RC23429 (W0411-125) November 12, 2004
Computer Science

IBM Research Report

Architecture and Policy for Adaptive Optimization in Virtual Machines

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, Peter F. Sweeney
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Architecture and Policy for Adaptive Optimization in Virtual Machines

MATTHEW ARNOLD, STEPHEN FINK, DAVID GROVE,
MICHAEL HIND and PETER F. SWEENEY
IBM Thomas J. Watson Research Center

Virtual machines for today's object-oriented programming languages face significant performance challenges, when compared to traditional static optimizers. Most notably, portable program representations force the virtual machine to perform most code generation and optimization at runtime. To manage runtime compilation, high-performance virtual machines have adopted sophisticated online *adaptive optimization systems* to manage runtime execution modes.

This article presents an architecture and policy framework for adaptive optimization in virtual machines. The architecture supports construction of an extensible adaptive optimization system based on independent components to manage profiling, decision-making, and recompilation. We present a policy framework for decision-making based on an analytic model of costs and benefits. We have implemented the system in the Jikes Research Virtual Machine, and present a detailed empirical evaluation.

Categories and Subject Descriptors: D.2.3 [**Coding Tools and Techniques**]: Object-oriented programming; D.3.2 [**Language Classifications**]: Object-oriented languages—*Java*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Java, virtual machines, online algorithms, profile-directed optimization

1. INTRODUCTION

Modern object-oriented programming languages, such as the Java™ programming language [Gosling et al. 1996] and C# [ECMA International 2002], support many dynamic features designed to enable flexible, modular, composable programs. Although these dynamic features enhance the programming model, they also introduce significant challenges for achieving high performance. In particular, language features such as dynamic class loading and reflection prevent straightforward applications of traditional static code generation and many forms of interprocedural optimization. Additionally, modern language implementations rely on portable "bytecode" program representations, which force the implementation to defer machine-specific code generation and optimization until runtime.

As a result, virtual machine (VM) implementors have invested significant effort in developing dynamic compilers, which execute at runtime during application execution. Because compilation occurs at runtime, VMs must judiciously trade-off optimization benefits with compilation overhead to maximize performance. Too much, or too little, runtime optimization will result in sub-par performance. On the bright side, runtime optimization also offers opportunities to improve performance, because the system can exploit runtime information to guide transformations.

As virtual machines incorporate a greater number of more sophisticated techniques, the systems face increased challenges in managing complexity. First, each optimization introduces new subsystems into the implementation, which all must interact to manage profiling, code generation, and optimization policy. Second, more advanced optimizations introduce more degrees of freedom into the policy decisions regarding what code to optimize, when to optimize, and what optimizations to apply. Today's systems rely on ad hoc heuristics to guide these policy decisions. However, as complexity increases, the difficulty in tuning these heuristics increases, and threatens to become a significant barrier to the adoption of sophisticated adaptive optimization technology.

To address these problems, this paper presents an architecture and policy framework for online adaptive optimization in virtual machines. The contributions of this work are

—an extensible software architecture for an adaptive optimization system that supports complex mechanisms, isolates separate concerns, and allows a low-overhead implementation,

—a policy framework for controlling adaptive optimization, based on a principled analytic model of system behavior, and

—a detailed experimental evaluation of the system, including evaluations of multi-level adaptive recompilation and some feedback-directed optimizations.

Our work differs from previous systems in that we rely on a unified methodology to support mechanisms and policies for adaptive optimization. This methodology allows us to build a system supporting a wide variety of complex adaptive techniques, while adhering to an extensible overall architecture and avoiding ad hoc heuristics to control each adaptive technique. Experimental results show that the system effectively manages a variety of adaptive techniques, with low overhead that allows high performance. We conclude that these approaches effectively help manage complexity in the implementation of an adaptive virtual machine.

The remainder of the paper is organized as follows. Section 2 presents the design requirements for a model-based online optimization system. Section 3 describes a general software architecture designed to meet those requirements. Section 4 presents the analytical model that is used by the system and demonstrates how the model supports various adaptive techniques. Section 5 presents the implementation of the Jikes RVM model-based online optimization system, an instantiation of the general architecture of Section 3. Section 6 presents an empirical assessment of the implementation on a suite of benchmarks. Section 7 presents a discussion of some of the issues we have encountered while building the Jikes RVM adaptive system. Section 8 presents related work and Section 9 concludes.

2. DESIGN REQUIREMENTS

A virtual machine's adaptive optimization system has the responsibility to manage execution modes of the running program. The adaptive optimization system may choose from among a collection of execution techniques, such as interpretation, non-optimized compilation, and optimized compilation of various flavors. For example, an adaptive optimization system may simply choose to interpret some methods and optimize others based on static properties of the methods, such as loop nesting. A

more sophisticated system will exploit runtime characteristics of the method and/or the program's dynamic execution to aid in this decision.

An effective adaptive optimization system must manage these modes effectively. With respect to performance, we highlight the following desirable characteristics:

Robustness. The system must execute a range of applications with varying execution times and runtime characteristics, without suffering pathologically bad performance in any scenario.

Responsiveness. The system must react quickly to a program's behavior and choose an appropriate execution strategy.

Low Overhead. When the system chooses to take actions, the benefits of these actions must outweigh their associated overhead. A low overhead implementation makes it easier to achieve this goal.

We assume the adaptive optimization system executes in the context of an *online* virtual machine runtime system, which executes a standard, modular, portable program representation such as Java bytecode. We assume that the system must rely solely on information carried by the program representation and information gathered online during the current program run. Thus, we do not consider systems that can cache persistent information between virtual machine runs, such as offline profiles of previous runs. We also exclude systems that rely on aggressive program analysis offline, because we assume the program representation does not provide a standard format to carry such analysis results. We also assume the program might invoke *dynamic class loading*; the program code might expand during execution as it loads new classes. This excludes the possibility of offline whole program analysis.

As a final requirement, we demand an *extensible* architecture. We envision a system that will evolve to incorporate ever more sophisticated optimization mechanisms and policies. The architecture must incorporate enough flexibility to accommodate additional mechanisms and changes in policy, to allow the system to further evolve as technology matures.

The next section presents a high-level architecture for an adaptive optimization system, designed to fulfill these requirements. Section 5 describes the Jikes RVM implementation of this architecture in some detail.

3. ARCHITECTURE

This section presents a general architecture for a virtual machine's adaptive optimization system, designed to fulfill the requirements of the previous section.

The adaptive optimization system architecture contains four components:

—*runtime measurements*: This component monitors the application's runtime behavior and produces profile data.

—*controller*: This component determines what profile data should be gathered, analyzes the profile data and determines which actions to perform regarding management of execution modes.

—*recompilation*: This component implements actions to change execution modes, including recompiling portions of the application.

—*knowledge repository*: This component provides a store of shared information that the other components can consult.

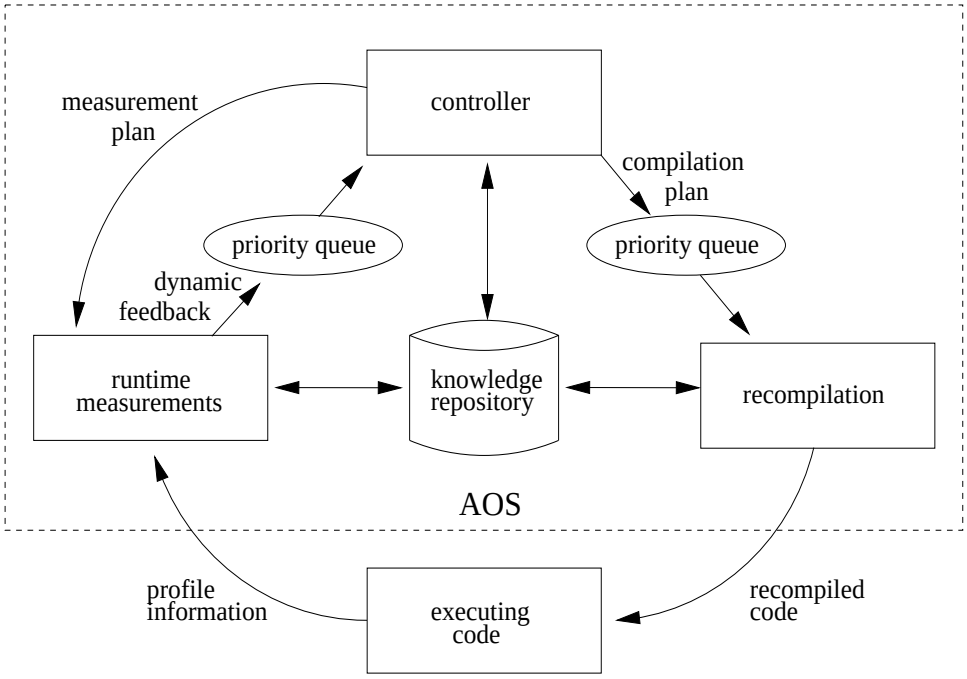


Fig. 1. Components of the adaptive optimization system architecture.

The first three components are *active*, each has one or more independent threads of control. The knowledge repository need not be active; it provides a passive data structure that the other threads consult as needed.

Fig. 1 shows a high-level illustration of the system architecture. The architecture includes priority queues, through which the active components communicate asynchronously with a producer-consumer pattern. The first priority queue handles communication between the runtime measurements component and the controller. The runtime measurements component collects profile information as the program runs, and produces summaries of the data. It uses the priority queue to notify the controller, who wakes and consumes the data.

The controller generates instructions for the other two active components in two ways: it can generate a *measurement plan*, or generate a *compilation plan*.

A *measurement plan* dictates to the runtime measurements component what information about the application's runtime behavior should be profiled. For example, the plan will indicate when profiling should start, what should be profiled, and for how long profile data should be collected. In addition, the measurement plan identifies the format of the profile data that the controller expects. When the runtime measurements component collects the specified body of information, it notifies the controller via the priority queue. The architectural design does not define the type of profile data that can be obtained by the runtime measurements component, or define how profile data is obtained. Section 5 will describe its implementation

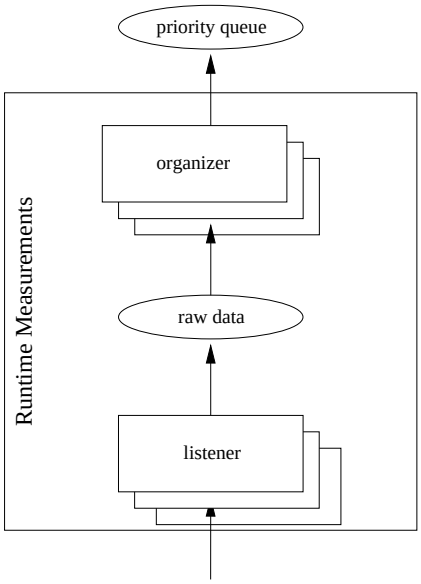


Fig. 2. Subcomponents of runtime measurement component.

in Jikes RVM.

A *compilation plan* tells the recompilation component to either optimize or instrument a region of code. For example, a compilation plan might indicate that a method should be recompiled at a particular optimization level, or indicate that a method should be recompiled with instrumentation that generates certain profile data. The architectural design does not define the unit of compilation or define the possible optimization or instrumentation options. After the recompilation component recompiles a region of code, it installs the newly compiled code for execution by the virtual machine.

The *knowledge repository* in Fig. 1 provides an in-memory repository for facts about the application and about the state of the adaptive optimization system. The runtime measurements component records profile data in the knowledge repository. For example, it could record the number of times a call site executes, for later use by the recompilation component to guide inlining decisions. The recompilation component records the history of compilation plans that it executes, which the controller can consult to decide if a new recompilation plan is warranted. The controller may choose to persistently record some subset of the knowledge repository to enable learning about application behavior across multiple runs.

Fig. 2 shows more details of the subcomponents of the runtime measurements component. The runtime measurements component has two types of subcomponents: listeners and organizers. *Listeners* gather raw profile data about the running application's behavior. When a listener has collected a specified volume of data, it wakes the governing organizers to process the raw data. *Organizer* subcomponents take the raw data and generate profile summaries, extracting from the profile data the information that is requested by the controller. *Organizers* may also initiate communication with the controller, by inserting notifications into the priority

queue.

As described, the architecture provides a clean separation of functionality between the adaptive optimization system components. The component boundaries isolate the various components from each other's implementations, which aids in supporting extensibility. The architecture allows the implementor to prototype new profiling mechanisms, profile-directed compilation options, or controller policies, with minimal disturbance to the rest of the system. The priority queues and plans provide relatively small interfaces through which the components communicate. The high-level architecture does not dictate any details regarding individual profiling, control, or compilation policies; it supports a wide variety of instantiations with different algorithmic and policy choices.

With three independent active components, the design supports a low overhead implementation. The implementation can ration cycles apportioned to each component as needed to throttle system overhead. By using a producer-consumer design, the controller and recompilation components idle until notified of work. Since inter-component communication occurs asynchronously, the system can make progress at whatever rate is appropriate, even if cycles for adaptive system management are scarce. Within the runtime measurements component, only listeners execute continually; organizers wake only occasionally to process raw data. So, it is imperative to implement only listeners with extremely low overhead.

As information flows through the adaptive optimization system, its volume shrinks. A listener generates a body of raw data; the organizer filters and summarizes the data and passes the summary information to the controller. The controller processes all relevant summary data, and generates a concise plan that encapsulates its decisions. With this strategy, each successive stage in the pipeline can apply increasingly sophisticated policies, since each successive stage executes less frequently and considers a progressively smaller input.

This architecture is consistent with an *autonomic element* [Kephart and Chess 2003] for self-optimization. An autonomic element is defined to have four components: *monitor*, *analyze*, *plan*, and *execute*, and a *knowledge* entity. In our adaptive optimization system, the runtime measurements provides the monitor functionality, the controller embodies the analyze and plan functionality, and the recompilation component performs the execute functionality. The knowledge repository comprises the autonomic knowledge entity.

4. MODEL-DRIVEN POLICY FOR ADAPTIVE OPTIMIZATION

As described in the previous section, the controller analyzes summarized profile data and makes decisions regarding what recompilation and profiling actions to perform. We now present a framework for controller policy decisions, based on an analytic model of costs and benefits.

Several previous studies [Plezbert and Cytron 1997a; Kistler and Franz 2003] have presented an analysis of recompilation decisions based on "break-even" points. These studies assume that recompilation of a particular method incurs some cost (time), and will deliver some benefit due to the faster execution of the recompiled method over the remainder of the program's life. Both [Plezbert and Cytron 1997a] and [Kistler and Franz 2003] presented analyses of break-even points in various

scenarios, to determine how long a program must run before adaptive recompilation becomes profitable.

Despite these analyses, all previous systems to our knowledge have relied on ad hoc heuristics to drive recompilation. The most common strategies rely on some form of counter, driving recompilation when a counter trips a threshold.

In contrast, our system attempts to evaluate the break-even point for each action using an online competitive algorithm. We rely on an analytic model to estimate the costs and benefits of each selective recompilation action, and evaluate the best actions according to the model predictions online.

A key advantage of this approach is that it allows a designer to extend the simple "break-even" cost-benefit model to account for more sophisticated adaptive policies, such as selective compilation with multiple optimization levels, on-stack-replacement, and long-running analyses.

This section presents a series of models variations that a controller can use to decide what profiling and recompilation actions should be taken to maximize expected system performance. After reviewing some notation, we present a basic recompilation model and six orthogonal enhancements.

By illustrating how to adapt the basic model to account for a variety of advanced profiling and optimization techniques, we intend to establish that model-driven is flexible and extensible. This discussion does not exhaustively explore every model variation. Some of the variations described here have been implemented in the Jikes RVM adaptive system (described in more detail in Section 5).

4.1 Model Notation

As described in Section 3, the adaptive system controller must decide, when awoken, whether to take one or more actions that affect execution of the system.

In general, each potential action will incur some *cost* and may confer some *benefit*. For example, recompiling a method will certainly consume some CPU cycles, but could speed up the program execution by generating better code. In this discussion we focus on costs and benefits defined in terms of time (CPU cycles). However, in general, the controller could consider other measures of cost and benefit, such as memory footprint, garbage allocated, or locality disrupted.

The controller will take some action when it estimates the benefit to exceed the cost. More precisely, when the controller wakes at time t, it considers a set of n available actions, the set $A = \{A_1, A_2, \ldots, A_n\}$. For any subset $S \in P(A)$, the controller can estimate the cost $C(S)$ and benefit $B(S)$ of performing all actions $A_i \in S$. The controller will attempt to choose the subset S that maximizes $B(S) - C(S)$. Obviously $S = \epsilon$ has $B(S) = C(S) = 0$; the controller takes no action if it cannot find a profitable course.

In practice, the precise cost and benefit of each action cannot be known; so, the controller must rely on estimates to make decisions.

The following subsections discuss specific instances of the model that can be used to drive adaptive optimization. Each instance relies on heuristics to calculate the costs and benefits of the abstract controller model. The following examples do not exhaustively cover all possible model incarnations; these represent straightforward instances that experimental results show work well in practice.

4.2 Basic Recompilation Model

This section presents the basic model the controller uses to decide which method to recompile, at which optimization level, and at what time.

Suppose that when the controller wakes at time t, and each method m is currently optimized at optimization level m_i, $0 \leq i \leq k$. Let M be the set of loaded methods in the program.

Let A_j^m be the action "recompile method m at optimization level j, or do nothing if $j = i$."

The controller must choose an action for each $m \in M$. The set of available actions is $\alpha = \{A_j^m \ | 0 \leq j \leq k, m \in M\}$.

Each action has an estimated cost and benefit:

—$C(A_j^m)$, the cost of taking action A_j^m, for $0 \leq j \leq k$.

—$T(A_j^m)$, the expected time the program will spend executing method m in the future, if the controller takes action A_j^m.

For $S \in \alpha$, define $C(S) = \Sigma_{s \in S} C(s)$. Given S, for each $m \in M$, define A_{min}^m to be the action $A_j^m \in S$ that minimizes $T(A_j^m)$. Then define $T(S) = \Sigma_{m \in M} T(A_{min}^m)$.

Using these estimated values, the controller chooses the set S that minimizes $C(S) + T(S)$. Intuitively, for each method m, the controller chooses the recompilation level j that minimizes the expected future compilation time and running time of m.

It remains to define the functions C and T for each recompilation action. The basic model models the cost C of compiling a method m at level j as a linear function of the size of m. The linear function is determined by an offline experiment to fit constants to the model.

The basic model estimates that the speedup for any optimization level j is constant. The implementation determines the constant speedup factor for each optimization level offline, and uses the speedup to compute T for each method and optimization level.

We assume that if the program has run for time t, then the program will run for another t units, and then terminate. We further assume program behavior in the future will resemble program behavior in the past. Therefore, for each method we estimate that if no optimization action is performed $T(A_j^m)$ is equal to the time spent executing method m so far.

4.3 Sampling-based Recompilation

Let $M = (m_1, \ldots, m_k)$ be the k compiled methods. When the controller wakes at time t, each compiled method m has been sampled $\sigma(m)$ times. Let δ be the sampling interval, measured in seconds. The controller estimates that method m has executed $\delta\sigma(m)$ seconds so far, and will execute for another $\delta\sigma(m)$ seconds in the future.

When driving recompilation based on sampling, the controller can limit its attention to the set of methods that were sampled in the previous sampling interval. This optimization does not lose precision; if the number of samples associated with a method has not changed, then the controller's estimate of the method's future execution time will not change. This implies that if the controller were to consider a method that does not appear in the previous sampling interval, the controller would

make exactly the same decision it did the last time it considered the method. This optimization, limiting the number of methods the controller must examine in each sampling interval, greatly reduces the amount of work performed by the controller.

Suppose the controller recompiles method m from optimization level i to optimization level j after having seen $\sigma(m)$ samples. Let S_i, S_j be the speedup ratios for optimization levels i and j, respectively. After optimizing at level j, we adjust the sample data to represent the system state as if it had executed method m at optimization level j since program startup. So, we set the new number of samples for m to be $\sigma(m) * \frac{S_i}{S_j}$. Thus to compute the time spent in m, we need know only one number, the "effective" number of samples.

4.4 Profile-Directed Inlining

We expect using online profile data to improve the efficacy of optimization. In fact, even if a method is currently optimized at the highest possible level, it might be profitable to recompile to benefit from newly acquired profile data. We enhance the basic recompilation model to account for this scenario.

As a concrete example, consider profile-directed inlining in our system. We maintain a sampled dynamic call graph (DCG), representing the pattern of runtime calls. Each edge in the DCG has a weight w, indicating the number of times that edge was sampled. Let W be the total weight of all edges in the DCG. Thus, $\frac{w}{W}$ is the percentage of the total samples for an edge.

To estimate the benefit of profile-directed inlining, we use the following simple model. We assume that inlining *all* edges in the DCG would enhance any optimization level by a constant factor X. So, we further assume that inlining an edge with weight w would improve the optimization efficacy by a factor of $b = X * \frac{w}{W}$.

Periodically, the controller considers the top $x\%$ of the call edges in the DCG as candidates to drive recompilation. For each such edge, the controller considers recompiling the caller m. For this consideration, the controller estimates that optimization improves by a factor of b; i.e., it reduces its estimate of $T(A_j^m)$ by a factor of b. In this way, the controller will gradually find it profitable to recompile methods in order to inline the dominant edges in the DCG.

4.5 Long-running actions

In some circumstances, it may be profitable for the system to perform some relatively cheap analysis to better determine the expected benefit of a costly action. For example, suppose the system would like to consider an expensive whole-program analysis to rewrite heap data structures. Assume the cost of this action is high, and that the benefit has high variance depending on the program. In this case, it makes sense to perform a cheap prepass analysis to determine whether the expensive analysis will likely pay off.

It is simple to accommodate this pattern of analysis and decision making into the controller model.

Let L be an action, assumed to have a large cost, $C(L)$, and a high variance, for the benefit, $B(L)$. For example, suppose that in many cases $B(L)$ is small, but in a few cases $B(L)$ is large. Suppose that we can partition L into a sequence of less costly actions $\{L_1, \ldots, L_n\}$, where the cost for each individual L_i is relatively small. Further suppose that each action L_i produces information that allows the

controller to better estimate the benefit of the entire sequence. Define $B(L)|L_i$ to be the expected benefit of executing the entire sequence $L_1, \ldots, L_n$, given that the system has already executed $L_1, \ldots L_i$ and considered the results of analysis.

At time t, let $j < n$ be the highest integer such that the program has previously executed L_j. Then the controller considers executing L_{j+1}, using the cost estimate $C(L_{j+1})$ and the computed benefit $B(L_{j+1}) = B(L)|L_j$.

