Phase-Based Adaptive Recompilation in a JVM

Dayong Gu
dgu1@cs.mcgill.ca

Clark Verbrugge
clump@cs.mcgill.ca

School of Computer Science, McGill University
Montréal, Québec, Canada

ABSTRACT

Modern JIT compilers often employ multi-level recompilation strategies as a means of ensuring the most used code is also the most highly optimized, balancing optimization costs and expected future performance. Accurate selection of code to compile and level of optimization to apply is thus important to performance. In this paper we investigate the effect of an improved recompilation strategy for a Java virtual machine. Our design makes use of a lightweight, low-level profiling mechanism to detect high-level, variable length phases in program execution. Phases are then used to guide adaptive recompilation choices, improving performance. We develop both an offline implementation based on trace data and a self-contained online version. Our offline study shows an average speedup of 8.7% and up to 21%, and our online system achieves an average speedup of 4.4%, up to 18%. We subject our results to extensive analysis and show that our design achieves good overall performance with high consistency despite the existence of many complex and interacting factors in such an environment.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers, Optimization, Run-time environments

General Terms

Design, Languages, Measurement, Performance

Keywords

Virtual machine, Java, adaptive optimization, runtime technique, hardware counters

1. INTRODUCTION

Many of today's Java Virtual Machines (JVMs) [28] employ *dynamic recompilation* techniques as a means of improving performance in Java programs. At runtime the dynamic Just-in-Time (JIT) compiler locates a "hot set" of important code regions and applies different optimizations, balancing the overhead costs of optimized (re)compilation with expected gains in runtime performance. Heuristically, the earlier the method is compiled to it's "optimal" optimization level the better. Naively assuming optimal means more optimizations, the potential for such improvements is illustrated schematically in Figure 1. In each of the 4 cases shown the x-axis is samples (normalized time), and the y-axis is optimization level; more time in a method at higher optimization levels heuristically means better performance, and so greater area under each curve indicates improved performance. The top left image shows a typical method history, compiled initially at a low level, and progressively recompiled to higher optimization levels. Better prediction of future behaviour allows a method to move more quickly between these steps (top right image), or to skip intermediate steps (bottom left image). In the bottom right graph a method is compiled to its highest optimization level immediately; this roughly represents an upper limit for the potential performance gains, at least assuming simple models of method execution and optimization impact.

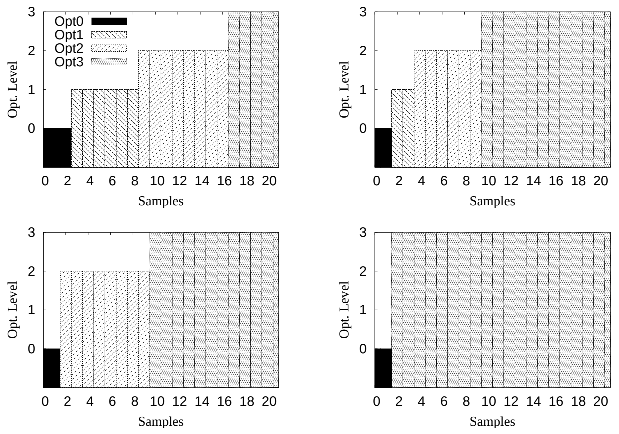

Figure 1: Sources of optimization due to improved recompilation decisions for a given method. Note that even in the "ideal" case of skipping all intermediate recompilation steps (bottom right) at least 1 sample is required to identify the hot method.

One of the key factors involved in improving or finding optimal recompilation choices for a given method is method *lifetime.* Method lifetime is an estimate of how much future execution will be spent in a given method based on current and past behaviour; techniques for estimating method lifetime are critical in making online recompilation decisions. A straightforward solution used in the Jikes RVM [1, 3, 2] adaptive recompilation component is to assume that the relative proportion of total execution time that will be spent in a given method is the same as its existing proportion: the ratio of future lifetime to past lifetime for every method is assumed to be 1.0. This is a generally effective heuristic, but as an extremely simple predictor of future method execution time it is not necessarily the best general choice for all programs or at all points in a program's execution.

Our work aims at investigating and improving the prediction of future method execution times in order to improve adaptive optimization decisions. To achieve better predictions we divide Java program execution into coarse phases; different phases imply different recompilation strategies, and by detecting or predicting phase changes we can appropriately alter recompilation behaviour. We perform an *offline* analysis of the practical "head space" available to such an optimization that depends on a *post mortem* analysis of program traces, allowing the method recompilation system to perform as in the bottom right graph of Figure 1. We also develop an *online* analysis that is more practical and dynamically gathers and analyzes phase information. To keep our online system lightweight, we base our phase analysis on hardware counter information available in most modern processors, recovering high-level phase data from low-level event data. Based on our Jikes RVM implementations we observe an average of 8.7% and up to 21% speed improvement in our benchmark suite using the offline approach, and an average of 4.4% and up to 18% speedup in our benchmarks using our online system, including all runtime overhead.

Although these results demonstrate significant potential, changes to the dynamic recompilation system introduce feedback in the sense that different compilation times and choices perturb future recompilation decisions. There are also many potential parameters of our design, and different kinds of benchmarks can respond quite differently to adaptive recompilation—programs with small, core method execution sets and long execution times can be well-optimized without an adaptive recompilation strategy, while programs with larger working sets and more variable behaviour should perform better with adaptive recompilation. We consider a number of confounding factors and include a detailed investigation of the source and extent of improvement in our benchmarks, including potential variability due to choice of recompilation algorithm. Our results show that our phase based optimization provides greater benefits in terms of performance, stability, and consistency than current designs or simpler optimizations.

Contributions of this work include:

- We demonstrate a lightweight system for obtaining high-level, variable length and coarse grained phase information for Java programs. Other phase analyses concentrate on finding fixed length and/or fine-grain periods of stability.
- We give the results of an offline study of the head space for optimization in the selection of hot-method recompilation points based on our phase information. In the case of repeated or allowed "warm up" executions our study represents an effective optimization by itself.
- We present a new dynamic, phase-based hot-method recompilation strategy. Our implementation incorporates online data gathering and phase analysis to dynamically and adaptively improve recompilation choices and thus overall program performance.
- We provide non-trivial experimental data, comparative results, and detailed analysis to show our design achieves significant and general improvement. Potential variation, identification of influences, and consideration of the precise source of improvements and degradations are important for optimizations to complex runtime services in modern virtual machines.

