Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 59 (2006) 64–81

Science of Computer Programming

www.elsevier.com/locate/scico

Phase-based visualization and analysis of Java programs

Priya Nagpurkar*, Chandra Krintz

Computer Science Department, University of California, Santa Barbara, United States

Received 4 October 2004; received in revised form 22 February 2005; accepted 21 March 2005
Available online 11 August 2005

Abstract

Extant Java Virtual Machines (JVMs) apply dynamic compiler optimizations adaptively, based on the partial execution of the program, with the goal of improving performance. Understanding and characterizing program behavior is of vital importance to such systems. Recent research, primarily in the area of computer architecture, has identified potential optimization opportunities in the repeating patterns in the time-varying behavior of programs. In view of this, we believe that by considering *time-varying*, i.e., *phase*, behavior in Java programs, adaptive JVMs can enable performance that exceeds current levels.

To enable analysis and visualization of phase behavior in Java programs and to facilitate optimization development, we have implemented a freely available, offline, phase analysis framework within the IBM Jikes Research Virtual Machine (JikesRVM) for Java. The framework couples existing techniques into a unifying set of tools for data collection, processing, and analysis of dynamic phase behavior in Java programs. The framework enables optimization developers to significantly reduce analysis time and to target adaptive optimization to parts of the code that will recur with sufficient regularity. We use the framework to evaluate phase behavior in the SpecJVM benchmark suite and discuss optimizations that are enabled by the framework.

Keywords: Program phases; Program behaviour visualization; Program analysis; Java

* Corresponding author.
E-mail addresses: priya@cs.ucsb.edu (P. Nagpurkar), ckrintz@cs.ucsb.edu (C. Krintz).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.07.004

1. Introduction

Dynamic program analysis and optimization is emerging as a promising technique for improving Java program performance. Using information gathered at run time, the dynamic compilation system can identify and implement profitable optimizations. 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 [23,21,6,24,8,20].

The notion of phases arises from the observation that program behavior can vary widely but also commonly exhibits repeating patterns [23]. To capture phase behavior in programs, we decompose the program into fixed-sized intervals. We then combine intervals with similar execution characteristics into a phase, regardless of temporal adjacency. As a result, a phase can be as small as a single interval and as large as the entire execution of the program.

Phase behavior, if present in Java programs, has the potential for enabling significant performance improvements in both JVM and program execution. However, to date, phase 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 make up an interval, i.e., at what *granularity*, should we observe 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)? How similar do the intervals have to be, to belong to the same phase? Studies have shown that the answers to these questions significantly impact the detection of phase boundaries [10], and thus, the degree to which phases can be meaningfully exploited.

To facilitate research into these questions and into the phase 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 JikesRVM [11], a freely available research 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 and then show how it can be used to visualize and analyze phase behavior in a set of commonly used Java benchmarks.

In the following section, we describe the design and implementation of our framework for the collection, categorization, and analysis of dynamic phase behavior in programs. In Section 3, we discuss how to employ the framework to simplify program analysis and to expose phase behavior and optimization opportunities in Java programs. In the remainder of the paper, we detail related work (Section 4) and present our conclusions and future work (Section 5).

2. JVM phase framework

To enable analysis and visualization of phase behavior in Java programs and to facilitate optimization development, we have implemented the phase analysis framework shown in

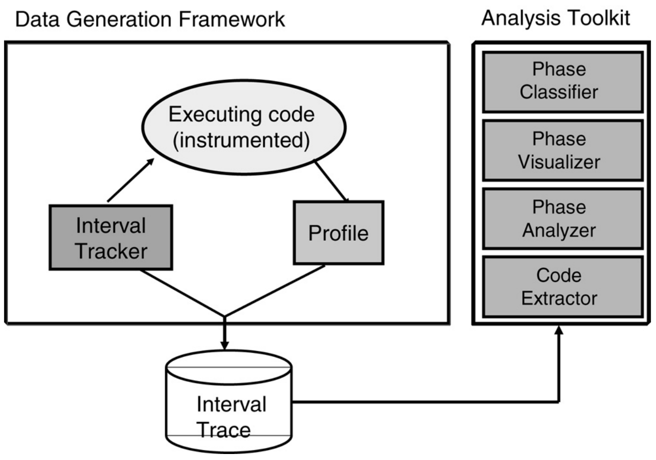


Fig. 1. JVM framework and toolkit for the analysis of phase behavior in Java programs.

Fig. 1. It consists of a *Data Generation Framework* and a *Data Processing Toolkit*. The data generation framework is responsible for capturing and storing the behavior of the program as it executes. It is a generic profile collection facility that can be employed by any utility that is able to extract dynamic behavior from programs, e.g., simulators, profilers, and virtual execution environments. In this work, we extended a Java Virtual Machine with the data generation framework to enable us to study phase behavior in Java programs.

The only constraint on data collection is the type of the profile and the output format. The profile type (collected by the *Profile Generation Component*) is basic block counts. Our framework employs this profile type as it not only captures code execution patterns, but has also been shown to mirror how the program exercises the underlying hardware resources [21]. We store the counters in an array, i.e., a vector, with length equal to the number of static basic blocks in the program. Each time a basic block is executed, its entry in the vector is incremented.

