and recall (compared with a "ground truth" definition of phases) and with small performance impact. To accomplish this, we first trace a number of executions of the program of interest, recording information about each call/return and conditional branch that occurs. These traces are suitable for computing "ground truth," that is, phases and phase changes according to a precise definition. Ground truth is defined in terms of all the branch instructions exercised in the program.

We also use the traces to develop an efficient real-time phase change detector, as follows. We choose (in a manner described later) a small number of branch instruction that we will instrument at run time—say four or eight branches. Next, we apply a clustering model called a *Gaussian mixture model* (GMM) McLachlan and Peel [2004]. The GMM model is trained given a desired number of

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland

DOI: http://dx.doi.org/10.1145/2972206.2972221

clusters as input, and probabilistic inference is then used to map each vector of feature values to a specific cluster. At run time, we count the instrumented branches, and when their total count reaches a threshold, we determine a cluster for the recent past. If it is different from the previous cluster, we judge the program to have transitioned from one phase to another.

Our method does not significantly impact performance because (1) it instruments only a small number of branches, (2) it evaluates the clustering model only occasionally, and (3) the clustering model is not overly expensive to evaluate. Our method corresponds well to ground truth because (1) the GMM model is faithful to the ground truth, even though it uses a much smaller number of features, and (2) the run time modeling does not diverge much from that of the training runs of the same program. We demonstrate this across a significant number of Java programs.

In addition to making real-time phase change detection practical, we also offer improved methodology for evaluating phase change detection techniques, described later.

Here is the plan of the paper. First, we introduce our data set and the tracing tool (Elephant Tracks), and describe the process of feature extraction (Section 2). In Section 3 we introduce the processes of feature selection and clustering, including our methodology of evaluating detector quality. Then in Section 4 we describe our real-time phase transition detection system. Section 5 presents the setting and methodology of our experiments, and offers results from our real-time phase transition detection system. We then discuss how other work is related to our approach (Section 6). Finally, in Section 7 we describe future work and conclude.

2. JAVA EVENT TRACES

For obtaining fine-grained traces of program features, we apply Elephant Tracks (ET) [Ricci et al., 2011, 2013], a dynamic program analysis tool for Java that produces detailed traces of a variety of program events. ET's original focus was on logging events relevant to garbage collection (object allocations and deaths, and heap mutations), along with method calls and returns (to provide additional temporal context). We modified ET to log each conditional branch that is executed, including the next instruction (i.e., which way the branch goes). We also include method calls and returns in the logs. Each branch instruction is identified by its containing class and method, and a unique index within the method. We employ means to map the features (events types) of different runs of the same program, which may load and exercise some different classes, into a commonly numbered space of features.

2.1 Feature Extraction

To build feature vectors, we group branch records together in batches of 100,000. A feature vector then indicates, for each static branch edge in the program (and each method entry and return), how many times that event occurred in that interval of 100,000 branches. (We also include binary features that indicate only whether a particular event occurred or not.)

Why 100,000? The choice of interval length must balance competing pressures. On the one hand, if the interval is too short, the feature counts may be noisy and overly sensitive to the exact boundaries between intervals. Furthermore, if the interval is too short, we need to increase the history length, which means we need to concatenate more feature vectors to get what we need, increasing the complexity of the predictive models. Finally, short intervals, if applied in the real-time setting, incur more overhead in computing the model's output. On the other hand, if the interval is too long, we will not detect phase changes as quickly, and we risk conflating phases.

While we did not explore a variety of interval lengths in our experiments, 100,000 branches per interval seems to be a reasonable value, and corresponds reasonably to previous work.

To shorten feature vectors, we drop features whose value is always zero across all runs of a program. These typically arise because of application code or library code that is not exercised. Such features often comprise 90% of the raw set of features, so this reduction is quite helpful for later processing steps.

2.2 Benchmarks Used

Most of our benchmarks come from the DaCapo Java benchmark suite [Blackburn et al., 2006], a set of open source, real world applications with non-trivial memory loads. DaCapo offers Java benchmarking to the programming language, memory management, and computer architecture communities. We use 7 DaCapo benchmarks, with 4 to 19 different inputs for each benchmark.

We add `javac` to the DaCapo benchmarks, the compiler from Java source code to bytecode provided as part of the Oracle Java Development Kit. We modified `javac` to force it to clear its cache of previously compiled classes between each compilation. This gives more iterative behavior and is intended to model a long-running server application. As it compiles a sequence of input source files, each file gives rise to similar, but still somewhat varying, profiles of the volume of live objects, as observed by Dieckmann and Hölzle [1999]. It is also easy to devise inputs to `javac` that require whatever amount of work is desired.

3. A MACHINE LEARNING MODEL FOR PHASE TRANSITION DETECTION

In our pursuit of accurate real-time phase transition detection, machine learning and feature selection play an important role. Our approach to detecting transitions is to cluster time intervals based on the similarity of their feature vectors. We consider there to be a phase *transition* whenever the current phase differs from the previous one. The specific clustering model we use is the Gaussian mixture model (GMM) McLachlan and Peel [2004]. This is a probabilistic model that can be seen as generalizing k-means clustering to incorporate information about the covariance structure of the clusters. The GMM assumes the data (feature vectors) are generated from a mixture of a finite number of Gaussian distributions (the clusters) with unknown parameters. Each feature vector has a probability of having been generated by the Gaussian distribution associated with each cluster. A feature vector is assigned to the cluster for which its probability is highest. We associate clusters with phases in a program.

At first it might seem strange that this would work. After all, a program can have an arbitrary number of phases, and to apply the GMM we must choose a number of clusters in advance. As our results will show, at least for the range of programs/runs we considered, the quality of the results is not strongly sensitive to the number of clusters we form, as long as there are more than just a few. Since different loop structures, recursions, etc., result in feature vectors that point in different directions (recall that feature vectors are based on counts of traversals of branch edges), in the end we found it not so surprising that this approach might work.

3.1 Feature Selection

As previously mentioned, even our reduced feature vectors from processed ET traces have thousands or tens of thousands of features. While the clustering algorithms might work on these, for real-time phase transition detection we cannot use all the features—the instrumentation cost would be very high. Rather, we need to choose a

smaller number of features that still capture the various phases. We have observed that many features in our vectors are highly correlated with one another, suggesting that there is a lot of underlying redundancy and that reduction in dimensionality by feature selection is likely to work well in practice.