4.6 Instrumentation-based Feedback Directed Optimization

A main benefit of online adaptation is the ability to optimize code based on online profile data. Arnold et al. [2002] investigated this strategy in Jikes RVM by instrumenting code to collect profile data, and then reoptimizing a method based on the profile data.

When is this process profitable? The discussion of the previous subsection can be used to guide controller decisions with a cost-benefit model.

Let L be a long-running action composed of a sequence of the following three actions:

(1) L_1: Recompile method m, inserting instrumentation that collects some class of profile data.
(2) L_2: Allow the instrumented version of m to run for some period, collecting the profile data P.
(3) L_3: Reoptimize m, using P.

Clearly each of L_1 and L_2 have no benefit, but some cost. The cost $C(L_1)$ represents the direct cost of recompiling m with instrumentation. The cost $C(L_2)$ represents the runtime overhead of collecting the profile data while running instrumented code.

The cost $C(L_3)$ represents the direct cost of reoptimizing m. The benefit $B(L_3)|L_2$ can be estimated based on the profile data P gathered during stage L_2.

4.7 Exploiting Deferred Compilation

Typically, when an adaptive optimization system recompiles a method, all new invocations of the method execute this new version, but currently executing invocations continue to use the existing version. On-stack replacement (OSR) is a powerful technique whereby a virtual machine can immediately transfer control into the new compiled version. OSR allows the system to speculatively optimize code, and potentially invoke the OSR mechanism to invalidate and replace the generated code when needed.

Self-91 introduced one application of on-stack replacement called *deferred compilation* [Chambers and Ungar 1991]. With deferred compilation, the system chooses to not generate code for some paths ("deferred paths") in the program. Should the program jump to an instruction for which code was not generated, the system instead jumps to a *uncommon branch extension* stub, which invokes OSR to invalidate the generated code and replace it with a version that includes code for the deferred path.

Deferred compilation changes the tradeoffs in compilation decisions. Fink and Qian [2003] extended the adaptive optimization model to account for deferred compilation and evaluated its effectiveness in Jikes RVM.

Deferred compilation can be used to make adaptive recompilation more aggressive. To this end, the analytic model accounts for the reduced compile time. When considering a method m for recompilation, the system can compute the percentage of the source code, P, that profile data indicates was dynamically reached. Assuming the optimizing compiler is linear in its costs, the cost of optimizing m can be estimated to be $P * C_j$. The result is that the analytic model accounts for the benefit of deferred compilation on a per-method basis.

In addition to deferred compilation, Fink and Qian [2003] also implemented *promotion*, whereby the system may optimize a long-running activation running baseline-compiled code. As described in Section 4.2, the adaptive optimization system considers optimizing a method based on the total time spent in that method in the past. To reconcile promotion with this policy, when the system optimizes a method, it starts to monitor time spent in *outdated* versions of the code (i.e., baseline-compiled versions). To consider promoting an outdated activation, the system uses the default analytic model, but estimates the future time T_j based on the time spent in the outdated code, rather than the total time spent in all compiled versions of the method.

5. IMPLEMENTATION

This section describes the implementation of the Adaptive Optimization System (AOS) in Jikes RVM version 2.3.1, released on December 8, 2003. The section begins with some relevant background information on Jikes RVM. It then presents an overview of AOS and describes the implementation of its two primary functions: selective optimization and feedback-directed inlining. It then discusses how the goals of extensibility and ease of experimentation influenced the design of the controller and concludes with a brief description of the debugging and logging facilities available in AOS.

5.1 Jikes RVM

Jikes RVM (Research Virtual Machine) is an open source research virtual machine for executing Java bytecodes.[1] Jikes RVM is written in the Java programming language augmented with a mechanism for writing snippets of unsafe code [Alpern et al. 1999]. In addition to providing a high-level strongly-typed development environment, this design decision allows the adaptive optimization techniques described in this paper to apply not only to application code, but also to the VM itself, including the compilers, the garbage collector, and the adaptive optimization system. Jikes RVM employs a compile-only strategy; before a method executes for the first time, the system compiles it to native code; methods are never interpreted.

[1] Jikes RVM was originally developed by researchers at IBM's T.J. Watson Research Center and called Jalapeño. In October 2001 the system was released as an open source project and has attracted a large user base among researchers and educators. This section highlights the characteristics of Jikes RVM that are most relevant to this work. A more comprehensive description of the system is given in [Alpern et al. 2000]. Publications, presentations, tutorials, teaching resources, and other information can be found at the Jikes RVM web site `www.ibm.com/developerworks/oss/jikesrvm`.

5.1.1 *Threading and Yieldpoints.* For each physical processor on the system, the system creates a `pthread`. Each pthread is associated with a virtual processor object that executes one or more Java threads in a *quasi-preemptive manner*, as follows. Each compiler generates *yield points*, which are program points where the running thread checks a dedicated bit in the virtual processor object to determine if it should yield to another thread. The compilers insert yield points in method prologues, method epilogues, and on loop backedges. In Jikes RVM 2.3.1, the system sets the thread-switch bit approximately every 10ms.

The adaptive optimization system piggybacks on this yieldpoint mechanism to gather profile data. The thread scheduler provides an extension point by which the runtime measurements component can install listeners that execute each time a yieldpoint is taken. Such listeners primarily serve to sample program execution to identify frequently-executed methods and call edges. Because these samples occur at well-known locations (prologues, epilogues, and loop backedges), the listener can attribute each sample to the appropriate Java source method.

The Jikes RVM implementation introduces a weakness with this mechanism, in that samples can only occur in regions of code that have yieldpoints. Some low-level Jikes RVM subsystems, such as the thread scheduler and the garbage collector, elide yieldpoints because those regions of code rely on delicate state invariants that preclude thread switching. These uninterruptible regions can distort sampling accuracy by artificially inflating the probability of sampling the first yieldpoint executed after the program leaves an uninterruptible region of code.

5.1.2 *Jikes RVM's compilers.* Jikes RVM invokes a compiler for one of three reasons. First, when the executing code reaches an unresolved reference, causing a new class to be loaded, the class loader invokes a compiler to compile the class initializer (if one exists). Second, the system compiles each method the first time it is invoked. In these first two scenarios, the initiating application thread stalls until compilation completes.

In the third scenario, which is the focus of this paper, the adaptive optimization system can invoke a compiler when profiling data suggests that *recompiling* a method with additional optimizations may be beneficial. The system supports both background and foreground recompilation. With background recompilation (the default), a dedicated thread asynchronously performs all recompilations. With foreground configuration, the system invalidates a compiled method, thus, forcing recompilation at the desired optimization level at the next invocation (stalling the invoking thread until compilation completes).

The system includes two compilers.

—The *baseline* compiler translates bytecodes directly into native code by simulating Java's operand stack. The compiler does not build an intermediate representation and does not perform register allocation. The baseline compiler generates native code that performs only slightly better than bytecode interpretation [Alpern et al. 2000], but produces this code quite quickly.

—The *optimizing* compiler [Burke et al. 1999; Fink et al. 2002] translates bytecodes into an intermediate representation, upon which it performs a variety of optimizations. All optimization levels include linear scan register allocation [Po-

letto and Sarkar 1999] and BURS-based instruction selection. The compiler's optimizations are grouped into several levels:

—*Level 0* consists of a set of flow-sensitive optimizations performed on-the-fly during the translation from bytecodes to the intermediate representation and some additional optimizations that are either highly effective or have negligible compilation costs. The compiler performs the following optimizations during IR generation: constant, type, non-null, and copy propagation, constant folding and arithmetic simplification, branch optimizations, field analysis, unreachable code elimination, inlining of trivial methods,[2] elimination of redundant nullchecks, checkcasts, and array store checks. As these optimizations reduce the size of the generated IR, performing them tends to reduce overall compilation time [Whaley 1999]. Level 0 includes a number of cheap local[3] optimizations such as local redundancy elimination (common subexpression elimination, loads, and exception checks), copy propagation, constant propagation and folding. Level 0 also includes simple control flow optimizations such as static basic block splitting, peephole branch optimization, and tail recursion elimination. Finally, Level 0 performs simple code reordering, scalar replacement of aggregates and short arrays, and one pass of intraprocedural flow-insensitive copy propagation, constant propagation, and dead assignment elimination.

—*Level 1* resembles Level 0, but significantly increases the aggressiveness of inlining heuristics. The compiler performs both unguarded inlining of final and static methods and (speculative) guarded inlining of non-final virtual and interface methods. Speculative inlining is driven both by class hierarchy analysis [Dean et al. 1995] and online profile data gathered by the adaptive system. In addition, the compiler exploits "preexistence" to safely perform unguarded inlining of some invocations of non-final virtual methods *without* requiring stack frame rewriting on invalidation [Detlefs and Agesen 1999b]. It also runs multiple passes of some of the Level 0 optimizations and uses a more sophisticated code reordering algorithm [Pettis and Hansen 1990].

—*Level 2* augments level 1 with loop optimizations such as normalization and unrolling; scalar SSA-based flow-sensitive optimizations [Cytron et al. 1991] based on dataflow, global value numbering, global common subexpression elimination, redundant and conditional branch elimination; and heap array SSA-based optimizations, such as load/store elimination [Fink et al. 2000], and global code placement.

Table I gives the average compilation rate and speed of compiled code for the compilers used in this paper on the PowerPC machine described in Section 6. For example, we can see that on average at Opt Level 2, the optimizing compiler is 209 times slower than the baseline compiler, but produces code that executes 6.6 times faster.

[2] A trivial method is one whose body is estimated to take less code space than 2 times the size of a calling sequence and that can be inlined without an explicit guard.

[3] The scope of a local optimization is one extended basic block.

Table I. Average compilation rates and execution speed of compiled code on the SPECjvm98 benchmark suite

Compiler	Bytecode Bytes/Millisecond	Speed
Baseline	377.76	1.0
Opt Level 0	9.29	4.26
Opt Level 1	5.69	6.07
Opt Level 2	1.81	6.61

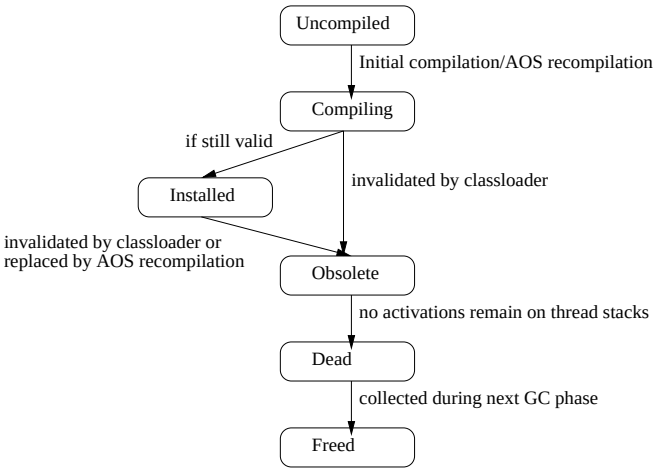


Fig. 3. Life cycle of a compiled method in Jikes RVM

5.1.3 *Life Cycle of a Compiled Method.* In early implementations of Jikes RVM's adaptive system [Arnold et al. 2000b], compilation required holding a global lock that serialized compilation and also prevented classloading from occurring concurrently with compilation. This bottleneck was removed in version 2.1.0 by switching to a finer-grained locking discipline to coordinate compilation, speculative optimization, and class loading. Since no published description of this locking protocol exists outside of the source code, we briefly summarize the life cycle of a compiled method here.

Fig. 3 shows the life cycle of a compiled method. When Jikes RVM compiles a method, it creates a compiled method object to represent this particular compilation of the source method. A compiled method has a unique id, and stores the compiled code and associated compiler meta-data. After a brief initialization phase, the compiled method transitions from *uncompiled* to *compiling* when compilation begins. During compilation, the optimizing compiler may perform speculative optimizations that can be invalidated by future class loading. Each time the compiler so speculates, it records a relevant entry in an invalidation database of the knowledge repository. Upon finishing compilation, the system checks to ensure that the current compilation has not already been invalidated by concurrent classloading. If it has not, then the system installs the compiled code, and subsequent invocations will branch to the newly created code.

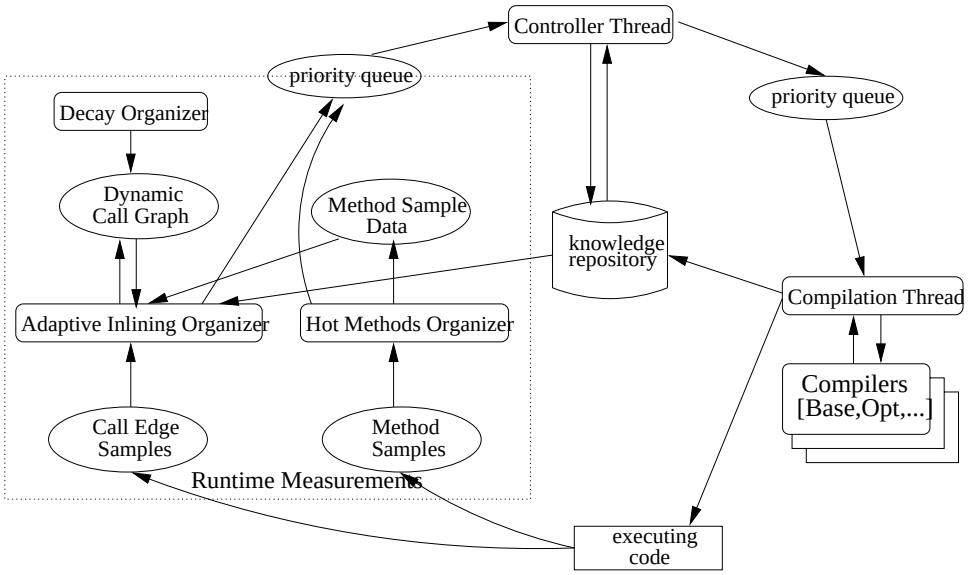


Fig. 4. Adaptive Optimization System as Implemented in Jikes RVM 2.3.1.

Each time a class is loaded, the system checks the invalidation database to identify the set of compiled methods to mark as obsolete, because this classloading action invalidates speculative optimizations previously applied to that method. A method may transition from either *compiling* or *installed* to *obsolete* due to a classloading-induced invalidation. A method can also transition from *installed* to *obsolete* when the adaptive system selects a method for optimizing recompilation and a new compiled method is installed to replace it.

Once a method is marked obsolete, it will never be invoked again. However, before the generated code for the compiled method can be garbage collected, all existing invocations of the compiled method must be complete. A compiled method transitions from *obsolete* to *dead* when no invocations of it exist on any thread stack. Jikes RVM detects this as part of the stack scanning phase of garbage collection; as stack frames are scanned, their compiled methods are marked as active. Any *obsolete* method that is not marked as active when stack scanning completes is marked as *dead* and the reference to it is removed from the compiled method table. It will then be freed during the next garbage collection.

5.2 Overview of AOS

Fig. 4 instantiates the general architecture of Fig. 1 to depict the main implementation artifacts of the Jikes RVM adaptive optimization system. The implementation consists of five Java threads: three organizer threads in the runtime measurements component, the controller thread, and the compilation thread. The various threads are loosely coupled, communicating with each other through shared queues and/or the knowledge repository. All queues in the system are blocking priority queues; if a consumer thread performs a dequeue operation when the queue is empty, it suspends until a producer thread performs an enqueue operation.

The adaptive optimization system performs two primary tasks: selective optimization and feedback-directed inlining. The next two sections describe the implementation of each.

5.3 Selective Optimization

The goal of selective optimization is to identify regions of code in which the application spends significant execution time (often called "hot spots"), determine if overall application performance is likely to be improved by further optimizing one or more hot spots, and if so to invoke the optimizing compiler and install the resulting optimized code in the virtual machine.

In Jikes RVM, the unit of optimization is a method. Thus, to perform selective optimization, first the runtime measurements component must identify candidate methods ("hot methods") for the controller to consider. To this end, it installs a listener that periodically samples the currently executing method at every taken yieldpoint. When it is time to take a sample,[4] the listener inspects the thread's call stack and records a single compiled method id into a buffer. If the yieldpoint occurs in the prologue of a method, then the listener additionally records the compiled method id of the current activation's caller. If the taken yieldpoint occurs on a loop backedge or method epilogue, then the listener records the compiled method id of the current method.

When the buffer of samples is full,[5] the sampling window ends. The listener then unregisters itself (stops taking samples) and wakes the sleeping Hot Method Organizer. The Hot Method Organizer processes the buffer of compiled method ids by updating the Method Sample Data. This data structure maintains, for every compiled method, the total number of times that it has been sampled. Careful design of this data structure (`VM_MethodCountData.java`) was critical to achieving low profiling overhead. In addition to supporting lookups and updates by compiled method id, it must also efficiently enumerate all methods that have been sampled more times than a (varying) threshold value. After updating the Method Sample Data, the Hot Method Organizer creates an event for each method that has been sampled in this window and adds it to the controller's priority queue, using the sample value as its priority. The event contains the compiled method and the *total* number of times it has been sampled since the beginning of execution. After enqueuing the last event, the Hot Method Organizer re-registers the method listener and then sleeps until the next buffer of samples is ready to be processed.

When the priority queue delivers an event to the controller, the controller dequeues the event and applies the model-driven recompilation policy described in Section 4.3 to determine what action (if any) to take for the indicated method. If the controller decides to recompile the method, it creates a recompilation event that describes the method to be compiled and the optimization plan to use and places it on the recompilation queue. The recompilation queue prioritizes events based on the cost-benefit computation.

[4] The sampling rate can be adjusted on the command line; for all of our experiments we used the default sampling interval of 10 ms.

[5] In our experiments we used the default buffer size of 20 samples.

When an event is available on the recompilation queue, the recompilation thread removes it and performs the compilation activity specified by the event. It invokes the optimizing compiler at the specified optimization level and installs the resulting compiled method into the VM.

Although the overall structure of selective optimization in Jikes RVM is similar to that described in [Arnold et al. 2000b], we have made several changes and improvements based on further experience with the system. The most significant change is that in the previous system, the method sample organizer attempted to filter the set of methods it presented to the controller. The organizer passed along to the controller only methods considered "hot". The organizer deemed a method "hot" if the percentage of samples attributed to the method exceeded a dynamically adjusted threshold value. Method samples were periodically decayed to give more weight to recent samples. The controller dynamically adjusted this threshold value and the size of the sampling window in an attempt to reduce the overhead of processing the samples.

Later, significant algorithmic improvements in key data structures and additional performance tuning of the listeners, organizers, and controller reduced AOS overhead by two orders of magnitude. These overhead reductions obviate the need to filter events passed to the controller. This resulted in a more effective system with fewer parameters to tune and a sounder theoretical basis. In general, as we gained experience with the adaptive system implementation, we strove to reduce the number of tuning parameters. We believe that the closer the implementation matches the basic theoretical cost-benefit model, the more likely it will perform well and make reasonable and understandable decisions.

5.4 Online Profile-Directed Inlining

Profile-directed inlining attempts to identify frequently traversed call graph edges and determine whether to recompile to inline these edges. In Jikes RVM, profile-directed inlining augments a number of static inlining heuristics. The role of profile-directed inlining is to identify high cost-high benefit inlining opportunities that evade the static heuristics and to predict the likely target(s) of `invokevirtual` and `invokeinterface` calls that could not be statically bound at compile time. More information on inlining in Jikes RVM can be found in [Hazelwood and Grove 2003].

To accomplish this goal, the system takes a statistical sample of the method calls in the running application and maintains an approximation of the dynamic call graph based on this data. Using this approximate dynamic call graph, the system identifies "hot" edges to inline, and passes the information to the optimizing compiler. The system may choose to recompile previously-optimized methods in order to inline hot call edges. The controller makes inlining decisions using the model described in Section 4.4.

As described previously, the system installs a listener that samples call edges whenever a yieldpoint is taken in the prologue or epilogue of a method. To sample the call edge, it records the compiled method id of the caller and callee methods and the offset of the call instruction in the caller's machine code into a buffer.

When the buffer of samples is full,[6] the sampling window ends. The listener then

[6] In our experiments we used the default buffer size of 200 samples.

unregisters itself (stops taking samples) and wakes the sleeping Adaptive Inlining Organizer. The Adaptive Inlining Organizer processes the sample buffer and updates the dynamic call graph accordingly. The dynamic call graph maintains a set of call edges, a triple of *(caller, callSite offset, callee)*, weighted by the number of times each edge was sampled. After it updates the dynamic call graph, the Adaptive Inlining Organizer analyzes the dynamic call graph to accomplish two tasks.

First, the Adaptive Inlining Organizer identifies any call edges that correspond to more than a threshold percentage of the sampled call edges. These edges are identified as hot call edges. Any subsequent compilation of a hot edge's caller method will be significantly more aggressive in attempting to inline the target (callee method) of the edge into the caller.[7] The system sets the initial edge hotness threshold fairly high, but periodically reduces it until reaching a fixed minimal value.[8] This forces inlining to be more conservative during program startup, but allows it to become progressively more aggressive as profiling data accumulates and becomes more reliable.

Second, the Adaptive Inlining Organizer determines if an opportunity to inline a hot edge has been missed because the method was last optimized before the edge became hot. Such methods could be recompiled (even at the same optimization level) for further speedup. To reduce the overhead of identifying the most profitable candidates, the organizer examines only methods that are already compiled at the maximum optimization level and that represent more than a threshold percentage of the total method samples.[9] The organizer examines the compiled method mapping information for each such hot method to determine if the method contains an attractive hot call edge. If any such methods are found, the organizer creates an event describing the opportunity and enqueues it for the controller to consider using the cost-benefit model described in Section 4.4.

After completing these two tasks, the Adaptive Inlining Organizer re-registers the edge listener and then sleeps until the next buffer of samples is available.

As described in more detail in the context of the Self-93 implementation [Hölzle 1995], the system should take care when recompiling a method previously compiled with feedback-directed inlining, to ensure that previous inlining decisions are not lost. The key issue in both Self-93 and in our system is that once a call edge is inlined, it may no longer appear in the profiling data used to drive the next round of inlining. Therefore, failure to preserve old inlining decisions can result in the system oscillating between two compiled versions of a method, each embodying a different set of inlining decisions. As in previous work, we solve this problem by ensuring that all methods inlined in the previous version of the method are also inlined in the new version. Of course, this can lead to inlining "cold" call edges if the application's behavior changes.

Unlike the implementation of selective optimization, profile-directed inlining relies on a threshold percentage to identify hot call edges. This is unsatisfying in

[7] The static inliner will not inline a callee method whose estimated inlined size exceeds 25 machine instructions. For hot edges, this size threshold is increased to 135 machine instructions.