The *Interval Tracker* collects basic block vectors for every fixed-length interval. The length of the interval is in terms of number of dynamic instructions executed. The Interval Tracker uses Hardware Performance Monitoring support to keep track of the number of dynamic instructions executed. Each vector thus represents the execution behavior of the program during that interval. The final *Interval Trace* is a set of basic block vectors per interval over the life of the program.

We implemented the data generation framework within the JikesRVM, an adaptive optimization Java Virtual Machine from IBM T.J. Watson Research Center [3]. JikesRVM is written in Java and provides extensive runtime services such as garbage collection and allocation, thread management, adaptive compilation, synchronization, and exception handling. It includes two compilers, a fast baseline compiler, and the optimizing compiler. The baseline compiler is very quick and generates code with execution speeds similar to that of interpreted code. The optimizing compiler incorporates extensive compiler technology and provides three different levels of optimization. As shown in Fig. 2, we

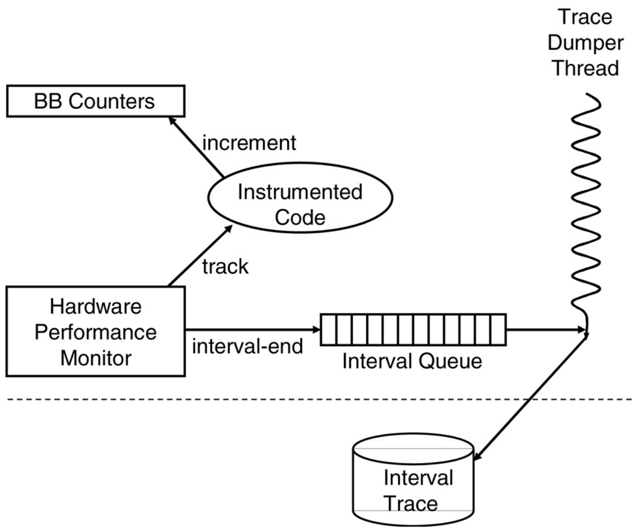


Fig. 2. Architecture of the data generation framework. The instrumented application updates counters for the current snapshot (profile) interval in an interval queue. When the hardware performance monitors (HPMs) indicate that the interval granularity has been reached, a background dumper thread is signalled. The dumper thread expels all past snapshots to an interval trace on disk. The HPMs collect hardware data on a per-thread data; we monitor only application threads.

extended this JVM to generate per-interval profiles, track intervals, and export an execution snapshot at regular intervals.

To generate the profile, we extended the optimizing compiler in the JikesRVM to insert instrumentation into every basic block. We instrument only application methods and library methods used by the application. However, this can be easily extended to include JVM methods (since this JVM is written in Java). The instrumentation consists of a counter identified by a unique identifier consisting of a method and basic block id.[1] We employed the JikesRVM configuration that uses only the optimizing compiler at the highest level of optimization. This allows us to investigate optimization opportunities that remain even after full optimization of the program.

We periodically store the basic block vector into an *Interval Queue* and reset the vector for use during the next interval. The period that the system uses is a modifiable granularity parameter, i.e., interval size, which is specified as the number of dynamic instructions executed by the program. When the interval queue fills up, a background trace dumper thread copies all stored intervals from the queue to a disk file. Once the dumper thread is scheduled, it remains scheduled until all past intervals are written.

The interval trace is analyzed offline by the *Data Processing Toolkit*. The Data Processing Toolkit consists of different components to collect intervals into phases (*Phase Finder*), visualize the phase behavior (*Phase Visualizer*), analyze the program behavior

[1] We must include the method identifier since basic block ids are not unique across methods.

within phases (*Phase Analyzer*), and extract important code sequences from within phases (*Code Extractor*). We describe each of these open-source, easily extensible, components in the following subsections.

2.1. Phase finder

To characterize program behavior into phases, we compare how every interval relates to every other interval. Intervals that are similar to each other constitute a phase. Note that these intervals need not be temporally adjacent. This functionality is provided by the *Phase Finder*, which consists of pluggable components that operate offline on the trace produced by the framework. The two components implemented by the phase finder compute the similarity between intervals and cluster intervals into phases.

The Phase Finder computes similarity by comparing two intervals and generating a value that indicates how similar the two intervals are in terms of their execution behavior. Any similarity metric, e.g., absolute element-wise difference, Manhattan distance, vector angle [13], etc., can be used to compute similarity. For this study, we compute similarity using the Manhattan (or city-block) distance. The Manhattan distance is the distance between two points measured along axes at right angles as against the Euclidean or straight-line distance. The Manhattan distance weighs differences in each dimension more heavily than the straight-line distance and is therefore more suitable for data with high dimensionality (which in our case is the number of static basic blocks).

To compute Manhattan distance, the phase finder first weights each basic block count by the static instruction count (within the block) and normalizes the weighted frequencies by dividing them by the sum of all weighted frequencies in the vector — we perform this step since we are interested relative rather than absolute values. The phase finder then computes the sum of the element-wise absolute differences between two vectors. A difference value of zero implies that the two vectors are entirely similar and 2 denotes complete dissimilarity.

The phase finder then clusters the intervals together into phases based on their similarity value. This component is also pluggable, i.e., any clustering algorithm, e.g., threshold-based clustering, k-means clustering, minimum spanning tree clustering, can be inserted, experimented with, and evaluated in terms of its efficacy for phase discovery. For this study, we employed a threshold-based approach. Intervals with Manhattan distances below the threshold are considered to be in the same phase. This simple mechanism enables users to adjust the threshold value to vary the number of intervals in each phase. We found experimentally that a threshold of 0.8 accurately reflects the how the program exercises the underlying hardware resources; it is the value we used in this study.

