McGill University
School of Computer Science
Sable Research Group

Phase-based adaptive recompilation in a JVM

Sable Technical Report No. 2007-4

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

May 25, 2007

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 8.5% and up to 21%, and our online system achieves an average speedup of 4.5%, 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.

1 Introduction

Many of today's Java Virtual Machines (JVMs) [36] 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.

Building a high-performance, adaptive recompilation strategy in a JVM requires making resource-constrained choices as to which methods to optimize, what set or level of optimization to apply, and when the optimized compilation should be done. 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. The upper 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 (upper right), or to skip intermediate steps (lower left). The area under the curve (rectangle) summarizes the "amount" of optimized method execution. On the bottom right 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.

One of the key factors involved in finding ideal 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 JikesRVM [1, 2, 4] 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 of Figure 1. We also develop an *online* analysis that is more practical and dynamically gathers and analyzes phase information.

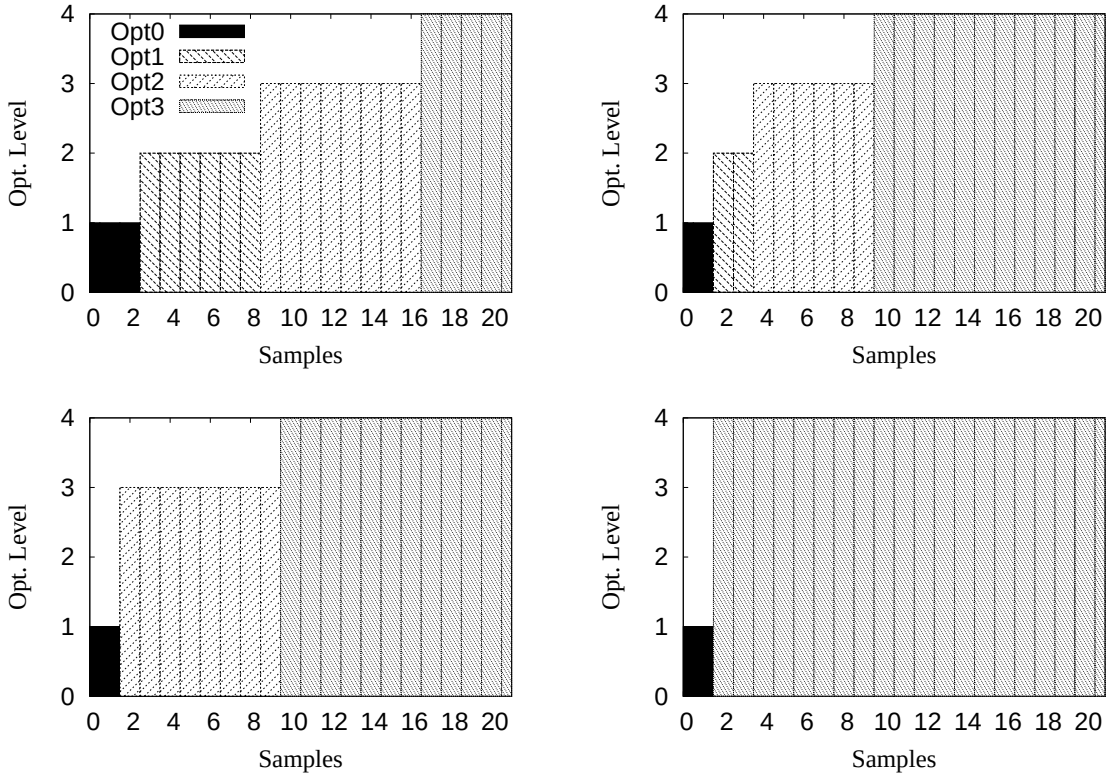

Figure 1: Sources of optimization due to improved recompilation decisions for a given method. In each case the x-axis is samples (normalized time), and the y-axis is optimization level. More time at higher optimization heuristically means better performance, and so the area under each curve roughly represents how well a method is optimized. Left to right on the top row are base recompilation behaviour and the result of more aggressive recompilation. The lower row shows the effects of skipping some intermediate recompilation steps (left), and of making an initial "ideal" choice, skipping all intermediate recompilation (right). Note that even in the latter case at least 1 sample is required to identify the hot method.

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 JikesRVM implementations we observe an average of 8.5% and up to 21% speed improvement in our benchmark suite using the offline approach, and an average of 4.5% 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

JIT compiling and optimizing all the code of a program can easily introduce far too much overhead, both in time and resources. JVM JIT compilers thus typically attempt to identify a smaller hot set on which to concentrate optimization effort. This kind of adaptive optimization allows sophisticated optimizations to be applied selectively, and has been widely explored in the community [4, 31, 40]. Most of this work focuses on methods as a basic compilation unit, but other choices are possible; Whaley, for instance, presents an approach to determining important intra-method code regions from dynamic profile data [54]. 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.

One of the crucial technical challenges for adaptive optimizations is to gather accurate profiling data with as low an overhead as possible. Profiles can be obtained from program instrumentation or from a sampling scheme. Instrumentation techniques are widely used in adaptive optimization: by inserting instrumentation into a program, we can gather accurate profiles at a fine granularity. For example, Dynamo [7] uses instrumentation to guide code transformations. Instrumentation techniques are also useful in program understanding; Daikon [22] is a system for dynamic detection of likely invariants in a program through instrumentation. Even commercial JVMs provide a basic instrumentation interface through Sun's JVMTI specification [51]. Unfortunately, instrumented profilers can also be fairly heavyweight, producing large

runtime overheads [14, 15]. This has inspired work on reducing instrumentation overhead, such as that by Kumar *et al.* in their *INS-op* system that optimizes (reduces) instrumentation points [33].

Alternatively, runtime profiles can be gathered by sampling. In a sampling-based approach, only a representative subset of the execution events are considered, and this can greatly reduce costs. Systems such as JikesRVM [2, 4], use a timer-based approach to determine sampling points. On some other systems, such as IBM's Tokyo JIT compiler [50] and Intel's ORP JVM [17], a count-down scheme is used. An optimization candidate is chosen when the corresponding *counter* reaches a pre-defined value. Arnold and Grove present an approach that combines the timer-based and count-down schemes [5]. Based on the original timer-based scheme in JikesRVM, a stride counter is set to control a series of non-contiguous burst count-down sampling actions.

A sampling-based strategy allows the system to reduce the profiling overhead with reductions in profiling accuracy as a tradeoff. Many techniques have been developed to reduce profiling overhead while maintaining profiling accuracy at a reasonable level. For instance, Zhuang *et al.* [55], for instance, develop an adaptive "bursting" approach to reduce the overhead while preserving accuracy. The key idea of this work is to perform detailed and heavy profiling, but only at selective points.