The remainder of this paper is organized as follows. In Section 2 we discuss related work on hot method set identification, profiling techniques, phase detection/prediction, and hardware counters. Our main data gathering system and phase prediction systems are described in Sections 3 and 4 respectively. Performance results and analytical measurements are reported in Section 5, and Section 6 provides detailed data analysis and discussion. Finally, we conclude and provide directions for future work in Section 7.

2. RELATED WORK

In a compiler based JVM (JIT), bytecode is compiled into native code immediately before it executes. However, the JIT strategy introduces compilation overhead before any code can execute. This can impose a heavy burden if complex optimization actions are employed during the course of compilation. JIT compilers thus typically attempt to identify a smaller hot set on which to concentrate optimization efforts. This kind of adaptive optimization allows sophisticated optimizations to be applied selectively, and has been widely explored in the community [23, 31, 3]. Most of this work focuses on methods as a basic compilation unit, but other choices are possible; For instance, Hansen's AF recompiled basic blocks and single-entry regions with loops selectively [19]. Whaley presents an approach to determining important intra-method code regions from dynamic profile data [40]. On the other hand, Chambers and Ungar apply optimizations across method boundaries via inlining [11].

Modern virtual execution environments often have a compiler with more than one optimization level. In general, code compiled at a higher optimization level provides faster speed as a trade-off for heavier compilation overhead. In a system with multiple optimization levels, only recompiling the most important (hot) code to a higher level is common sense, *i.e.*, making selective optimization. In a system such as in SELF-93 [21], all methods are first compiled at a non-optimizing level, with the optimizing compiler invoked only for frequently executed methods. SELF-93 uses method invocation counts to determine hot methods, the counts decaying over time. Detlef and Agesenuse use a fast JIT compiler and a slow "traditional" compiler adaptively [14]. They found that a combination of the fast JIT and judicious use of the slow JIT on the longest running methods showed the best results on their benchmark suite. Sun's HotSpot Server JVM [31] uses a technique similar to the one in SELF-93. The IBM Mixed-mode interpreter system also relies on invocation counts to determine recompilation decisions [37]. In addition to a counter-based selective optimization heuristics, Intel's ORP JVM [13] also uses a count-down scheme to identify hot methods.

All these counter-based policies rely on various heuristic tuning values. Recently, more theoretically involved policies have received more and more attention. Kistler *et al.* consider a sophisticated online decision for driving compilation in the Oberon virtual machine [24]. Each compiler phase estimates its own speedup based on a rich set of profile data. Arnold *et al.* use call stack sampling to support a model-driven optimization policy in Jikes RVM, relying on a cost-benefit model [3]. Krintz also provides a dynamic compilation system based on Jikes RVM [25]. Offline profiling results for the top hottest methods are annotated and works as a suggestion for a compilation task to the adaptive engine. Our offline mechanism follows a similar style, but stores all recompilation history from multiple runs and makes a summary trace from the traces of these multiple executions. More recently, Buytaert *et al.* present an HPM-Sampling mechanism which is more effective in finding optimization candidates for dynamic compilation than sample or counter based sampling mechanisms [9].

In all these efforts, recompilation overhead is reduced by avoiding compiling and optimizing rarely used code, based on either the assumption that "future = past," or by using simple counter-based schemes to determine relative execution frequency. Our work here augments these approaches by concentrating on the specific problem of providing additional predictive information to the adaptive engine of a JVM in order to improve optimization decisions, rather than providing the concrete adaptive optimization framework itself.

Program phase information can be used to locate stable or repetitive periods of execution at runtime, and has been incorporated into various adaptive optimizations and designs for dynamic system reconfiguration [5, 12, 15, 22, 34]. Nagpurkar *et al.* present a flexible scheme to reduce network-based profiling overhead based on repetitive phase information gathered from remote programs [30]. Their *phase tracker* is implemented using the SimpleScalar hardware simulator [8]. Data for phase analysis may in general be gathered through offline analysis of program traces, or through online techniques. Nagpurkar *et al.* present an online phase detection algorithm that detects stable, flat periods in program execution as useful phases [29], and further provides a set of accuracy scoring metrics to measure the stability and length of the detected phases. Phases based on various statistics are of course also possible, and many different data measurements have been considered for phase analysis work. Dhodapkar *et al.* make a comparison between several detection techniques based on *basic block vectors, instruction working sets* and *conditional branch counts* [16]. Phase data is also often employed for high level program understanding [36, 17].

Most phase analysis techniques are based on fixed-length intervals, aiming to detect stable periods of program execution [5, 35, 20]. For programs with complex control flow, such as Java and other object-oriented programs, at the levels of granularity useful for optimization there may be no actual flat and stable phases, even if there is obvious periodicity. For such situations the majority of techniques and associated quality metrics are not sufficient to capture or accurately present program phases. Basic problems with phase granularity are starting to be considered; Lau *et al.* point out the intrinsic problem of fixed interval designs being "out of synchronization" with the actual periodicity of the execution, and graphically show that it is necessary to study variable length intervals [26]. Here we use actual hardware data to detect coarse, variable length, recurrent phases in a program and use it to give useful advice to the adaptive engine of a JVM.

To actually gather hardware data we make use of the specialized hardware performance counters available in modern processors. Hardware counters can provide important micro-architectural performance information that is difficult or impossible to derive from software techniques alone. These data allows the program behaviour to be more directly understood from the viewpoint of the underlying hardware platform, and although low level, this information can be used for guiding higher level adaptive behaviour. Barnes *et al.* use hardware profiling to detect hot code regions and apply code optimizations efficiently [6]. Schneider and Gross present a runtime compiler framework using instruction level information provided by hardware counters to detect hot spots and bottlenecks [33]. Their work provides a platform to study the relation between dynamic compiler decisions and hardware specific properties. Kistler and Franz describe the *Oberon* system that performs continuous program optimization [24]. They describe the benefits of using hardware counters in addition to software based techniques as crucial components of their profiling system. Hardware information obtained from hardware counters can also be used to improve static compilers; Cavazos *et al.* use performance counters to as a means of determining good compiler optimization settings [10]. Other works based on hardware event information can be found in [32, 38, 18]. Many software applications and libraries are available to access these counters, including VTune, PMAPI, PCL and PAPI [7]. In this work, we use the PAPI library.