2.2. Phase visualizer

The *Phase Visualizer* consumes the similarity values from the phase finder and maps them to one of 65536 different grayscale values to generate 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.

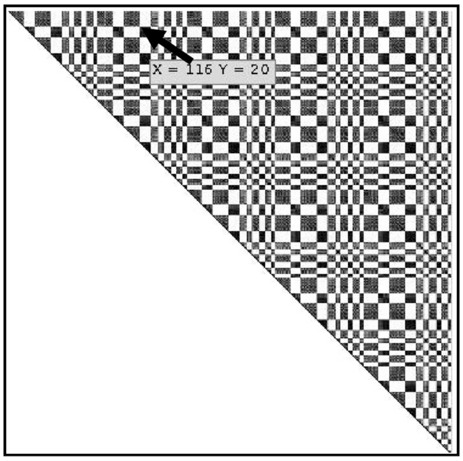


Fig. 3. Phase Visualizer. The visualizer is a Java program that displays interval data in the portable graymap format. The axes are interval id; intervals for this run are 5 million instructions. There are a total of 1312 intervals (x- and y-axis entries). The program is the Compress SpecJVM benchmark executed with input size 100.

An image produced by the phase analyzer is shown in Fig. 3. The data in the figure was taken from the phase trace of SpecJVM benchmark Compress using input size 100. Each image is a *similarity matrix* [22]; the x-axis and y-axis are increasing interval identifiers. An interval is a period of program execution (specified during trace collection); we assign interval ids in increasing order starting from 0. In this figure we use an interval size of 5 million instructions, which generates 1312 intervals. 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, phase boundaries, and repeating phases over time.

2.3. Phase analyzer and code extractor

We also developed two tools to 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 data can be reported as weighted or unweighted counts. An unweighted count is the number of times a basic block or method executes during the phase or interval. A weighted count is this same number multiplied by the number of instructions in each block.

For all 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 we employ all of the tools in the toolkit in the next section.

3. Employing the framework

In this section, we describe different ways in which our JVM phase framework can be used to understand and analyze phase behavior in Java programs and to guide adaptive optimization.

Before examining how each of the components in the toolkit can be applied, it is worth mentioning that our framework allows researchers to experiment with the granularity (of the interval size) and the similarity, parameters which have been shown to significantly impact phase-shift detection [10]. The interval size dictates the granularity at which we study program behavior. If the interval size is too large, we lose information about the time-varying behavior within the interval. If, on the other hand, it is too small, the behavior captured is more fine-grained than required. The similarity threshold influences the length of phases and the extent to which program behavior within the phase is uniform. Lower similarity thresholds generate a small number of long phases, whereas higher similarity thresholds generate many short phases. The choice of both interval size and similarity threshold can be governed by several factors, e.g., the application being analyzed, the intended use of phase information, etc. For example, if the phase information is to be used in dynamic optimization by the JVM, the similarity threshold ensures that the execution characteristics within that phase are similar enough for the intended optimization to be applicable to the entire phase.

The data we present in this section was gathered by executing Java benchmarks on a 1.13 GHz x86-based single-processor Pentium III machine running Redhat Linux, kernel v2.4.5., patched with perfctr for performance monitoring counters support. We used JikesRVM version 2.2.2 build 4-22-03 with an extension for Hardware Performance Monitoring support for x86 provided by the PANIC group at Rutgers University [18]. We use Hardware Performance Monitoring support to track the number of dynamic instructions executed. The benchmarks we examined are described in Table 1.

3.1. Visual analysis

As mentioned previously, we visualize program phase behavior using an upper triangle of an $N \times N$ similarity matrix, where N is the number of intervals in the program's

Table 1
Description of the benchmarks used

Program	Description
Compress	SpecJVM (201) Compression utility
DB	SpecJVM (209) Database access program
Jack	SpecJVM (228) Java parser generator based on the Purdue Compiler Construction Tool set
Javac	SpecJVM (213) Java to bytecode compiler
Jess	SpecJVM (202) Expert system shell: Computes solutions to rule-based puzzles
Mpegaudio	SpecJVM (222) Audio file decompressor Conforms to ISO MPEG Layer-3 spec.
Mtrt	SpecJVM (227) Multi-threaded ray tracing implementation

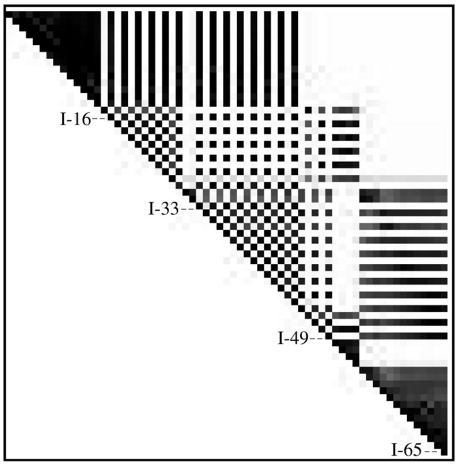


Fig. 4. Similarity graph for Mtrt input size 10.

execution. Each entry in a row or column represents an interval. Intervals are listed in each row or column in the order in which they occur in the program. An entry in the matrix at position (x, y) is a pixel colored to represent the similarity between interval x and interval y (black is similar, white is completely dissimilar). 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 pixels denote intervals in the same phase.

