Visualization and Analysis of Phased Behavior in Java Programs

Priya Nagpurkar Chandra Krintz
Computer Science Department
University of California, Santa Barbara
{priya,ckrintz}@cs.ucsb.edu

Abstract

To enable analysis and visualization of phased behavior in Java programs and to facilitate optimization development, we have implemented a freely-available, phase analysis framework within JikesRVM. The framework couples existing techniques into a unifying set of tools for data collection, processing, and analysis of dynamic phased behavior in Java programs. The framework enables program and optimization developers to significantly reduce analysis time and target optimization (by-hand or automatic) to parts of the code that will recur with sufficient regularity. We use the framework to evaluate phased behavior in the SpecJVM benchmark suite.

1 Introduction

Recent research in the area of feedback-directed, hardware-based optimization has identified potential optimization opportunities in the repeating patterns in the time-varying behavior, i.e., *phases*, of programs [11, 3, 12]. Phases represent intervals during program execution that are similar (repeat). Hence, phase information can be used to reduce analysis and profiling overhead (analysis of an interval in a phase is the same as that for all intervals in the phase), and to identify repeating behavior that can be exploited with code specialization.

Phased behavior, if present in Java programs, has the potential for enabling significant performance improvements in both JVM and program execution. Phases can be used to reduce the overhead of online instrumentation and analysis and to guide optimization, adaptation, and specialization [5, 14, 15]. However, to date, phased behavior in Java programs has not been thoroughly researched. Moreover, there are many open questions about the various components of the methodology of phase behavior collection. For example, how many instructions, i.e., at what *granularity*, should we consider as the minimum phase size (program behavior)? Is this size application specific? In addition, how do we measure the *similarity* between program behaviors so that we distinguish different behaviors (phases)? Studies have shown that the answers to these questions significantly impact the detection of phase boundaries [6], and thus, the degree to which phases can be exploited.

To facilitate research into these questions and into the phased behavior in Java programs, we have developed a toolkit and JVM extensions for the collection and visualization of dynamic phase data. In addition, our framework enables researchers to experiment with the various parameters associated with phase detection and analysis, e.g., granularity and similarity. Our framework incorporates phase collection techniques used by the binary optimization and architecture communities into a freely-available JVM. Our toolset is intended for use offline to gather phase data and to simplify and facilitate phase analysis as part of the design and implementation process of high-performance Java programs and JVM optimizations. We first describe the system in detail then show how it can be used to visualize and analyze phased behavior in a set of commonly used Java benchmarks.

2 JVM Phase Framework

To better understand the potential benefits from exploitation of phased behavior in Java programs, we 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. The framework consists of a *data generator* and a *set of tools for the extraction and analysis of phased behavior*. In this section, we describe the implementation of each and discuss how each component can be parameterized to facilitate research and investigation into the customization of phase collection and analysis.

We implemented the phase framework as an extension to the Jikes Research Virtual Machine (JikesRVM) from IBM T.J. Watson Research [1]. JikesRVM is an adaptive optimization system that uses online instrumentation to apply optimization dynamically to methods of a Java program that account for a significant portion of the execution time. We extended this JVM with instrumentation for basic blocks that causes basic block frequencies to be collected into an array, i.e., a basic block vector, during program execution. In addition, we apply the highest level of optimization to all application and library methods.

To capture the time-varying nature of program behavior, we decompose program execution into *intervals*, each representing the execution of a fixed number of dynamic instructions. We can specify interval size as a command-line parameter; we use a value of 5 million in this study. At the end of each interval, we take a snapshot of the basic block vector and store it in an *Interval Queue*; a background thread periodically empties the queue to disk. Upon program termination, a file on disk contains a trace of per-interval basic block vectors for the program.

We next describe the second component of our phase framework: A set of tools for the extraction and analysis of phased behavior from the profile information produced by the data generator. The toolkit consists of four tools: an image generator and visualizer, a phase finder, a phase analyzer, and a code extractor. The tools are written in Perl and Java and are easily extensible.