The features we select for clustering need to satisfy two characteristics: high diversity (low correlation with each other), and low cost to collect. Increasing the diversity of features gives GMMs more ability to identify cluster structure in data, potentially increasing the ability of our model to differentiate various phases. Selecting features that cost less to collect from a running program is important for us in building a real-time system.

To achieve high diversity, we randomize the order of the features, and then select features sequentially. For each new candidate feature, we compute its correlation with previously selected features, and reject the candidate if it is highly correlated (i.e., correlation above a chosen threshold) with any previously selected feature. We use a threshold of 0.80 for the Pearson correlation coefficient. We iterate this process until we have selected a target number of features. In our experiments we try various numbers of features, and find that a surprisingly small number of features is adequate. We also perform this randomized feature selection process a number of times, and report the distribution of results.

After choosing features using the filtered randomized process just described, for each chosen feature we consider replacing it with a lower cost proxy feature. The point of this step is to reduce the run-time cost of collecting feature event counts. A chosen proxy must be highly correlated (we use the same threshold as before) with the originally chosen feature. We sort the features in increasing order of cost to collect, assuming that the number of times of the feature's event occurs is a good estimate of that feature's cost. For each originally selected feature, find the first feature in the cost-sorted list that is highly correlated with the original, and replace the original with that feature (or keep the original, if we do not find a lower cost one). We do not explicitly demand that the new feature be uncorrelated with the other original features as this property should be sufficiently maintained by transitivity of correlation.[1] In sum, we choose a small number of features, uncorrelated with each other, and of low cost to count at run time. This helps guarantee that our run-time overheads for real-time phase transition detection will be as small as possible and the quality high.

How long does this process take? Figure 1 shows the total time required to run ET, generate feature vectors from traces, and select 12 features, for each of the benchmarks. There are two stacked bars for each benchmark program, one showing the time that would be required on a uniprocessor (sum of times for the various traces of that benchmark) and one the time required on a multiprocessor large enough to process concurrently whatever can proceed in parallel. Note that in the multiprocessor case, the traces can be produced concurrently and each trace's feature vectors can be constructed as soon as that trace finished. However, feature selection can proceed only after all traces' feature vectors are available. After that the 30 selections of 12 features can proceed concurrently. The figure shows that trace production with ET dominates the time required, taking an average of nearly 8 hours, and up to 19 hours, on a uniprocessor. On a multiprocessor this becomes an average of 5 hours, with the longest time being 12 hours. Running the GMM model and inserting instrumentation into the programs takes negligible time (seconds to at most a few minutes) on this time scale. We observe the ET focuses more on functionality than on speed, and it might be

[1] In other words, if features A and B have low correlation and features A and A' have high correlation, then features A' and B will have low correlation.

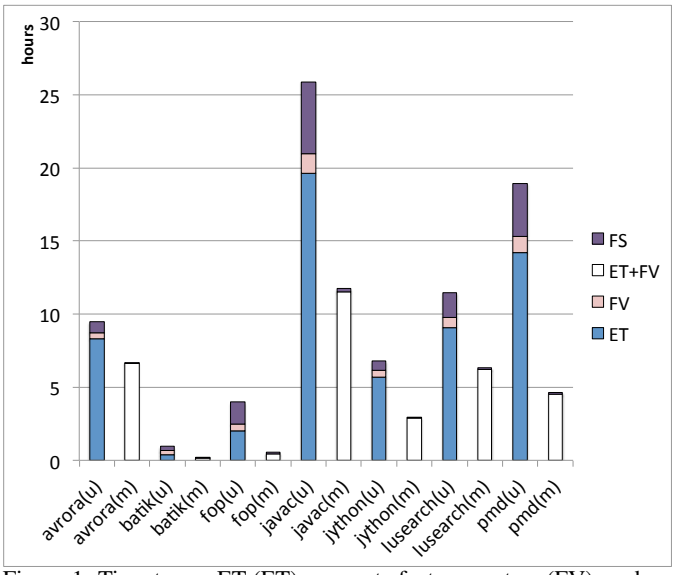


Figure 1: Time to run ET (ET), generate feature vectors (FV), and select 12 features (FS) on a uniprocessor (u) and multiprocessor (m)

possible to build a more streamlined tracing tool. However, detailed recording of every conditional branch, call, and return will still be costly. Processing ET traces to produce feature vectors is a small component of overall ocst, but feature *selection* is more significant on a uniprocessor, because we perform it 30 times. (We would recommend doing it this many times in practice and choosing the best feature set found.) Fortunately the 30 selections can be done in parallel, making this step negligible in the multiprocessor case.

3.2 Offline Evaluation of the GMM Clustering Approach

To understand the quality of results we might achieve with GMM clustering, we compare it with the detector of Nagpurkar et al., hereafter referred to as ND (for Nagpurkar et al.'s Detector), and with a detector that simply makes phase transition decisions at random, which we call Random. We use the baseline definition of phases and transitions of Nagpurkar et al. [2006], hereafter called Baseline, described further below. Baseline defines phases in terms of a parameter one must supply, the minimum phase length (MPL).

ND works by considering what amounts to our feature vectors (in their weighted set model), but with all features included. Their method indicates phase transitions according to the similarity of the current window's feature vector with that of the previous window.

The Random scheme works by simply choosing "in phase" with probably p and "in transition" with probability $1-p$, where p is the proportion of windows that Baseline identifies as in-phase.

Baseline works by building up phases from *complete repetitive instances* (CRIs). A CRI is either a complete recursive call (not nested within another call of the same routine), which can include just a single non-recursive call, or a loop execution (all iterations) within a (possibly) recursive execution. The scheme concatenates adjacent CRIs and indicates in-phase if the concatenated CRIs meet the minimum phase length (MPL) criterion. Otherwise it indicates in-transition.

To evaluate GMM, ND, and Random, we explored two scoring metrics, the Accuracy Score of Nagpurkar et al. [2006], and an F-score based on measuring precision and recall, with shifted weights (so that phase transitions need not be recognized exactly when they occur). All scoring is based on comparing a given scheme's judgment of in-phase vs. in-transition against Baseline's determination.

The accuracy score of Nagpurkar et al. [2006] is a weighted sum of three comparison measures. The first, which they call "correlation" (more usually called accuracy), is the fraction of windows in which the judgments agree. The other two measures are based on how well a scheme's phase transition boundaries agree with Baseline. Their "sensitivity" measure is the fraction of Baseline boundaries that the detector also identifies as boundaries. Their third measure is the fraction of a detector's reported boundaries that match Baseline boundaries. The total score is a weighted sum of these, where the weights are 50% correlation, and 25% each for the other two measures. (For details of what is recognized as a matched or unmatched boundary, we refer the reader to their paper.)