Phase work can be generally considered with respect to phase *detection* and/or phase *prediction.* Detection techniques work in a reactive manner; program behaviour changes are observed only after the occurrence of the change. A *predictive* mechanism is clearly more desirable for optimization purposes. Prediction techniques can be roughly divided into two types: *statistical prediction* and *table-based prediction*.

Statistical predictors estimate future behaviour based on recent historical behaviour [21]. Many statistical predictors have been developed, including *last value*, *average(N)*, *most frequent(N)* and the *exponentially weighted moving average* (EWMA(N)) predictors. Statistical predictors have been widely used in optimizations based on *(return) value prediction* [13, 24, 39, 41]. Hu *et al.* present a *parameter stride* predictor that predicts return values as a constant offset from one parameter [28]. *Table-based* predictors allow for more arbitrary correlation of program activity and predicted behaviour. Mappings between events or states and predictions of the future are stored in a table and dynamically updated when large behaviour changes are identified. Pickett and Verbrugge develop a *memoization* predictor for *return value prediction* that correlates function arguments with likely return values [41]. Sherwood and Sair use a table-based technique to to perform *run length encoding phase prediction* based on patterns in low level branch data [46]. Duesterwald *et al.* give a general study on predicting program behaviour [21], comparing statistical and table-based models operating on fixed size intervals. Their experimental results show that table-based predictors can cope with program behaviour variability better than statistical predictors. Our prediction technique is largely table-based as well; we use a mixed global/local history and give prediction results with a confidence probability.

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 [8, 16, 19, 29, 44]. Nagpurkar *et al.* present a flexible scheme to reduce network-based profiling overhead based on repetitive phase information gathered from remote programs [38]. Their *phase tracker* is implemented using the SimpleScalar hardware simulator [12]. 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 [37], and further provide 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* [20]. Phase data is also often employed for high level program understanding [23, 49].

Most phase analysis techniques are based on fixed-length intervals, aiming to detect stable periods of program execution [8, 27, 45]. 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 [34]. 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 [9]. Schneider and Gross present a runtime compiler framework using instruction level information provided by hardware counters to detect hot spots and bottlenecks [43]. 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 [32]. They describe the benefits of using hardware counters in addition to software based techniques as crucial components of their profiling system. Other works based on hardware event information can be found in [3, 26, 42, 52]. Many software applications and libraries are available to access these counters, including VTune [18], PMAPI [30], PCL [10] and PAPI [11]. In this work, we use the PAPI library.

3 Basic System

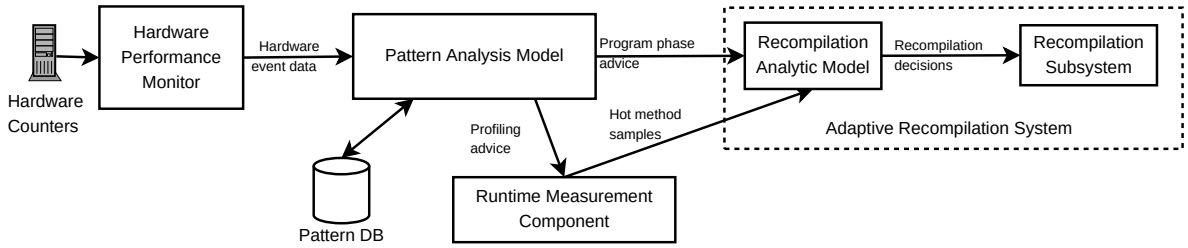

Figure 2: The cooperation among hardware performance monitor, pattern analysis model and adaptive optimization components.

Our system design is based on an extension to the current recompilation system in JikesRVM. Figure 2 shows the overall structure and components of our base system, and how it integrates with JikesRVM. Raw hardware event data is obtained through the *hardware performance monitor* (HPM), a pre-existing component in JikesRVM. The pattern analysis model detects "patterns" in the hardware data. Through comparison with previous patterns stored in the *pattern database*, the pattern analysis model detects 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 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 [35]. Our implementation mainly samples the "L1 instruction cache miss" event, an event known to correlate well with method execution behaviour [26]. The HPM component of JikesRVM 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 event 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; precise details of the pattern construction process and parameter justification are available in a technical report [25].

Our technique operates by summarizing low-level behaviour with short bit-vectors that encode the overall pattern of variation. We 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.

- *Levels*: A basic discretization is applied to variations in event density data to coarsen the data and help identify changes that indicate significant shifts in behavior. We compute the density of events over time for each sample. By comparing the density of the current sample with that of the previous sample, we obtain a variation V. This variation V is discretized to to a corresponding level, P_V. In our experiments we use 4 discrete levels.

- *Shapes*: We determine *shapes* by observing the direction of changes, positive or negative, between consecutive samples. Complexity in shape construction is mainly driven by determining when a pattern begins or ends.

  Each shape construction is represented by a pair $(P_V, \overline{v})$, where P_V is a 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 data with level P_V, if there is no shape under construction a new construction begins with an empty vector: $(P_V, [\,])$. Otherwise, there must be a shape under construction $(Q_W, \overline{v})$.

If $Q_W = P_V$, or we have seen $Q_W > P_V$ less than n times in a row, then shape construction continues based on the current shape construction $(Q_W, \overline{v})$: a bit indicating whether $V > 0$ or not is added to the end of $\overline{v}$.

The following conditions terminate a shape construction:

1. If we find $Q_W < P_V$ we consider the current shape building complete and begin construction of $(P_V, [\,])$. Increases in variation of event density are indicative of a significant change in program behavior.
2. If we find $Q_W > P_V$, n times in a row the current shape has "died out". In this case we also consider the current shape building complete. In our experiments we have found $n = 2$ is sufficient for good performance.
3. If in $(Q_W, \overline{v})$ 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.

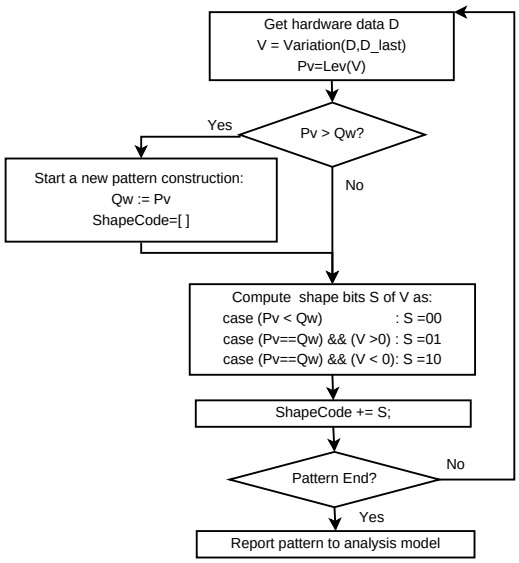

Figure 3: A flow chart for pattern construction.