2.1 Phase Visualizer

The phase visualizer consumes the phase trace and generates a portable graymap image from it. This image can be viewed using any image viewer; however, we developed our own Java-based viewer that enables users to point (using the mouse) to a pixel on the image and view the interval coordinates. These coordinates allow the user to identify intervals within a visualized phase.

An image produced by the phase analyzer is shown in Figure 1. The data in the figure was taken from the phase trace of SpecJVM benchmark _201_compress using input size 100. Each image is a *similarity matrix* [10]; the x-axis and y-axis are increasing interval identifiers (ids). An interval is a period of program execution (specified during trace collection); we assign interval ids in increasing order starting from 0.

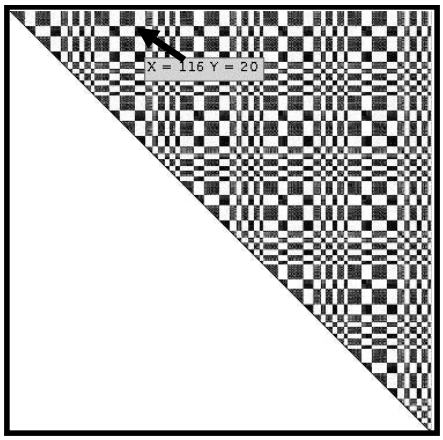

Figure 1: Phase Visualizer on Compress100

The visualizer omits data in the lower triangle since it is symmetric with the upper triangle. Each point, with coordinates *x* and *y*, denotes how similar interval *y* is to interval *x*. Dark pixels indicate high similarity. White indicates no similarity. A user can read the figure by selecting a point on the diagonal; by then traversing the row, she can visualize the degree to which the intervals that follow are similar. The grayscale depiction of similarity enables identification of phases, phases boundaries, and repeating phases over time. We discuss how we compute interval similarity in the next section.

2.2 Phase Finder

To determine which intervals are similar (and thus, the pixel color displayed by the visualizer), the second tool that we developed is the *Phase Finder*. The tool consists of two components: One that computes the similarity between intervals and another that clusters intervals into phases.

Two intervals are similar if the execution behavior that each represents is correlated. As such, we use the *vector distance* between the basic block vectors representing the two intervals. Specifically, we compute similarity as the Manhattan distance between two vectors: the sum of the element-wise absolute differences between two normalized vectors. The Manhattan distance is a value between 0 and 2. A difference value of zero implies that the two vectors are entirely similar and 2 indicates complete dissimilarity.

Other similarity metrics, e.g., vector angle [7], can be plugged into this component to enable evaluation of the efficacy of different techniques. Once we compute interval similarity, we map the similarity value to one of 65536 different grayscale values to generate the portable graymap image.

The clustering component, uses interval similarity to determine which intervals should be included in a phase. This component is also pluggable, i.e., any clustering algorithm can be inserted, experimented with, and evaluated in terms of its efficacy for phase discovery. We currently implement a simple threshold-based mechanism that identifies similar intervals for this component. The user provides a Manhattan distance threshold (between 0 and 2). This component then extracts intervals with Manhattan distances below the threshold specified. This simple mechanism enables users to adjust and experiment with the threshold value.

2.3 Phase Analyzer and Code Extractor

As part of our toolset, we also developed two tools that enable users to extract statistics as well as code from each phase or interval: The *Phase Analyzer* and the *Code Extractor*.

The phase analyzer generates and filters data to aid in the analysis of phases and individual intervals. It lists the intervals in each phase as well as how often the phase occurs and in what durations over the execution of the program. The phase analyzer extracts details about the behavior of individual intervals or entire phases. For example, it reports the number of phases found, the number of instructions in each phase (over time), and how many instructions occur in dissimilar intervals that interrupt the different phases. Moreover, it lists sorted basic block and method frequencies. This information is useful for identifying “hot spots” in the execution.

For all of the data reported by the phase analyzer, we include a number of filters that significantly simplify analysis of the possibly vast amounts of data generated for a program. For example, a user can specify a threshold count below which data is not reported. This enables users to analyze only the most frequent data. In addition, data can be combined into cumulative counts or into a number of categories, e.g., instructions, basic blocks, methods, and types of instructions.