3. BASIC SYSTEM

Our system design is based on an extension to the current recompilation system in Jikes RVM. Figure 2 shows the overall structure and components of our base system, and how it integrates with Jikes RVM. Raw hardware event data is obtained through the *hardware performance monitor* (HPM), a pre-existing component in Jikes RVM. The pattern analysis model detects "patterns" in the hardware data. Through comparison with previous patterns stored in the *pattern database*, the pattern analysis model determines the current phase of an executing program. Phase information is then used to give advice on the program phase to the adaptive recompilation engine, and also to control the behaviour of the runtime measurement component. By taking phase advice into account, the adaptive recompilation engine is able to make better adaptive recompilation decisions, as we show in our experimental data. Below we provide more detailed descriptions of the core components of our implementation design and environment.

3.1 Hardware Performance Data

Hardware performance data is acquired by reading hardware-specific performance counters. Fundamentally, the hardware counters in modern processors are a set of registers. Values in these registers can represent a variety of hardware events, such as machine cycles, instruction and data L1/L2 cache misses, TLB misses, and branch predictor hits/misses. Counter data reflects the performance of the underlying hardware directly and collecting it imposes little overhead.

Critically, although hardware counter data is low level it

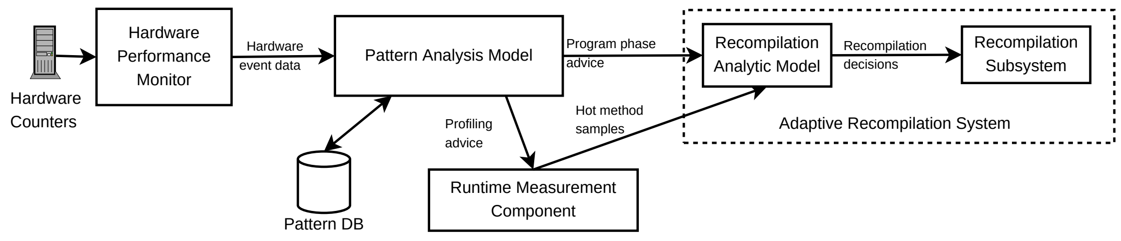

Figure 2: The cooperation among hardware performance monitor, pattern analysis model and adaptive optimization components.

can be related to high level aspects of program behaviour. Lau *et al.* show there is a strong relation between hardware performance and code signatures or instruction working sets [27].

Our implementation mainly samples the "L1 instruction cache miss" event, an event known to correlate well with method execution behaviour. Although this is only one of many possible choices, Gu and Verbrugge have previously shown the importance of "L1 instruction cache miss" on JVM performance [18]. Of course the L1 I-cache is a global resource, subject to potential pollution by other processes. We minimize external "noise" by stopping inessential processes for testing. On the other hand, any program optimization should be effective in a real life system; necessary system processes produce unavoidable impacts, and as we will show our system is quite robust.

The HPM component of Jikes RVM is used to gather our raw hardware event data. To ensure a lightweight design our system samples events only at each process context switch point; in a typical benchmark this produces several thousand sample points per benchmark run.

3.2 Hardware Pattern Construction

To detect coarse grained and variable length phases the input hardware event data is inspected for patterns. Our *pattern analysis model* discovers simple patterns by observing how event density changes over time, and looking for distinct sequences of change. There are many parameters possible in such a design, and here we provide an overview of an approach optimized for accuracy, generality, and simplicity.

Our technique operates by summarizing low-level behaviour with short bit-vectors that encode the overall pattern of variation. We make use of a "second order" approach that considers variation in hardware event counts rather than absolute counts themselves as the basic unit to focus the technique on detecting changes in behaviour, heuristically important for identifying phases. The actual algorithm for translating hardware event data to bit-vector patterns involves first coarsening the (variation in) data into discrete *levels*, and then building a corresponding bit-vector *shape* representation. The final combined shape and level structure is a pattern, which can be stored and tested for repetition. In our experiments we discretized variation in I-cache miss density into 4 levels, higher level indicated greater importance. Shape construction is more complex and proceeds as follows; a helpful full example of pattern construction is shown in Figure 3.

Shapes are simply sequences of bits observing the direction of change, positive or negative, between consecutive event data. Complexity in shape construction is mainly driven by determining when a shape begins or ends; we use the discrete level of the data that initiated the shape, with data level functioning as a heuristic indicator of pattern strength.

Each shape construction is represented by a pair $(p, \overline{v})$, where p is the level associated with the beginning of the shape, and $\overline{v}$ is a bit-vector encoding the sign (positive, negative) of successive changes in event density. Given a new event e with level p_e, if there is no shape under construction a new construction begins with an empty vector: $(p_e, [\,])$. Otherwise, there must be a shape under construction $(q_f, \overline{w})$. If $q_f = p_e$, or we have seen $q_f > p_e$ less than n times in a row, then the shape is assumed to continue; a bit indicating whether the current variation difference is positive or not is added to the end of $\overline{w}$.

The following conditions terminate a shape construction.

1. If we find $q_f < p_e$ we consider the current shape building complete and begin construction of $(p_e, [\,])$. Increases in level are indicative of a significant change in program behavior.

2. If we find $q_f > p_e$ to occur n times in a row the current shape is considered to have "died out," and the pattern is also completed. In our experiments we have found $n = 2$ is sufficient for good performance.

3. If in $(q_f, \overline{v})$ we find $|\overline{v}|$ has reached a predefined maximum length we also report the current construction as complete. In our experiments we use a maximum of 10 bits as a tradeoff of storage cost and expressiveness in patterns.

Once terminated a shape becomes a full pattern available for use in pattern analysis. Of course the same or a similar pattern construction strategy can be applied to any hardware event counter, and in general any scalar event data. As mentioned, in our actual system we make use of the instruction cache miss density as a hardware event, found useful by others and confirmed effective in our own experiments. Section 6 discusses this issue further, but a deeper investigation of different events and event combinations is left for future work.

3.3 Adaptive Recompilation in Jikes RVM

The adaptive recompilation system [3] of Jikes RVM involves three main subsystems. A *runtime measurement component* is responsible for gathering method samples. An *analytic model* reads this data and makes the decision on whether to recompile a method and the appropriate optimization level. The recompilation plan is fed to the *recompilation subsystem* which carries out the actual recompilation.