A overview of the pattern construction algorithm is shown in Figure 3. After obtaining hardware data D, we compute the variation V between D and the same data (D_{last}) for the previous interval. V is then mapped from a real value to an integer value $P_V \in \{0, \ldots, n\}$, representing the level of V. As mentioned in the formal description of this algorithm, Q_W to represents the level of the pattern currently under construction. Initially the value of Q_W is set to -1 to indicate no pattern is under construction. If $P_V > Q_W$ then we are facing a larger and hence more important variation than the one that began the current pattern construction. The current pattern is thus terminated and a new pattern construction associated with level P_V begins. The value of P_V is assigned to Q_W and the shape code vector (denoted as *ShapeCode* in Figure 3) is blanked. Otherwise ($P_V \leq Q_W$) and the current pattern building continues.

The actual pattern encoding is based on the relation between P_V, Q_W and the sign of V. Two bits will be appended to the current *ShapeCode* each time a pattern grows: 01 means a positive variation at level Q_W, 10 represents a negative variation at level Q_W, and 00 means either a positive or negative variation at a level below Q_W. Binary 1s in our scheme thus indicate points of significant change. Construction continues until

one of the pattern termination conditions is met, at which point we report the pattern to the pattern analysis model. A concrete example of the construction of a pattern is shown in Figure 4.

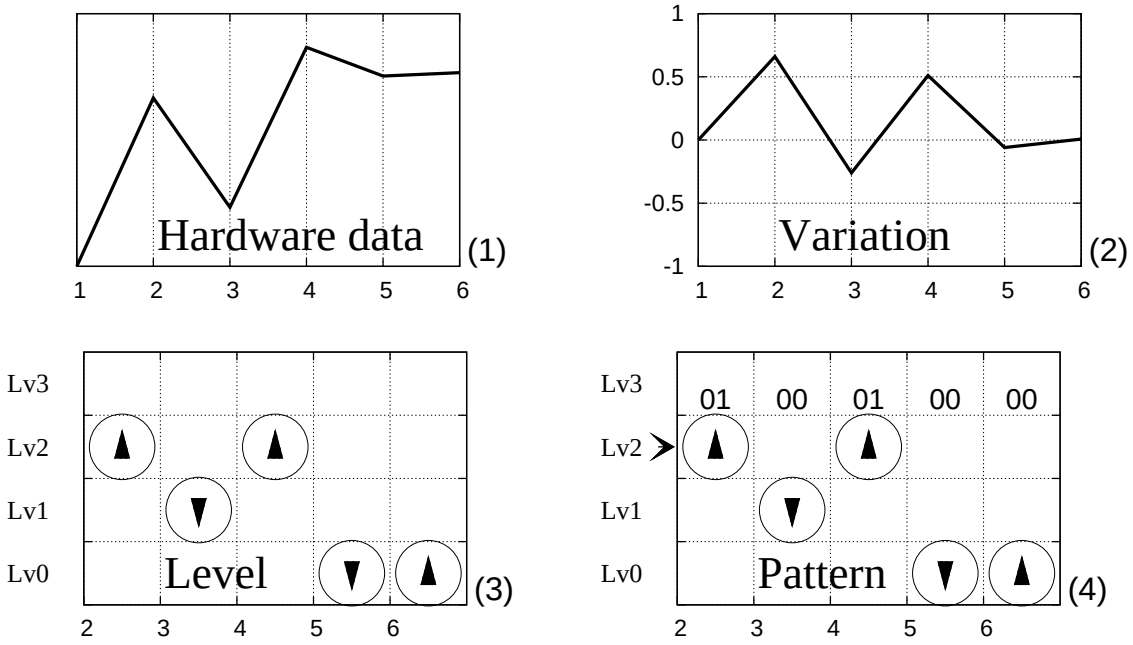

Figure 4: Pattern construction example. (1) Acquire the raw hardware data. (2) Calculate the variation between consecutive points. (3) Coarsen the variation into different levels; the triangles inside each circle show the direction (negative/positive) of variation. (4) The final pattern construction 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.

The same pattern construction strategy can be applied to any hardware event counter, and in general any scalar event data. 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 more thorough investigation of different events and event combinations is left for future work.

3.3 Pattern analysis and prediction

Pattern analysis and prediction consumes created hardware patterns. Here we further examine the patterns to discover repetitive phases and generate predictions of future program behaviour. All created patterns are stored in a *pattern database.* The recurrent pattern detection and prediction are based on the information in the pattern database and the incoming pattern.

The recurrent detection is straightforward: if we find a newly created pattern that shares the same pattern code as a pattern stored in the pattern database we declare it to have recurred. An actually repetitive phase, however, is not declared unless the current pattern also matches the prediction results.

The prediction strategy we use is a variant of fixed-length, local/global mixed history, table-based prediction. Unlike more direct table-based methods our predictions include an attached “confidence” value; this allows us to track multiple prediction candidates and select the most likely.

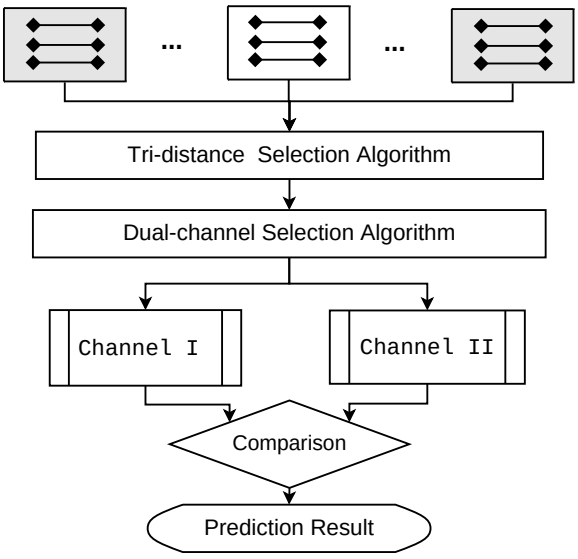

Figure 5: Overview of the prediction mechanism.

Figure 5 gives an overview of our prediction scheme. For each pattern, we keep the three most popular repetition "distances" from a former occurrence to a later one; our experiments showed that three candidates provided a good balance of performance and accuracy. Prediction updates are performed by heuristically evaluating these distances for a given incoming pattern to find the most likely, variable length pattern repetition. Our *tri-distance selection algorithm* updates the likely choices for an incoming pattern p by tracking three repetitions $D_i,\ i \in \{0, 1, 2\}$:

- For each D_i we keep a repetition length L_i, measured by subtracting time stamps of occurrences, and a "hotness" value H_i.
- The difference T_i between the current pattern occurrence p and the ending point of each of D_i is calculated.
- If the difference between T_i and L_i is smaller than a threshold T, the hotness H_i is increased. Otherwise, H_i is decreased.
- If the difference between T_i and L_i is larger than T for all three D_i, we replace the D_j associated with the lowest hotness with a new D_j. The length L_j is based on the distance to the closest of the current set of D_i, and hotness H_j, is initialized to a constant value representing a low but positive hotness in order to give the new pattern a chance to become established.
- We use the D_i with the greatest hotness as the prediction result; H_i further functions as a confidence value for this prediction.