Finally, to analyze the program code that makes up a phase, we developed a code extraction tool. By inputting intervals identified by the visualizer and statistics generated by the phase analyzer, users can use the code extractor to dump code blocks of interest. The granularity of the dump can be specified to be a single basic block, a series of basic blocks, or an entire method. We show how the tools in the toolkit can be employed in the next section.

3 Phase Behavior in Java Programs

The data we present in this section was gathered by executing Java benchmarks on a 1.13Ghz x86-based single-processor Pentium III machine running Redhat Linux v2.4.5 We used JikesRVM version 2.2.2. The programs we examined are from the SpecJVM98 benchmark suite [13].

Each of the similarity graphs that follow, can be read, as described previously, as an N x N similarity matrix, where N is the number of intervals in the program’s execution. An entry in the matrix at position (x,y) is a pixel colored to represent the similarity between interval x and interval y. The diagonal is black since an interval is

entirely similar to itself. To see how an interval relates to the remaining program execution, we locate the interval of interest, say x, on the diagonal and move right along the row. Dark areas in the row identify intervals that are similar in behavior to interval x.

As an example, consider the similarity matrix for the multi-threaded ray trace benchmark, Mtrt, when we execute it with input size 10 (Figure 2). Mtrt executes 65 intervals of 5 million instructions. We start at the top left corner of the matrix and move right along the x-axis. As we move right, we encounter dark pixels until we reach interval 15. That is, the initial phase, phase-1, begins at interval 0 and continues through interval 15. Interval 15 is entirely dissimilar and therefore belongs to another phase, which we call phase-2. After interval 15, the intervals alternate between phase-1 and phase-2 until we reach interval 44. From interval 44 until the end of the execution, the intervals are completely dissimilar to phase-1.

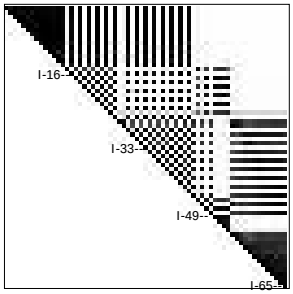

Figure 2: Similarity graph for Mtrt10

Up to this point, we have visually discerned two phases. We have concluded that the intervals in phase-2 and intervals 44 through 65 are completely different from the intervals in phase-1. Now, we must investigate how intervals from interval 44 through 64 relate to each other. We do this by locating interval 44 on the diagonal and evaluating its row in the same way. We can observe two different phases in this row. It is important to note here that the dark intervals we encounter in row 44 are in no way related to the dark intervals in phase-1 even though the color may be the same. That is, a row in a similarity matrix identifies the similarity between the row interval and all future intervals only.

Figure 3 shows the phases found by our phase-finder using a similarity threshold of 0.8 for Mtrt. We use a figure to depict the output of the phase finder. Each pattern indicates a different interval; there were 3 phase detected. Using the phase analyzer and code extractor, we can further evaluate each phase by analyzing the commonly executing methods. The most frequently executed methods in phase-1 are *ReadPoly* in class Scene and <*init*> in class *PolyTypeObj*. In phase-2 the most frequently executed methods are *CreateChildren* and *CreateFaces* in class *OctNode*. phase-3 then renders the scene by frequently executing *Intersect* from class *OctNode*, *Combine* from class *Point*, and *RenderScene* from class *Scene*.

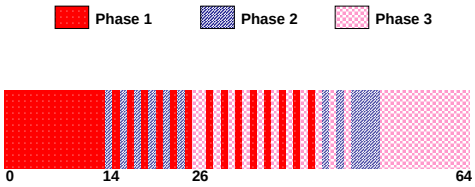

Figure 3: Phases for Mtrt10 (thresh: 0.8)

Figure 4 shows the interval similarity matrices for all of the SpecJVM benchmarks; we showed the omitted benchmark, Compress, previously. Visual analysis of the benchmark data provides insight into the phase behavior in the programs and also enables us to target our efforts for further analysis using the other tools as we did above for Mtrt. We can see that each of the benchmark programs exhibits very different patterns.