In our experiments, we found that this measure did not discriminate well in various cases. In particular, when there are few phase transitions, it gives a high score even if a detector reports no transitions at all. Likewise, Random achieves a high score when there are few transitions. Therefore we developed an alternative, based on the F-score used in document retrieval and similar evaluations. The F-score combines precision and recall into a single number. Precision is the fraction of detected transitions that correspond to actual ones in Baseline, while recall is the fraction of actual Baseline transitions found by the detector. Given precision p and recall r, the F-score is defined as $2 \cdot \frac{p \cdot r}{p+r}$.

However, given that detection may lag actual transitions, we find it reasonable to allow for approximate matching of transitions, but with a lowered score. Therefore, we developed a scheme that gives a weighted value to transitions that are not perfectly aligned. We call it the *Shifted Weighted F (SWF) score*, and it works as follows. We consider a set of window sizes 1 through δ. Assuming that there are N original time periods, for a given window size w we consider all $N-(w-1)$ distinct overlapping windows containing w consecutive elements. For each window, we consider a transition to be found by a given detector (or Baseline) if for the detector (respectively, Baseline) any element in the window indicates transition. Thus, window size 1 considers perfectly aligned transitions, while window size 2 considers those within one time step of one another, etc. We compute precision and recall on the vectors of length $N-(w-1)$ and, to penalize for non-alignment, divide the result by w. We sum these values for w from 1 up to a chosen limit δ. To normalize the result, we divide by the similar number obtained from comparing a detector (or Baseline) with itself, as appropriate for precision and recall. It is these normalized p and r values that we use to compute SWF.

Here is how we choose δ. If windows are very large, then we will almost always find transitions within them. Therefore, we desire δ small enough that the likelihood (for Random) that there is no transition within a window will be at least α for some chosen α. This results in the following equation:

$$\delta < \frac{\log \alpha}{\log(1-(1/\mathrm{MPL}))}. \quad (1)$$

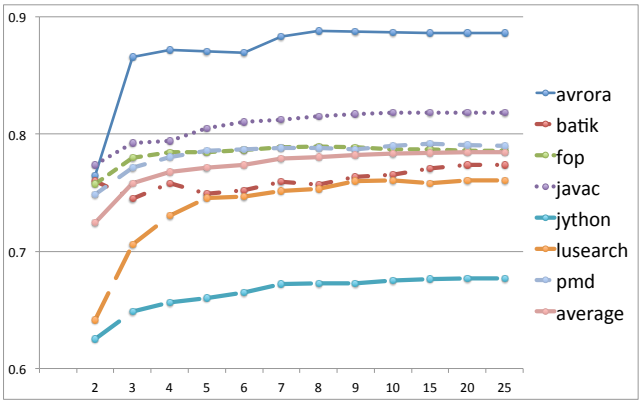
MPL appears in this equation because, in Baseline, MPL gives the minimum possible distance between phase transitions. Equation 1 expresses the solution to this equation, which gives the probability of no phase transition within δ steps:

$$\alpha > (1-(1/\mathrm{MPL}))^{\delta} \quad (2)$$

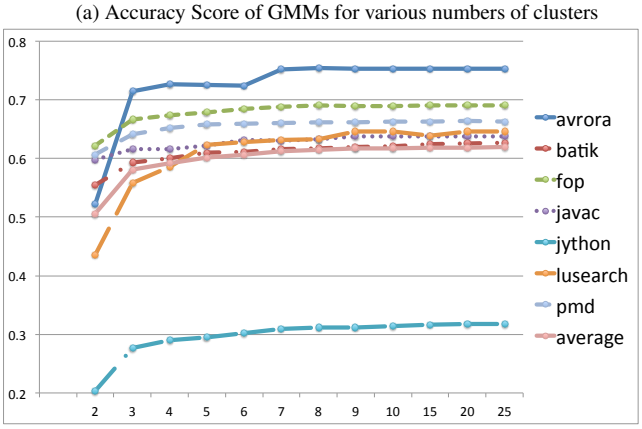
We chose to use $\alpha = 0.1$ to determine our δ value.

3.2.1 Number of Clusters

The number of GMM clusters we choose to use is an important parameter of our real-time phase transition detection system. Having more clusters increases the cost of the run-time phase change detection function, which is linear in the number clusters. Having fewer clusters, though, may decrease the quality of the detector. In order to choose an appropriate number of clusters, we developed GMM clusterings with various numbers of clusters. Figures 2a and 2b show the scores obtained offline, i.e., using the originally obtained values for the selected features. We see that four clusters gives a fairly high score, and going beyond eight clusters does not give much additional improvement. This shows that it is possible to reduce from thousands of features to just a few bits of information in order to identify phase transitions.



(a) Accuracy Score of GMMs for various numbers of clusters



(b) SWF Score of GMMs for various numbers of clusters

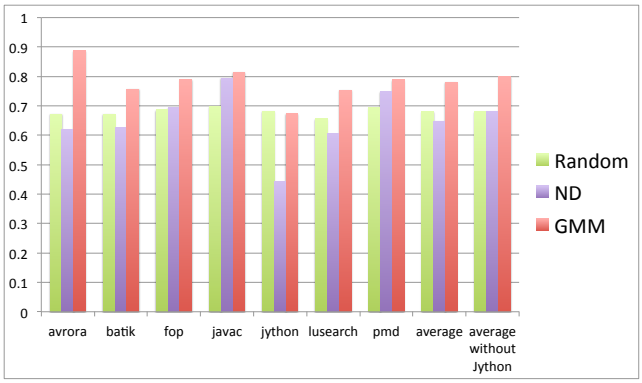
Figure 2: Performance of GMMs with various numbers of clusters

Here is how the graphs are derived. For each benchmark run (trace) we develop 30 different selections of features, and for each of those feature sets, we developed a clustering and scored it. A given point is the arithmetic mean across the 30 clusterings of each trace. In some cases, due to limitations of time available, we developed fewer than 30 selections of features, and in such cases we indicate the different number. The "average" line is the arithmetic mean of the plotted benchmark scores (i.e., weighting each benchmark equally).