With the current prediction updated we then make a final prediction from the global set of pattern updates. In this case we use two global prediction "channels" to limit the cost of choosing among all possible patterns. Our *dual-channel selection algorithm* is similar to the tri-distance selection algorithm: if the current prediction matches one or both of the prediction channels the channel hotness is increased by the prediction confidence, and if it matches neither then the coldest channel is replaced. The hottest channel then determines the global prediction result.

3.4 Adaptive recompilation system in JikesRVM

The adaptive recompilation system [4] of JikesRVM 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 levels, based on an estimate of the potential benefit of each level. For each optimization level i $(0 \leq i \leq N)$, JikesRVM 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:

- 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 JikesRVM.

- T_i: The expected time of m at level i, if it is not recompiled. In the original implementation, the system assumes:

  $$T_i = T_p \tag{1}$$

- 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 (*Offline*) 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. Use of multiple runs accommodates minor variations in performance; sources of noise in recompilation data is 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 JikesRVM'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 aggression of the recompilation subsystem in JikesRVM. 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	**HW Event Behaviour**	**Recompilation**
Newborn	No recurrence of patterns	Less aggressive
Young	Recurrence of patterns	More aggressive
Mature	Less new patterns More old patterns	Moderately aggressive
Rejuvenated	More new patterns Invalidation of old patterns	More aggressive

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

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 aggression 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 aggression 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 *aggression* 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 in Formula 1 to:

$$T_i = T_p * \textit{futureEstimator} \tag{2}$$

Figure 6 shows a high level overview of the complete online algorithm. 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 $[0.9, 5.0]$.

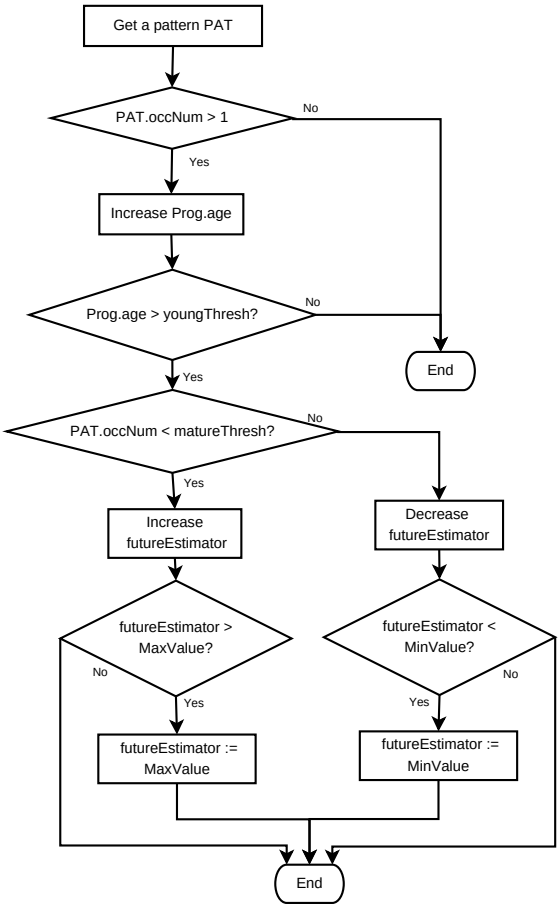

Figure 6: An overview of the algorithm used in the computation of the *futureEstimator*.

5 Experimental Results

Experimentally we evaluated the performance of both our offline and online solutions. Our implementations are built upon JikesRVM 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 [47], and two larger examples, SOOT [53] 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 SPECJBB2000 [48] which 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 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.

Benchmark	**Original**	**Offline**		**Online**		**Benchmark Characteristics**	
	Time(s)	Time(s)	Improvement (%)	Time(s)	Improvement (%)	Patterns	Optimized methods
compress	15.75	15.55	1.3	15.73	0.1	157.9	17.6
db	37.97	37.22	2.0	37.72	0.6	450.5	25.3
jack	22.59	20.08	11.2	19.78	12.5	343.5	90.0
javac	11.78	10.72	9.4	11.10	5.7	193.9	36.9
jess	18.11	14.25	21.3	14.87	17.9	204.5	50.0
mpegaudio	20.24	17.81	12.1	19.79	2.3	103.6	58.9
mtrt	15.14	14.29	6.4	15.42	-1.8	58.8	36.4
raytrace	14.35	13.30	7.3	14.21	0.8	63.9	35.3
soot	303.12	278.45	8.1	291.28	3.9	2542.3	408.2
PseudoJbb	753.95	705.90	6.4	735.62	2.5	7832.8	331.8
Average	-	-	8.5	-	4.5	-	-

Table 2: Execution results, number of patterns created in the online version, and number of methods optimized for SPECJVM98, SOOT and PSEUDOJBB. Values are the arithmetic average of the middle 11 out of 15 runs.

To gain greater insight into the source of improvement, and inspired by our intuition as to potential performance gains in introductory Figure 1, we also developed more abstract, analytical measures that summarize the *amount* of optimized code executed. Our abstract measures of optimization quality are shown in Figure 7 and Figure 8. For space reasons we cannot show all such results in detail, so the analytical results are selected to be representative of the different kinds of observed behaviour.

To measure the relative proportion of code executed at different optimization levels we developed a *method-level speed* (MLS) metric that can be applied to individual methods in individual program executions. MLS is computed as the sum of the time, measured in samples, spent at different optimization levels, weighted by the proportion of time at each optimization level. Each partial sum for an optimization level in this calculation is scaled by an estimate of optimization quality, namely the *speed* of the code under the given optimization level; JikesRVM provides fixed estimates for these latter values. Figure 7 shows the results for a measurement of MLS for the three methods with the largest MLS values in JACK,ordered from top to bottom. The x-axis in these graphs is time, measured in samples, while the y-axis is the estimated speed for