The crucial point is the decision-making strategy of the analytic model. This selects between different optimization

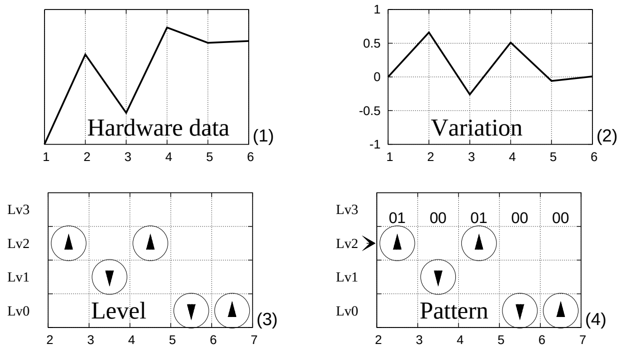

Figure 3: Pattern construction example. (1) Acquire the raw hardware data. (2) Calculate the variation between consecutive points. (3) Coarsen the variation into different levels; the triangles inside each circle show the direction (negative/positive) of variation. (4) The final pattern construction results; the arrow on the y-axis indicates that we obtain a level 2 pattern; the number above each circle shows the 2-bit code for each variation. The four trailing zeros are omitted (the pattern has died out), and the final pattern code is 010001.

levels, based on an estimate of the potential benefit of each level. The specific optimizations applied in each level can be found in [3]. For each optimization level i ($0 \leq i \leq N$), Jikes RVM gives an estimate of the execution speed Sp_i of a method m. The value of N can be different for different platforms; in our system, $N = 3$. A recompilation decision is then made based on the following computations [3]:

- T_p: The time of the program already spent in m. It is computed as $T_p = \mathit{SampleNumber} * \mathit{TPS}$ TPS stands for "time per sample," a constant value in Jikes RVM.
- T_i: The expected time of m at level i, if it is not recompiled. In the original implementation, the system assumes: $T_i = T_p$
- C_j: The cost of recompiling method m at level j, for $i \leq j \leq N$.
- T_j: The expected time the program will spend in m in the future, if it is recompiled at level j: $T_j = T_i * \frac{Sp_i}{Sp_j}$

The analytic model chooses the level j that minimizes the value of $C_j + T_j$, the compile time overhead and expected future time in m. If $C_j + T_j < T_i$, then m will be recompiled to level j.

4. PHASE ANALYSIS

Improvements to the prediction model used by the adaptive recompilation engine have the potential to improve performance, executing highly optimized code more often and decreasing the overhead of successive recompilations. We investigate the improvement from two perspectives. The first is an offline technique based on trace data; this mainly serves to give a sense of the maximal benefit that could be reached given optimal information. The second is a purely online implementation, that uses our low-level profiling and dynamic phase systems to improve predictions.

4.1 Offline Trace-Driven Mechanism

Recompiling a hot method to an ideal optimization level at the earliest point in program execution will in general maximize the benefit of executing optimized code, as well as eliminate further potential compilation overhead from the method. For a recompilation mechanism based on runtime sampling data, knowledge of the final optimization level of a method at the time when the first sample of it is taken represents ideal results with minimal profiling overhead. Optimality is bounded by the accuracy of the estimation, including heuristic choices that balance optimization costs and benefits. Here we implement an offline trace-driven optimization technique to discover the approximate improvement head space if optimal choices are made in the sense of maximizing the heuristic benefit.

Implementation of the offline mechanism is straightforward. A set of traces from training runs is gathered, analyzed, averaged, and used in a subsequent replays of the program to select an appropriate optimization level for each recompiled method. Note that this is not necessarily the highest optimization level possible—as part of the cost/benefit trade-off, not all methods will have been, or necessarily should be recompiled to the highest level of optimization. Use of multiple runs accommodates minor variations in performance; sources of noise in recompilation data are discussed more fully in Section 6.

Implementation details include that:

- First, training data is gathered; a Java program is executed N times to produce trace files $T_i (1 \leq i \leq N)$.
- Each trace T_i is composed of a set of pairs $< M, L_i >$. M is a particular method, and L_i is the last and highest optimization level of M in T_i.
- A summary trace T_s is constructed, composed of pairs $< M, L_s >$, where $L_s = Max(L_1, L_2, ..., L_N)$ for a given M.
- In the tested runs, T_s is loaded at the beginning of execution. Each time a method sample M is taken, if we can find a record $< M, L_s >$ for it in T_s, we recompile M to level L_s directly, and mark the recompilation as a final decision. No further compilation will be applied to M.
- It is possible that speed gains due to better adaptive recompilation allows a method not recompiled in any training run to be added to the hot set in an actual run. If we cannot find a record for M in T_s, M is treated per Jikes RVM's original recompilation strategy. Note that in our experiments such cases are rare and involved infrequently executed methods; the impact of this divergence in hot set identification is reasonably expected to be small.

Performance results from the offline strategy are given in Section 5.1. On some benchmarks the benefit obtained is quite significant, confirming both the potential available to a more flexible online optimization, and the value of our offline design as an optimization unto itself.

4.2 Online Mechanism

The success of an online recompilation system depends on the accuracy of method *lifetimes*, or the future time spent in a method, as well as other recompilation cost and benefit estimates. Underestimating future method execution time results in missed optimization opportunities, while overestimating runs the risk of being overly aggressive in compilation, wasting time on unnecessary recompilations and/or high optimization levels. This is particularly true early and late in program executions, where code execution variability is high and the expectation of continued behaviour is lower. This can also occur when programs make major phase changes, shifting into markedly different modes of execution. The kernel of our online mechanism is thus a system that detects coarse grained and variable length program phases and uses this information to control the relative aggressiveness of the recompilation subsystem in Jikes RVM. The resulting improved recompilation choices improve overall program performance.