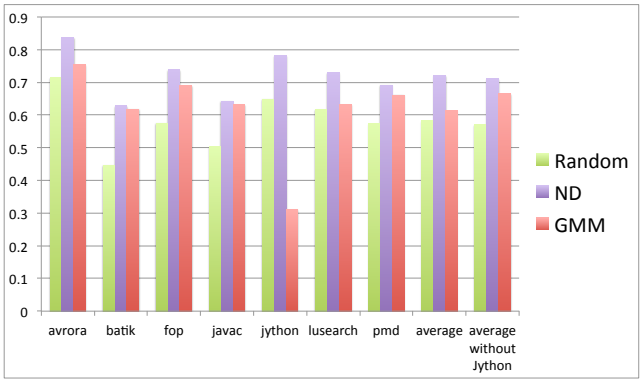
3.2.2 Quality of the Two Scoring Methods

To show why we believe the SWF score is a better metric than the Accuracy of Nagpurkar et al. [2006], we compare Random with GMM and ND in the same graphs. Figure 3a shows that Random obtains a score of about 70%, about the same as ND. Figure 3b shows how the SWF score discriminates between Random and the other detectors. Therefore we believe that the SWF score is a more appropriate metric for these comparisons. Therefore, after this figure we use only the SWF score to evaluate detectors.

3.2.3 Comparison of the Detectors



(a) Accuracy Score for GMM, ND, and Random



(b) SWF Score for GMM, ND, and Random

Figure 3: Performance of GMM, ND, and Random detectors. The offline GMMs shown use 8 selected features and 8 clusters. The threshold for ND is 0.8.

Again considering Figure 3a, we see that, even though our GMMs use fewer features than ND (which uses complete feature sets), they actually perform better on the Accuracy Score, an average of 78% versus ND's 65%, which is lower than Random's 68%. The GMM approach achieves the best score for all programs except `jython`. Turning to our SWF scoring metric in Figure 3b, we see that (again, except for `jython`) the relative quality order of the schemes is ND (72%), GMM with eight features (61% with `jython`, 67% without), and Random (58%). This is actually less surprising since GMM uses less information than ND. Thus, GMMs might not give the best results for an offline analysis unconcerned with real-time performance, but give surprisingly good quality in the face of using such little information. Note that ND uses a similarity threshold parameter, which affects the quality of its results. We used the best value of those reported by Nagpurkar et al. [2006], namely 0.8. The plotted GMM score values were developed as described for Figures 2a and 2b.

3.2.4 *Offline Cross Validation of GMMs*

The results given for GMMs so far evaluate a GMM on the same trace used to develop that GMM, what is sometimes called "self-test" evaluation. This is reasonable for ND since their method is entirely after the fact. However, we aim for real-time use, which implies developing a model from some number of program runs—the training set—and using that model on new program runs. Since we are developing program-specific phase transition detectors, we use new runs of the same program, but on different inputs. Thus, to evaluate our scheme for real-time use, we need to consider *cross-validation* results. We use "leave-one-out" cross-validation. If we have k runs of a given program, then each cross-validation fold trains on $k-1$ of the runs and evaluates on the run not trained on. We do this leaving out each of the k runs in turn, and average the results.

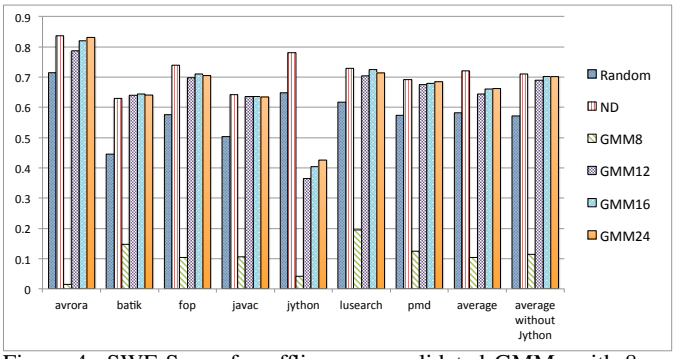


Figure 4: SWF Score for offline cross-validated GMMs with 8, 12, 16, and 24 selected features, including Random and ND for comparison. All GMMs use 8 clusters. The threshold for ND is 0.8.

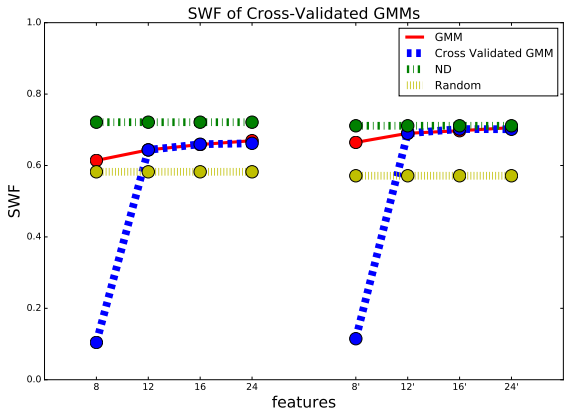


Figure 5: SWF Score for self-test GMM, cross-validated GMM, ND, and Random detectors, varying the number of features used by the GMMs. All GMMs use 8 clusters. The threshold for ND is 0.8. The unprimed x-axis give the number of features for averages that include `jython`, and the primed one for those that do not.

Figure 4 presents cross-validation results for GMMs using 8, 12, 16, and 24 features. While 8 features did well under self-test, we see that we need at least 12 features for a good quality detector under cross-validation, and that 16 and 24 features offer little additional improvement. We include ND and Random in the figure for comparison, but of course they are invariant under cross-validation. Figure 5 shows the average SWF scores, with and without `jython` (an outlier), in more detail, showing both self-test and cross-validation results for GMMs that use various number of features.

Here is why we believe 8 features did not work well under cross-validation. When self-testing, all the selected features will take non-zero values in the run being used for evaluation. In cross-validation, it is entirely possible that one or more selected features will *always* be zero in the run used for evaluation, thus throwing the model off. Given the small number of features, this can substantially impact detector quality. Using more features tends to mitigate this: the models tends to depend less on each individual feature. Also, using more features will tend to improve quality, up to the point of over-fitting on the training data.

4. IMPLEMENTING THE REAL-TIME DETECTOR

For our run-time measurements we used a server system consisting of two CPUs, each an Intel Xeon E5-2690 v3, clocked at 2.6 GHz. Because these CPUs will otherwise adjust their voltage and clock frequency, we pinned the clock speed to the design point of 2.6 GHz, for consistent results. The server has a large amount of memory and disk space, not relevant to these relatively small programs. These processors have AVX2 vector units, which we exploit to speed the run-time computation of the current cluster for detecting phase transitions.