[8] In our experiments we used the default values of an initial threshold of 6% and a minimal threshold of 1%. After each sampling window, the threshold is halved until it reaches the minimal value.

[9] In our experiments, we used the default threshold value of 0.25%.

several dimensions, and is an aspect of the system that has not evolved very much from its initial implementation as described in [Arnold et al. 2000b]. The most critical weakness of this scheme for identifying hot edges is that it introduces several parameters that must be tuned to maximize performance and that do not correspond to any clean theoretical model. The system is also forced to rely on the periodic decay of the edge weights in the dynamic call graph to remain somewhat responsive to application phase shifts. A more theoretical basis for profile-directed inlining remains an open topic for future work.

5.5 Controller Implementation

A primary design goal for the adaptive optimization system is to enable research in online feedback-directed optimization. Therefore, we require the controller implementation to be flexible and extensible. As we gained experience with the system, the controller component went through several major redesigns to better support our goals.

The controller is a single Java thread that runs an infinite event loop. After initializing AOS, the controller enters the event loop and attempts to dequeue an event. If no event is available, the dequeue operation blocks (suspending the controller thread) until an event is available. All controller events implement an interface with a single method: `process`. Thus, after successfully dequeuing an event the controller thread simply invokes its `process` method and then, the work for that event having been completed, returns to the top of the event loop and attempts to dequeue another event. This design makes it easy to add new kinds of events to the system (and thus, extend the controller's behavior), as all of the logic to process an event is defined by the event's `process` method, not in the code of the controller thread.

A further level of abstraction is accomplished by representing the recompilation strategy as an abstract class with several subclasses. The `process` method of a hot method event invokes methods of the recompilation strategy to determine whether or not a method should be recompiled, and if so at what optimization level. The cost-benefit model itself is also reified in a class hierarchy of models to enable extension and variation. This set of abstractions enable a single controller implementation to execute a variety of strategies. Strategies such as the models of Sections 4.3, 4.6 and 4.7, and several counter-based strategies (described in the subsequent section) were all implemented by extending the `VM_RecompilationStrategy` and/or `VM_AnalyticModel` class.

Another useful mechanism for experimentation is the ability to easily change the input parameters to AOS that define the expected compilation rates and execution speed of compiled code for the various compilers. By varying these parameters, one can easily cause the default multi-level cost-benefit model to simulate a single-level model (by defining all but one optimization level to be unprofitable). One can also explore other aspects of the system, for example the sensitivity of the model to the accuracy of these parameters [Arnold et al. 2000a]. We found this capability to be so useful that the system supports a command line argument that causes it to optionally read these parameters from a file.

As one indication of the flexibility and ease of experimentation enabled by this design, the empirical results reported in the next section were all gathered by

```
  30:891457047728888 Compiled ScannerInputStream.read ()I with baseline compiler in 0.20 ms

  90:891457136817287 Controller notified that method ScannerInputStream.read ()I(14402) has 4.0 samples
  92:891457139813016  Estimated cost of doing nothing (leaving at baseline) to ScannerInputStream.read ()I is 40.0
  92:891457139830219  Estimated cost of OPT compiling ScannerInputStream.read ()I at O0 is 40.42, total future time is 49.81
  92:891457139842466  Estimated cost of OPT compiling ScannerInputStream.read ()I at O1 is 65.99, total future time is 72.58
  92:891457139854029  Estimated cost of OPT compiling ScannerInputStream.read ()I at O2 is 207.44, total future time is 213.49

 110:891457166901172 Controller notified that method ScannerInputStream.read ()I(14402) has 9.0 samples
 111:891457168378722  Estimated cost of doing nothing (leaving at baseline) to ScannerInputStream.read ()I is 90.0
 111:891457168396493  Estimated cost of OPT compiling ScannerInputStream.read ()I at O0 is 40.42, total future time is 61.54
 111:891457168409562  Estimated cost of OPT compiling ScannerInputStream.read ()I at O1 is 65.99, total future time is 80.81
 111:891457168421097  Estimated cost of OPT compiling ScannerInputStream.read ()I at O2 is 207.44, total future time is 221.06
 111:891457168435937 Scheduling level 0 recompilation of ScannerInputStream.read ()I (plan has priority 28.46)
 112:891457169879779 Recompiling (at level 0) ScannerInputStream.read ()I
 114:891457173293360  Recompiled (at level 0) ScannerInputStream.read ()I

 150:891457227058078 Controller notified that method ScannerInputStream.read ()I(14612) has 5.11 samples
 151:891457228691160  Estimated cost of doing nothing (leaving at O0) to ScannerInputStream.read ()I is 51.12
 151:891457228705466  Estimated cost of OPT compiling ScannerInputStream.read ()I at O1 is 66.26, total future time is 102.14
 151:891457228717124  Estimated cost of OPT compiling ScannerInputStream.read ()I at O2 is 208.29, total future time is 241.24

<....many similar entries....>

 998:891458599006259 Controller notified that method ScannerInputStream.read ()I(14612) has 19.11 samples
 999:891458599561634  Estimated cost of doing nothing (leaving at O0) to ScannerInputStream.read ()I is 191.13
 999:891458599576368  Estimated cost of OPT compiling ScannerInputStream.read ()I at O1 is 54.38, total future time is 188.52
 999:891458599587767  Estimated cost of OPT compiling ScannerInputStream.read ()I at O2 is 170.97, total future time is 294.14
 999:891458599603986 Scheduling level 1 recompilation of ScannerInputStream.read ()I (plan has priority 2.61)
1000:891458601308856 Recompiling (at level 1) ScannerInputStream.read ()I
1002:891458604580406  Recompiled (at level 1) ScannerInputStream.read ()I

1018:891458628022176 Controller notified that method ScannerInputStream.read ()I(15312) has 18.41 samples
1019:891458629548221  Estimated cost of doing nothing (leaving at O1) to ScannerInputStream.read ()I is 184.14
1019:891458629563130  Estimated cost of OPT compiling ScannerInputStream.read ()I at O2 is 170.97, total future time is 340.06
```

Fig. 5. Selected AOS logfile entries for a hot method of _213_javac

running a single Jikes RVM image and varying recompilation strategies, analytic models, and compiler parameters on the command line.

5.6 Logging and Debugging

Complex non-deterministic systems such as the Jikes RVM adaptive system present enormous challenges for system understanding and debugging. Virtually all of the profiling data collected by the runtime measurements component results from non-deterministic timer-based sampling at taken yieldpoints. The exact timing of these interrupts, and thus, the profile data, differs somewhat each time an application executes. Furthermore, many optimizing compiler optimizations rely on online profiles of conditional branch probabilities. So, even if the same method is being compiled at the same optimization level from one run to the next, it could be compiled slightly differently depending on profiles of branch probabilities.

The primary mechanism we use to manage this complexity is the ability to log all of the online profile data gathered during one run and feed it back to the system in a subsequent run. As methods are dynamically compiled, the system prints a logging message. At the end of the run, the adaptive system can optionally dump the profile-derived call graph, profile-directed inlining decisions, and the branch probabilities of all instrumented conditional branches. This log of methods and the files of profile data can then be provided as inputs to a driver program (`OptTestHarness`) which replays series of compilation actions and then optionally executes the program. Usually a fairly rapid binary search of methods being compiled and/or the supporting profile data suffices to narrow the cause of a crash to a small set of actions taken by the optimizing compiler. Although this does not enable a perfectly accurate replay of a previous run, in practice we have found that it suffices to reproduce almost all crashes causes by bugs in the optimizing compiler.

In addition to these mechanisms, which mainly help debugging the optimizing compiler, the adaptive system can generate a log file that contains detailed information about the actions of its organizer and controller threads. It supports multiple levels of logging that enable increasing levels of detailed logging (and associated logging overhead). Fig. 5 shows a subset of the log entries associated with the method `read` of the class `spec.benchmarks._213_javac.ScannerInputStream`, one of the hotter methods of the SPECjvm98 benchmark _213_javac. To improve readability of the figure, the package prefix is removed from the method name. The first pair of numbers are the controller clock (number of timer interrupts since execution began) and the value of the hardware cycle counter `VM_Time.cycles()` for the log entry. These log entries show the cost-benefit values computed by the controller for various possible optimization actions and the progression of the method from baseline compilation through two optimizing recompilations (first at level 0 and then at level 1).

For example, at controller clock time 92, we see four entries that give the estimated total future time (the sum of the compilation cost and the total future execution time in a method) for performing no recompilation and for each optimization level. Because the total future time for not recompiling (40) is less than the other alternatives (49.81, 72.58, and 213.490), the method is not scheduled for recompilation. However, at controller clock time 110, the method has been sampled more often. Thus, the total future time estimate is updated, resulting in two recompilation actions (level 0 and level 1) that are more attractive than taking no recompilation action. Because level 0 gives the least future time, this decision is chosen by placing a recompilation event in the recompilation priority queue. The priority for the event is the difference between the future time for the new level and the future time for current execution ($90 - 61.54 = 28.46$).

6. EMPIRICAL ASSESSMENT

This section empirically evaluates the adaptive optimization architecture and policy described in the earlier sections of this paper. We evaluate the importance of performing selective optimization, specifically looking at the benefits of model-driven recompilation, using multiple optimization levels, and performing feedback-directed optimizations. In addition, we evaluate the overhead introduced by our system and characterize some of the decisions that it makes, such as the number of methods recompiled over time.

6.1 Benchmarks

The first column of Table II lists the benchmarks used in this study. The first seven benchmarks are the SPECjvm98 benchmark suite [Standard Performance Evaluation Corporation b]. `SPECjbb2000` is an emulator of a typical Java business application [Standard Performance Evaluation Corporation a]. `Pseudojbb` is a modified version of SPECjbb2000 that executes a fixed number of transactions, rather than executing for a fixed time period. `Ipsixql` is a benchmark of persistent XML database services [colorado bench]. The `xerces` benchmark measures a simple XML Parser exercise, using the Apache Xerces Java2 parser [xerces]. `Daikon` is a dynamic invariant detector from MIT [daikon]. `Kawa` exercises a Java-based Scheme system [kawa]. `Soot` is a Java bytecode analysis and transformation

Table II. Description of benchmarks.

Benchmark	Medium	Large	Steady-state
compress	1 iter w/ `-s10` input	1 iter w/ `-s100` input	37 iters w/ -s100
jess	1 iter w/ `-s10` input	1 iter w/ `-s100` input	62 iters w/ -s100
db	1 iter w/ `-s10` input	1 iter w/ `-s100` input	23 iters w/ -s100
javac	1 iter w/ `-s10` input	1 iter w/ `-s100` input	37 iters w/ -s100
mpegaudio	1 iter w/ `-s10` input	1 iter w/ `-s100` input	35 iters w/ -s100
mtrt	1 iter w/ `-s10` input	1 iter w/ `-s100` input	76 iters w/ -s100
jack	1 iter w/ `-s10` input	1 iter w/ `-s100` input	48 iters w/ -s100
ipsixql	1 iter w/small workload	1 iter w/large workload	37 iters large workload
xerces	SAXCount exercise	DOMCount exercise	-
daikon	Stack example	Queue Example	-
kawa	2K line scheme input	12K line scheme input	-
soot	HelloWorld input	SpecApplication input	-
saber	Standard reporting	HTML reporting	-
xalan	10K XML input	-	-
pseudojbb	12,000 transactions	200,000 transactions	-
SPECjbb2000	-	-	5 min warm up/ 5 min timed

framework from McGill University [soot]. `Saber` is a J2EE code validation tool developed within IBM [Reimer et al. 2004]. `xalan` is an XSLT processor [xalan].

None of the results reported conform to the official SPEC run rules, so our results do not directly or indirectly represent a SPEC metric.

In the experiments that follow, we evaluate performance in each benchmark in one of two "regimes": *total-time* and *steady-state*.

For *total-time* regime experiments, we measure the time for the benchmark to perform a fixed body of work, starting from a cold start. For each benchmark, we have defined a "medium" configuration and a "large" configuration for the total-time experiments.[10] The second and third columns of Table II list the inputs used to define the medium and large configurations of each benchmark.

For *steady-state* regime experiments, we configured each benchmark to run for a fixed amount of *time*, and define a *warm up* period during which performance is not recorded. This regime evaluates the performance that is eventually converged upon in a long running application. We chose a 5 minute warm up period followed by 5 minutes of timed execution. SPECjbb2000 fits naturally into this regime and was configured with a 5 minute warm up and 5 minute timed interval. The remaining benchmarks were executed in an iterative driver, and configured to continue for 10 minutes. For the first 5 minutes timings were ignored; for the second 5 minutes, the average time per iteration is reported. The last column of Table II defines the steady-state regime measured for each benchmark. Several benchmarks cannot execute correctly in a loop since they rely on static state; we could not define steady-state regimes in these cases.

Table III characterizes the behavior of the benchmarks in our suite for each of the regimes. The first column of each regime reports the number of methods executed, the second column shows the total number of bytecodes compiled (in K), and the

[10] Xalan currently has only one input. A second input will be added in the final version

Table III. Characterization of benchmarks.

	Medium			Large			Steady-state		
Benchmark	Meth Exe	K BC Exe	Time (s)	Meth Exe	K BC Exe	Time (s)	Meth Exe	K BC Exe	Time (s)
compress	243	22	3.08	243	22	23.97	243	22	14.95
jess	662	42	1.58	675	43	15.65	675	43	7.38
db	258	24	0.74	262	24	23.33	262	24	22.12
javac	939	86	2.36	967	87	21.83	967	87	14.82
mpegaudio	416	67	5.14	415	67	27.22	415	67	19.06
mtrt	368	31	3.60	369	31	13.24	369	31	6.57
jack	477	52	2.68	478	52	17.29	478	52	10.99
ipsixql	459	31	3.76	490	33	23.26	490	33	13.72
xerces	719	64	6.26	822	71	11.79	-	-	-
daikon	1671	140	11.64	1673	141	63.38	-	-	-
kawa	1794	96	3.64	3496	161	16.19	-	-	-
soot	1215	111	12.00	1734	126	53.55	-	-	-
saber	1970	218	7.40	4372	377	20.15	-	-	-
xalan	1582	142	2.14	-	-	-	-	-	-
pseudojbb	587	51	5.00	597	51	53.88	-	-	-
SPECjbb2000	-	-	-	-	-	-	962	85	2.10

third column shows the running time when executed with the best performing configuration of Jikes RVM (multi-level model with FDO). For the steady-state configurations, the table reports the time for one iteration.

6.2 Data Collection Methodology

Our empirical evaluation was performed using a version of Jikes RVM extracted from the open source CVS repository as of January, 20, 2004, with minor bug fixes applied.[11] All experiments use a `FastAdaptiveCopyMS` boot image, i.e., the garbage collector is a non-generational copy/mark-sweep hybrid [Blackburn et al. 2004]. This collector gave the highest performance for this version of Jikes RVM. The system was run on a dedicated IBM RS/6000 Model F80 with six 500MHz processors and 4GB of main memory, running AIX 4.3.3. For all experiments, we restricted the system to use only one processor. This decision places the most demands on the adaptive optimization system, because for single-threaded benchmarks, the system cannot offload background compilation onto a spare processor.

The first three experiments in Sections 6.4 – 6.6 were performed with feedback-directed optimizations disabled. Section 6.7 considers feedback-directed optimizations in detail. Sections 6.8 and 6.9 present runtime characteristics of the adaptive system with feedback-directed optimizations enabled.

All performance timings reported in this section were computed by taking the median of N runs, where N is 10 for the medium and large regimes, and 3 for the steady-state regime.[12] Tables IV–VII in Appendix B report the performance

[11] This system corresponds to the 2.3.1 release described in Section 5 with an additional month's worth of development and bug fixes.

[12] Fewer runs were used for the steady-state regime for two reasons. First, each benchmark runs for 10 minutes so there are practical reasons of collecting the data. Second, the timings reported are the average performance over 5 minutes, so there is already some degree of noise reduction.

measurements in detail, including the variance over the N runs as well the raw execution times.

6.3 Training Methodology

The models and heuristics used to derive adaptive recompilation typically depend on some characterization of system performance. To train the models and heuristics, we partition the benchmark set into a *training set* and a *production set*. We define the training set to be the "size 100" runs of the SPECjvm98 benchmarks, and the production set to be all other codes.[13]

As discussed in Section 5, the Jikes RVM adaptive optimization system model is calibrated by constants that represent the compilation rate and expected speedup provided by each optimization level available. To gather these constants, we ran the benchmarks in the training set in a non-adaptive mode for each compiler of interest (Baseline, Opt 0, Opt 1, or Opt 2); in this non-adaptive mode, each application method is compiled at a fixed optimization level the first time it is invoked, and no profiling or recompilation is performed. We used a `production` image (assertions turned off and all VM classes precompiled) with a 400MB heap and the `GenMS` (generational with mark-sweep mature space) garbage collection policy, which is Jikes RVM's highest performing memory management system at the time of the measurements.

To gather speedup data we ran each benchmark for five iterations and used the time of the best run; this methodology factors out compilation time (which happened during the first run) and thus evaluates the quality of the code generated by the compiler being used. This process was repeated for each compiler, and the overall speedup for each compiler was computed by taking the geometric mean across all benchmarks in the training set.

To measure compilation rate data, we ran each benchmark for one iteration and measured the number of bytecodes compiled and the time spent compiling. We computed the average compilation rate (bytecodes per millisecond) for each benchmark and then computed the geometric mean across all benchmarks for each optimization level.

This training configuration allowed us to measure the observed compilation rate for each optimization level, and derive relative speedup numbers by comparing the performance difference between the optimization levels. The resultant constants determined were presented in Table I of Section 5; these constants were used in the analytic controller model throughout these experiments.

6.4 Selective Optimization

Before evaluating the tradeoff between various selective optimization policies, we first verify the fundamental assumption that a selective optimization policy is necessary. This section addresses the question: *how beneficial is selective optimization?*

To answer this question, we define a single-mode execution strategy called JIT. Under this strategy, the system compiles each method with a fixed optimization

[13] This decision reflects common industry practice, where the VM may be tuned for a small set of standard industry benchmarks.

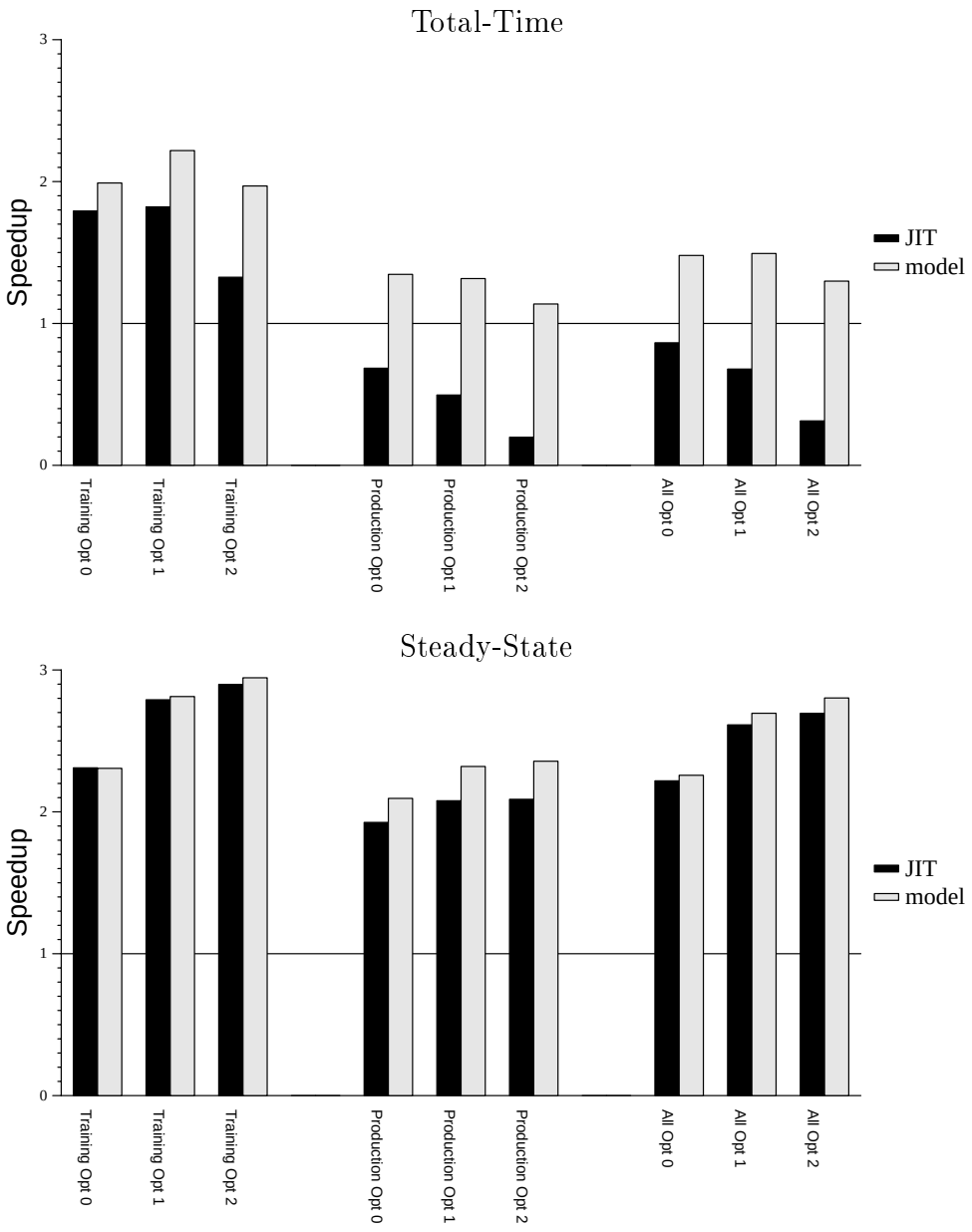


Fig. 6. Summary performance comparison between JIT and single-level models normalized to baseline. Results for individual benchmark are presented in Appendix A Figures 16 and 17.

level (baseline, Opt level 0, Opt level 1, or Opt level 2) the first time the method is invoked.

To make a fair comparison, we evaluate the performance of this JIT strategy relative to a system that performs selective optimization using only a single optimization level. This single-level adaptive system first compiles each method with the baseline compiler, and then uses the cost-benefit model to apply the optimizing compiler to the hot methods; thus, we refer to this system as a *single-level model*.

Fig. 6 summarizes performance of the JIT and the single-level model strategies across all benchmarks in both "total-time" and "steady-state" regimes. The figure presents performance normalized to the speed of a system that uses the baseline compiler only. The first group of bars, labeled "Training", reports a summary of the results for the training benchmarks only (described in Section 6.3). The middle group of bars, labeled "Production", reports the summary for the non-training benchmarks, and the final group, labeled "All" reports the summary over all benchmarks. All summaries are computed using the geometric mean.