For example, consider the similarity matrix for the benchmark Mtrt when we execute it with input size 10 in Fig. 4. Mtrt executes for 65 intervals of approximately 5 million instructions each. 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 till 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.

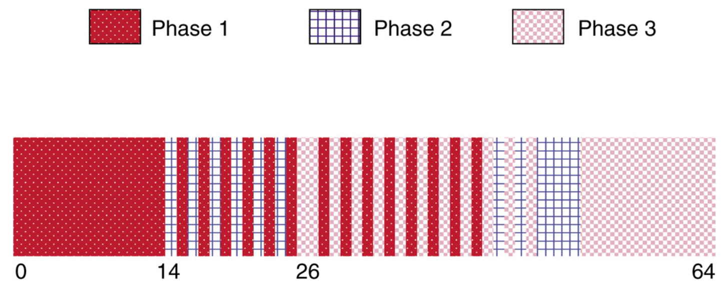


Fig. 5. Phases for Mtrt with similarity threshold 0.8. The *x*-axis represents the interval identifier. The program's execution has been broken down into 65 intervals of 5 million instructions each. The pattern of each interval indicates the phase it belongs to.

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.

Fig. 5 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 phases detected. Using the phase analyzer, 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.

Figs. 6 and 7 show the interval similarity matrices for all of the benchmarks; we have omitted Compress since we include it in a later discussion. The first page matrices show the phase behavior of the programs when they are executed using input size 100; the second page shows the matrices for input size 10. Visual analysis of the benchmarks 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 [27,25]. This phase in these four benchmarks is noticeably different from the rest of the execution. This is particularly evident in case of Javac input size 100. 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

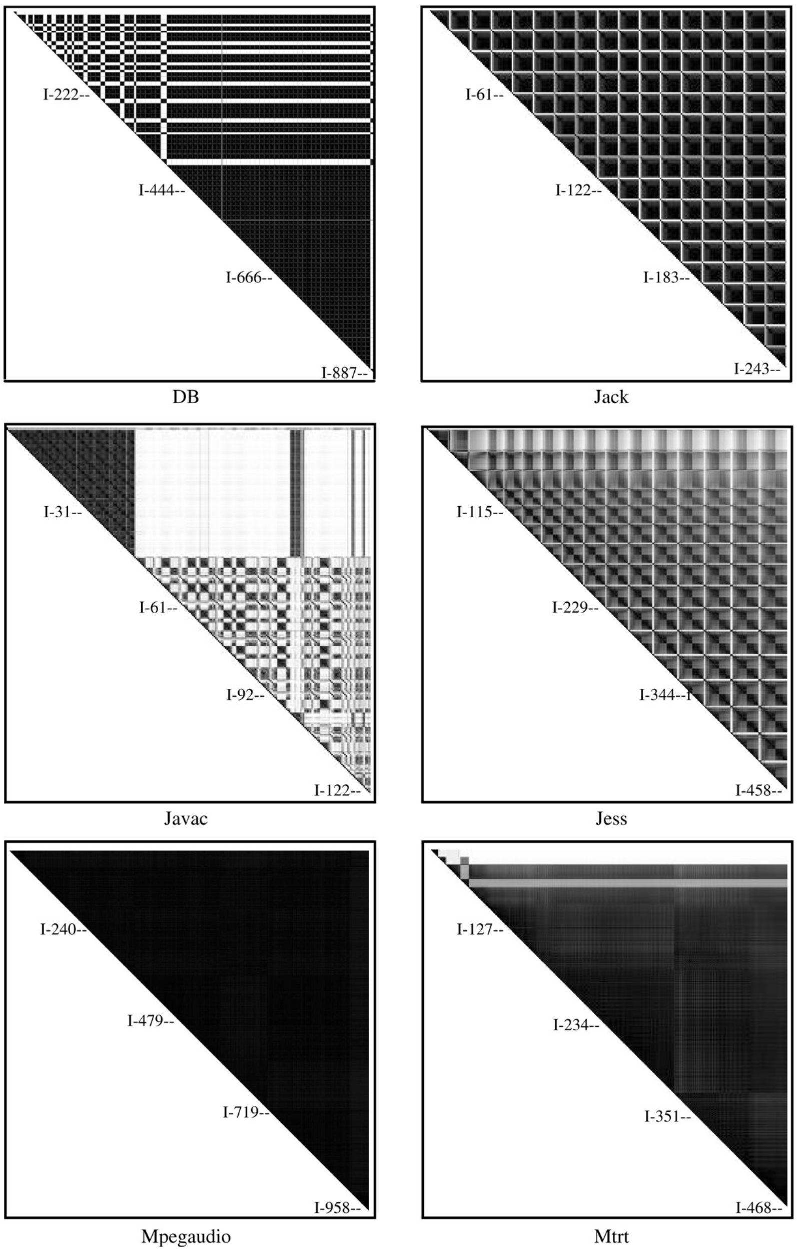


Fig. 6. Similarity graphs for the SpecJVM benchmarks with input size 100. The interval length used is 5 million instructions. The number of intervals, n, vary and each graph is a $n \times n$ matrix with the x and y axes representing the interval identifier. The lower triangle is a mirror image of the 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.