In terms of software, the operating system is Red Hat Enterprise Linux 7.0. We used IBM's J9 Java Virtual Machine, build 2.4 for Java 1.6.0. All the software uses 64-bit mode.

We implemented the phase transition detector as a Java agent, using the Java Virtual Machine Tool Interface (JVMTI). Each instrumented feature calls a Java method to increment a counter for that feature. It also counts down a total number of events. Once the desired number of events is reached, the Java counting method calls a native method in the agent, which obtains the counter values and then, using vector intrinsics supplied by `gcc`, determines the highest probability cluster using probabilistic inference.

To insert the instrumentation into the Java code, we use a modified version of Elephant Tracks that instruments only the selected features. It does this by applying bytecode rewriting to the compiled classes of the application. We perform these rewrites as the code loads. Note that we iterate runs of a benchmark program to factor out JVM JIT compilation costs, etc., which also factors out our bytecode rewriting. However, the cost of that rewriting is not extreme, and is incurred only on start up. The modified version of Elephant Tracks also automatically generates the Java class holding the event counters and counter increment methods, which the Java agent then loads into the JVM.

5. EXPERIMENTS

We now consider the run-time overhead of our real-time detector, and its quality as a detector.

5.1 The cost of instrumented program

We compare the cost of running with and without the instrumentation that our systems adds. For each benchmark and input, we selected features some number of times, and then perform multiple runs of the instrumented program. We also perform 30 runs of each program and input with no instrumentation. Table 1 shows the number of feature selections and runs for each number of features.[2] We did runs with 8 features early on, to assess the acceptability of the overhead of our instrumentation. These runs were self-test version (mode ST in the table). Given the poor score of 8 features, we did not both doing cross-validated runs for that case. The runs for 12 and 16 features are cross-validated, however (mode CV).

We insert instrumentation into a program's class files in advance, using a different version of our Java agent that simply rewrites byte codes at the instrumented code locations. At run time we use a very simple agent that just collects and dumps out phase detection statistics. Overall, we use three agents, one when generating the full ET trace, one when instrumenting classes, and one when running the instrumented application.

We find, as one would expect, that the run-time overhead tends to increase as we instrument more features. The smallest number of features that gives good detection quality is 12, resulting in average overhead of 2.0% (see Table 2). Using 16 features increases the overhead to 5.0%. It may seem curious that the 8 and 12 feature cases have the similar average overhead, but recall that the 8 feature case is self-test, so all 8 features are used, while the other cases are cross-validated and the tested program may not exercise all of the chosen features.

[2]In a final version of the paper we intend to report 30 feature selections and 30 runs in all cases. Some of these numbers are lower now due to limitations of time.

Table 1: Number of feature selections, number of runs of each selection, and mode (Self-Test of Cross-Validated) for each number of features

Features	Selections	Runs	Mode
8	30	30	ST
12	30	30	CV
16	30	30	CV

Figure 6 shows the distribution of relative running time for each program, showing uninstrumented and instrumented versions with various numbers of features. We developed the plotted data as follows. For each trace we determined the median running time for the uninstrumented version. This is the value to which the other data are normalized. (This applies to Table 2 as well.) Thus the uninstrumented case shows its relative median as being 1. For instrumented versions, the distribution includes multiple runs under each of the randomized feature selections.

For timing data we normalize by median values since it is more likely that a run will be perturbed so as to lengthen its running time rather than shorten it, so the median gives a better estimate of typicality than the mean would. The distributions show that indeed there is skew in the data toward higher values, but that the 25th to 75th percentiles are mostly not too dispersed.

Returning to Table 2, in addition to observing that the average overhead for 12 features is less than 2%, we see that only 13 of 48 traces incur overhead of 1% or more. Curiously, a number of program inputs perform better with instrumentation. We can only speculate as to why, but possibilities include impact on the JIT compiler, branch predictors, and data and instruction caches. Of course one would expect the cost to go up, but if some inner loops happen to run faster we could see this kind of effect. We found a small number of outliers in the underlying data. For example, `lusearch-default` typically runs in about 14 seconds, but one instrumented run took 91.6 seconds. Also some `pmd` runs took 39 times longer when instrumented. Figure 6 shows the distribution of times and shows that those high overhead points are indeed rare outliers.

To test the statistical significance of the difference in running times, we performed T-tests comparing the instrumented and uninstrumented time distributions (see Table 3). The 8 feature case shows quite weak significance ($p = 0.17$ means that there is a 17% chance that the two distributions were drawn from the same underlying distribution). With higher numbers of features the significance is higher. We conclude that the effect is real, not just chance in our measurements. We also performed a Mann-Whitney U test, as a check since the T-test is sound only for more or less normal distributions. It gave the same result.

5.2 Quality of the Real-time Detector

There are a number of reasons why real-time phase detection might work differently from original program runs. A program may not execute in exactly the same way, even on the same input—even deterministic Java programs may be affected by background threads in the Java system. However, an effect of more direct concern is that we cannot easily trigger our phase detection function at the same points during execution that the offline method uses. This is because

Table 2: Ratio of running time of instrumented programs with 8, 12, and 16 features to the uninstrumented versions.