In the total-time regime, performance of the JIT strategy generally declines as the optimization level increases. Because these programs are relatively short running (0.7—63.3 seconds), the system cannot afford to apply the costly level 2 optimizations to every method. In fact, in the production and overall set, *any* single JIT compilation strategy underperforms the baseline compiler, which produces code faster than the three levels of the optimizing compiler.

In contrast, in the steady-state regime, JIT configuration performance increases as we apply more sophisticated optimizations. In our steady-state experiments, the JIT configurations will compile each method before the timing period commences, so high quality code can be obtained for free.

In the total-time regime, model-driven selective optimization outperforms the corresponding JIT strategy in each case. In the overall set, the model-driven strategy improves performance over the JIT by 71% (optimization level 0) to 414% (optimization level 2). Furthermore, the model-driven strategies outperforms the baseline compiler by 48% to 30%. Clearly, selective optimization is necessary to achieve competitive performance in the total-time regime.

In the steady-state regime, the model-driven strategy delivers comparable performance to the corresponding JIT strategy in each case. Counter-intuitively, the model-driven strategies outperform the corresponding JIT strategies by 1.8% to 4.9%. We believe that the selective optimizer benefits from delayed compilation. The selective strategy delays compilation until more of the class hierarchy becomes available; this delay allows more effective class hierarchy-based optimizations such as inlining.

In summary, selective optimization significantly outperforms any fixed execution mode considered. The model constrains optimization enough to perform well in the total-time regime, but optimizes aggressively enough to match or exceed the best performance in the steady-state experiments.

Figs. 16 and 17 in the Appendix A presents the detailed data for each individual benchmark, which Fig. 6 summarizes. In the total-time regime each single-level model outperformed its corresponding JIT strategy in 26 of 29 benchmarks with maximum gains of 86.5%, 92.1%, and 98.8% for levels 0, 1, and 2, respectively. In the worse case, the JIT strategy beat the single-level model by 30.0%, 50.2%, and

33.5% for levels 0, 1, and 2, respectively. In the steady-state regime each single-level model outperformed its corresponding JIT strategy in 5 of 9 benchmarks at levels 0 and 1 and 6 of 9 benchmarks at level 2. The maximum gains are 17.9%, 21.8%, and 23.3% for levels 0, 1, and 2, respectively. In the worse case, the JIT strategy beat the single-level model by 4.0%, 2.6%, and 2.4% for levels 0, 1, and 2, respectively.

6.5 Model-driven policy evaluation

Most other VMs described in the literature have used counter-driven policies to drive selective optimization. This section compares the model-driven single-level selective optimization, that was presented in the previous section, with counter-driven single-level selective optimization. In particular, this section addresses the question: *how beneficial is model-driven selective optimization?*

For these experiments, we implemented a counter-driven recompilation policy that is similar to those reported in [Suganuma et al. 2001]. As in the previous experiments, we restrict the system to two modes of execution: baseline-compiled code, and optimized code at a single optimization level. The system associates a counter with each method, and the baseline compiler inserts instrumentation to increment a method's counter when either a) the method is invoked, or b) a backward branch is taken within the method. When the counter trips a predetermined threshold, the system recompiles the method with the optimizing compiler.[14]

To determine the counter thresholds, we performed an exhaustive search over possible threshold values for the benchmarks in the training set. We selected the threshold that gave the best overall performance on the training set. Figs. 13 – 15 in Appendix A show the performance curves used to tune the counter thresholds.

Fig. 7 compares the performance of the counter-driven policies and the model-driven policy in the total-time regime. On the training set, the performance of the two strategies is roughly comparable; the model outperforms the counters by 0.4% (level 0), 0.9% (level 1), and 4.6% (level 2). On the production set, the model outperforms the counters by 16.9% (level 0), 3.4% (level 1), and 9.4% (level 2).

Fig. 7 also compares the performance of the counter- and model-driven policies in the steady-state regime. In all configurations, performance of the two policies is within 1%. This is expected because in steady-state, all the critical methods have already been optimized in the warm up phase.

Summarizing these results, we see that when restricting to selective single-level optimization, the model can match or exceed the performance of the best tuned counter strategy. This is significant because the counter strategies required exhaustive tuning using multiple runs to explore the space of possible counter values. The model strategies were also *trained*, but they were not exhaustively *tuned*; the model's constants are derived by measuring (once) the characteristics of the compilers in the system. While this distinction may seem subtle, the impact can be substantial. Training our model takes roughly 30 minutes using our current training benchmarks; collecting the tuning data presented in Figs. 13 – 15 of Appendix A required several days of CPU time.

[14] By default, Jikes RVM performs compilation using an asynchronous compilation thread. This mechanism was used for all configurations to allow a fair comparison.

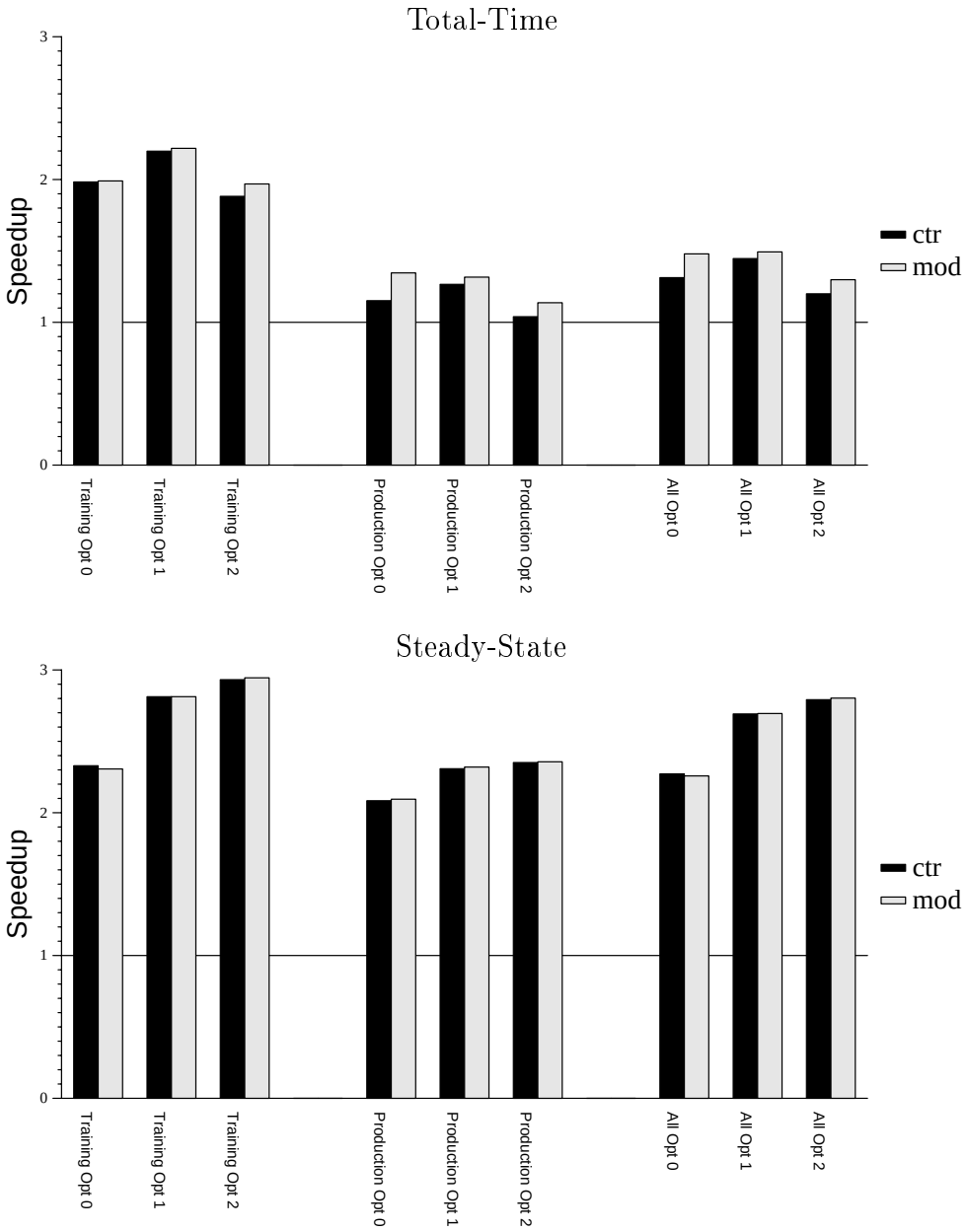


Fig. 7. Summary performance comparison between counter and single-level models normalized to baseline. Results for individual benchmark are presented in Appendix A, Figures 18 and 19.

Even more important, we believe that the model will be more robust with respect to tuning, and will be more likely to provide good performance for programs outside the training set. This hypothesis is supported by our data, as the model strategies significantly outperformed the counter strategies in the production set for the total-time regime.

Figs. 18 and 19 in the Appendix A presents the detailed data for each individual benchmark, which Fig. 7 summarizes. In the total-time regime each single-level model outperformed its corresponding counter strategy in 16 of 29 benchmarks at level 0, in 15 of 29 benchmarks at level 1, and 23 of 29 benchmarks at level 2. The maximum gains are 72.0%, 26.0%, and 40.4% for levels 0, 1, and 2, respectively. In the worse case, the counter strategy beat the single-level model by 11.2%, 10.6%, and 10.4% for levels 0, 1, and 2, respectively. In the steady-state regime each single-level model outperformed its corresponding counter strategy in 1 of 9 benchmarks at levels 0, in 4 of 9 benchmarks at level 1, and in 6 of 9 benchmarks at level 2. The maximum gains are 1.8%, 1.1%, and 2.3% for levels 0, 1, and 2, respectively. In the worse case, the JIT strategy beat the single-level model by 4.6%, 0.5%, and 1.4% for levels 0, 1, and 2, respectively.

6.6 Multiple optimization levels

All of the adaptive strategies evaluated so far have been using single-level selective optimization, i.e., using only a single level of optimization for hot methods. As discussed in Section 4, it is straightforward to define the model-driven recompilation policy to consider multiple optimization levels, and we evaluate such a system in this section. This section addresses the question: *how beneficial is multi-level selective optimization?*

With multi-level selective optimization, there are multiple optimization levels that can be chosen and a method may be optimized multiple times, each time at a successively higher optimization level. Fig. 8 compares the multi-level selective optimization system to a single-level system using each optimization level. The bars are normalized to the multi-level system, so a bar below 1.0 indicates that the multi-level system performs better than that single-level system. For the total-time regime, the best single-level strategy for the production set is optimization level 0; it outperforms the other single-level strategies, and is 0.3% better than the multi-level system. For the overall data set, optimization level 1 is the best single-level strategy; it outperforms the other single-levels strategies and is within 1% of the performance of the multi-level system.

In the steady-state regime, the best single-level strategy is optimization level 2. Optimization level 2 overall outperforms level 0 and level 1 by 24.1% and 4.0%, respectively. However, the multi-level model matches the best single-level model.

In summary, although single-level selective optimization performs well in both regimes, a system with optimization level 1 delivers the best performance in the total-time regime, while optimization level 2 delivers the best performance in steady-state. The multi-level model matches the best single-level in each regime, and there is no single-level strategy that can match the multi-level strategy's overall performance in both regimes.

Fig. 20 in the Appendix A presents the detailed data for each individual benchmark, which Fig. 8 summarizes. In the total-time regime, the performance of the

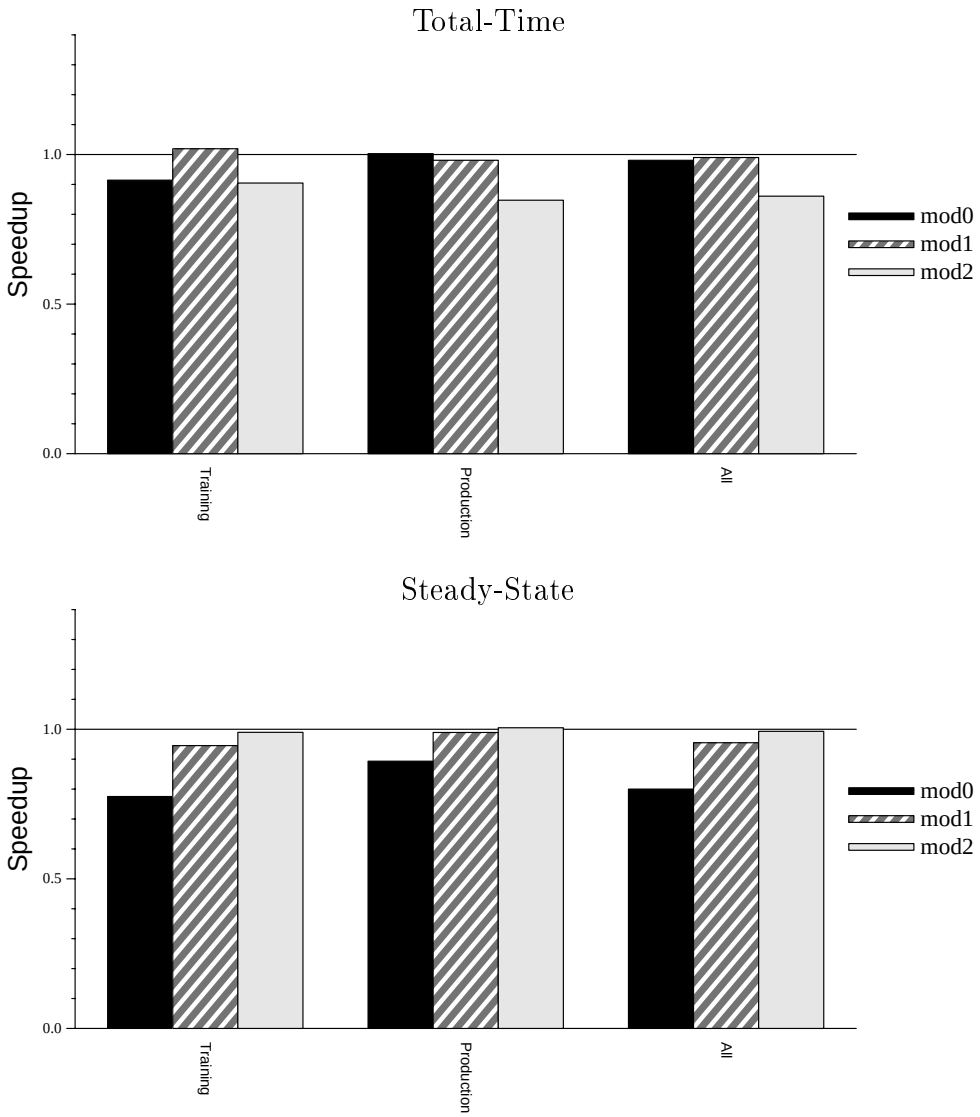


Fig. 8. Summary performance comparison between single-level models and multi-level model normalized to the multi-level model. Results for individual benchmark are presented in Appendix A, Figure 20.

multi-level model with respect to a single-level model ranges from -6% to 31%, -12% to 21%, -9% to 44%, for optimization levels 0, 1, and 2, respectively. In the steady-state regime, the performance of the multi-level model with respect to a single-level model ranges from -4% to 53%, -2% to 27%, -1% to 5%, for levels 0, 1, and 2, respectively.

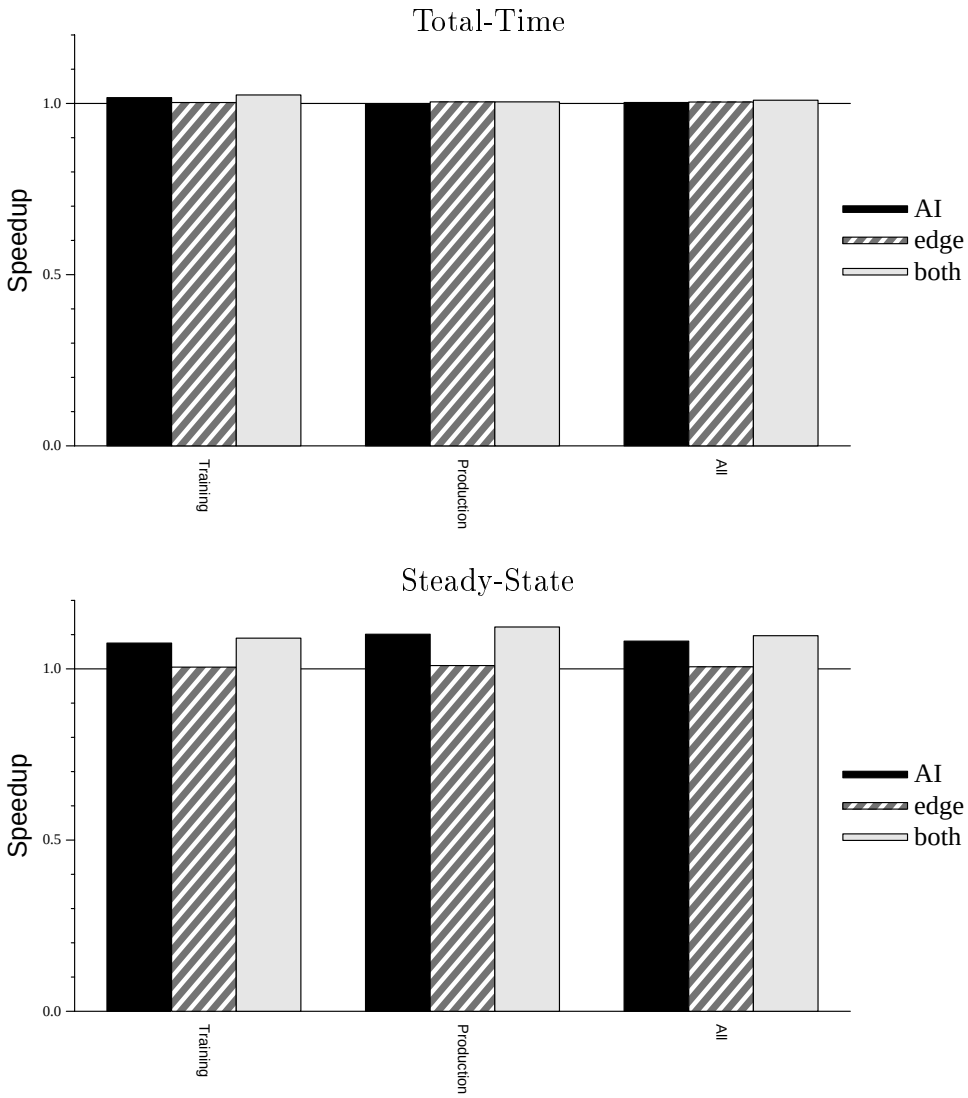


Fig. 9. Summary performance comparison between feedback-directed optimizations and the multi-level model normalized to multi-level. Results for individual benchmark are presented in Appendix A, Figure 21.

6.7 Feedback-directed Optimization

This section evaluates feedback-directed optimization. In particular, this section addresses the question: *how beneficial are feedback-directed optimizations?*

As described in Sections 4.4 and 4.6, Jikes RVM includes two types of feedback-directed optimizations: adaptive inlining and optimizations driven by intraprocedural control-flow graph edge profiles. Fig. 9 evaluates the performance of these optimizations. The results are normalized to multi-level selective optimization with-

out FDO (described in the previous subsection).

The results shows that in the total-time regime, on average feedback-directed optimizations have little impact, improving performance overall by less than 0.5%. This is not surprising because in the total-time regime, the programs do not run long enough for the system to collect and act on online profile data. However, feedback-directed optimizations entails extra overhead to collect profile data, so enabling feedback-directed optimization has the potential to degrade performance, rather than improve it. Although the feedback-directed optimizations do not improve average performance in the total-time regime, the results demonstrate that performance was not degraded either.

Fig. 9 also evaluates performance in the steady-state regime. These benchmarks run long enough that the benefits of feedback-directed optimization become evident. Adaptive inlining delivers more overall performance (8.1%) than edge-driven optimizations (0.6%). However, there appears to be some synergy when combining the two approaches as both adaptive inlining and edge-driven optimizations increase overall performance by 9.7%.

Fig. 21 in the Appendix A presents the detailed data for each individual benchmark, which Fig. 9 summarizes. In the total-time regime, the performance of feedback-directed optimizations with respect to the multi-level model ranges from -1.4% to 13.6%. In the steady-state regime, the performance of feedback-directed optimizations with respect to the multi-level model ranges from 1.0 to 20.0%.

6.8 Implementation overhead

As described in Section 3, our adaptive system architecture uses multiple asynchronous communicating threads, which process a fair volume of profile data. This subsection evaluates the overhead in this architecture by examining where a running application spends its time.

Fig. 10 characterizes the overhead of the system for the total-time and steady-state regimes. Table VIII in Appendix A presents the same data as Fig. 10 in table form. For each benchmark we report the time spent in various threads of a multi-level model configuration with feedback-directed optimizations (adaptive inlining and edge counts) enabled. We partition the execution time into four categories: application, garbage collection, recompilation, and AOS. The last category is the time spent in all organizers and the controller threads, i.e., it is the overhead in profiling the application looking for methods to recompile and the decision-making process of deciding which methods to compile. The actual recompilation time is in the third category, recompilation.

Because some benchmarks are typically run with a harness that forces a garbage collection, thus, inflating the overhead value for this component, we record the overhead timings once the application begins and ends, i.e., we do not count the harness time. This is achieved by modifying the harness to call special VM methods to signify the beginning and end of the application and an iteration of application for the steady-state regimes. The steady-state regimes record the execution for all iterations (unlike the performance results, which discarded the first 5 minutes of execution).