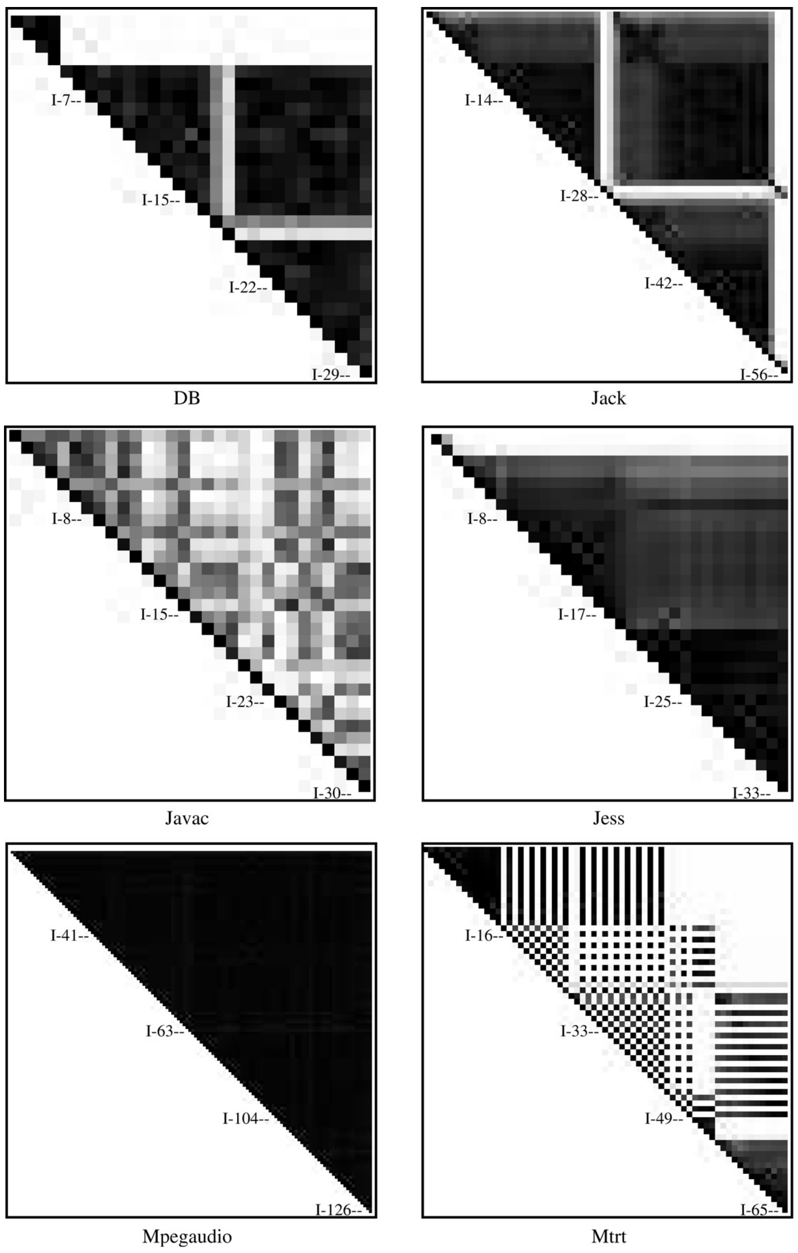


Fig. 7. Similarity graphs for the SpecJVM benchmarks with input size 10. The interval length used is 5 million instructions. The number of intervals, n, vary and each graph is a $n \times n$ matrix with the x and y axes representing the interval identifier. The lower triangle is a mirror image of the 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.

intervals 202–212, depicted by the dark vertical bar in the right part of the startup phase. They are rarely executed in all other phases. For input size 10, the same benchmarks exhibit startup phases in varying sizes. For example, for DB and Mtrt, both input size 10 and size 100 have the same length startup phase; The duration of the startup phase for Javac and Jess are different across inputs. All other benchmarks, exhibit no perceivable startup phase.

Other programs exhibit other interesting phase features. Mpegaudio for both input sizes shows no apparent phase behavior. That is, each intervals is very similar to every other. Mtrt for input size 100 shows very dark intervals also, however there is a perceivable pattern that traverses the matrix. 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 for input size 100 and twice for input size 10. The reason for this is because this benchmark is a parser generator that generates the same parser 16 times for input size 100 and twice for input size 10. Our framework correctly identifies this repeating phase behavior.

3.2. Code analysis

To demonstrate the use of the *Phase Analyzer* and *Code Extractor* components of the toolkit, we use them to analyze a frequently occurring phase in the SpecJVM Compress benchmark. For this experiment, we collected phase data using an interval size of 5 million instructions. We then employed the phase finder to extract highly similar intervals (using a Manhattan-distance difference threshold of 0.01). The phase we selected includes 11 intervals distributed across the execution of the program. We used the phase analyzer to filter weighted basic block counts so that we could immediately identify the most frequently executed basic block in the phase. Finally, we passed this basic block into the code extractor which dumped the register-based, low-level intermediate code (that uses the JikesRVM object model [2]) shown in Fig. 8. We prefix this code in the figure with source code of equivalent semantics and include comments for clarity.