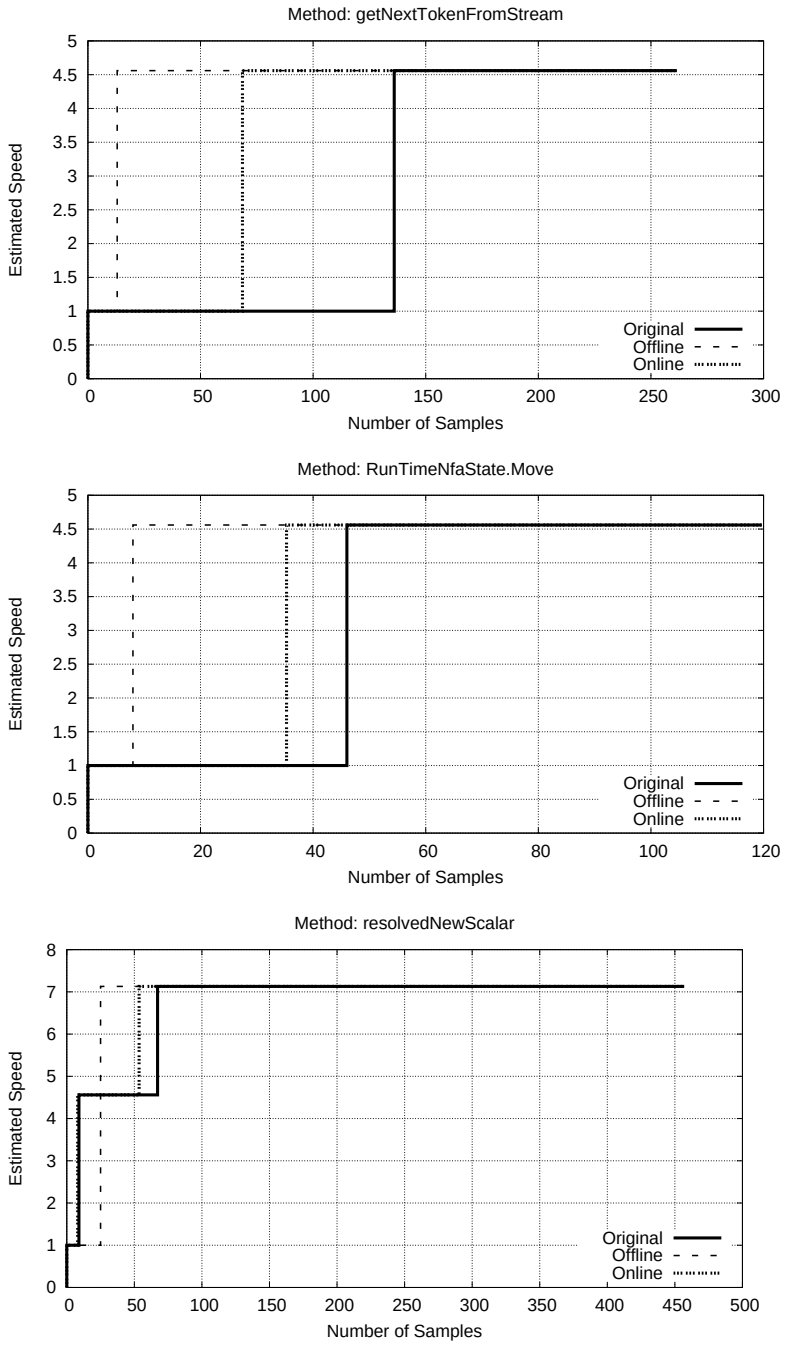

Figure 7: Dynamic *Method Level Speed* measurements over time for each of our baseline, offline and online recompilation approaches. Each graph is a distinct method from JACK.

different optimization levels in JikesRVM. An upward step in the graph corresponds to a recompilation at a higher optimization level. The size of the area under each curve gives an estimate of how MLS changes under different recompilation strategies—greater area means greater use of optimized code, and hence heuristically improved performance.

In Figure 8 we show a summary of the same basic property, but summarized over the entire execution and all

methods. To simplify calculations, method contributions are weighted here not by actual number of runtime samples, but by static method size. This provides a more approximate picture of behaviour, akin to a static versus dynamic analysis, but also demonstrates the effect is robust in the face of different and less precise forms of evaluation. In these figures the x-axis is normalized execution time, and the y-axis is "weighted optimized methods", a sum of weighted method size of all sampled methods, where each weighted sum is again scaled by the appropriate optimization *speed* factor provided by JikesRVM. The interpretation of these graphs is similar to that used for Figure 7; a higher curve means there are more methods optimized to a higher level and the execution speed should be faster, with the area underneath approximating relative amount and quality of optimized code executed.

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 does achieve significant improvements on some benchmarks. On JESS, it improves execution time by 21.3%. On JACK, JAVAC and MPEGAUDIO, the improvements are also quite large. On average, the offline version saves 8.5% of the execution time, although the effect is not uniform; for some benchmarks, such as COMPRESS and DB, there is little to no improvement at all. We will discuss these benchmark-specific behaviours in more detail in Section 6.

In the weighted optimized methods graphs, the curves for our offline implementation are shown as dashed lines. Corresponding with the faster execution speeds, these curves are also the highest ones in these graphs. Interestingly, in most of the benchmarks, there is only one major upwards trend. In the graph for SOOT, however, there are two such increasing phases. This shows the existence of programs with multiple major phases that can require large and relatively abrupt changes in identified hot method sets.

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 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 4.5% improvement, about 53% of the possible performance improvement demonstrated in the offline version. For the 4 benchmarks that responded most positively to the offline version, the improvement online is on average 9.6%, or 71% of the offline result.

In the weighted optimized methods graphs, the curves for the online version are shown as dotted lines, and typically lie between the curves for the offline and original implementations. In the graph for SOOT (the bottom graph in Figure 8), the online curve reflects the multiple phases that are more clearly seen in the offline curve; our online system correctly identifies the rejuvenated phase, as we discuss in more detail in Section 6.1.

Further details on performance can be seen in the behaviour of specific methods, as shown for JACK in Figure 7. As with the weighted optimized method results, the offline version has the greatest area and provides higher optimization earlier, with the online implementation lying between the offline and original versions. Note the bottom graph (*resolvedNewScalar*) shows the offline implementation optimizing the method later than both the original and online versions. This is a result of resource management in the recompilation system, prioritizing requests for relatively fast lower levels of optimization over more expensive requests for longer, highly optimized compilations.