Trace	Inst 8	Inst 12	Inst 16
avrora-default	0.999	0.971	0.972
avrora-small	1.020	1.003	1.001
batik-default	1.031	1.039	1.000
batik-large	1.010	0.997	0.974
batik-mapSpain	1.005	1.020	0.979
batik-moonPhases	1.008	1.020	0.988
batik-small	1.004	1.020	0.976
batik-strokeFont	1.017	1.016	0.989
batik-vuelo	1.004	1.030	0.983
fop-apache-borders	0.974	0.977	1.738
fop-apache-list	0.952	0.956	1.765
fop-apache-readme	0.951	0.959	1.794
fop-apache-table	0.955	0.946	1.666
fop-default	0.965	0.879	1.966
fop-small	0.816	0.814	0.995
fop-test-afp	1.104	0.979	0.943
fop-test-pdf	1.009	1.012	0.974
fop-test-rtf	1.016	1.018	0.956
fop-xmlmind-userguide	1.012	0.998	0.973
fop-test-pcl	0.959	2.181	1.016
javac-analyzetrace	0.998	1.026	1.055
javac-asm	0.986	1.020	0.979
javac-crystal	1.100	0.961	0.757
javac-HashCacheLines	0.996	0.983	0.963
jython-html	0.626	1.494	1.311
jython-scsa	0.939	1.470	1.284
jython-small	0.889	1.350	0.946
lusearch-default	1.430	1.069	1.066
lusearch-large	1.039	0.856	0.853
lusearch-small	1.054	0.911	0.901
pmd-ant	1.129	0.922	0.898
pmd-ast	1.126	0.938	0.909
pmd-cpd	1.000	0.898	0.876
pmd-dcd	1.196	0.744	0.720
pmd-default	1.189	0.930	0.901
pmd-dfa	1.033	0.917	0.891
pmd-jaxen	1.162	0.995	0.974
pmd-jsp	0.924	0.875	0.850
pmd-parsers	1.017	0.973	0.948
pmd-properties	1.109	0.988	0.974
pmd-renderers	0.985	0.971	0.957
pmd-rules	0.952	0.935	0.928
pmd-sourcetypehandlers	0.956	0.939	0.942
pmd-symboltable	1.033	1.023	1.007
pmd-typeresolution	1.035	1.030	1.004
pmd-util	1.008	0.982	0.963
pmd-large	0.998	0.960	0.953
pmd-middle	0.996	0.956	0.942
Average	1.015	1.020	1.050

we are not counting *all* the conditional branches. At best we can count some number of occurrences of the much smaller number of features that we *do* have. We choose a feature with small variation from window to window across the run, and we use it as a proxy for time, triggering our phase detection function when this feature has occurred the average number of times that it occurs in a window in the original ET trace. We first compare the real-time detector against offline GMMs.

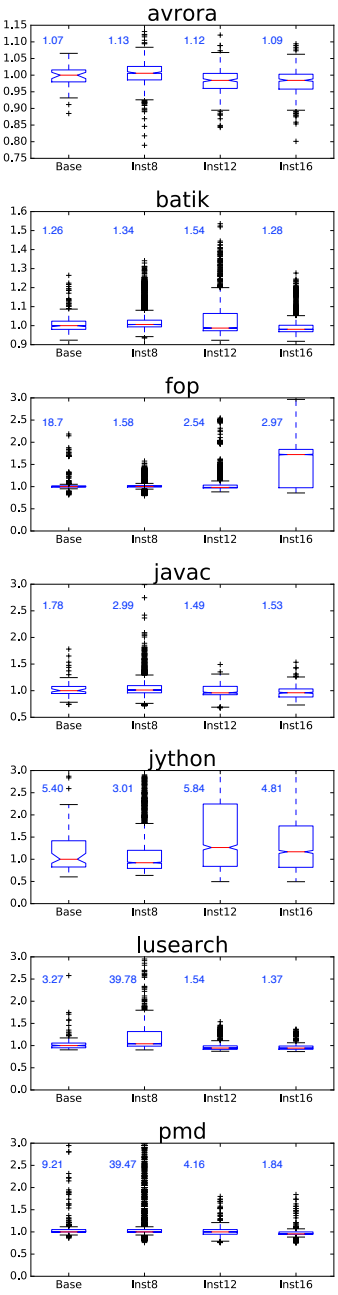


Figure 6: Execution time distributions of uninstrumented and instrumented traces. The number above each distribution is the maximum of the distribution.

Figure 7 shows that the transitions detected by the real-time detector are not highly correlated with the transitions detected by offline GMMs. The highest SWF score the real-time detector achieves, when comparing with the offline GMMs, is 87%, and the average is 76%. This is because the detectors use different time intervals. We conclude that a more appropriate comparison must adjust for that difference.

The most interesting comparison is against Baseline, i.e., "ground truth," but given the different lengths of time intervals, how shall we compare? First, we assume that each detector's intervals are of uniform length in real time. Then, if detector B reports a transition anywhere during a given interval of detector A, we consider B to

Table 3: T-test of execution time of programs run with and without instrumentation

Trace	Inst 8	Inst 12	Inst 16
avrora-default	0.44	0.00	0.00
avrora-small	0.00	0.28	0.40
batik-default	0.04	0.02	0.50
batik-large	0.20	0.41	0.02
batik-mapSpain	0.23	0.01	0.00
batik-moonPhases	0.02	0.00	0.00
batik-small	0.33	0.02	0.01
batik-strokeFont	0.00	0.00	0.02
batik-vuelo	0.33	0.00	0.04
fop-apache-borders	0.16	0.18	0.00
fop-apache-list	0.10	0.12	0.00
fop-apache-readme	0.09	0.13	0.00
fop-apache-table	0.10	0.06	0.00
fop-default	0.24	0.01	0.00
fop-small	0.00	0.00	0.46
fop-test-afp	0.00	0.15	0.00
fop-test-pdf	0.02	0.02	0.00
fop-test-rtf	0.02	0.02	0.00
fop-xmlmind-userguide	0.01	0.40	0.00
fop-test-pcl	0.02	0.00	0.00
javac-analyzetrace	0.48	0.21	0.05
javac-asm	0.20	0.10	0.11
javac-crystal	0.02	0.20	0.00
javac-HashCacheLines	0.31	0.03	0.00
jython-html	0.01	0.00	0.03
jython-scsa	0.28	0.00	0.01
jython-small	0.17	0.00	0.33
lusearch-default	0.00	0.02	0.03
lusearch-large	0.30	0.03	0.03
lusearch-small	0.21	0.04	0.03
pmd-ant	0.09	0.14	0.08
pmd-ast	0.07	0.07	0.02
pmd-cpd	0.50	0.02	0.01
pmd-dcd	0.15	0.11	0.09
pmd-default	0.15	0.18	0.10
pmd-dfa	0.36	0.16	0.10
pmd-jaxen	0.01	0.37	0.04
pmd-jsp	0.24	0.13	0.09
pmd-parsers	0.15	0.18	0.04
pmd-properties	0.13	0.14	0.01
pmd-renderers	0.33	0.20	0.10
pmd-rules	0.21	0.14	0.12
pmd-sourcetypehandlers	0.17	0.10	0.11
pmd-symboltable	0.01	0.05	0.32
pmd-typeresolution	0.01	0.02	0.38
pmd-util	0.39	0.28	0.12
pmd-large	0.47	0.03	0.02
pmd-middle	0.47	0.23	0.17
Average	0.17	0.11	0.08

have reported a transition for that interval. If we compare against Baseline, and use the same number of time intervals for each detector as just described, we obtain the SWF scores shown in Figure 8.