The existence of basic startup, core execution, and shutdown phases are well known. Our phase identification is based on identifying *age*, but further allows programs to *rejuvenate*, as a means of allowing for the identification of multiple major execution phases. These phases imply distinct patterns of control for recompilation, and are classified as follows:

- Newborn: At startup a Java program tends to spend time on a set of methods that perform initialization actions, and these are often not executed after basic setup is complete. When considering whether past behaviour is a good predictor of future behaviour we can heuristically expect that the future execution time of a given method will be less than the past: $Future < Past$.
- Young: After a period of time, the program comes into the main application or kernel code and will spend a comparatively long time on the same set of methods. Methods executed at this stage are likely to be executed even more in the future: $Future > Past$.
- Mature: After the program works within its kernel code for a while, we consider the program to be *mature*. In this case, we assume the runtime profiling subsystem has gathered enough samples to support the recompilation engine in determining suitable optimization levels. Here the original estimate that future and past performance will be similar is most valid: $Future \approx Past$.
- Rejuvenated: Experience with coarse grained phase analysis of Java programs shows some programs will have distinct, kernel-based phases, and at runtime will have more than one hot method set. When a program enters a new hot set it thus transitions to the young phase again. Once so *rejuvenated* as such, however, we have a slightly more cautious estimate as to the future behaviour of the new hot set: $Future > Past$.

Phase	Hardware Pattern Behaviour	Recompilation
Newborn	No recurrence	Less aggressive
Young	Recurrences	More aggressive
Mature	Less new patterns More old patterns	Moderately aggressive
Rejuvenated	More new patterns	More aggressive

Table 1: Program phase, hardware patterns, and recompilation aggressiveness.

The second column of Table 1 describes how program phases are heuristically determined from the underlying hardware event data. Changes in how lower-level patterns are identified in the data suggest corresponding changes in the program code, and thus phase or age. At program startup a wide variety of "execute-once" startup code is executed, and few recurring low-level patterns are found. A young program will start to show significant recurrences of new patterns as it begins to execute its kernel code. The mature phase is detected by noticing the balance tipping from discovery of new patterns to recurrence of old patterns, and the rejuvenated phase by a subsequent loss of old patterns and introduction of new ones.

Understanding program phase allows for heuristic control of the relative aggressiveness of the recompilation engine. In cases where the future performance is not equal to the past the expected execution time should be appropriately scaled. The third column in Table 1 gives a summary of how age affects the behaviour of the recompilation engine. A newborn program is less likely to repeat its behaviour, and recompilation should be more conservative. A young program enters into its kernel; the new code is likely to be executed much more than it has been in the past, and recompilation becomes aggressive. As the execution enters a mature phase aggressiveness is decreased; in such a relatively stable environment the recompilation engine is expected to have sufficient past data for making good decisions. A program that enters a new significant kernel of execution requires again ramping up the aggressiveness of recompilation.

The *aggressiveness* of the adaptive recompilation engine is controlled by using a scaling parameter in the estimation of future execution times. We introduce a variable *futureEstimator* and change the definition of T_i from $T_i = T_p$ to: $T_i = T_p * futureEstimator$. This is integrated with phase analysis as follows. Each hardware pattern PAT has a field *occNum* which remembers the number of occurrences. If the adaptive recompilation model finds a recurring PAT, such that, $PAT.occNum$ is more than one, the estimated "age" of a program ($Prog.age$) is increased. When $Prog.age$ is larger than a threshold *youngThresh*, the program has left the newborn phase and become young. From then on, each time there is a *fresh* pattern PAT such that the occurrence number is less than a threshold *matureThresh*, the value of *futureEstimator* is increased; otherwise its value is decreased. A larger value of *futureEstimator* drives the adaptive recompilation model to make more aggressive recompilation decisions, assuming methods will run for longer than currently estimated. Fixed upper and lower bounds protect the *futureEstimator* value from diverging in cases of extended bursts of fresh or mature patterns. Based on earlier experiments we limit *futureEstimator* to the range $[0.9, 5.0]$.

5. EXPERIMENTAL RESULTS

Experimentally we evaluated the performance of both our offline and online solutions. Our implementations are built upon Jikes RVM 2.3.6 with an adaptive compiler, and runs on an Athlon 1.4GHz workstation with 1GB memory, under Debian Linux with a 2.6.9 kernel.

Benchmarks used in this work include the industry standard SPECjvm98 suite benchmark antlr, bloat, fop, pmd and xalan from the DaCapo suite and two larger examples, soot [39] and PseudoJbb (pJbb). Soot is a Java optimization framework which takes Java class files as input and applies optimizations to the bytecode. In our experiments, we run soot on the class file of benchmark javac in

SPECjvm98 with the `-app -O` options, which performs all available optimizations on application classes. PseudoJbb is a variant of SPECjbb2000which executes a fixed number of transactions in multiple warehouses. In these experiments it executes from one to eight warehouses with 100 000 transactions in each warehouse. For SPECjvm98 we use the S100 input size. For DaCapo benchmarks, we use the large input size.

For performance evaluation we measured our benchmarks quantitatively using a baseline (original), and using our offline and online strategies. Overall execution time for the online approach includes all overhead for phase analysis and low-level profiling. In the case of the offline approach the overall execution time includes the overhead of processing the recompilation trace. Full results for our benchmarks in absolute and relative terms are shown in Table 2.

5.1 Offline Mechanism

The results of our offline mechanism in absolute terms as well as relative improvement over the original version are given in the third and fourth columns of Table 2. The offline version achieves significant improvements on many benchmarks, and is positive for nearly all. On jess, it improves execution time by 21.3%. On jack, javac, mpegaudio and bloat the improvements are also quite large. On average, the offline version saves 8.7% of the execution time. The effect, however, is not uniform, and for some benchmarks, such as compress, db and fop, there is only marginal change. Nevertheless, in an overall sense the results of our offline mechanism confirm the potential for a comparatively large benefit by improving the recompilation strategy. We will discuss these benchmark-specific behaviours in more detail in Section 6.

5.2 Online Mechanism

The execution time results for the online mechanism are shown in the fifth and sixth columns of Table 2. For benchmarks where the offline version shows a large improvement, the online version also generally performs well. We obtain up to nearly 18% improvement for jess, quite close to the 21% improvement found for jess offline. On average the online version achieves a 4.4% improvement, about 51% of the possible performance improvement demonstrated in the offline version. For the 6 benchmarks that responded most positively ($> 10\%$) to the offline version, the improvement online is on average 9.0%, or 65% of the offline result.

Measuring the impact of actual phase identification and predictions choices beyond raw performance is more difficult. Experimentally, our thresholds and other heuristic parameters have been optimized within reasonable ranges on a subset of benchmarks (SPECjvm98, soot and PsuedoJbb); the fact that DaCapo benchmarks respond similarly is encouraging. We further compare our design with simple approaches to changing recompilation aggressiveness in Section 6.2.