The basic block is the fourth block in the decompress()V virtual method in the class Decompressor, and contains a single loop. The visualizer and analyzer show that this loop is not only hot (as can be discovered using extant profiling approaches) but that it recurs throughout execution — indicating that it is important to optimize this code (or perhaps the layout of the array objects so that the copy is more efficient) early on in the execution of the program and that the optimization will be amortized throughout the lifetime of the execution. Moreover, the execution of this code is interleaved with that from the compress methods (indicated by the alternating phase behavior in the visualization of the compress benchmark). This periodic behavior indicates that it may be possible to overlap optimization of decompress()V with execution of compress.

Using the Phase Analyzer and Code Extractor in this manner helps us to understand the program's behavior in terms of code execution patterns. We can use these components to identify the dominant methods and basic blocks in different phases and subject them to further analysis. We can also use these components to detect specialization opportunities by analyzing whether the same method is being used differently in different phases or whether the basic blocks in different phases belong to entirely different methods.

```
/*
do {
   a = y.obj2.ary[- - y.obj2.index];
   y.obj1.ary[y.obj1.counter++] = a;
} while (index != 0)
*/

LABEL BB4                        //start of BB4
R1 = R2(-52)                     //y.obj1
R4 = R2(-72)                     //y.obj2
R4(-20)- -                       //y.obj2.index- -
R3 = R4(-16)                     //y.obj2.ary[]
R6 = R4(-20)                     //y.obj2.index
array_bounds_check(R6,R3(-4))    //R6: y.obj2.index
                                 //R3(-4): y.obj2.ary.length
R5 = R1(-16)                     //y.obj1.counter (old)
FP(-20) = R5                     //spill R5
FP(-20)++                        //y.obj1.counter++ (new)
FP(-36) = R4                     //spill R4 (=y.obj2)
R4 = FP(-20)                     //new y.obj1.counter
R1(-16) = R4                     //y.obj1.counter = new
R1 = R1(-20)                     //y.obj1.ary[]
array_bounds_check(R5,R1(-4))    //R5: old y.obj1.counter
                                 // R1(-4): y.obj1.ary.length
R3 = R3(+R6)                     //y.obj2.ary[y.obj2.index]
R1(+R5) = R3                     //y.obj1.ary[old y.obj1.counter]
R4 = FP(-36)                     //restore R4 (=y.obj2)
cmp R4(-20), 0                   //compare y.obj2.index, 0
jne BB4                          //goto LABEL BB4 if !=
```

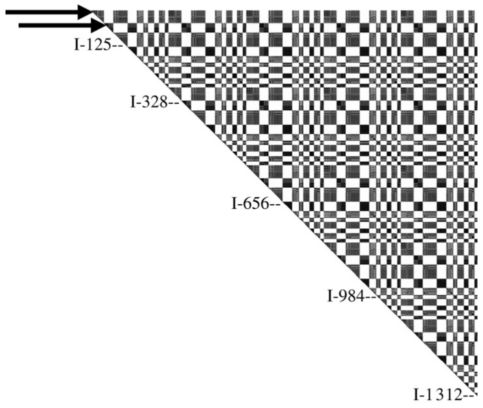
Fig. 8. Code extracted using the phase framework and toolkit for SpecJVM benchmark executed using input size 100. The code is contained in the most frequently executed basic block in the phase being analyzed. We prefix the low-level intermediate code dumped by the code extractor with the source code equivalent.

3.3. *Cross-input analysis*

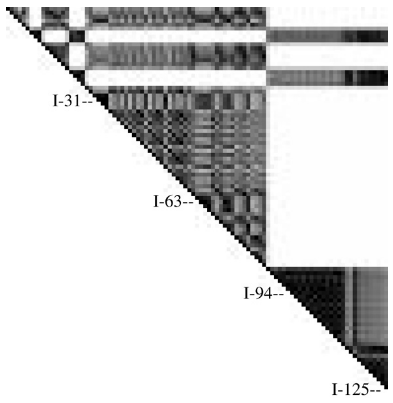
Commonly, offline profiling techniques are limited since they are dependent upon the input used [15]. When a different input is used, the assumptions about the execution change and hence, optimizations performed based on a cross-input, offline profile may be incorrect and impose needless overhead. As such, we are interested in when offline profiling techniques are likely to be effective. We can use our analysis framework to perform such cross-input analysis.

Fig. 9, in graphs (a) and (b), shows the similarity matrices for the Compress SpecJVM benchmark for input size 100 and size 10, respectively. The total number of intervals is 1312 for size 100 and 125 for size 10. The interval size in both graphs is 5 million instructions. The two graphs appear very different at first glance.

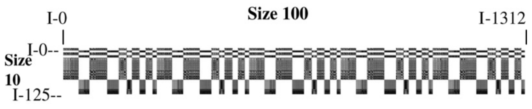
Fig. 9(c) shows the similarity matrix *across* the two Compress inputs. We compute this matrix using the phase finder to identify similarities across to different runs of the same program. Since the static number of basic blocks are the same, we can compare the two vectors as if they were from the same program. Now, however, the similarity matrix is not square. The rows are from input 10 and the columns are from input 100.



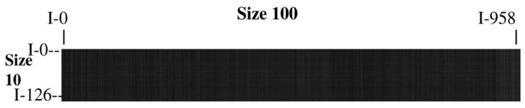
(a) Compress size 100.



(b) Compress size 10.



(c) Cross-input similarity for compress.



(d) Cross-input similarity for Mpegaudio.

Fig. 9. SpecJVM benchmark similarity matrices. Matrix (a) is for Compress and input size 100, matrix (b) is Compress and input size 10, matrix (c) is cross-input similarity for Compress, and matrix (d) is cross-input similarity for Mpegaudio.