The results show that the time spent managing the adaptive optimization system is minimal; the AOS threads consume 0.42% and 0.13% of total cycles in total-

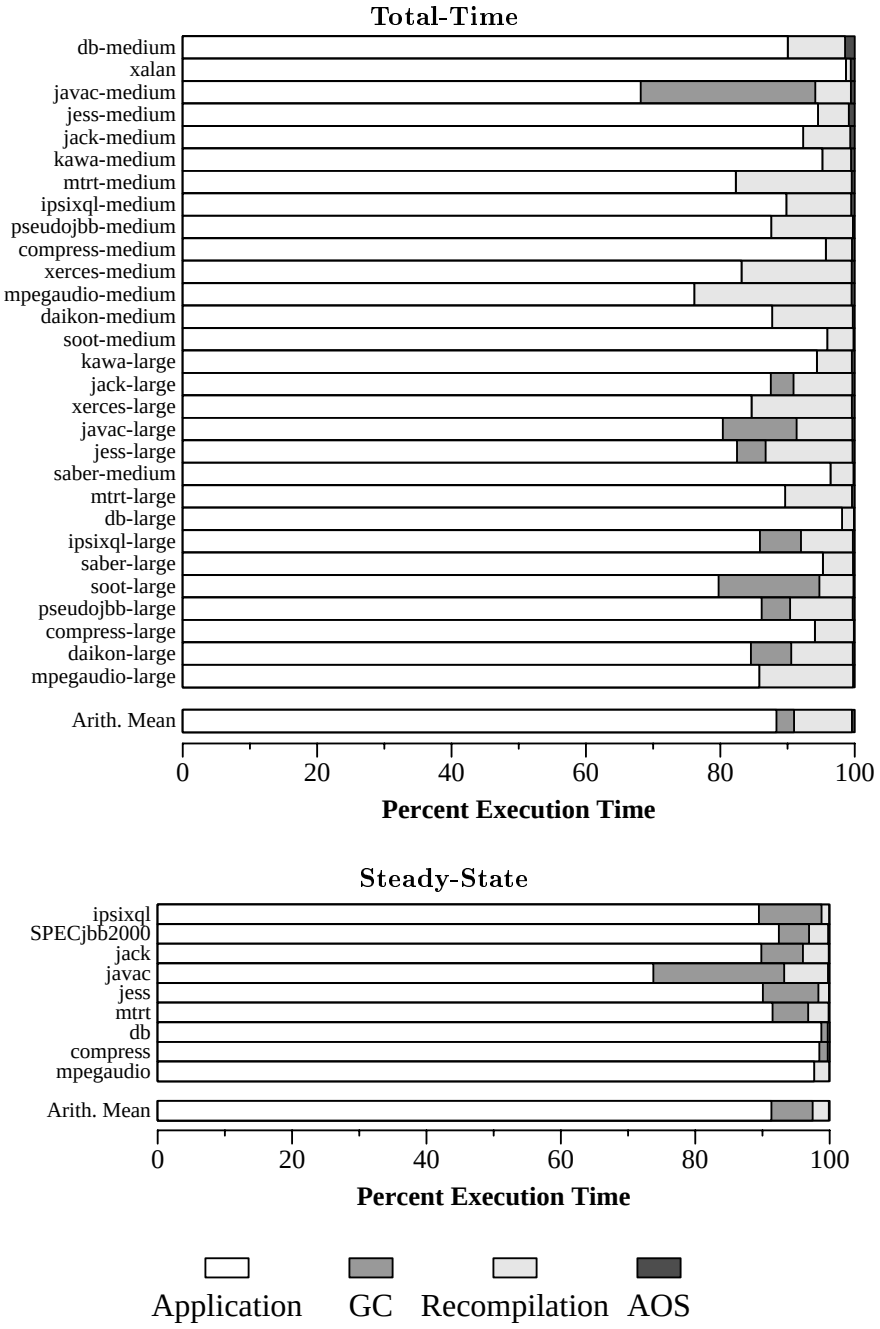


Fig. 10. Breakdown of time spent in various Jikes RVM threads.

time and steady-state regimes, respectively. In the total-time regime, the system spends 8.56% of the cycles recompiling, but only 6.14% in the steady-state. As a comparison, 3.28% and 6.14% of the cycles are spent in garbage collection in the total-time and steady-state regimes.

In summary, the overhead of the AOS threads is less than 0.5%, on average, fulfilling the design requirement for low overhead mentioned earlier in Section 2.

6.9 Recompilation activity

Finally, we show some data to qualitatively illustrate the behavior of the system. Fig. 11 shows the percentage of methods that are recompiled by the adaptive system in the total-time and steady-state experiments. The left graphs give the percentage of methods that are never optimized and optimized, respectively for the total-time and steady-state regimes. For the total-time regime, the system recompiled between 0.19–33.02% of all methods, with an average of 7.65%. For the steady-state regime, the system recompiled between 5.43–49.79% of all methods, with an average of 23.58%. This latter average corroborates the well-known intuition that the programs spend most of their time in a small subset of their methods [Knuth 1970]; the adaptive system tracks this behavior.

The right two figures of Fig. 11 further explore the methods that are optimized. For the methods that are optimized, it shows the breakdown of the final optimization level chosen for each method. In the total-time regime, the system recompiles the vast majority (95.03%) of methods at optimization level 0 or 1. In the longer-running steady-state programs, 27.14% of methods reach the highest optimization level, 2.

Figure 12 presents recompilation activity when all seven SPECjvm98 benchmarks run in a single VM instance. Three histograms show different orders of the benchmarks. The top graph lists the benchmarks in the numerical order (_201_compress ... _228_jack). The next graph lists the benchmarks in reverse order. The final graph list is the same as the first graph, but with _213_javac first.

The x-axis represents time broken into buckets of equal size. The height of each bar represents the number of methods recompiled during that time unit. The shading of the bars shows the optimization level at which the method was compiled. The tick marks below the x-axis show when each benchmark started and finished executing.

The table in Figure 12 summarizes final optimization level of each method. The first number represents the percent of all compiled methods, while the number in parenthesis is the absolute number of compilations.

These figures confirm that most of the recompilation activity occurs near the beginning of each benchmark. For benchmarks like db, which have a small number of very hot methods, a small amount of compilation occurs early in the execution, and no additional compilation activity occurs for the remainder of the benchmark. However, for programs like javac, which spreads its execution over a large number of methods, recompilation activity can be seen throughout the programs entire execution.

These graphs also confirm that changing the order the benchmarks are run within

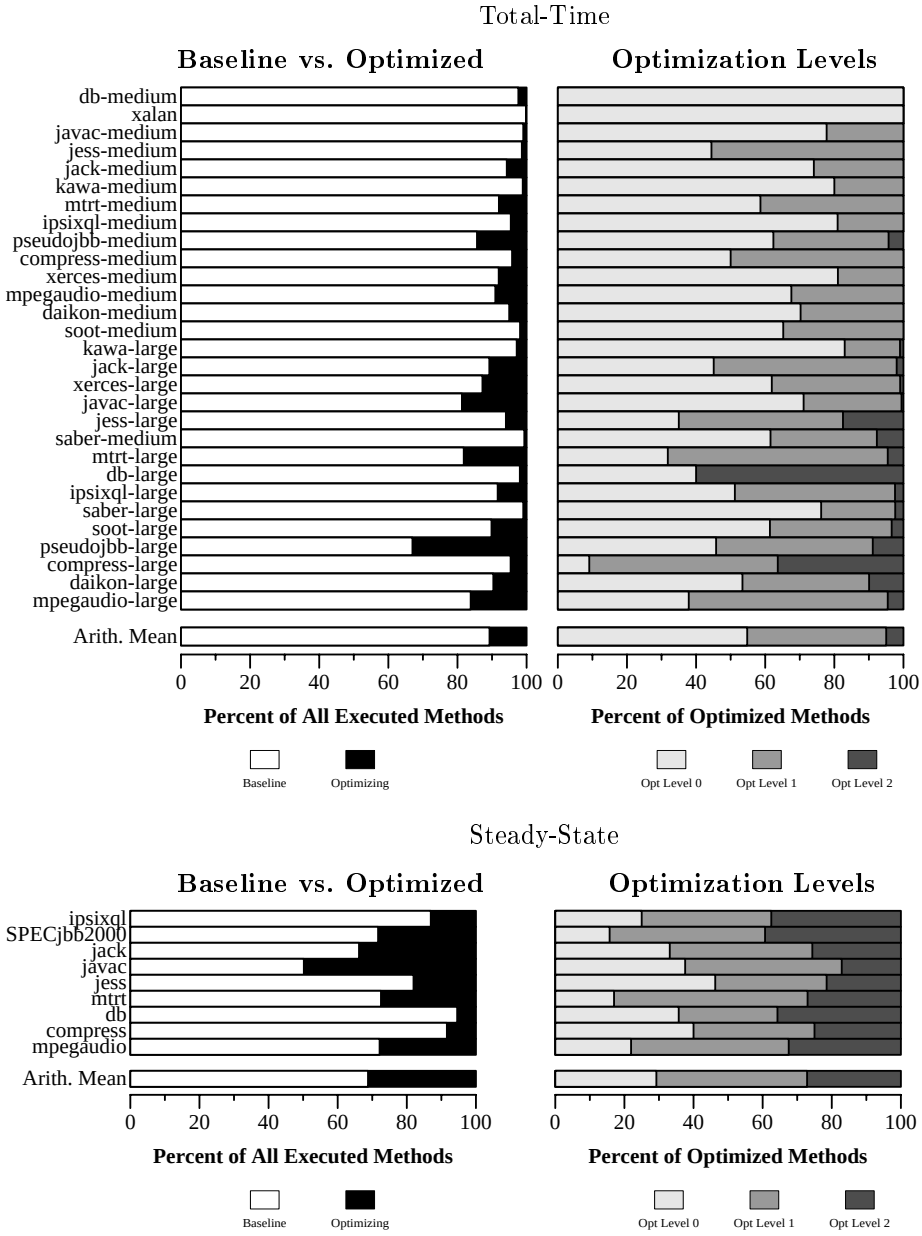
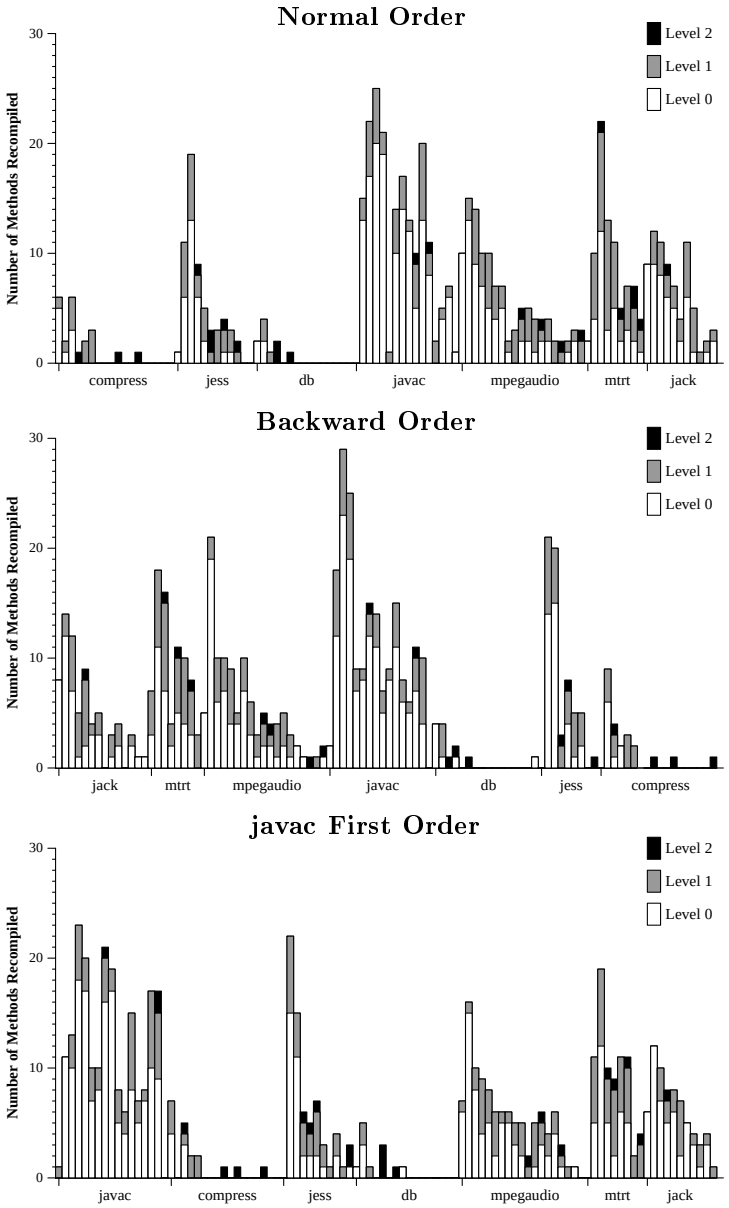


Fig. 11. The left graphs show the percent of methods that were compiled with the only baseline compiler versus those that were eventually compiled with the optimizing compiler. The graphs on the right breakdown show a breakdown of the final optimization level for methods that were compiled with the optimizing compiler.



Benchmark	Baseline	Total Opt	Opt 0	Opt 1	Opt 2
normal	81.46 (1731)	18.54 (394)	9.79 (208)	7.67 (163)	1.08 (23)
backward	81.73 (1736)	18.27 (388)	9.70 (206)	7.63 (162)	0.94 (20)
javacFirst	80.71 (1715)	19.29 (410)	10.78 (229)	7.39 (157)	1.13 (24)
Arith. Mean	81.30 (1727)	18.70 (397)	10.09 (214)	7.56 (160)	1.05 (22)

Fig. 12. Recompilation behavior over time for all SPECjvm98 benchmarks in one VM instance. The first three charts graphically report the recompilation behavior, while the table presents the raw data.

the VM does not have much effect on the compilation activity of each benchmark.[15] This result is desirable, and is not surprising because our current model makes each compilation decision based solely on the number of samples of the method; it does not take into account how much time has passed since the VM began execution. This is an improvement over a previous version of the system reported in [Arnold et al. 2000a].

7. DISCUSSION

Building, improving and maintaining the Jikes RVM adaptive system has been a multi-year, multi-person project. In this section we discuss some of the lessons we have learned in the process.

7.1 Implementation Details Matter

Although the adaptive optimization system rests on a theoretically solid foundation, various "implementation details" still play an important role in determining overall performance. This section discusses some ways in which relatively small implementation choices impacted overall system design and performance in significant ways.

7.1.1 *Finding the Right Low-Overhead Solution.* When we initially built the adaptive system, some component implementations were less mature than others. Occasionally, this immaturity manifested itself in the form of overly simplistic, poor-performing implementations of basic operations and data structures. In addition to direct performance impact, these poor implementations can lead to contortions in other system components that attempt to compensate for apparent "inherent overhead" of basic operations. Instead, the basic operations simply should have been implemented in a more efficient manner from the start.

For example, one of the key operations in selective optimization is identifying methods in which the application spends significant time (finding the "hot spots"). The system uses timer-based sampling; each time a method was sampled, the system increments the method's counter. Periodically, an organizer would scan all of the counters and report methods whose counters exceeded threshold values to the controller. The initial implementation was biased toward efficient incrementing of the counters. It maintained an array of counters, indexed by method id; the increment operation was a simple array load, increment, and store. However, finding the "interesting" counters required the organizer to scan the entire counter array (which contained tens of thousands of entries). In this implementation, identifying frequently executing methods imposed an overhead of approximately 1% [Arnold et al. 2000b], virtually all resulting from the periodic scans. As a result, we spent significant effort tuning the interval at which the organizer scanned the counter array and devising adaptive schemes that increased/decreased this interval based on ad hoc heuristics. We eventually abandoned this implementation and switched to one in which samples accumulate in a buffer. When the buffer is full, the organizer updates a data structure in which all non-zero counters are kept. All methods

[15] These benchmarks do share some common library routines, but the application code for each benchmark is independent.

that were sampled in the previous window (i.e., that were in the buffer) are reported to the controller. With this new implementation, even with a small (and non-adaptive) buffer size of 20 samples, method sample organizer overhead is only 0.04%.

Hazelwood and Grove implemented context-sensitive feedback directed inlining in Jikes RVM [Hazelwood and Grove 2003]. In the process they discovered that the hash function used by the call graph data structure was needlessly creating a large number of temporary objects as it computed the hashcode. By fixing this single operation, they reduced the overhead of the Adaptive Inlining Organizer by a factor of 50. This qualitatively changed the research results since the profiling technique then became quite practical to apply online.

The current Jikes RVM adaptive system implementation is in many ways simpler than that described in [Arnold et al. 2000b]. By reducing overhead and paying attention to implementation details, we were able to remove complex feedback loops that were intended to reduce profiling and decision-making overhead. The current system is more effective and imposes significantly less overhead. In [Arnold et al. 2000b] we reported adaptive system overhead (not including recompilation) of 3.6% on short-running programs and 3.0% in steady-state. In Jikes RVM version 2.3.3, we see overheads of 0.42% and 0.13%, respectively. Since basic overhead is now much less of a concern, we can instead focus on improving the accuracy of the profiling information and the sophistication of the controller's analytic model.

7.1.2 *Accounting for Optimization.* As the adaptive optimization system recompiles methods, the system must account for how the gathering profile data reflects the state of the system over a time period that contains both the old and new code. Consider in particular the method samples that are used to drive selective optimization. Initially, we did not consider the issue of recompilation, and set the initial counter for all versions of compiled methods to a default value of 0. For frequently executed methods, this introduced an undesirable delay between the initial optimization of a method at a moderate optimization level (O0 or O1) and subsequent re-optimizations at even higher optimization levels. We eventually determined that the correct approach (both theoretically and empirically) was to transfer the samples from the old compiled version of the method to the new one, scaling them down by the expected speedup. While in retrospect this decision is obviously correct, it took a while to identify this issue which manifested as a measurable, but subtle, drain on overall system performance.

7.1.3 *Adaptive Compiler DNA.* As described in Section 5.1, the controller's analytic model depends strongly on measurements of compilation rate and the expected benefits from optimization (the "compiler DNA"). We obtain these values by performing a series of experiments[16] to determine average compilation rate and speedup for each optimization level. These values need to be gathered on each of Jikes RVM's platforms, and are usually recomputed before each major release of the system. However, for a given release of Jikes RVM on a given platform, a single set of values will be used on machines of varying clock speeds and overall

[16]The scripts used to drive these experiments and compute the DNA values are included in the Jikes RVM open source release in `rvm/src/tools/computeCompilerDNA`.

performance. Not surprisingly, compilation rate needs to be scaled to account for these differences or overall system performance can degrade. Therefore, the adaptive system automatically times all baseline compilations and maintains a dynamic estimate of the baseline compilation rate. This value is then used to scale up or down the compilation rates encoded in the DNA value to match compilation time estimates to the actual hardware on which Jikes RVM is running.

7.1.4 *Yieldpoint Placement.* Initially, Jikes RVM placed yieldpoints in method prologues and on loop backedges. This placement was deemed sufficient for the purposes of quasi-preemptive thread scheduling (the original reasons for compiler-generated yieldpoints). Once the adaptive system started to use yieldpoints for timer-based sampling we discovered that this placement was not sufficient for accurately determining which methods the application was spending time. In particular, this placement created a blind spot in large, loop-free leaf methods. In some programs (`mpegaudio` was a notably extreme example), significant execution time is spent in such methods, and they were never selected for optimization. To fix this problem, we added epilogue yieldpoints to Jikes RVM.

A related issue is the tension between removing "useless" yieldpoints in optimized code and retaining sufficient yieldpoints to ensure accurate profiling. This tradeoff is particularly severe in "trivial" methods where the presence of a yieldpoint prevents the optimizing compiler from eliding the normal prologue/epilogue sequence. For example, a trivial getter method that simply returns an instance field is normally compiled by the optimizing compiler into two instructions that take 6 bytes of IA32 machine code. If we force a yieldpoint in this method, then the full prologue/epilogue sequence must be generated and the method will contain 15 executed IA32 instructions (plus the off branches of the yieldpoints) and consume 64 bytes of machine code. As a result, the system currently drops yieldpoints from single basic block leaf methods. This is almost always appropriate as methods that are this simple will virtually never benefit from high levels of optimization. However, it introduces a blind-spot in the profile-derived dynamic call graph. If this trivial method is frequently invoked via a virtual call, by eliding its yieldpoint we preclude the possibility of profile-directed inlining of this callee into its caller. An alternative approach would be to implement a shrink-wrapping optimization that would only execute the normal prologue/epilogue sequence when the yieldpoint was actually taken. This would mitigate the runtime costs of inserting yieldpoints in small methods, but not the increase in code space.

7.2 Comprehensive Performance Evaluation is Critical

As the Jikes RVM adaptive system has improved over the years, so has our methodology for evaluating it. We have expanded the benchmark suite with more realistic benchmarks, implemented more sophisticated strategies to compare against, and automated the process of collecting and reporting performance data to allow more frequent and robust comparisons.

Although we always knew that comprehensive performance evaluation is a good practice, we substantially underestimated its importance. We are now convinced that it is a basic requirement for achieving even moderate levels of robustness. Every time we enhanced our performance evaluation methodology, we uncovered new bugs

in the system; this happened repeatedly, regardless of our level of confidence in the previous system.

For example, in our original evaluation of the system [Arnold et al. 2000b] we did not compare our sample-based model against an alternative strategy, such as method counters. After doing so, we discovered a small number of benchmarks for which the model performed poorly relative to the counter-based strategy; after investigating, we discovered and fixed several performance related bugs that previously went undetected.

The reality is that the complexity of virtual machines continues to increase, and subtle interactions between subcomponents makes them particularly hard to model and predict. Thorough correctness and performance testing is a necessary component to achieve even a moderate level of confidence in the system.

7.3 Coping with Non-Determinism

The adaptive optimization system (AOS) introduces an additional level of non-determinism in two ways. As Section 5.2 stated, the AOS is implemented as five Java threads that all compete for the same system resources as the running application. In general, time-based scheduling of threads is non-deterministic, because of other system activity that also compete for resources, such as the operating system's daemon processes. Therefore, for different runs of the same application, the order and frequency that methods are sampled may differ, because many samples are taken at thread switch time. Second, because recompilation decisions are driven by time-based samples, the order that methods are recompiled also differs from run to run. However, the order that methods are optimized may also affect the order and frequency that methods are sampled: the number of times a method is sampled is expected to decrease when the method is optimized because it runs faster and executes for a smaller portion of the total execution time.

This additional non-determinism makes debugging and benchmarking more difficult. If a bug depends on the order that methods execute, and the order changes because of non-determinism, then reproducing the bug becomes difficult. To deal with this non-determinism, we invested significant effort into logging mechanisms and replay with offline-profiles. Debugging requires either pouring through the logs, or replaying the AOS's decisions to deterministically reproduce the execution.

Drawing conclusions about the performance between different runs of an application is difficult, because the optimization level of a method and when that method is optimized differs from run to run. In response, many of our users have adopted a "pseudo-adaptive" approach to eliminate this non-determinism. After executing an application for a fixed number of runs and recording the final optimization level of each method, the method's minimum optimization for a majority of runs is calculated. This calculated optimization level is then used by an oracle to determine the optimization level that the method is compiled at the first time the method is invoked, fixing the time when a method is optimized and fixing the level of optimization. The pseudo-adaptive approach increases the level of confidence of the conclusions that are drawn about the performance between different runs of an application.