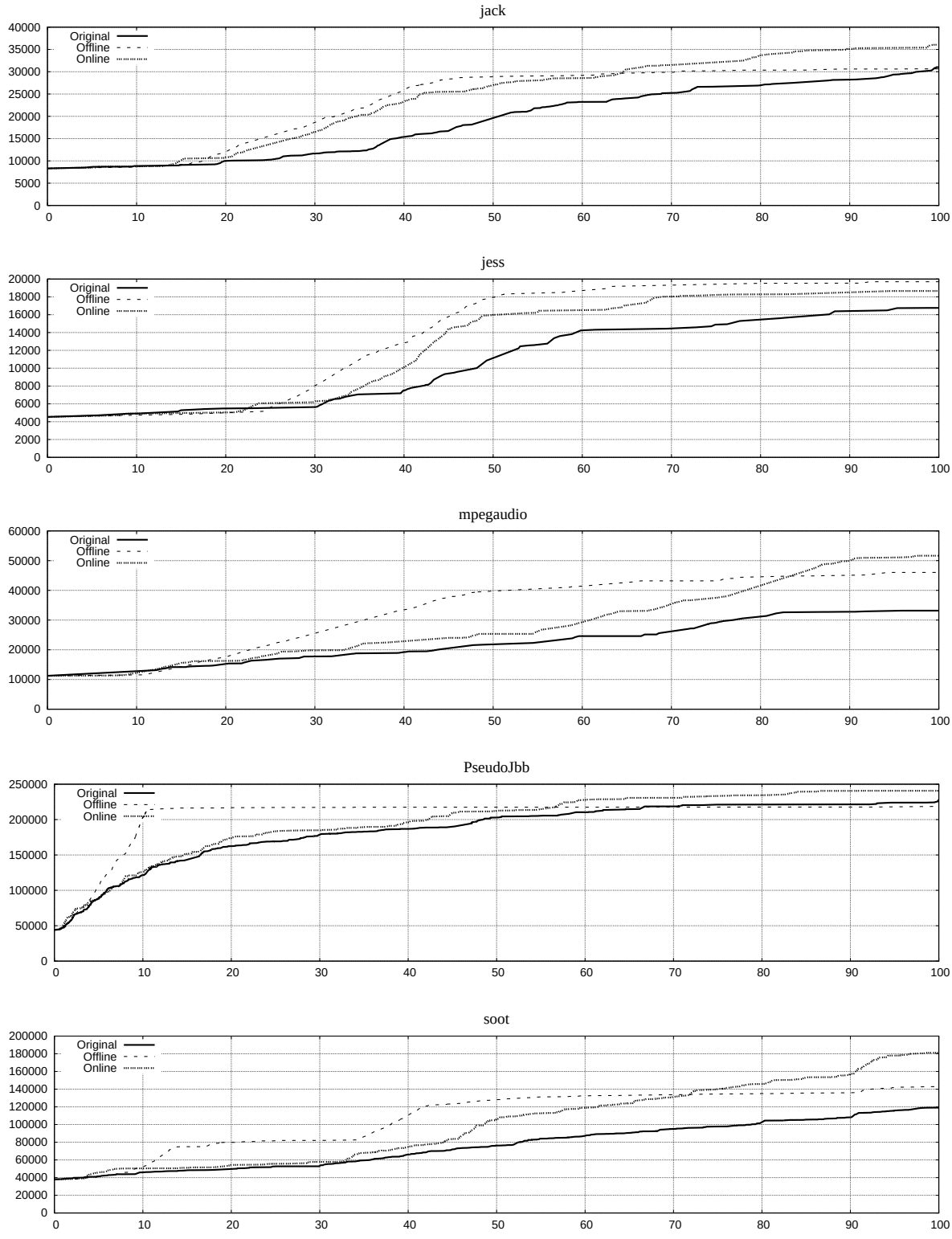

Figure 8: Weighted optimized methods: JACK, JESS, MPEGAUDIO, PSEUDOJBB and SOOT. In each of these graphs the x-axis is normalized time and the y-axis is the "weighted method sum," a heuristic measurement of the amount of optimized execution as described in Section 5.

5.3 Variance and overhead

Figure 9 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.

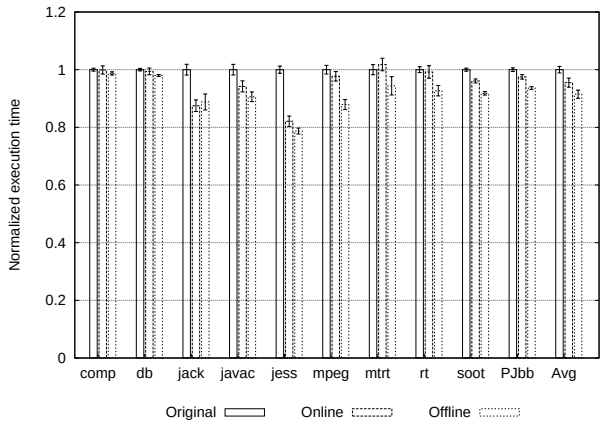

Figure 9: Normalized execution time of SPECJVM98, SOOT and PSEUDOJBB with 99% confidence interval errorbars for each of our three test scenarios: original, online and offline.

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 JikesRVM with an implementation of our online technique that computes phases as normal but does not actually change the adaptive recompilation settings (*futureEstimator*). Figure 10 shows the computed relative overhead. On average there is a 1.33% 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 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.

We have chosen algorithmic parameters to include resource requirements (eg use of tri-selection and dual-channel approaches), and performed extensive initial experimentation and numerical validation of the parameter space to justify our main approach; this numerical evaluation is described in [25]. Here we discuss various factors that can influence our performance, and present data validating the general stability and effectiveness of our design. We first consider different benchmark characteristics that are important in our

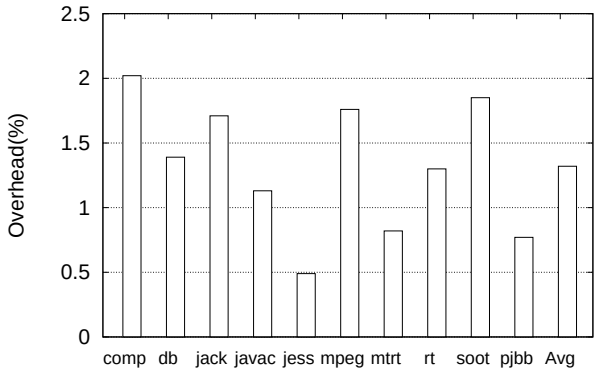

Figure 10: Relative overhead in the online system compared with the original. Overhead comes from sources such as hardware monitoring, pattern construction, phase prediction, and building control events for the recompilation component.

approach. This is followed by a detailed comparison of our design with other simple optimizations to the recompilation system, again showing the practicality of our work and the generally good quality of the result.

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.

Benchmark execution time

In our benchmark suite, the SPECJVM98 benchmarks finish in a comparatively short time while SOOT and PSEUDOJBB execute for an order of magnitude or so longer, and also recompile many more methods than other benchmarks, as seen in the last column of Table 2. Longer running programs have an advantage in that recompilation has more data to work with as there are more sample points. Furthermore, any reduction in speed due to less optimal recompilation choices can be amortized over a longer period and often a larger hot set. For shorter programs our mechanism helps the VM locate the hot set more quickly; the reduction in overhead obtained by promoting methods more quickly to their final optimization level is also a greater benefit. This factor can be seen in the results for the longer and shorter running programs. SOOT and PSEUDOJBB show an average improvement of 7.3% and 3.2% using offline and online analyses respectively, while the other, shorter benchmarks improve on average of 8.9% and 4.8%.

Hot set stability

We observe that many programs contain a single hot set of methods that is more-or-less stable over the course of execution. Some benchmarks, however, do have large, distinct execution phases, and show a clear

hot set variation. SOOT in our benchmarks demonstrates this quite clearly; in Figure 8, the SOOT curve of the offline version obviously has multiple stages. Each large incline corresponds to a major change in the hot set.