In DB, Javac, Jess, and Mtrt there is a clear startup phase. Existing adaptive systems have shown that it can be profitable to consider startup behavior separately from the remaining execution [15, 14]. This phase in these four benchmarks is noticeably different from the rest of the execution. This is particularly evident in Javac. Using further analysis of the number of instructions executed by different methods (using the phase analyzer), we find that the most popular methods are *read*, *scanIdentifier*, and *xscan* during the first 90 intervals. They are again the most popular methods during intervals 202-212, depicted by the dark vertical bar in the right part of the startup phase. These methods are rarely executed in all other phases.

Other programs exhibit other interesting phase features. Mpegaudio shows no apparent phased behavior. That is, each interval is very similar to every other. Mtrt shows very dark intervals also, however there is a perceivable

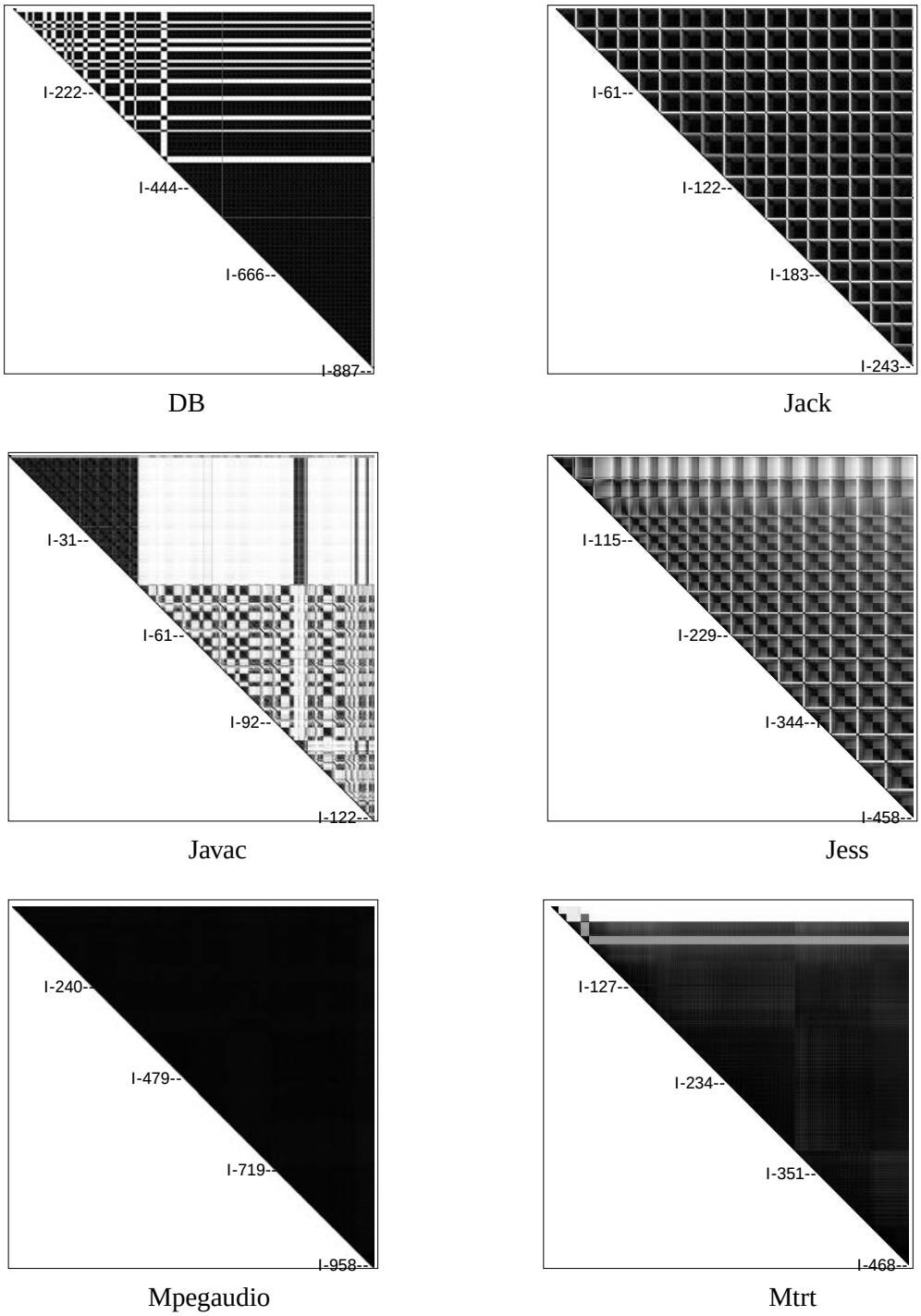

Figure 4: Similarity graphs for the SpecJVM benchmarks with input size 100. The interval length used is 5 million instructions. The number of intervals, n, vary and each graph is a n x n matrix with the x and y axes representing the interval identifier. The lower triangle is a mirror image of the upper one and is masked for clarity. Each point on the graph indicates the similarity between the intervals represented by that point. Dark implies similar and light implies dissimilar. The diagonal is dark, since every interval is entirely similar to itself.

pattern that traverses the matrix of Mtrt. Jack exhibits a very regular pattern: 16 rows of almost perfect squares. Output from our phase analyzer for Jack reveals that the code does repeat itself 16 times. The reason for this is that this benchmark is a parser generator that generates the same parser 16 times. Our framework correctly identifies this repeating phase behavior.

4 Related Work

Runtime phased behavior of programs has been previously studied and successfully exploited primarily in the domain of architecture and operating systems [8]. The basis of our framework is to combine existing techniques that have proven to be successful in these domains within an adaptive JVM context. [9] and [10] are two such techniques. The authors of these works propose to use basic block distribution analysis to capture phases in a program's execution. They use phase information to reduce architectural simulation time by selecting small representative portions of the program's execution for extensive simulation. Basic block vectors are used to characterize program behavior across multiple intervals of fixed duration and are classified into phases using Manhattan distance and k-means clustering. [11] presents an online version of such phase characterization along with a phase prediction scheme. The authors of this work also describe additional applications to configurable hardware. Our framework employs this methodology within the data gathering component and phase finding tool to enable collection and analysis of phase data in Java programs.