Nevertheless, non-determinism may prevent pathological behavior because of the randomness that it introduces. For example, a system that uses counters can intro-

duce a long compilation pause time if a set of methods all exceed the recompilation counter threshold around the same time. Using a time-based sampling approach can eliminate these long compilation pause times.

7.4 Continuous Recompilation

Our empirical assessment demonstrated that continuous recompilation is a necessity. Section 6.6 demonstrated that any fixed strategy that initially compiles all the methods at a particular optimization level will underperform a model-based scheme that optimizes methods only when it is profitable. In addition, phase shifts may require that a method be recompiled to be optimized in the context of a new phase. Section 6.9 demonstrated that in fact applications that execute for a non-trivial amount of time are compiled (and recompiled) throughout the application's execution. This implies that even optimized code needs to be profiled. Earlier systems, such as Self [Hölzle and Ungar 1996], did not support the profiling of optimized code, and missed opportunities for improved performance. Of course, with continuous recompilation, compiled code can introduce a slow memory leak unless dead code is reclaimed as described in Section 5.1.3.

One issue to consider in adaptive optimization systems is what optimization actions should occur when a long-running application reaches a steady-state? Should the system continue to try to improve performance, or should it pronounce victory until a new phase of execution is detected? The basic recompilation model (Section 4.2) and its implementation (Section 5.5) address this issue nicely. It will only consider methods for recompilation that have been recently sampled. When a long-running application reaches a steady-state, all of its executing methods will likely have reached their highest optimization level. If these are the only methods that are sampled, the controller will correctly decide to take no recompilation action, despite the fact that their expected future time will continue to increase.

7.5 Modeling Compilation Cost and Expected Benefit

We have presented a principled approach to designing an adaptive system controller, which uses a cost-benefit model to estimate tradeoffs and select profitable actions. As we have argued, the model can accommodate a variety of extensions to model sophisticated adaptive policies, providing a straightforward policy by which new adaptive optimizations can be incorporated into a virtual machine. We believe this approach reduces complexity of implementation and tuning compared to the various ad hoc approaches previously presented.

The model-driven policy relies on estimates of costs and benefits of potential actions. The implementation as described relies on coarse, insensitive estimates of the costs and benefits of compiler optimization, based on offline measurements of system behavior. The current implementation predicts compilation time to be a linear function of method size, and the expected speedup from a particular optimization level to be the mean speedup observed from a number of experiments performed offline.

In theory, the model-driven controller might perform better with more accurate estimates of compilation time and expected speedup. However, obtaining more accurate estimates is a difficult open problem. We have attempted to predict speedup of individual methods based on static characteristics such as loop depth and various

measures of a method's instruction mix.[17] So far, all results have been negative; we have not been able to identify a cheap set of static method characteristics that demonstrate a significant correlation with expected speedup.

We believe the most challenging open problem in this area concerns how to accurately model the compilation costs and the indirect benefits of inlining. One possible approach for modeling the indirect benefits is inlining trials [Dean and Chambers 1994], an approach whereby the compiler monitors the effects of inlining as it compiles, and builds a database to predict future effects. It remains an open question whether empirical optimization approaches such as inlining trials and techniques as employed in the ADAPT system [Voss and Eigemann 2001] would improve overall performance compared to the current model-driven policies. We believe online empirical optimization policies hold great promise and represent fertile ground for future research.

8. RELATED WORK

The fundamental ideas of adaptive optimization have existed for several decades. This section focuses on the work most relevant to adaptive optimization in virtual machines. We refer the reader to Arnold et al. [2005] for a more comprehensive survey of this field and to Aycock [2003] for an in-depth review of the genesis of adaptive optimization and a survey of dynamic compilation techniques in domains other than virtual machines.

8.1 Influential Adaptive Optimization Systems

Several key systems stand out in the history of adaptive optimization technology. The most influential include the Adaptive Fortran system [Hansen 1974], the ParcPlace Smalltalk virtual machine [Deutsch and Schiffman 1984], the Self project [Chambers and Ungar 1991; Hölzle and Ungar 1994], and many Java virtual machines [Paleczny et al. 2001; Suganuma et al. 2000; Suganuma et al. 2001; Arnold et al. 2000b; Arnold et al. 2002; Cierniak et al. 2000; Cierniak et al. 2003; Adl-Tabatabai et al. 2003; BEA 2003].

Hansen's Adaptive Fortran (AF) system [Hansen 1974] provided the first in-depth exploration of issues in online adaptive optimization. His 1974 thesis describes many of the challenges facing adaptive optimizers with regard to selective optimization, including models and heuristics to drive recompilation, dealing with multiple optimization levels, and online profiling and control systems.

The AF compiler produced an intermediate form that could either be directly interpreted, or further optimized. The system would selectively apply optimizations to basic blocks or loop-like "segments" of code. The compiler associated a counter with each basic block, which triggered selective compilation of each code sequence according to empirically-derived heuristics. After empirical tuning, Hansen settled on a system with three levels of optimization in addition to direct interpretation. The compiler applied selective optimization to portions of an individual method. The compiler did not perform inlining, and thus could not selectively optimize across procedure boundaries. The experimental results showed that AF's adaptive

[17] These unpublished experiments were joint work with Barbara Ryder and Jeanne Ferrante.

optimization provided better overall performance than JIT-only strategies with any single optimization level.

The ParcPlace Smalltalk virtual machine [Deutsch and Schiffman 1984] was the first modern virtual machine. The system introduced many of the core concepts used today, including a full-fledged dynamic compiler, inline caches to optimize polymorphic dispatch, and native code caches. The system architecture supported three mutually exclusive execution strategies: an interpreter, a simple non-optimizing compiler, and an optimizing compiler. The execution strategy was chosen prior to execution. The Smalltalk system also introduced a code cache to deal with managing space for optimized code. When space was tight, their system would discard native code and regenerate it as needed. Subsequently, code caches have been employed in other contexts for platforms in which code space is scarce (e.g. [Bala et al. 2000]).

The Self project [Chambers and Ungar 1991; Hölzle and Ungar 1994] developed more advanced techniques, such as polymorphic inline caches, on-stack replacement, dynamic deoptimization, advanced selective compilation with multiple compilers, profile-directed inlining integrated with adaptive recompilation, and type prediction and splitting. The introduction of the Java programming language [Gosling et al. 1996] resulted in significant commercial resources being dedicated to virtual machine implementations [Paleczny et al. 2001; Suganuma et al. 2000; Suganuma et al. 2001; Cierniak et al. 2000; Adl-Tabatabai et al. 2003; BEA 2003]. The introduction of the Common Language Runtime [Meijer and Gough] further increased the importance of creating high-performing virtual machines. Researchers have also explored advanced adaptive optimization ideas in other languages [Kistler and Franz 2003].

8.2 Optimization Strategies

The ParcPlace Smalltalk virtual machine [Deutsch and Schiffman 1984] was the first to introduce a dynamic compiler to improve program performance. The system used a straightforward runtime compilation policy: compile each method the first time the method is called. This system did not recompile methods. The Self-91 system [Chambers and Ungar 1991] and several Java system [Adl-Tabatabai et al. 1998; Krall 1998; Yang et al. 1999] also employed this technique.

More recent virtual machines exploited selective optimization strategies to better balance performance and overhead. Many systems would initially execute all methods with a quick execution engine (an interpreter or non-optimizing compiler) and then monitor the execution to find candidates for compilation by an optimizing compiler. Several VMs have adopted this approach, such as Self-93 [Hölzle and Ungar 1994], the HotSpot server JVM [Paleczny et al. 2001], the IBM DK for Java Technology, version 1.3.1 [Suganuma et al. 2000; Suganuma et al. 2001], the ORP/JUDO system [Cierniak et al. 2000], the Jikes RVM/Jalapeno system [Arnold et al. 2000b; Arnold et al. 2002; Jikes RVM], the JRocket JVM [BEA 2003], and the J9 VM [Grcevski et al. 2004]. The IBM DK for Java Technology, HotSpot, and the J9 VM use an interpreter for initial execution. Self-93, Jikes RVM, ORP, and JRocket use a non-optimizing compiler for the initial execution engine.

Two important components of these systems are the online profiling system, and the policy for selecting candidates for optimization. Most VMs merge these

decisions into one component, often relying on ad hoc solutions for making these decisions. In contrast, our architecture decouples these decisions by assigning their responsibility to separate components.

Because these components operate online during program execution, the system must minimize their overhead to maximize performance. Two mechanisms have emerged for obtaining a low-overhead, coarse-grained data to drive selective optimization: *counters* [Hölzle and Ungar 1994; Paleczny et al. 2001; Cierniak et al. 2000; Cierniak et al. 2003] and *sampling* [BEA 2003; Arnold et al. 2000b]. The counter mechanism associates a method-specific counter with each method. The counter counts method invocations and, optionally, loop iterations in the same or different method-specific counter. Because this introduces overhead, this mechanism is not employed in optimized code. The sampling mechanism periodically interrupts the system and records the method (or methods) on the top of the call stack. Often, an external clock triggers sampling, which allows significantly lower overheard than incrementing a counter on each method invocation. However, using an external clock as the trigger introduces non-determinism, which can complicate system debugging. Some systems [Suganuma et al. 2001] employ a hybrid approach, using counters for initial compilation and sampling for further identifying recompilation candidates. As mentioned in Section 5, our implementation uses a timer-trigger call stack sampling approach. This profiling approach enables a straightforward conversion into estimates of time spent in methods, a requirement of the cost-benefit model that this article advocates.

Based on the profile data, the decision-making component selects methods or code sequences for optimization. Most systems [Hölzle and Ungar 1996; Cierniak et al. 2000; Suganuma et al. 2000; Paleczny et al. 2001; Suganuma et al. 2001] employ a fixed threshold strategy, where a value (method counter or sample counter) exceeding a fixed threshold triggers the compilation of one or more methods. Choosing an appropriate threshold value for all benchmarks is quite challenging. This article advocates using a more theoretically-grounded policy (a cost-benefit model) for selecting which methods to optimize and at what levels of optimization. This approach allows lower optimization levels to be skipped if a method is deemed to be particularly important. Other schemes require methods to proceed sequentially through optimization levels, if they are recompiled.

Plezbert and Cytron [1997b] considered several online strategies for selective optimization. Their "Crossover" strategy is an online two-competitive algorithm that guarantees that the online cost will be at most twice the offline (or optimal) cost. They presented a simulation-based study on C compilation with a file-based granularity. They compared "JIT-only" and selective optimization approaches, as well as considering a background compilation thread that uses a spare processor to continuously compile. The study simulated a number of scenarios and calculated break-even points that indicate how long a program must run to make a particular recompilation heuristic profitable.

Kistler and Franz [2003] performed a similar analysis in a more realistic study using a virtual machine for Oberon. They considered a more sophisticated online decision procedure for driving compilation, in which each compiler phase estimates its own speedup based on a rich set of profile data. Kistler and Franz performed an extensive study of break-even points based on this model, but did not implement

the model-driven online algorithm in the virtual machine.

Other studies have evaluated the viability of selective optimization. Arnold et al. [2000] quantify selective optimization's potential for outperforming a fixed JIT strategy using previously gathered profile information. Radhakrishnan et al. [2000] used the Kaffe Virtual Machine to establish the maximum performance improvement possible by interpreting, rather than compiling, cold methods.

Detlefs and Agesen [1999a] studied the tradeoff between an interpreter and two compilers. Using an oracle to determine the longest-running methods, they found that a combination of the fast JIT and judicious use of the slow JIT on the longest-running methods provided the best results.

Arnold et al. [2000a] study the sensitivity of an earlier version of the Jikes RVM cost-benefit model to changes in the compiler DNA values, concluding that minor changes to DNA values do not impact performance.

Krintz and Calder [2001] present an annotation framework that can be used to reduce the cost of dynamic compilation by communicating offline analysis information to be used to guide a dynamic compiler. In latter work, Krintz [2003] shows how to combine offline profiles collected from a previous run with online profile data to improve performance. Sandya [2004] uses a model to combine offline and online profile data to drive selective optimization.

Serrano et al. [2000] describe the *QuickSilver* system, which attempts to avoid the costs of dynamic compilation. Using their approach, an application is pre-executed and the compiled images are written to a file before the VM terminates. When a compiled method is needed during subsequent executions of the application, the dynamic compiler checks to see if a precompiled image of the method exists, and if so relocates and links the code into the new VM environment. If no image exists, the dynamic compiler compiles the method.

8.3 Feedback-Directed Inlining

The Self implementations introduced progressively sophisticated optimizations to predict types, and inline and split code based on static type estimates. Self-93 [Hölzle and Ungar 1996] augmented these techniques with *type feedback*, where the VM would provide the runtime compiler with a profile of receiver types collected from the current run. The Self compiler used this information to choose inlining candidates and to guide the transformations to deal with inlined dynamic dispatch. The reported results show significant speedup (1.7x improvement) from using type feedback, and show that the profile-directed approach results in significantly better code than a more sophisticated optimizer (Self-91) that relied on static type estimates.

The Self-93 adaptive optimization system incorporated inlining decisions into the recompilation policy, walking the call stack to find a suitable root method to recompile with inlining. The HotSpot JVM [Paleczny et al. 2001] adopted the Self-93 technique of driving recompilation policies based on inlining decisions. It performs guarded inlining when class hierarchy analysis or the profile indicated a single receiver type.

Dean and Chambers [1994] presented Inlining Trials, an approach to more systematically determine inlining decisions. In this work, the Self compiler would tentatively inline a call site, and monitor compiler transformations to quantify the

resultant effect on optimizations. The virtual machine maintained a history of inlining decisions and resultant effects, and would drive future inlining decisions based on the history. This approach could help guide more intelligent inlining decisions because the effect of inlining on optimizations is difficult to predict. Waddell and Dybvig [1997] report a similar approach for a dynamic Scheme compiler.

Several studies [Arnold et al. 2000b; Suganuma et al. 2002; Arnold et al. 2002; Adl-Tabatabai et al. 2003; Cierniak et al. 2000] report on fully automatic online profile-directed inlining for Java that improves performance by factors of approximately 10-17%, as compared to comparable strategies that ignore profile data.

Arnold et al. [2002] augment the inlining scheme in Jikes RVM [Arnold et al. 2000b], which is similar to what was used in this article, by profiling hot methods to determine hot basic blocks within those methods. Inlining budgets for call sites in such hot blocks are increased.

Suganuma et al. [2002] explored online profile-directed inlining heuristics, relying on an approximation of the dynamic call graph collected by instrumenting hot target methods for short time periods. They concluded that for non-tiny methods, heuristics based solely on profile data outperformed strategies that also rely on static heuristics.

The StarJIT compiler [Adl-Tabatabai et al. 2003] uses call site frequency to augment inlining heuristics and improve guarded devirtualization decisions. Hazelwood and Grove [2003] explored more advanced inlining heuristics that consider both static and dynamic characteristics of the call stack.

8.4 Other related work

Dynamic binary optimizers (such as Dynamo [Bala et al. 2000]) share similar goals (improve performance) and constraints (profiling and optimization occur at runtime) with VM adaptive optimization systems, but operate on a lower-level program representation. Most of these systems typically employ a threshold-based strategy based on execution counters for profiling and recompilation solutions. There are also fully automated profiling systems [Hookway and Herdeg 1997; Zhang et al. 1997; Anderson et al. 1997] that use transparent low overhead profiling to improve performance of future executions.

Another area of research considers runtime optimizations that exploit invariant runtime values [Auslander et al. 1996; Grant et al. 1999; Marlet et al. 1999; Leone and Lee 1998; Consel and Noël 1996; Marlet et al. 1999; Poletto et al. 1997]. Because such values are not statically determinable, these systems provide optimization opportunities not available with static compilation. The main disadvantage of these techniques is that they rely on programmer directives to identify the regions of code to be optimized.

More details about online profile directed optimizations can be found in [Arnold et al. 2005].

9. CONCLUSIONS

We have presented an architecture and policy framework for online adaptive optimization in virtual machines. The architecture provides a clean separation of concerns that helps manage complexity as a virtual machine evolves to incorporate ever-more sophisticated optimizations. From our experience with Jikes RVM,

we conclude that the architecture succeeded by allowing us to extend the system with a variety of new adaptive technologies while maintaining robust performance characteristics.

The most fundamental difference between our approach and all previous work lies in the approach to adaptive optimization policy. All previous systems rely on ad hoc heuristics to guide these policy decisions. However, as complexity increases, the difficulty in tuning these heuristics increases. In current industry practice, VM vendors laboriously tune a variety of counter thresholds to meet performance goals. Hansen lamented the situation in 1974:

> Determining optimization counts heuristically has its limitations, for we found it hard to change an optimization count so only a portion of the performance curve is affected. Therefore, if any appreciable progress is to be made, a more theoretical basis for determining them must be developed. [Hansen 1974], p.112.

We have presented a controller policy framework based on an analytic model of costs and benefits, in an attempt to provide a more theoretical basis for adaptive optimization policy. The current instantiation performs well on the benchmarks we have considered, although many limitations and open issues remain. Nevertheless, we believe the model-based framework provides a solid foundation for building the next generation of adaptive virtual machines.

REFERENCES

Adl-Tabatabai, A.-R., Bharadwaj, J., Chen, D.-Y., Ghuloum, A., Menon, V., Murphy, B., Serrano, M., and Shpeisman, T. 2003. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal 7,* 1 (Feb.), 19–31.

Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V. M., and Stichnoth, J. M. 1998. Fast, effective code generation in a just-in-time Java compiler. *ACM SIGPLAN Notices 33,* 5 (May), 280–290. Published as part of the proceedings of PLDI'98.

Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. 2000. The Jalapeño virtual machine. *IBM Systems Journal 39,* 1 (Feb.), 211–238.

Alpern, B., Attanasio, C. R., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J. J., Hummel, S. F., Sheperd, J. C., and Mergen, M. 1999. Implementing Jalapeño in Java. *ACM SIGPLAN Notices 34,* 10 (Oct.), 314–324. Published as part of the proceedings of OOPSLA'99.

Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., tak A. Leung, S., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. 1997. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems 15,* 4 (Nov.), 357–390.

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. 2000a. Adaptive optimization in the Jalapeño JVM: The controller's analytical model. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3).*

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. 2000b. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices 35,* 10 (Oct.), 47–65. Proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00).

ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P. F. 2005. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE 93,* 2. Special issue on Program Generation, Optimization, and Adaptation.

ARNOLD, M., HIND, M., AND RYDER, B. G. 2000. An empirical study of selective optimization. In *13th International Workshop on Languages and Compilers for Parallel Computing.* 49–67.

ARNOLD, M., HIND, M., AND RYDER, B. G. 2002. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices 37,* 11 (Nov.), 111–129. Proceedings of the 2002 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'02).

AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S. J., AND BERSHAD, B. N. 1996. Fast, effective dynamic compilation. *ACM SIGPLAN Notices 31,* 5 (May), 149–159. Published as part of the proceedings of PLDI'96.

AYCOCK, J. 2003. A brief history of just-in-time. *ACM Computing Surveys 35,* 2, 97–113.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices 35,* 5 (May), 1–12. Published as part of the proceedings of PLDI'00.

BEA. 2003. BEA weblogic JRockit: Java for the enterprise. Technical white paper available at http://www.bea.com.

BLACKBURN, S., CHENG, P., AND MCKINLEY, K. 2004. Myths and reality: The performance impact of garbage collection. In *ACM SIGMETRICS — Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems.* 25–36.

BURKE, M. G., CHOI, J.-D., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M. J., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 1999. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference.* 129–141.

CHAMBERS, C. AND UNGAR, D. 1991. Making pure object-oriented languages practical. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications.* 1–15.

CIERNIAK, M., ENG, M., GLEW, N., LEWIS, B., AND STICHNOTH, J. 2003. The open runtime platform: A flexible high-performance managed runtime environment. *Intel Technology Journal 7,* 1, 5–18.

CIERNIAK, M., LUEH, G.-Y., AND STICHNOTH, J. M. 2000. Practicing JUDO: Java under dynamic optimizations. *ACM SIGPLAN Notices 35,* 5 (May), 13–26. Published as part of the proceedings of PLDI'00.

colorado bench. Colorado Bench. http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench.

CONSEL, C. AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 145–156.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (Oct.), 451–490.

daikon. The Daikon dynamic invariant detector. http://pag.csail.mit.edu/daikon.

DEAN, J. AND CHAMBERS, C. 1994. Towards better inlining decisions using inlining trials. In *LISP and Functional Programming.* 273–282.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *9th European Conference on Object-Oriented Programming.* 77–101.

DETLEFS, D. AND AGESEN, O. 1999a. The case for multiple compilers. In *OOPSLA'99 VM Workshop: Simplicity, Performance and Portability in Virtual Machine Design.* 180–194.

DETLEFS, D. AND AGESEN, O. 1999b. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming.* 258–278.

DEUTSCH, L. P. AND SCHIFFMAN, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*. 297–302.

ECMA INTERNATIONAL. 2002. Standard ECMA-334, C# Language Specification. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

FINK, S., GROVE, D., AND HIND, M. 2002. The design and implementation of the Jikes RVM optimizing compiler. Tutorial Notes from OOPSLA'02. Available at www.ibm.com/developerworks/oss/jikesrvm/information/presentations.shtml.

FINK, S., KNOBE, K., AND SARKAR, V. 2000. Unified analysis of array and object references in strongly typed languages. In *Seventh International Static Analysis Symposium (2000)*.

FINK, S. J. AND QIAN, F. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization*. IEEE Computer Society, 241–252.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison Wesley.

GRANT, B., PHILIPOSE, M., MOCK, M., EGGERS, S. J., AND CHAMBERS, C. 1999. An evaluation of run-time optimizations. *ACM SIGPLAN Notices 34,* 5 (May), 293–304. Published as part of the proceedings of PLDI'99.

GRCEVSKI, N., KILSTRA, A., STOODLEY, K., STOODLEY, M., AND SUNDARESAN, V. 2004. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Usenix 3rd Virtual Machine Research and Technology Symposium (VM'04)*.

HANSEN, G. J. 1974. Adaptive systems for the dynamic run-time optimization of programs. Ph.D. thesis, Carnegie-Mellon University.

HAZELWOOD, K. AND GROVE, D. 2003. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization*. IEEE Computer Society, 253–264.

HÖLZLE, U. 1995. Adaptive optimization for Self: Reconciling high performance with exploratory programming. Ph.D. thesis, Stanford University.

HÖLZLE, U. AND UNGAR, D. 1994. A third generation SELF implementation: Reconciling responsiveness with performance. *ACM SIGPLAN Notices 29,* 10 (Oct.), 229–243.