5.3 Variance and Overhead

Figure 4 shows 99% confidence intervals for our original, offline, and online data measurements. Our evaluation is experimentally quite stable and deterministic, with confidence ranges for the three variations generally showing good separation. Note that the intervals for jack are among the largest and have clear overlap; the $\approx 1\%$ performance gain for jack online as opposed to offline could be attributed to data variance and/or the intrinsic imprecision of simple optimization benefit/cost estimates. We discuss accuracy and noise concerns in depth in the following section.

Overhead in profiling systems is always a major design concern. In our case we make use of hardware counters that are sampled at every process context switch; at a few tens of machine cycles per read and only on the order of thousands of context switches over a program's lifetime this technique is extremely cheap. Pattern construction and phase analysis provide the bulk of our actual overhead, and to measure total overhead costs we compared the original, baseline Jikes RVM with an implementation of our online technique that computes phases as normal but does not actually change the adaptive recompilation settings (*futureEstimator*). The *overhead* column of Table 2 shows the computed relative overhead. Overhead comes from sources such as hardware monitoring, pattern construction, phase prediction, and building control events for the recompilation component. On average there is a 1.43% slowdown across these benchmarks due to our data gathering and phase analysis system. There is always room for improvement, but this relatively small cost is in most cases greatly exceeded by the benefit, and demonstrates the practical low overhead of our technique; again, speedup and other experimental data includes all overhead.

6. DISCUSSION

Initial recompilation choices necessarily affect later recompilation choices, and there are many potential parameters and choices in our, or any, recompilation design. A good understanding of potential variation and relative performance gain is therefore important to making good, general selections of recompilation strategies. Here we discuss various factors that can influence our performance, and present data further validating the general stability and effectiveness of our design. We first consider different benchmark characteristics that are important in our approach. This is followed by a comparison of our design with other simple optimizations to the recompilation system, again showing the practicality of our work and the generality of our good results.

6.1 Benchmark Characteristics

Benchmarks in our study demonstrate a wide range of responses to our optimization. Several benchmark-specific factors can be seen to influence whether and where performance will be realized using our techniques. Benchmark length, the stability of the hot set, as well as more general sensitivity of the program to our profiling and optimization systems can all affect the relative success.

In our benchmark suite, five benchmarks, bloat, pmd, xalan, soot and PseudoJbb execute for an order of magnitude or so longer than the other shorter benchmarks. Longer benchmarks recompile many more methods, providing more opportunity for potential improvement in applying optimizations to the recompilation strategy. On the other hand, longer running programs have more data to work with: there simply are more sample points, and any reduction in speed due to less optimal recompilation choices can be amortized over a longer period of execution. From our experimental results, we find that a comparatively moderate improvement is produced by both offline (on average 9.8%) and online (on average 4.9%) strategies.

Benchmark	Original	Offline		Online			Benchmark Prop.	
	Time (s)	Time (s)	Impr. (%)	Time (s)	Impr. (%)	Overhead (%)	Patterns	Opt.Methods
compress	15.75	15.55	1.3	15.73	0.1	2.02	157.9	17.6
db	37.97	37.22	2.0	37.72	0.6	1.39	450.5	25.3
jack	22.59	20.08	11.2	19.78	12.5	1.71	343.5	90.0
javac	11.78	10.72	9.4	11.10	5.7	1.13	193.9	36.9
jess	18.11	14.25	21.3	14.87	17.9	0.49	204.5	50.0
mpegaudio	20.24	17.81	12.1	19.79	2.3	1.76	103.6	58.9
mtrt	15.14	14.29	6.4	15.42	-1.8	0.82	58.8	36.4
raytrace	14.35	13.30	7.3	14.21	0.8	1.30	63.9	35.3
antlr	47.29	41.99	11.2	44.95	4.9	2.12	236.4	134.5
bloat	262.88	219.03	16.9	230.47	12.5	1.65	1891.6	232.5
fop	22.37	22.51	-0.7	22.80	-1.9	1.69	132.6	57.8
pmd	290.85	258.15	11.2	281.34	3.3	1.70	893.8	218.1
xalan	804.56	750.49	6.7	783.63	2.5	1.07	3432.7	359.4
soot	303.12	278.45	8.1	291.28	3.9	1.85	2542.3	408.2
PseudoJbb	753.95	705.90	6.4	735.62	2.5	0.77	7832.8	331.8
Average	-	-	8.7	-	4.4	1.43	-	-

Table 2: Execution results, number of patterns created in the online version, and number of methods optimized. Values are the arithmetic average of the middle 11 out of 15 runs. The *Impr.* (Improvement) columns show the percentage change in performance over the original.

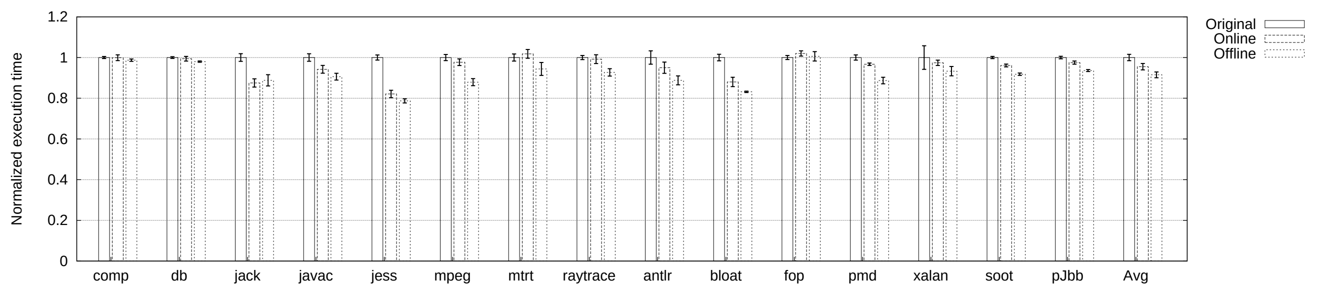

Figure 4: Normalized execution time with 99% confidence interval errorbars for each of our three test scenarios: original, online and offline.