We find that cross-validated RT gives good results, better than ND, when comparing with Baseline, and much better than Random. RT 12 achieves an average SWF score of 81%, compared to 58% for Random and 72% for ND. If we omit `jython` from the results—for which GMMs did not show good results in offline comparisons,

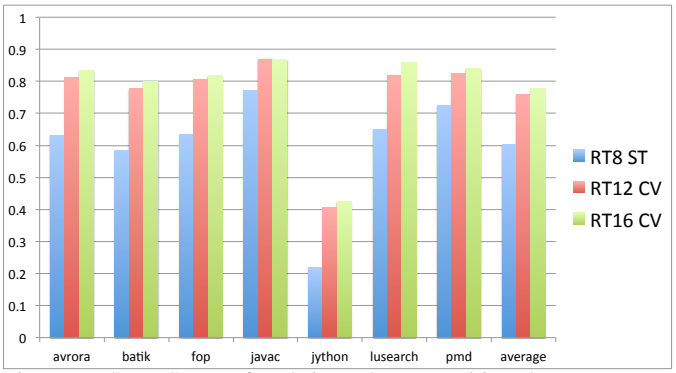


Figure 7: SWF Score of real-time phase transition detectors vs offline GMMs. RT = Real-Time; the numbers indicate the number of features; ST = Self-Test; CV = Cross-Validation. All GMMs use 8 clusters.

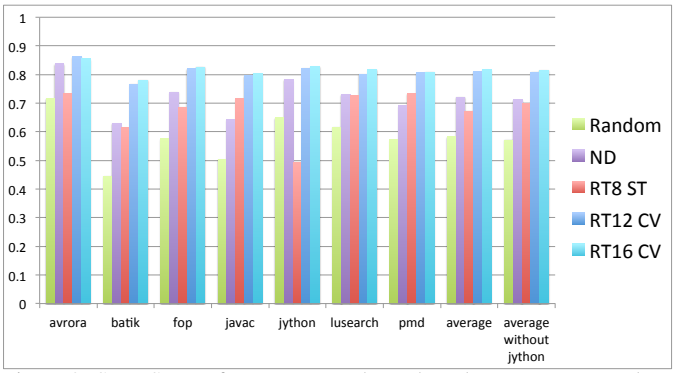


Figure 8: SWF Scores for RT, ND, and Random detectors compared with Baseline. The number indicate the number of features for the RT scheme; ST = Self-Test; and CV = Cross-Validation. All RT detectors use 8 clusters. The threshold for ND is 0.8.

either—RT 12's average score remains 81%, compared with ND's 71%. In Figure 7, we find that RT 12's detections follow `jython` better than the offline GMMs. In sum, our real-time phase transition system can detect phase changes with good precision and recall and with small impact on execution time.

6. RELATED WORK

Dhodapkar and Smith [2003] present an overview of proposed techniques for detecting program phase changes and discuss some of the issues related to the definition of program phases. They also present a comparison of techniques that use unbounded hardware resources and compare practical hardware implementations based on working set signatures and accumulator tables. Duesterwald et al. [2003] study the time-varying behavior of programs using metrics derived from hardware counters on two different micro-architectures. Georges et al. [2004] propose an offline method-level phase analysis approach for Java workloads that includes computing execution time for each method invocation, subsequently analyzing the dynamic call graph, and measuring performance characteristics for each of the selected phases. Wang et al. [2012] propose a phase detection method for loop-based programs on a multiprocessor system-on-a-chip (MPSoC). Based on the "hot" functions of the program, located by a cross compiling tool based on ARM's RealView Development Suite (RVDS), they target function optimization on a Hadoop cluster. These approaches require hardware support or use interrupts to gain hardware-specific information. While it is conceivable that interrupt driven methods could operate in real time on stock hardware, we

are not aware of this having been applied to specific Java programs. Our approach is designed for Java, and works with stock Java virtual machines on stock hardware.

Shen et al. [2004] detect locality phases using variable-distance sampling, wavelet filtering, and optimal phase partitioning. Singer and Kirkham [2008] investigate the possibility of using concept information within a framework for dynamic analysis of programs. They demonstrate two different styles of concept visualization, which show the proportion of overall time spent in each concept and the sequence of concept execution, respectively. Pirzadeh et al. [2011] propose a trace exploration approach based on examining trace execution phases by adopting the Term Frequency, Inverse Document Frequency (TF-IDF) technique and the cosine similarity measure. Benomar et al. [2014] propose an automatic approach for the detection of high-level execution phases from previously recorded execution traces, based on object lives, and without the specification of parameters or thresholds. The method is simple, based on the heuristic that different phases tend to involve different objects. These papers are interested in phases as part of program analysis and understanding, so the methodologies they use are offline, not real time.

In early work, Sherwood et al. [2002] developed automatic techniques that are capable of finding and exploiting the large scale behavior of programs (behavior seen over billions of instructions). Nagpurkar and Krintz [2004] developed a framework for Java virtual machines that allows application developers, as well as architects of dynamic optimization systems, to visualize, investigate, and experiment with phase behavior data in Java programs. Lau et al. [2005] show graphically that there exists a hierarchy of phase behavior in programs, and they motivate the need for variable length intervals. Cho and Li [2006] uses wavelet techniques to represent program phases at multi-resolution scales to analyze, quantify, and classify the dynamics and complexity of program phases. Watanabe et al. [2008] use an LRU cache for observing a working set of objects, and interpret a sharp rise in the frequency of cache updates as a phase transition. Wimmer et al. [2009] perform local phase detection based on trace trees, which are a collection of frequently executed code paths through a code region, and are generated by trace recording and compilation. Pirzadeh et al. [2010] present a phase detection approach that identifies when and where, during the execution of a program, the methods that implement a particular phase start to disappear as new methods begin to emerge, indicating the beginning of another phase. Zhang et al. [2015] propose an investigation of the potential of multilevel phase analysis (MLPA), where phase analyses with different granularity are combined to improve overall accuracy. These approaches provide rich and useful results, but only after the fact—they cannot be applied in real time.