HÖLZLE, U. AND UNGAR, D. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems 18,* 4 (July), 355–400.

HOOKWAY, R. J. AND HERDEG, M. A. 1997. DIGITAL FX!32: Combining emulation and binary translation. *Digital Technical Journal 9,* 1, 3–12.

Jikes RVM. Jikes Research Virtual Machine (RVM). http://www.ibm.com/developerworks/oss/jikesrvm.

kawa. Kawa, the Java-based Scheme system. http://www.gnu.org/software/kawa.

KEPHART, J. O. AND CHESS, D. M. 2003. The vision of autonomic computing. *IEEE Computer 36,* 1 (Jan.), 41–50.

KISTLER, T. AND FRANZ, M. 2003. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems 25,* 4 (July), 500–548.

KNUTH, D. E. 1970. An empirical study of FORTRAN programs. RC 3276.

KRALL, A. 1998. Efficient JavaVM Just-in-Time compilation. In *International Conference on Parallel Architectures and Compilation Techniques*, J.-L. Gaudiot, Ed. 205–212.

KRINTZ, C. 2003. Coupling on-line and off-line profile information to improve program performance. In *The Internation Symposium on Code Generatino and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization*. 69–78.

KRINTZ, C. AND CALDER, B. 2001. Using annotations to reduce dynamic optimization time. *ACM SIGPLAN Notices 36,* 5 (May), 156–167. Published as part of the proceedings of PLDI'01.

LEONE, M. AND LEE, P. 1998. Dynamic specialization in the Fabius system. *ACM Computing Surveys 30,* 3es (Sept.), 1–5. Article 23.

MARLET, R., CONSEL, C., AND BOINOT, P. 1999. Efficient incremental run-time specialization for free. *ACM SIGPLAN Notices 34,* 5 (May), 281–292. Published as part of the proceedings of PLDI'99.

MEIJER, E. AND GOUGH, J. Technical overview of the Common Language Runtime.

PALECZNY, M., VICK, C., AND CLICK, C. 2001. The Java Hotspot server compiler. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'01).* 1–12.

PETTIS, K. AND HANSEN, R. C. 1990. Profile guided code positioning. *ACM SIGPLAN Notices 25,* 6 (June), 16–27. Published as part of the proceedings of PLDI'90.

PLEZBERT, M. P. AND CYTRON, R. K. 1997a. Does 'just in time' = 'better late than never'? In *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, ACM, Ed. ACM Press, New York, NY, USA, 120–131.

PLEZBERT, M. P. AND CYTRON, R. K. 1997b. Does "just in time" = "better late than never"? In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 120–131.

POLETTO, M., ENGLER, D. R., AND KAASHOEK, M. F. 1997. tcc: A system for fast, flexible, and high-level dynamic code generation. *ACM SIGPLAN Notices 32,* 5 (May), 109–121. Published as part of the proceedings of PLDI'97.

POLETTO, M. AND SARKAR, V. 1999. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems 21,* 5 (Sept.), 895–913.

RADHAKRISHNAN, R., VIJAYKRISHNAN, N., JOHN, L. K., AND SIVASUBRAMANIAM, A. 2000. Architectural issues in Java runtime systems. In *Sixth International Symposium on High Performance Computer Architecture (HPCA-6).* Toulouse, France, 387–398.

REIMER, D., SCHONBERG, E., SRINIVAS, K., SRINIVASAN, H., ALPERN, B., JOHNSON, R., KERSHENBAUM, A., AND KOVED, L. 2004. Saber: Smart analysis based error reduction. In *International Symposium on Software Testing and Analysis.* 252–262.

SANDYA, S. M. 2004. Jazzing up JVMs with off-line profile data: does it pay? *ACM SIGPLAN Notices 39,* 8 (Aug.), 61–71.

SERRANO, M., BORDAWEKAR, R., MIDKIFF, S., AND GUPTA, M. 2000. Quicksilver: a quasi-static compiler for Java. *ACM SIGPLAN Notices 35,* 10 (Oct.), 66–82. Proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00).

soot. Soot, a Java Bytecode Analysis and Transformation Framework. http://www.sable.mcgill.ca/software/#soot.

STANDARD PERFORMANCE EVALUATION CORPORATION. SPECjbb2000 Java Business Benchmark. http://www.spec.org/jbb2000.

STANDARD PERFORMANCE EVALUATION CORPORATION. SPECjvm98 Benchmarks. http://www.spec.org/jvm98.

SUGANUMA, T., OGASAWARA, T., TAKEUCHI, M., YASUE, T., KAWAHITO, M., ISHIZAKI, K., KOMATSU, H., AND NAKATANI, T. 2000. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal 39,* 1 (Feb.), 175–193.

SUGANUMA, T., YASUE, T., KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. 2001. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices 36,* 11 (Nov.), 180–195. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

SUGANUMA, T., YASUE, T., AND NAKATANI, T. 2002. An empirical study of method in-lining for a Java just-in-time compiler. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'02).* 91–104.

VOSS, M. J. AND EIGEMANN, R. 2001. High-level adaptive program optimization with ADAPT. *ACM SIGPLAN Notices 36,* 7 (July), 93–102.

WADDELL, O. AND DYBVIG, R. K. 1997. Fast and effective procedure inlining. In *4th International Symposium on Static Analysis* (Paris, France).

WHALEY, J. 1999. Dynamic optimization through the use of automatic runtime specialization. M.S. thesis, Massachussetts Institute of Technology.

xalan. Xalan-Java XSLT processor. http://xml.apache.org/xalan-j.

xerces. Xerces2 Java Parser Readme. http://xml.apache.org/xerces2-j/index.html.

YANG, B.-S., MOON, S.-M., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. C., KIM, S., EBCIOGLU, K., AND ALTMAN, E. 1999. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. In *International Conference on Parallel Architectures and Compilation Techniques*. 128–138.

ZHANG, X., WANG, Z., GLOY, N., CHEN, J. B., AND SMITH, M. D. 1997. System support for automated profiling and optimization. In *16th Symposium on Operating Systems Principles*. Operating Systems Review, 31(5). 15–26.

A. ADDITIONAL DATA

This appendix presents additional figures that were mentioned, but not included in Section 6.

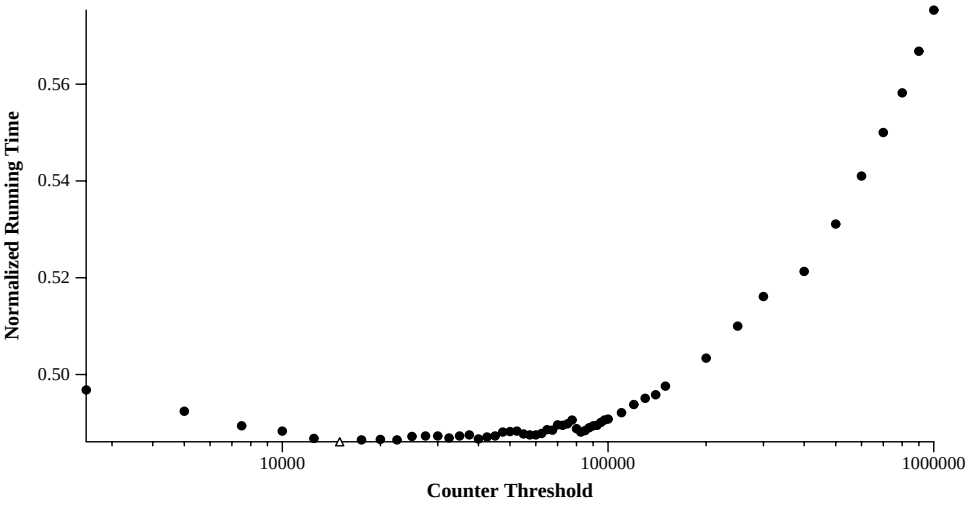


Fig. 13. Tuning results for single-level counter strategy using optimization level 0. The x-axis specifies the counter thresholds using a log scale and the y-axis gives the running time relative to baseline compiler. The best-performing counter value, 15,000, is shown with a triangle. This value is used in Section 6.5.

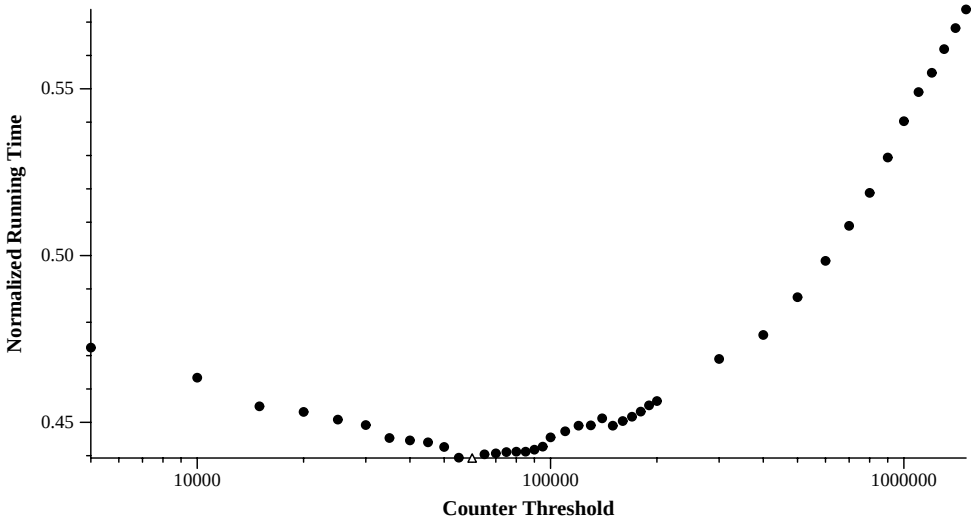


Fig. 14. Tuning results for single-level counter strategy using optimization level 1. The x-axis specifies the counter thresholds using a log scale and the y-axis gives the running time relative to baseline compiler. The best-performing counter value, 60,000, is shown with a triangle. This value is used in Section 6.5.

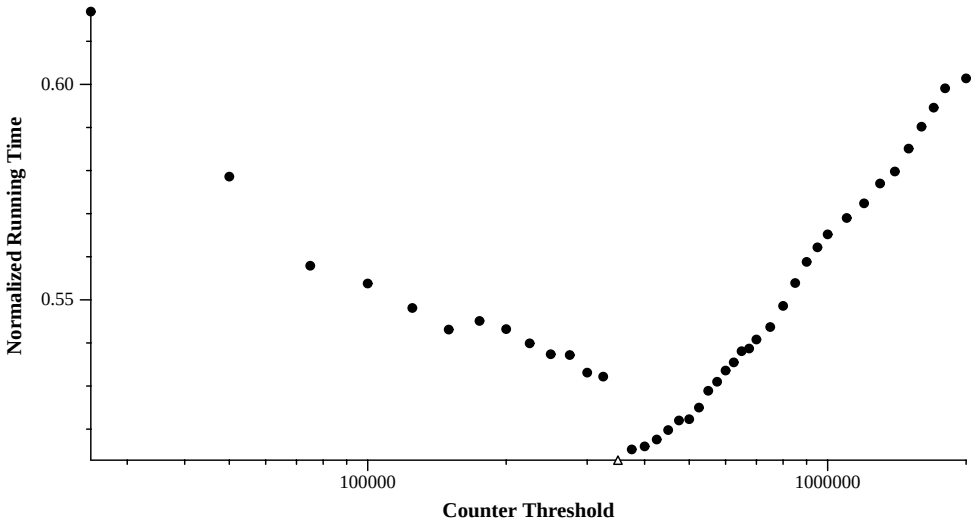


Fig. 15. Tuning results for single-level counter strategy using optimization level 2. The x-axis specifies the counter thresholds using a log scale and the y-axis gives the running time relative to baseline compiler. The best-performing counter value, 350,000, is shown with a triangle. This value is used in Section 6.5.

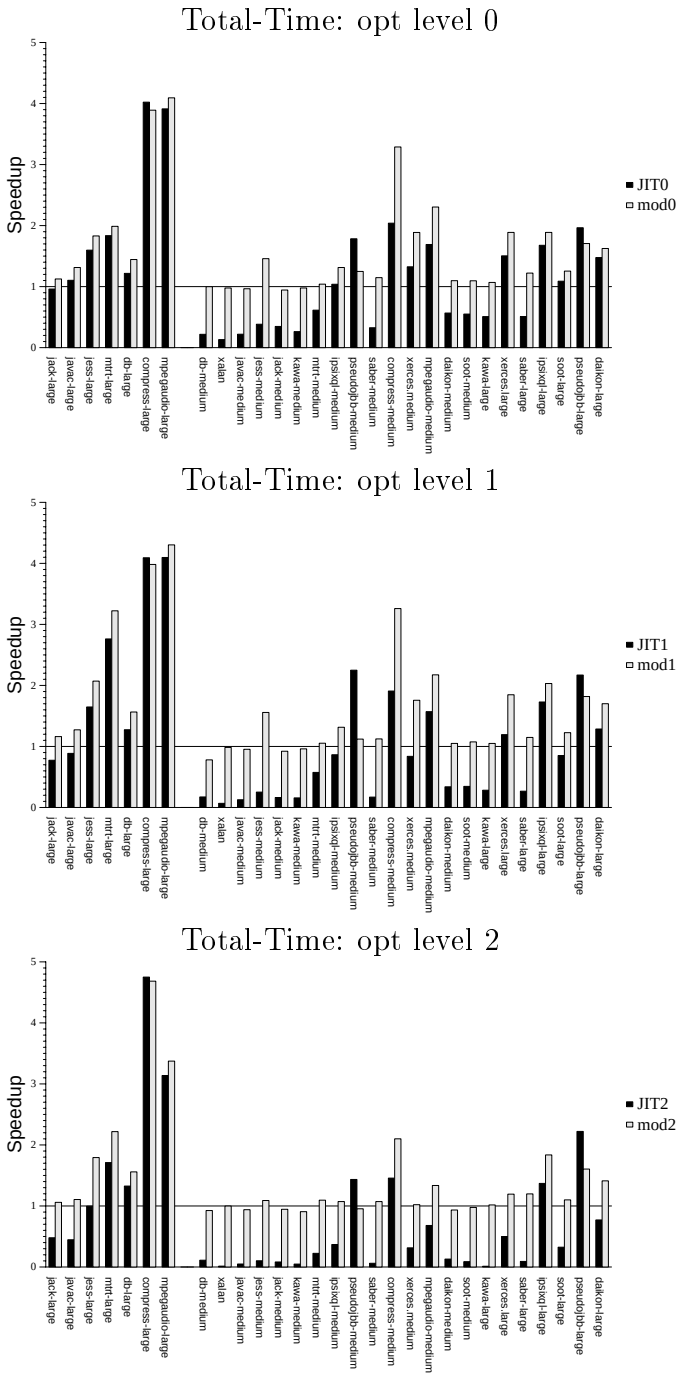


Fig. 16. Detailed total-time performance comparison between JIT and single-level models, normalized to baseline. The training set benchmarks are on the left, the production set is on the right. A summary of this data was presented in Figure 6.

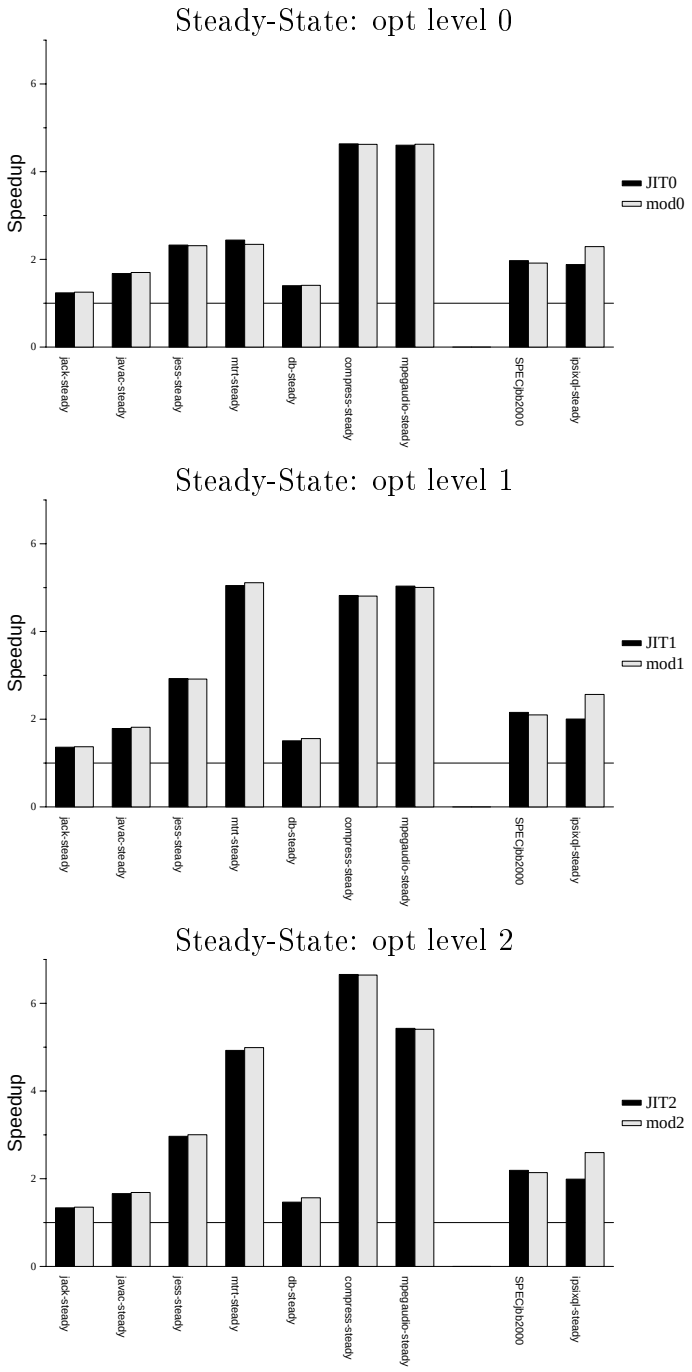


Fig. 17. Detailed steady-state performance comparison between JIT and single-level models, normalized to baseline. A summary of this data was presented in Figure 6.

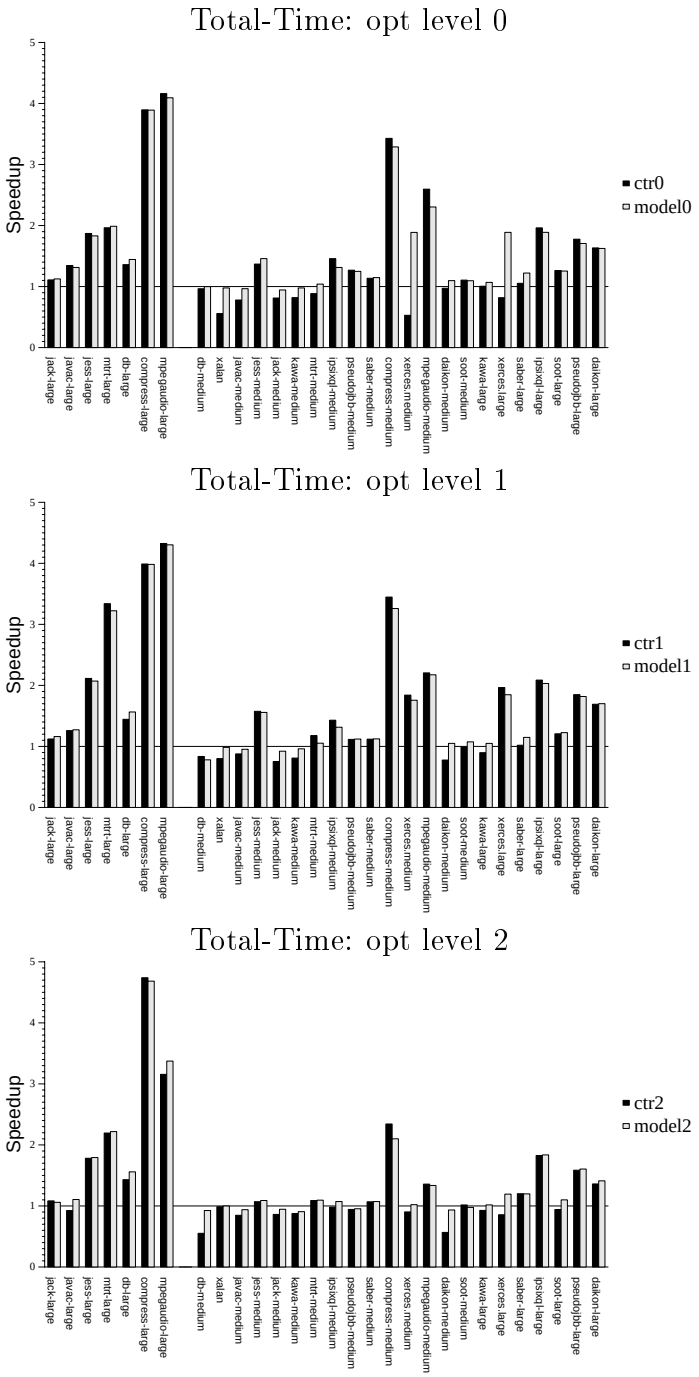


Fig. 18. Detailed total-time performance comparison between counter and single-level models, normalized to baseline. A summary of this data was presented in Figure 7.