Using our offline implementation with perfect knowledge of the future, we can detect the hot set variation or *rejuvenated* phase correctly and quickly, resulting in relatively steep slopes upward as the new hot set is optimized. The original implementation, on the other hand, has no apparent sensitivity to this change in program behaviour and shows a gradually increasing curve with no obvious bursts of optimization. Our online implementation achieves an intermediate level between these two. It has a moderate sensitivity to the hot set variation and goes through a couple of smaller steps at approximately the same points in time, rising more quickly to the level of the offline analysis.

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. The online curves of JACK, MPEGAUDIO and SOOT in Figure 8 tend to rise above even that of the offline curve by the end of execution, indicating that optimized recompilation may be being overused, recompiling and optimizing methods that will only be used in the final fraction of program execution. We experimented with identifying a termination phase, but termination tends to look like any other phase change (rejuvenation) with our current pattern analysis and data. Solutions based on incorporating extra, high level information such as knowledge of termination-specific methods may be more profitable. In practice, these sub-optimal online decisions at termination time do not have an overly large impact, and so we leave reducing this "tail" problem to future work.

Appropriateness of data source

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 this does impact not only what can be measured but also how it can be measured, and of course other choices and event combinations are possible.

Although a good choice in general, for some benchmarks I-cache miss rate provides somewhat reduced information. RAYTRACE and MPEGAUDIO, 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 marginally positive improvements, while MTRT shows a 2% 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. 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 the chance to gather enough samples to recompile a method 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. We previously showed, for instance, that 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 [26]. 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 comparison with simple approaches

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 aggression of recompilation choices, and we must consider that similar effects could be achieved by simply making the the JikesRVM estimator more aggressive without adaptation.

Testing the effects of trivial, constant increases in recompiler aggression 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 JikesRVM with no hardware monitoring or phase analysis, but incorporating our scaled time estimate formula in Formula 2 with *futureEstimator* set to different fixed, constant factors to increase recompiler aggression. 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×; this represents the range of average increase in aggression used by our online system for benchmarks in our suite (Table 3, last row).

futureEstimator	**compress**	**db**	**jack**	**javac**	**jess**	**mpegaudio**	**mtrt**	**raytrace**	**soot**	**PseudoJbb**
1.5×	0.997	0.991	0.987	0.970	0.924	0.960	1.017	0.983	0.966	0.991
2.0×	0.970	1.008	1.041	0.955	0.879	0.924	1.039	1.010	0.950	0.978
2.5×	1.018	1.022	1.063	0.975	0.856	0.925	1.127	1.057	0.945	0.969
3.0×	1.018	1.025	1.080	0.991	0.852	0.948	1.151	1.053	0.969	0.975
online	0.999	0.993	0.876	0.942	0.821	0.978	1.018	0.990	0.961	0.976
online average	3.06	1.98	2.16	2.40	2.34	2.44	2.22	1.99	1.35	1.09

Table 3: Fixed setting of *futureEstimator* versus the online version. The "online average" row shows the average *futureEstimator* value used in the online version, weighted proportionally over program execution.

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 aggression overall is not a generally effective strategy. This is more apparent graphically, as seen in Figure 11. Some benchmarks have a large variance in performance as *futureEstimator* changes, and some are relatively unaffected. For all benchmarks except

MPEGAUDIO and COMPRESS, 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 aggression settings.

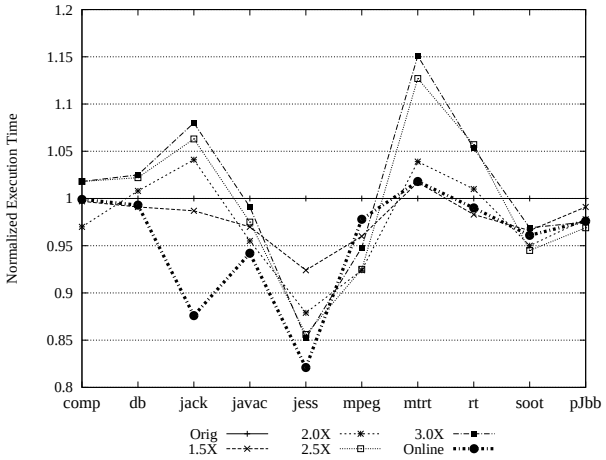

Figure 11: Normalized execution time for benchmarks using different recompilation optimization strategies.

Recompilation algorithm sensitivity

We can separate benchmarks into those that exhibit a low sensitivity to recompilation decisions (less than $\approx 5\%$ variance between approaches), and those that show relatively high variance due to such choices. The former are shown in Figure 12 and the latter in Figure 13.

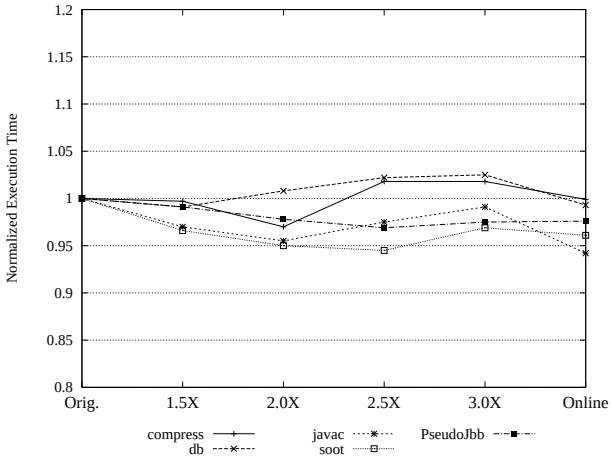

Figure 12: Normalized execution time for benchmarks using different recompilation optimization strategies. These benchmarks seem insensitive to strategy.

The less sensitive benchmarks in Figure 12 correspond reasonably well with our discussion of benchmark-specific behaviours that impair the effectiveness of our technique. SOOT and PSEUDOJBB are long-running with large hot sets, while COMPRESS and DB contain hot sets that are easily identified under all scenarios.

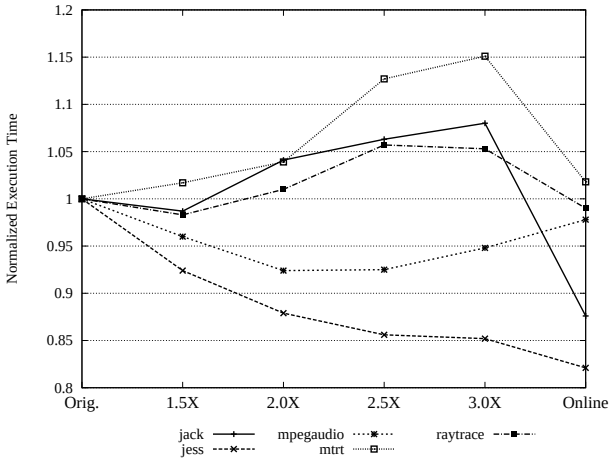

Figure 13: Normalized execution time for benchmarks using different recompilation optimization strategies. These benchmarks are quite sensitive to strategy.