The matrix indicates that there may be potential for cross-input optimization. Two different alternating (non-contiguous) patterns are visible through out the entire execution of input 10. This is much like the pattern that occurs for input 100 alone. We also analyzed Mpegaudio in a similar way to evaluate whether the lack of phase behavior that occurs in both inputs is the same behavior across inputs. The data is shown in Fig. 9(d) and indicates that the same basic blocks are executing at the same frequency and duration for both input 10 (126 intervals) and input 100 (958 intervals) as indicated in the matrix.

4. Related work

This paper extends and elaborates on our initial investigation into phase visualization and analysis for Java described in [17]. We classify other related work into analysis, detection and prediction of phases in program execution and phase behavior in the context of dynamic, feedback-directed software systems.

4.1. Phase research

Runtime phase behavior of programs has been previously studied and successfully exploited primarily in the domain of architecture and operating systems [16]. The basis of our framework is to combine existing techniques that have proven successful in these domains within an adaptive JVM context.

Ref. [21] and [22] present 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. The programs analyzed include benchmarks from the SPEC95 and SPEC2000 benchmark suites. Basic block vectors consisting of basic block frequencies are used to characterize program behavior across multiple intervals of fixed duration. Basic block vectors for different intervals are then compared using Manhattan distance and finally classified into phases using k-means clustering. [23] presents an online version of such phase characterization along with a phase prediction scheme. This prior work describes additional applications to configurable hardware.

Dhodapkar and Smith [6] and Duesterwald et al. [8] stress the importance of exploiting phase behavior to tune configurable hardware components. Dhodapkar and Smith compare working set signatures across intervals using a similarity measure called relative working set distance to detect phase changes and identify repeating phases. Duesterwald et al. use hardware counters to study the time-varying behavior of programs and use it in the design of online predictors for two different micro-architectures. This exploits the periodicity in program behavior and allows a predictive approach to dynamic optimization as opposed to a reactive one. In other work [7], the authors compare three different metrics that characterize phase behavior. The metrics are basic block vectors, branch counters and instruction working sets.

The papers described above study phase behavior at the architectural level, and for C programs. Our aim is to enable collection and analysis of code-level phase data for Java programs.

Hind et al. [10] rigorously examine the fundamental problem of detecting shifts in program phase behavior. They focus on the dependence of two parameters that impact phase detection: granularity and similarity. They also demonstrate for the SpecJVM benchmark suite that phase detection depends on the choice of these parameter values. The authors model program behavior as a string of values; they define granularity as the atomic units of comparison that make up the strings and similarity as a function that computes how similar strings are. The authors consider a similarity function as consisting of a model (that produces a similarity value of 1 if there is perfect similarity and 0 if there is no similarity between two strings), a breakup (a function defines how substrings of the strings being compared have their similarity values computed), and a threshold (a constant value that indicates that two strings are similar if their similarity value is greater than it). The authors present similarity data (which they call similarity graphs) for a specific instance of their model.

Our system is a concrete example of the abstraction defined in this prior work. Our framework enables visualization of the range of similarity values across intervals in Java programs. In addition, the visualizations enable investigations by researchers beyond phase-shift detection; they allow users to identify, study, and reason about repeating phases, their length, and their internal code characteristics. Moreover, each of the parameters of our framework can be set to empirically evaluate the practical impact of the abstract parameters defined in this prior work. For our empirical analysis of the SpecJVM benchmarks, we employ a granularity (interval size) of 5 million instructions, our unit of comparison is dynamic basic block count, we compute similarity using Manhattan distance, and we use a similarity threshold of 0.8 in our discussion. Our similarity graphs are grayscale visualizations of the similarity values between intervals given this parameterization.

4.2. Dynamic optimization software systems

Papers on runtime optimizers in execution environments [2,12,1,24] and binary translators [5,9] have also discussed the benefit of considering phase behavior.

Arnold et al. [4] mention the use of phase-shift detection to trigger re-gathering of profiles that drive dynamic optimizations in the JikesRVM. Dynamo [5] uses phase-shift detection in its code cache policy. However, none of these systems currently use the various characteristics of phase behavior, namely periodicity and repetition, to drive dynamic optimizations.

In [13], Kistler and Franz use online phase-shift detection to trigger re-optimization in their continuously optimizing system for Oberon System 3. Change in program behavior is detected by observing whether the footprint of the profile has changed significantly in the last two consecutive time intervals. They use a similarity measure based on the geometric angle between the two profile vectors as against the vector difference that we use. Though the method of characterizing program behavior using profile vectors and comparing execution across two intervals is similar, their system does not study or exploit repeating patterns in time-varying behavior. Their primary aim is to detect change in program behavior.

5. Conclusions and future work

Dynamic program analysis and optimization are vital for enabling high performance in Java programs. Extant JVMs employ adaptive optimization techniques based on profile data gathered while the program is executing. The goal of adaptive optimization is to *learn* from a partial execution of the program what portions of the remaining execution can be optimized. Recent research has shown that it might be possible to exploit repeating patterns in the time-varying behavior of programs with feedback-directed optimizations. However, phase behavior in Java programs has not been thoroughly studied before.

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 unifying set of tools for data collection, processing, and analysis of dynamic phase behavior in Java programs. The framework is highly extensible and can be used by ourselves and others to investigate and empirically evaluate currently open questions about phase behavior analysis and its exploitation for Java programs.