The performance of shorter benchmarks, however, varies much more, mainly due to differences in of their method sets and invocation orders. Our offline mechanism achieves an on average 13.1% improvement on the five sensitive benchmarks (JACK, JAVAC, JESS, MPEGAUDIO, and ANTLR) and only a 3.26% on the five worst benchmarks (COMPRESS, DB, JAVAC, JESS, MTRT, RAYTRACE, and FOP). Online the average improvement for these two groups are 8.7% and −0.4%.

An unfortunate side effect of our optimization for detecting rejuvenation, or variations in the hot set is a certain overzealousness of optimization toward the end of execution. Termination tends to look like any other phase change or rejuvenation with our current pattern analysis and data, and optimized recompilation may be overused, recompiling and optimizing methods that will only be used in the final fraction of program execution. Solutions based on incorporating extra, high level information such as knowledge of termination-specific methods may be possible. In practice, these sub-optimal online decisions at termination time do not have an overly large impact; termination time does not dominate execution for our benchmarks and so we leave reducing this "tail" problem to future work.

It is interesting that low level events can expose high level behaviour, even for complex, object-oriented programs with non-trivial control flow. We have successfully used the I-cache miss rate as a base event, but other choices and event combinations are of course possible and desirable. RAYTRACE, MPEGAUDIO and FOP, for instance, have a relatively small instruction working set. Thus we observe only slight changes in I-cache performance, and as can be seen from the 2nd-last column in Table 2 our pattern creator finds significantly fewer patterns in these cases. This provides less information to the recompilation engine, and thus recompilation choices are not much better than in the original version: RAYTRACE and MPEGAUDIO show very marginally positive improvements, while FOP shows a 1.9% reduction. The fact that performance even in this situation is close to the original and not significantly degraded is evidence of the low overhead of our implementation design in general, and sample-based hardware monitoring specifically.

Other benchmarks have instruction working-sets large enough to produce significant misses as different code paths are exercised, allowing our online solution to identify patterns easily. The performance difference resulting from the improved information is evident in benchmarks such as JACK, JESS, and JAVAC. JESS' impressive performance may also be partially due to its extremely low overhead. Some benchmarks, however, exhibit cache performance changes, but the actual hot method set remains quite small. If a small set of methods are called frequently, as for COMPRESS and DB, the original adaptive recompilation engine has a chance to gather enough samples to recompile the important methods relatively quickly. In these cases, the potential improvement available by reducing the delay of recompilation is small. The marginal benefit achieved by our offline solution can be mainly attributed to reductions in optimization overhead due to skipping redundant intermediate recompilations for some methods.

Programs can also exhibit *bias* with respect to different hardware events. Some programs like JESS and JACK are highly "instruction cache sensitive", meaning that from a processor-level point of view the instruction cache performance has a large impact on the execution time of the program [18]. On the other hand, DB and especially COMPRESS are highly data cache biased. There is obviously limited room to improve performance from the code side if data usage has a dominating impact. In these cases even the offline version only obtains a small improvement. We expect that programs with large memory requirements and hence garbage collection overhead, heavy I/O, and so forth will also respond less well to our design, as in general programs that are dominated by other costs than code execution speed will receive reduced benefits from adaptive code optimization techniques.

The above discussion suggests that monitoring different or multiple hardware events may be a route to further optimization. We have explored a few hybrid forms of pattern-building based on combinations of I-cache miss rate, D-cache miss rate, branch instruction counts, and brand prediction miss rates. So far, these designs have not shown useful improvement above that of one based on a simple I-cache miss rate; further exploring this space is, however, potentially fruitful future work.

6.2 Stability and Comparisons

Understanding which benchmarks can work well is important, but differentiating them online may be non-trivial, and a good recompilation system should perform reasonably well over a range of benchmarks. For our adaptive system to be useful it is also important to know that the adaptivity is effective. Both our online and offline strategies generally increase the aggressiveness of recompilation choices, and we must consider that similar effects could be achieved by simply making the the Jikes RVM estimator more aggressive without adaptation.

Testing the effects of trivial, constant increases in recompiler aggressiveness provides a baseline that shows both the variability of performance of different recompilation strategies and in comparison with our online approach, the actual impact of adapting to program phases. We evaluate several versions of Jikes RVM with no hardware monitoring or phase analysis, but incorporating our scaled time estimate formula, $T_i = T_p * \mathit{futureEstimator}$, with *futureEstimator* set to different fixed, constant factors to increase recompiler aggressiveness. Table 3 shows the normalized overall execution time for our benchmarks when the future time estimate of methods is increased by values between 1.5× and 3.0×; the last column shows the range of average increase in aggressiveness used by our online system for our benchmarks.

The data in Table 3 shows that there is certainly no one fixed setting that is optimal for all benchmarks; benchmarks respond differently, and simply increasing aggressiveness overall is not a generally effective strategy. Some benchmarks have a large variance in performance as the *futureEstimator* parameter changes, and some are relatively unaffected. For all benchmarks except MPEGAUDIO and FOP, our online version is optimal or within variance of optimal. In comparison with simple approaches, our online design provides stable and good results overall, significantly more so than the base version or any of the constant aggressiveness settings. Our approach is expected to bring benefit to all VMs with more than one optimization level, by pushing the optimization point(s) earlier.

7. CONCLUSIONS AND FUTURE WORK

For many programs, sub-optimal choices in recompilation can result in reduced performance. We have shown how improvements to recompilation strategy can result in better performance, and provided a design using coarse grained, variable length phase prediction to adaptively improve recompilation choices. Using offline trace data for prediction provides an experimental high performance watermark for such a technique, and functions as a useful optimization when program executions are repeated exactly. Our fully online implementation makes choices based on dynamically acquired data, and exhibits both low overhead and good overall performance.

Multiple factors influence performance in a recompilation system, and to show meaningful improvement a close evaluation of performance under different scenarios and with different levels of detail is important. We have explored our optimization in terms of execution time from a variety of perspectives, and provided comparative evaluations of our designs where possible. Detailed examination of benchmark behaviour reveals that benchmarks respond in different ways to the relative aggression of a recompilation engine, further complicating the optimization space. We consider a wide variety of benchmark-specific factors, including high level considerations such as overall runtime and low level influences such as the density of hardware event data. Under these highly variable and "noisy" conditions our adaptive online system achieves a significantly improved performance.