JAVAC is a marginal inclusion; like RAYTRACE it has a small working set, but falls within the threshold of insensitive benchmarks in our simple binary division.

More sensitive benchmarks where recompilation decisions can have a relatively large performance impact are shown separately in Figure 13. Adaptivity accommodates benchmarks where greater aggression usually improves performance such as JESS, and benchmarks where greater aggression decreases performance, such as JACK and MTRT. A more detailed view of typical benchmark behaviour found in our experimental data is shown in Figure 14, with the upper row showing the normalized performance of benchmarks that improve or degrade performance as an almost linear function of recompiler aggression. For benchmarks such as SOOT and MPEGAUDIO, however, a "sweet spot" exists in terms of overall aggression, in both cases here around 2.0–2.5. Adaptation is not as successful overall for MPEGAUDIO while for SOOT adaptation finds a good performance level, albeit in a context where the total performance variation is small. Universally good performance under these conditions is hard to achieve; however, the online system, generally does quite well in adapting to different benchmark conditions and is clearly an overall better choice than either the current or other fixed aggression schemes.

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, and further validated our results with analytical measurements. Detailed examination of benchmark behaviour reveals that benchmarks respond in different

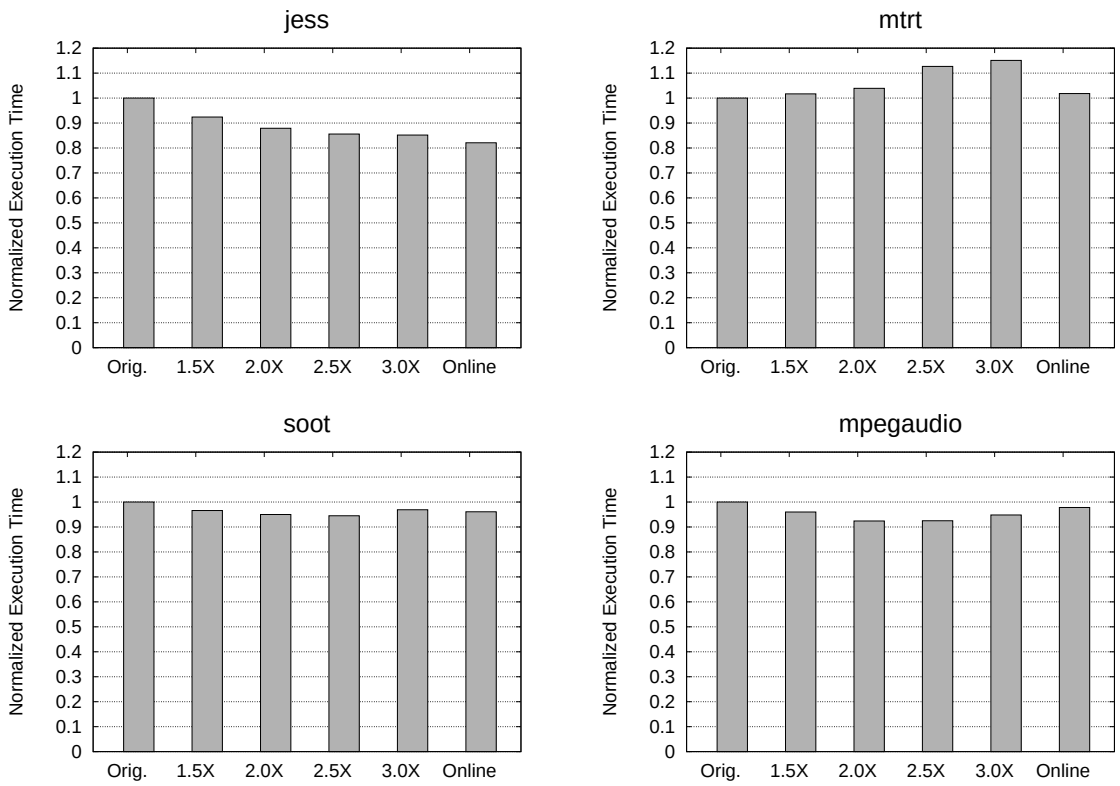

Figure 14: Typical behaviour of benchmarks in response to different recompilation strategies. More aggressive recompilation is in general good for benchmarks like JESS (upper left), bad for others like MTRT (upper right), while some such as SOOT and MPEGAUDIO have an intermediate sweet spot in terms of overall recompiler aggression. In the first three cases the online system adapts well; for MPEGAUDIO the online performance is improved over the baseline but does not achieve optimal performance.

ways to the relative aggression of a recompilation engine, and we considered 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.

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*, aggregating profile data from multiple executions may be a useful way of moving online performance closer to that of offline performance [6]. 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.

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. Predicting 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.

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: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 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] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, Nov. 1997.

[4] 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.

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

[6] M. Arnold, A. Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 297–311, New York, NY, USA, 2005. ACM Press.

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

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

[9] 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 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 233–244, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[10] R. Berrendorf, H. Ziegler, and B. Mohr. PCL-the performance counter library. `http://www.fz-juelich.de/zam/PCL/`.

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

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

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

[14] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization, 1999.

[15] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. *SIGPLAN Not.*, 33(11):139–149, 1998.

[16] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209, New York, NY, USA, 2002. ACM Press.

[17] 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.

[18] I. Corporation. VTune performance analyzer. `http://www.intel.com/software/products/vtune/`.

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

[20] 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.

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

[22] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

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

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

[25] D. Gu and C. Verbrugge. Using hardware data to detect repetitive program behavior. Technical Report SABLE-TR-2007-2, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, March 2007.

[26] D. Gu, C. Verbrugge, and E. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, New York, NY, USA, June 2006. ACM Press.

[27] 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.

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

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

[30] IBM. Pmapi. `http://www.alphaworks.ibm.com/tech/pmapi`.

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

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

[33] N. Kumar, B. R. Childers, and M. L. Soffa. Low overhead program monitoring and profiling. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 28–34, New York, NY, USA, 2005. ACM Press.

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

[35] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, page 220. IEEE Computer Society, March 2005.

[36] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

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

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

[39] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT '99*, pages 303–313. IEEE, 1999.

[40] 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.

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

[42] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 189–198, Oct. 2004.

[43] F. Schneider and T. R. Gross. Using platform-specific performance counters for dynamic compilation. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, October 2005.

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

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

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

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

[48] Standard Performance Evaluation Corporation. SPECjbb2000. `http://www.spec.org/osg/jbb2000`, 2000.

[49] 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.

[50] 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: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 180–195, New York, NY, USA, 2001. ACM Press.

[51] Sun Microsystems, Inc. The Java Virtual Machine Tools Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.

[52] 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:Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, May 2004.

[53] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[54] J. Whaley. Partial method compilation using dynamic profile information. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 166–179, New York, NY, USA, 2001. ACM Press.

[55] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM Press.