The authors in [2] and [4] stress the importance of exploiting phased behavior to tune configurable hardware components. In the former, the authors compare working set signatures across intervals using a similarity measure called *relative working set distance* to detect phase changes and identify repeating phases. In [4], the authors use hardware counters to study the time-varying behavior of programs and use it in the design of online predictors for two different microarchitectures. In other work [3], the authors compare three different metrics that characterize phased behavior. The metrics are basic block vectors, branch counters and instruction working sets.

Hind et al. [6] examine the fundamental problem of phase shift detection and analyze its dependence on the two parameters that define phased behavior, granularity and similarity. They demonstrate that for the SpecJVM benchmark suite, observed phase behavior depends on the choice of parameter values. Our framework allows the user to specify and experiment with both, possibly application-specific, parameters.

5 Conclusion

To enable the study of time-varying behavior, i.e., *phased* behavior, in Java programs, we developed an offline, phase visualization and analysis framework within the JikesRVM Java Virtual Machine. The framework couples existing techniques from other research domains (architecture and binary optimization) into a unified set of tools for data collection, processing, and analysis of dynamic phased behavior in Java programs. The framework enables program and optimization developers to significantly reduce analysis time and to target parts of the code that will recur with sufficient regularity.

References

[1] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2000.

[2] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[3] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. In *36th Annual International Symposium on Microarchitecture*, December 2003.

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

[5] S. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization (CGO)*, March 2003.

[6] M. Hind, V. Rajan, and P. Sweeney. The Phase Shift Detection Problem is Non-Monotonic. Technical Report RC23058, IBM Research, 2003.

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

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

[9] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[10] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages*, October 2002.

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

[12] M. Smith. Overcoming the challenges to feedback-directed optimization. In *ACM Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, July 2000.

[13] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[14] John Whaley. Partial Method Compilation using Dynamic Profile Information. In *Proceeding of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA*, pages 166–179. ACM Press, October 2001.

[15] L. Zhang and C. Krintz. Code unloading. Technical Report 2003-14, University of California, Santa Barbara, 2003.