Nagpurkar et al. [2006] focus on the enabling technology of online phase detection using a similarity model. The method is online in that it works via a single pass over trace data. It consumes profile elements from a trace and transforms them into a sequence of similarity values that represent the degree of similarity between recent profile elements. It employs a similarity analyzer that determines whether the similarity is sufficient to signify that execution is in a phase or in transition between phases. Otte and Richardson [2007] present a semi-online program phase analysis using hidden Markov model (HMM) approaches. They present two approaches to the problem of phase detection and prediction: a Vector Quantized hidden Markov model (VQ-HMM) and a Continuous Density hidden Markov model (CD-HMM), achieving a 90% score in their best result. They work in a single forward pass over a stream of information about the program. Even though these techniques can achieve high accuracy with lower cost than offline phase detection systems, they do not operate in real time. Even if they did, the cost of analyzing each branch as it occurs would slow program execution dramatically.

7. CONCLUSIONS AND FUTURE WORK

We have presented a method for *real-time* phase transition detection for Java programs. After applying a training protocol to a program of interest, our method can detect phase changes at run time for that program with good precision and recall and without significant performance impact.

We trace a number of executions of the program of interest, recording information about each call/return and conditional branch that occurs, and use the traces to develop an efficient offline real-time phase transition detector. We choose a small number of branch instructions that we will instrument at run time and apply Gaussian mixture models (GMMs) to cluster vectors of branch execution counts. We use seven DaCapo benchmarks, plus a modified version of `javac`, to test our system. Our experiments show that our offline GMM detector can achieve results similar to, and even better than, the detector of Nagpurkar et al. [2006] and a detector that reports transitions randomly with probability equal to the average rate of transitions as determined by a reference definition of phases and transitions.

In comparing detectors, we found that the scoring scheme of Nagpurkar et al. [2006] did not discriminate well between the random detector and others that are more accurate, so we developed a new comparison method that we call the Shifted Weighted F (SWF) score.

To make run-time measurements, we implemented the phase transition detector as a Java agent, using the Java Virtual Machine Tool Interface (JVMTI). We count the instrumented branches, and when their total count reaches a threshold, we determine a cluster for the recent past. If that cluster is different from the previous cluster, we judge the program to have transitioned from one phase to another.

Our experimental results show that our method does not significantly impact performance of the running program, and the detector corresponds well with the underlying definition of phases ("ground truth"). The SWF score that our real-time phase transition detector achieves is similar to that of offline ND and better than the random detector.

In the future, we hope to apply similar techniques to programs written in other languages, ranging from statically compiled languages such as C and C++ to dynamic scripting languages such as Python.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1320498.

References

O. Benomar, H. Sahraoui, and P. Poulin. Detecting program execution phases using heuristic search. In *Search-Based Software Engineering*, pages 16–30. Springer, 2014.

S. M. Blackburn, R. Garner, C. Hoffman, A. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, S. Z. Guyer, A. Hosking, M. Jump, J. E. B. Moss, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Widerman. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 2006 ACM International Conference on Object-Oriented*

Programming Systems, Languages, and Applications (OOPSLA 2006), pages 169–190, Portland, OR, October 2006.

C.-B. Cho and T. Li. Complexity-based program phase analysis and classification. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006*, pages 105–113. ACM, 2006.

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

S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmark. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 92–115, London, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5. URL http://dl.acm.org/citation.cfm?id=646156.679838.

E. Duesterwald, C. Caşcaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2003*, pages 220–231. IEEE, 2003.

A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 270–287, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. . URL http://doi.acm.org/10.1145/1028976.1028999.

D. Gu and C. Verbrugge. Phase-based adaptive recompilation in a JVM. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 24–34. ACM, 2008.

W. ke Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, PLDI '06, pages 332–340. ACM, 2006.

J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2005*, pages 135–146. IEEE, 2005.

G. McLachlan and D. Peel. *Finite mixture models*. John Wiley & Sons, 2004.

P. Nagpurkar and C. Krintz. Visualization and analysis of phased behavior in Java programs. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, pages 27–33. Trinity College Dublin, 2004.

P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–123. IEEE Computer Society, 2006.

M. Otte and S. Richardson. An HMM applied to semi-online program phase analysis (CU-CS-1034-07). Technical report, Univ. of Colorado Boulder, 2007.

N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 150–162. IEEE, 2007.

H. Pirzadeh, A. Agarwal, and A. Hamou-Lhadj. An approach for detecting execution phases of a system for the purpose of program comprehension. In *Eighth ACIS International Conference on Software Engineering Research, Management, and Applications (SERA), 2010*, pages 207–214. IEEE, 2010.

H. Pirzadeh, A. Hamou-Lhadj, and M. Shah. Exploiting text mining techniques in the analysis of execution traces. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 223–232. IEEE, 2011.

N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Tool demonstration: Elephant Tracks—generating program traces with object death records. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, pages 39–43, Kongens Lyngby, Denmark, Aug. 2011. ACM.

N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: Portable production of complete and precise GC traces. In P. Cheng and E. Petrank, editors, *International Symposium on Memory Management, ISMM '13, Seattle, WA, USA - June 20 - 20, 2013*, pages 109–118. ACM, 2013. ISBN 978-1-4503-2100-6. . URL http://doi.acm.org/10.1145/2464157.2466484.

Y. Roh, J. Kim, and K. H. Park. A phase-adaptive garbage collector using dynamic heap partitioning and opportunistic collection. *IEICE TRANSACTIONS on Information and Systems*, 92(10): 2053–2063, 2009.

X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 165–176, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. . URL http://doi.acm.org/10.1145/1024393.1024414.

T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review*, 36(5):45–57, 2002.

J. Singer and C. Kirkham. Dynamic analysis of Java program concepts for visualization and profiling. *Science of Computer Programming*, 70(2):111–126, 2008.

C. Wang, X. Li, D. Dai, G. Jia, and X. Zhou. Phase detection for loop-based programs on multicore architectures. In *2012 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 584–587. IEEE, 2012.

Y. Watanabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache. In *Proceedings of the 2008 International Workshop on Dynamic Analysis; held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 8–14. ACM, 2008.

C. Wimmer, M. S. Cintra, M. Bebenita, M. Chang, A. Gal, and M. Franz. Phase detection using trace compilation. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ)*, pages 172–181. ACM, 2009.

F. Xian, W. Srisa-an, and H. Jiang. Microphase: An approach to proactively invoking garbage collection for improved performance. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 77–96, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. . URL http://doi.acm.org/10.1145/1297027.1297034.

C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM 2006, pages 174–183. ACM, 2006.

W. Zhang, J. Li, Y. Li, and H. Chen. Multilevel phase analysis. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2): 31, 2015.