It is important to note that our design and choice of monitoring I-cache misses applies to and reflects the performance of the entire JVM, and not any specific component. Measuring the impact on specific subsystems is not practical due to the overhead introduced and potential further perturbation of results. Thus while a complete breakdown of impact on varying JVM components or specific optimizing actions such as an individual recompilations would be ideal, our experimental results reflect the real difficulty in separating the impact on different JVM components or subsystems.

There exist a large number of possible extensions to this work. The success of our approach, like most adaptive online systems, depends on the extent of variability in runtime execution data. We have expended a great deal of effort to understand and experimentally validate potentially critical factors, ensuring our approach is a generally robust optimization. Further understanding and detection of benchmark characteristics may improve our design, and could also be used to help select benchmark-specific responses by the adaptive optimization system. *Profile repositories,* aggre-

Benchmark	**FutureEstimator Setting**					**Online Aggressiveness**
	1.5×	**2.0×**	**2.5×**	**3.0×**	**Online**	
compress	0.997	0.970	1.018	1.019	0.998	3.05
db	0.991	1.008	1.022	1.025	0.993	1.98
jack	0.987	1.041	1.063	1.080	0.876	2.16
javac	0.970	0.955	0.975	0.991	0.942	2.40
jess	0.924	0.879	0.856	0.852	0.821	2.34
mpegaudio	0.960	0.924	0.925	0.948	0.978	2.44
mtrt	1.017	1.039	1.127	1.151	1.018	2.22
raytrace	0.983	1.010	1.057	1.053	0.990	1.99
antlr	0.954	1.019	1.030	1.075	0.951	1.86
bloat	0.915	0.972	0.899	0.902	0.877	1.87
fop	0.997	1.084	1.068	1.065	1.019	1.81
pmd	0.985	1.008	1.006	1.013	0.970	1.71
xalan	0.990	0.981	0.985	1.001	0.974	1.91
soot	0.966	0.950	0.945	0.969	0.961	1.35
PseudoJbb	0.991	0.978	0.975	1.025	0.976	1.09

Table 3: Fixed setting of *futureEstimator* versus the online version. Values are normalized execution times (smaller is better), except for the last column. The "online aggressiveness" column shows the average *futureEstimator* value used in the online version, weighted proportionally over program execution.

gating profile data from multiple executions may be a useful way of moving online performance closer to that of offline performance [4]. Mixing profile data from multiple runs or using offline/online hybrid data might also help with the "tail problem" of predicting the termination phase of a program.

Different and finer grain phase categorizations are possible of course. We intentionally exploit coarse grained phase information to allow complex optimizations time to act and improve performance. Startup phases are well-known, but the use of high level and variable length phase information, when cheaply gathered, is also obviously of value. Detecting major phase changes may be useful for scheduling garbage collection, heap data reorganization or any other design for larger scale adaptive execution. Additional or different hardware event data may be useful for more "data-centric" applications, and part of our current investigations include the use of multiple and hybrid hardware event sources. A potentially very fruitful direction for future work is in directly associating software level units with the major hardware performance variations. There are many possible approaches, but among other benefits this would enable better identification of hot methods for recompilation.

Acknowledgments

This research has been supported by Le Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

8. REFERENCES

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *OOPSLA '99*, pages 314–324, Oct. 1999.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, Apr. 2005.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[4] M. Arnold, A. Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *OOPSLA '05*, pages 297–311, 2005.

[5] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general purpose architectures. In *MICRO 33rd*, pages 245–257, Dec. 2000.

[6] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. mei W. Hwu. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *MICRO 35th*, pages 233–244, 2002.

[7] S. Brown, J. Dongarra, N. Garner, K. London, and P. Mucci. PAPI. `http://icl.cs.utk.edu/papi`.

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

[9] D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. D. Bosschere. Using HPM-Sampling to drive dynamic compilation. In *OOPLSA '07*, Oct. 2007.

[10] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO '07*, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[11] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *OOPSLA '91*, pages 1–15, 1991.

[12] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02*, pages 199–209, New York, NY, USA, 2002. ACM Press.

[13] M. Cierniak, M. Eng, N. Glew, B. Lewis, and

J. Stichnoth. The open runtime platform: a flexible high-performance managed runtime environment: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):617–637, 2005.

[14] D. Detlefs and O. Agesen. The case for multiple compilers. In *OOPSLA'99 Workshop on Peformance Portability, and Simplicity in Virtual Machine Design*, pages 180–194, 1999.

[15] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02*, pages 233–244, 2002.

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

[17] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in Java workloads. In *OOPSLA '04*, pages 270–287, Oct. 2004.

[18] D. Gu, C. Verbrugge, and E. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06*, pages 111–121, June 2006.

[19] G. J. Hansen. *Adaptive Systems for the Dynamic Run-time Optimization of Programs*. PhD thesis, Carnegie-Mellon University, 1974.

[20] M. J. Hind, V. T. Rajan, and P. F. Sweeney. Phase shift detection: A problem classification. Technical Report IBM Research Report RC-22887, IBM T. J. Watson, August 2003.

[21] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[22] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *ISCA '03*, pages 157–168, New York, NY, USA, 2003. ACM Press.

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

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

[25] C. Krintz. Coupling on-line and off-line profile information to improve program performance. In *CGO '03*, pages 69–78, 2003.

[26] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals to find hierarchical phase behavior. In *ISPASS'05*, March 2005.

[27] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS '05*, page 220, March 2005.

[28] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[29] P. Nagpurkar, M. Hind, C. Krintz, P. Sweeney, and V. Rajan. Online phase detection algorithms. In *CGO '06*, Washington, DC, USA, March 2006. IEEE Computer Society.

[30] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *CGO '05*, pages 191–202, 2005.

[31] M. Paleczny, C. A. Vick, and C. Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[32] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS-XI*, pages 189–198, Oct. 2004.

[33] F. Schneider and T. R. Gross. Using platform-specific performance counters for dynamic compilation. In *LCPC'05*, October 2005.

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

[35] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01*, pages 3–14, 2001.

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

[37] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *OOPSLA '01*, pages 180–195, 2001.

[38] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *VM'04*, May 2004.

[39] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99*, page 13. IBM Press, 1999.

[40] J. Whaley. Partial method compilation using dynamic profile information. In *OOPSLA '01*, pages 166–179, New York, NY, USA, 2001. ACM Press.