As part of future work, we plan to use our phase analysis and visualization toolkit to investigate optimization and specialization opportunities that arise in Java programs. For example, we plan to use phases to identify opportunities to unload unneeded native code from the system to avoid unnecessary garbage collection [27], to perform dynamic voltage scaling to reduce power consumption [14,26,19], and to identify and exploit opportunities for code specialization. In addition, we plan to investigate techniques that use phase behavior to guide dynamic code reorganization, e.g., partial inlining, outlining, and cache-conscious code layout.

Acknowledgments

We thank the reviewers for their invaluable insights on and suggestions for the final version of this paper. This work was funded in part by NSF grants No. EHS-0209195 and No. ST-HEC-0444412, and an Intel/UCMicro external research grant.

References

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, J. Stichnoth, Fast, Effective code generation in a just-in-time Java compiler, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, May 1998, pp. 280–290.
[2] B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. Russell, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, J. Whaley, The Jalapeño Virtual Machine, IBM Systems Journal 39 (1) (2000) 211–221.
[3] M. Arnold, S. Fink, D. Grove, M. Hind, P. Sweeney, Adaptive optimization in the Jalapeño JVM, in: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, October 2000, pp. 47–65.
[4] M. Arnold, M. Hind, B. Ryder, Online feedback-directed optimization of Java, in: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, November 2002, pp. 111–129.
[5] V. Bala, E. Duesterwald, S. Banerjia, Dynamo: A transparent dynamic optimization system, ACM SIGPLAN Notices 35 (2000) 1–12.
[6] A. Dhodapkar, J. Smith, Managing multi-configuration hardware via dynamic working set analysis, in: 29th Annual International Symposium on Computer Architecture, ISCA, May 2002, pp. 233–244.
[7] A. Dhodapkar, J. Smith, Comparing program phase detection techniques, in: 36th Annual International Symposium on Microarchitecture, MICRO, December 2003, p. 217.
[8] E. Duesterwald, C. Cascaval, S. Dwarkadas, Characterizing and predicting program behavior and its variability, in: International Conference on Parallel Architecture and Compilation Techniques, PACT, September 2003, p. 220.
[9] K. Ebcioglu, E. Altman, M. Gschwind, S. Sathaye, Dynamic binary translation and optimization, IEEE Transactions on Computers 50 (2001) 529–548.
[10] M. Hind, V. Rajan, P. Sweeney, Phase shift detection: A problem classification, IBM Research Report 22887, August 2003.
[11] IBM Jikes Research Virtual Machine (RVM), http://www-124.ibm.com/developerworks/oss/jikesrvm.
[12] The Java HotSpot Virtual Machine, Technical White Paper. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.ps.
[13] T. Kistler, M. Franz, Continuous program optimization: A case study, ACM Transactions on Programming Languages and Systems (TOPLAS) (2003) 500–548.
[14] U. Kremer, J. Hicks, J. Rehg, A Compilation framework for power and energy management on mobile computers, in: 14th International Workshop on Parallel Computing, LCPC, August 2001, pp. 115–131.
[15] C. Krintz, Coupling on-line and off-line profile information to improve program performance, in: International Symposium on Code Generation and Optimization, CGO, March 2003, pp. 69–78.
[16] A. Madison, A. Bates, Characteristics of program localities, Communications of the ACM 19 (5) (1976) 285–294.

[17] P. Nagpurkar, C. Krintz, Visualization and analysis of phased behavior in Java Programs, in: ACM International Conference on the Principles and Practice of Programming in Java, PPPJ, June 2004, pp. 27–33.

[18] T. Nguyen, PANIC Laboratory at Rutgers University. http://www.panic-lab.rutgers.edu/.

[19] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. Irwin, J. Hu, H. Hsu, U. Kremer, Energy conscious compilation based on voltage scaling, in: ACM SIGPLAN Conference on Language, Compiler and Tool Support for Embedded Systems, LCTES, June 2002, pp. 2–11.

[20] X. Shen, Y. Zhong, C. Ding, Locality phase prediction, in: 11th International Conference on Architectural Support for Programming Languages, ASPLOS, October 2004, pp. 165–176.

[21] T. Sherwood, E. Perelman, B. Calder, Basic block distribution analysis to find periodic behavior and simulation points in applications, in: International Conference on Parallel Architectures and Compilation Techniques, PACT, September 2001, pp. 3–14.

[22] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, in: 10th International Conference on Architectural Support for Programming Languages, ASPLOS, October 2002, pp. 45–57.

[23] T. Sherwood, S. Sair, B. Calder, Phase tracking and prediction, in: 30th Annual International Symposium on Computer Architecture, ISCA, June 2003, pp. 336–349.

[24] M. Smith, Overcoming the challenges to feedback-directed optimization, in: ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo), January 2000, pp. 1–11.

[25] J. Whaley, Partial method compilation using dynamic profile information, in: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA, October 2001, pp. 166–179.

[26] F. Xie, M. Martonosi, S. Malik, Compile-time dynamic voltage scaling settings: opportunities and limits, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, June 2003, pp. 49–62.

[27] L. Zhang, C. Krintz, Profile-driven code unloading for resource-constrained JVMs, in: ACM International Conference on the Principles and Practice of Programming in Java, PPPJ, June 2004, pp. 83–90.