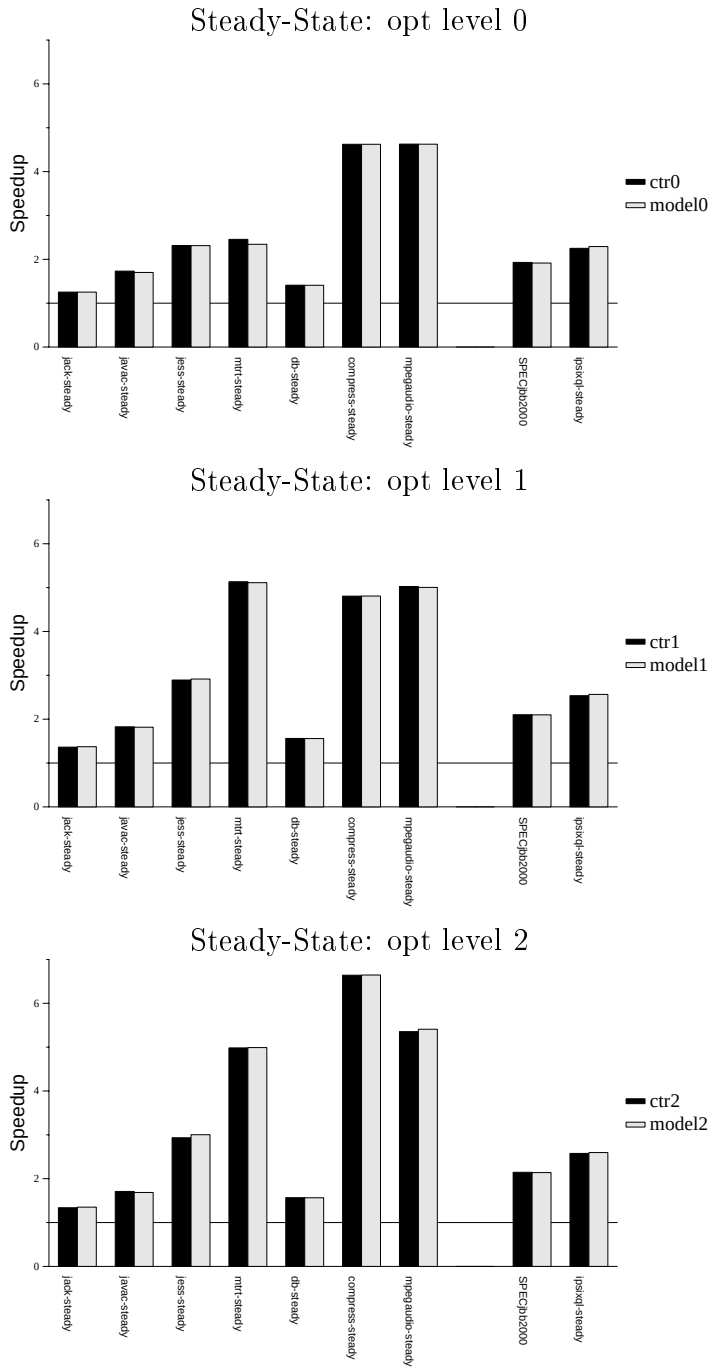


Fig. 19. Detailed steady-state performance comparison between counter and single-level model, normalized to baseline. A summary of this data was presented in Figure 7.

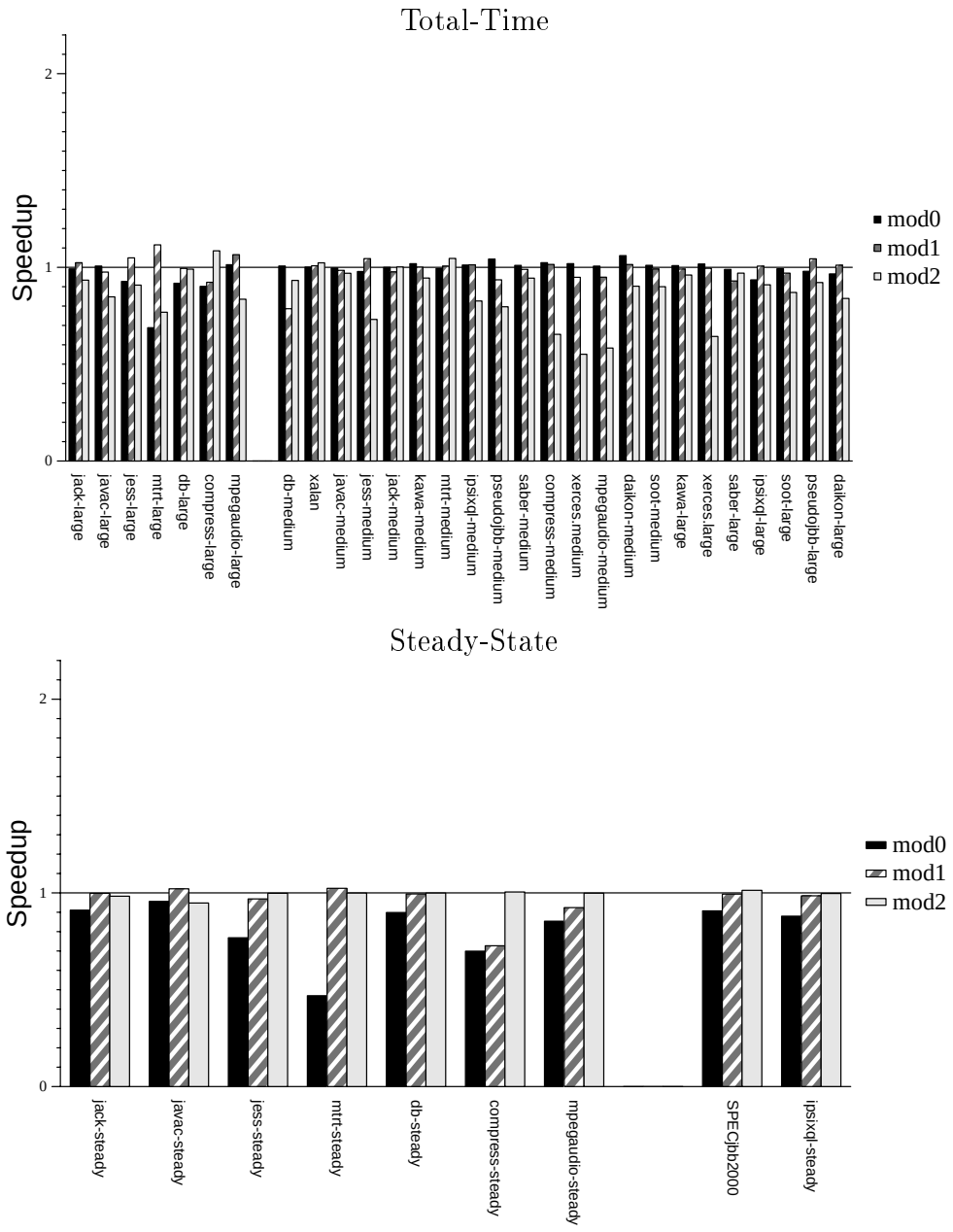


Fig. 20. Detailed performance comparison between single-level models and multi-level model, normalized to multi-level model. A summary of this data was presented in Figure 8.

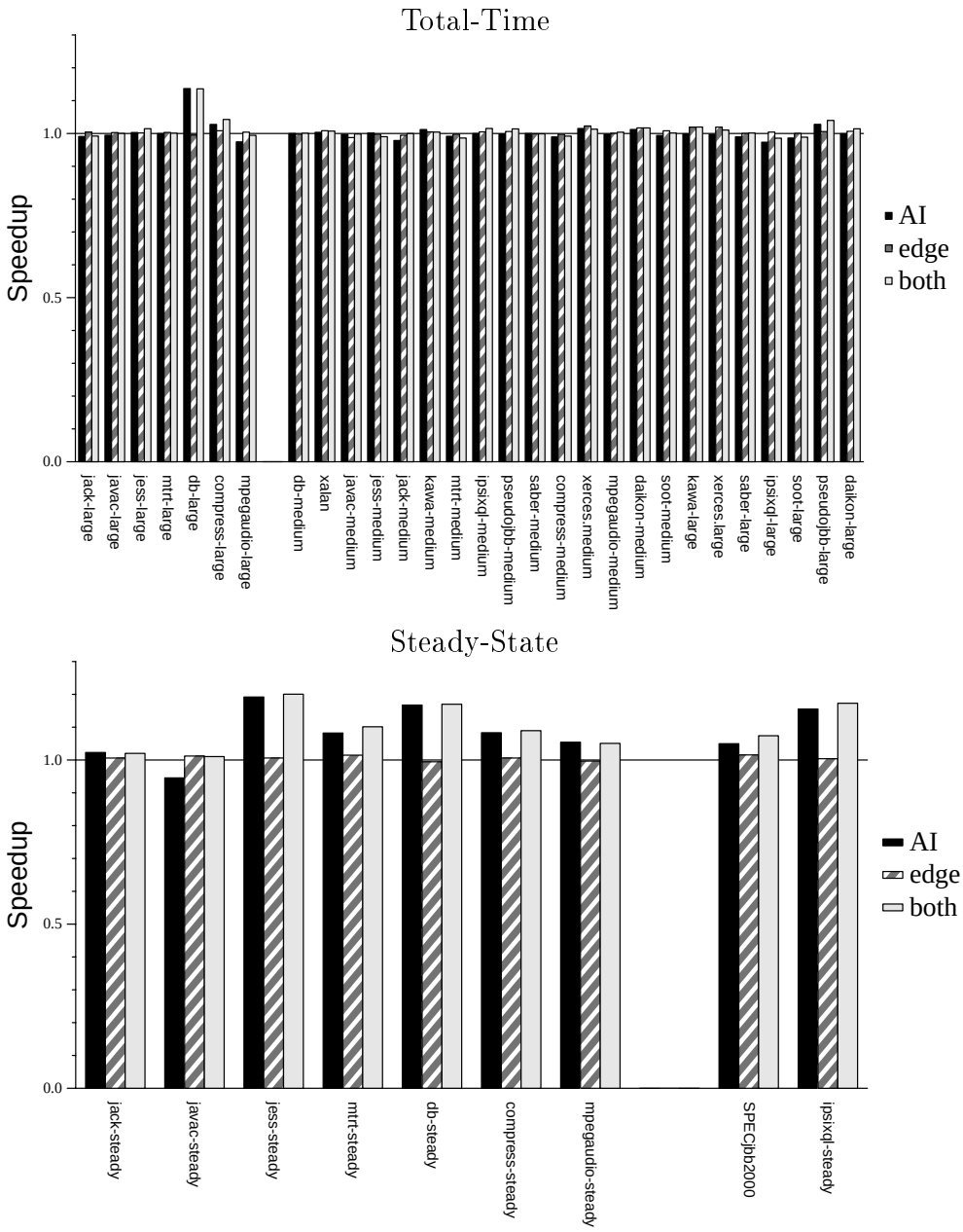


Fig. 21. Detailed performance comparison between feedback-directed optimizations and multi-level model, normalized to multi-level model. A summary of this data was presented in Figure 9.

B. ONLINE APPENDIX

This appendix will be available as an online appendix.

Table IV. Raw times for Total Time regime. The first number is running time in seconds, computed by taking the average of 10 runs. The number in parentheses is the standard deviation as a percentage of the average. For example, the baseline configuration of `db-medium` has 1 standard deviation of values within .27% of the average.

Benchmark	base	ctr0	ctr1	ctr2	opt0	opt1	opt2
db-medium	0.73(0.27)	0.76(0.41)	0.88(0.51)	1.32(1.72)	3.36(0.19)	4.26(0.28)	6.59(0.09)
xalan	2.10(0.21)	3.76(0.32)	2.63(0.42)	2.13(0.39)	15.98(0.15)	30.97(0.10)	148.91(0.04)
javac-medium	2.27(0.36)	2.91(0.61)	2.59(0.16)	2.68(0.31)	10.32(0.20)	17.98(0.48)	46.35(0.08)
jess-medium	2.33(0.08)	1.70(0.25)	1.48(0.31)	2.18(0.31)	6.08(0.19)	9.27(0.43)	22.92(0.07)
jack-medium	2.53(0.21)	3.12(0.38)	3.36(0.30)	2.94(0.13)	7.29(0.13)	15.52(0.49)	30.80(0.17)
kawa-medium	3.59(17.56)	4.23(12.39)	4.28(11.94)	3.84(4.38)	12.89(2.65)	21.29(2.36)	70.03(1.14)
mtrt-medium	3.73(0.26)	4.18(2.61)	3.16(0.16)	3.42(0.26)	6.08(0.32)	6.49(0.29)	16.64(0.12)
ipsixql-medium	4.94(0.49)	3.40(0.73)	3.46(0.52)	5.05(0.50)	4.77(0.60)	5.73(0.56)	13.41(0.11)
pseudojbb-medium	6.04(0.61)	4.76(0.55)	5.43(0.48)	6.42(0.51)	3.39(0.47)	2.69(0.33)	4.30(5.67)
saber-medium	8.40(0.23)	7.40(0.34)	7.52(0.38)	7.88(0.42)	25.79(0.07)	49.41(0.13)	137.76(0.24)
compress-medium	9.82(0.05)	2.86(0.12)	2.85(0.30)	4.19(0.27)	4.82(0.21)	5.15(0.21)	6.74(0.08)
xerces-medium	11.72(0.03)	22.31(2.71)	6.37(0.31)	13.00(0.25)	8.85(0.13)	13.99(0.13)	37.36(0.07)
mpegaudio-medium	11.82(0.02)	4.55(0.07)	5.36(0.09)	8.70(0.10)	6.99(0.10)	7.53(0.26)	17.36(0.07)
diakon-medium	12.23(0.17)	12.59(0.20)	15.75(0.21)	21.59(0.11)	21.57(0.11)	36.14(0.10)	94.26(0.03)
soot-medium	13.14(0.18)	11.87(0.19)	13.27(0.27)	12.93(0.25)	23.89(0.10)	38.01(0.06)	150.83(0.09)
kawa-large	17.52(6.11)	17.40(4.82)	19.23(3.92)	18.69(3.96)	33.98(1.36)	61.12(2.56)	1400.84(0.83)
jack-large	19.48(0.20)	17.57(0.14)	17.38(0.22)	17.99(0.26)	20.28(0.12)	25.18(0.31)	40.67(0.12)
xerces-large	22.14(0.08)	27.09(1.67)	11.28(0.36)	25.86(0.20)	14.72(0.09)	18.55(0.08)	44.32(0.05)
saber-large	24.91(0.37)	23.79(1.22)	24.42(0.28)	20.76(0.30)	48.93(0.19)	93.40(0.15)	272.97(0.09)
javac-large	28.42(0.14)	21.13(0.18)	22.58(0.17)	30.75(0.24)	25.76(0.09)	32.12(0.53)	63.71(0.09)
jess-large	31.23(0.05)	16.69(0.24)	14.77(0.14)	17.53(0.23)	19.55(0.11)	18.97(0.26)	31.33(0.06)
mtrt-large	38.27(0.42)	19.48(0.34)	11.45(0.26)	17.45(0.16)	20.87(0.36)	13.87(0.28)	22.37(0.08)
db-large	41.59(0.27)	30.59(0.49)	28.45(3.54)	28.96(2.31)	34.21(0.23)	32.68(0.33)	31.82(3.44)
ipsixql-large	46.19(0.08)	23.57(0.56)	22.14(0.22)	25.28(0.14)	27.54(0.17)	26.76(0.26)	33.72(0.12)
soot-medium	66.84(0.07)	52.91(0.12)	55.38(0.18)	70.98(0.18)	61.41(0.16)	78.52(0.04)	206.04(0.16)
pseudojbb-large	97.28(0.15)	54.78(0.16)	52.66(0.28)	61.31(0.21)	49.53(0.14)	44.82(0.21)	43.85(0.28)
compress-large	107.89(0.03)	27.78(0.78)	27.09(0.37)	22.77(0.09)	26.83(0.12)	26.38(0.18)	22.72(0.13)
diakon-large	108.12(0.05)	66.20(0.14)	63.95(0.11)	79.51(0.24)	73.25(0.08)	84.10(0.09)	140.28(0.03)
mpegaudio-large	109.03(0.01)	26.21(0.64)	25.21(0.06)	34.55(0.09)	27.87(0.02)	26.62(0.04)	34.75(0.04)

Table V. Raw times for Total Time regime. The first number is running time in seconds, computed by taking the average of 10 runs. The number in parentheses is the standard deviation as a percentage of the average. For example, the baseline configuration of `db-medium` has 1 standard deviation of values within 3.44% of the average.

Benchmark	mod0	mod1	mod2	mod012	mod012_ai	mod012_edge	mod012_fdo
db-medium	0.73(3.44)	0.92(9.41)	0.79(3.37)	0.73(3.68)	0.74(2.49)	0.74(0.91)	0.74(2.94)
xalan	2.14(0.89)	2.14(0.68)	2.11(1.25)	2.15(0.42)	2.15(0.78)	2.14(0.54)	2.14(0.82)
javac-medium	2.35(0.66)	2.39(1.18)	2.40(2.06)	2.38(2.86)	2.35(0.92)	2.36(1.28)	2.36(1.35)
jess-medium	1.61(1.71)	1.51(1.73)	2.13(1.97)	1.57(3.18)	1.57(1.92)	1.57(2.68)	1.58(1.55)
jack-medium	2.67(2.24)	2.79(4.67)	2.69(1.33)	2.68(1.87)	2.72(1.98)	2.69(2.22)	2.68(2.11)
kawa-medium	3.68(20.18)	3.63(12.79)	3.93(19.65)	3.55(6.49)	3.68(18.10)	3.60(13.12)	3.64(13.96)
mtrt-medium	3.59(1.98)	3.43(6.60)	3.40(1.93)	3.57(1.62)	3.59(1.36)	3.58(1.38)	3.60(1.46)
ipsixql-medium	3.74(2.39)	3.77(2.14)	4.64(2.41)	3.81(2.25)	3.81(1.79)	3.80(1.41)	3.76(2.11)
pseudojbb-medium	4.87(1.56)	5.33(3.24)	6.34(3.71)	5.06(1.72)	5.05(1.62)	5.03(1.95)	5.00(1.72)
saber-medium	7.33(0.75)	7.47(1.23)	7.82(1.45)	7.41(1.28)	7.39(0.49)	7.40(0.68)	7.40(0.49)
compress-medium	2.99(0.85)	3.02(1.06)	4.67(1.94)	3.08(2.03)	3.10(1.31)	3.06(1.28)	3.08(1.25)
xerces-medium	6.19(1.23)	6.66(1.91)	11.49(1.81)	6.35(2.23)	6.28(1.71)	6.20(2.31)	6.26(2.54)
mpegaudio-medium	5.12(1.18)	5.45(1.81)	8.83(3.57)	5.15(0.73)	5.16(0.88)	5.17(0.97)	5.14(1.21)
diakon-medium	11.15(0.77)	11.65(2.12)	13.06(3.08)	11.82(1.24)	11.68(1.10)	11.65(1.44)	11.64(1.04)
soot-medium	12.00(0.55)	12.27(1.05)	13.59(2.56)	12.15(1.10)	12.21(0.68)	12.02(0.23)	12.10(0.57)
kawa-large	16.34(4.52)	16.51(3.27)	17.25(6.27)	16.30(2.11)	16.55(3.90)	16.14(3.88)	16.19(4.28)
jack-large	17.31(0.37)	16.77(0.60)	18.39(0.85)	17.16(0.35)	17.30(0.86)	17.12(0.55)	17.29(0.62)
xerces-large	11.70(0.60)	12.02(1.34)	18.60(2.41)	11.98(1.32)	11.98(1.27)	11.74(1.22)	11.79(1.87)
saber-large	20.42(0.61)	21.71(0.48)	21.03(3.69)	20.22(0.64)	20.37(0.69)	20.19(0.70)	20.15(0.32)
javac-large	21.64(0.46)	22.37(0.43)	25.72(0.98)	21.81(0.54)	21.88(0.97)	21.69(0.63)	21.83(0.77)
jess-large	17.06(0.75)	15.12(0.61)	17.46(1.31)	15.81(0.75)	15.70(1.37)	15.76(0.68)	15.65(1.08)
mtrt-large	19.32(1.83)	11.77(2.19)	17.26(1.01)	13.27(1.02)	13.34(1.54)	13.21(0.59)	13.24(0.74)
db-large	28.83(0.38)	26.66(1.12)	26.73(0.63)	26.52(1.15)	23.17(2.26)	26.57(0.40)	23.33(1.17)
ipsixql-large	24.44(0.57)	22.72(0.64)	25.17(0.88)	22.89(0.69)	23.44(0.64)	22.75(0.47)	23.26(0.76)
soot-medium	53.23(0.35)	54.59(0.62)	60.65(1.89)	52.80(0.35)	53.69(0.54)	52.84(0.33)	53.55(0.71)
pseudojbb-large	57.06(0.41)	53.50(0.43)	60.60(0.87)	55.88(0.54)	54.34(0.73)	55.50(0.39)	53.88(0.99)
compress-large	27.71(0.33)	27.08(0.48)	23.06(0.49)	25.03(0.51)	24.29(0.43)	24.78(0.26)	23.97(0.15)
diakon-large	66.61(0.29)	63.64(0.66)	76.70(0.77)	64.19(0.45)	64.33(0.72)	63.92(0.53)	63.38(0.55)
mpegaudio-large	26.61(0.25)	25.36(0.27)	32.18(0.86)	26.99(0.63)	27.86(2.26)	26.87(0.40)	27.22(1.60)

Table VI. Raw times for steady-state regime. The first number is running time in seconds, computed by taking the average of 3 runs. The number in parentheses is the standard deviation as a percentage of the average. For example, the baseline configuration of `SPECjbb-300` has 1 standard deviation of values within .39% of the average.

Benchmark	base	ctr0	ctr1	ctr2	opt0	opt1	opt2
SPECjbb-300	4.76(0.39)	2.47(0.35)	2.26(0.66)	2.22(0.09)	2.41(0.29)	2.21(0.09)	2.17(0.21)
jack-long	15.42(0.23)	12.34(0.47)	11.31(0.14)	11.51(0.41)	12.46(0.12)	11.32(0.08)	11.54(0.07)
javac-long	26.61(0.19)	15.39(0.16)	14.58(0.08)	15.57(0.43)	15.88(0.09)	14.89(0.06)	16.04(0.06)
jess-long	26.67(0.02)	11.52(0.06)	9.23(0.10)	9.09(0.12)	11.47(0.06)	9.11(0.05)	8.99(0.03)
mtrt-long	36.18(0.11)	14.73(0.11)	7.05(0.02)	7.26(0.17)	14.84(0.22)	7.17(0.07)	7.34(0.35)
db-long	40.62(0.37)	28.94(0.87)	26.13(0.44)	25.97(0.84)	29.17(0.55)	27.00(0.27)	27.76(0.01)
ipsixql	41.86(0.22)	18.61(0.07)	16.52(0.09)	16.26(0.08)	22.27(0.23)	20.90(0.06)	21.04(0.04)
compress-long	107.46(0.23)	23.28(0.12)	22.38(0.07)	16.20(0.06)	23.22(0.04)	22.32(0.04)	16.17(0.01)
mpegaudio-long	108.58(0.01)	23.50(0.26)	21.61(0.07)	20.28(0.06)	23.58(0.01)	21.56(0.00)	20.00(0.02)

Table VII. Raw times for steady-state regime. The first number is running time in seconds, computed by taking the average of 3 runs. The number in parentheses is the standard deviation as a percentage of the average. For example, the baseline configuration of SPECjbb-300 has 1 standard deviation of values within .65% of the average.

Benchmark	mod0	mod1	mod2	mod012	mod012_ai	mod012_edge	mod012_fdo
SPECjbb-300	2.48(0.65)	2.27(0.73)	2.23(0.31)	2.26(0.76)	2.15(0.35)	2.22(0.59)	2.10(0.47)
jack-long	12.31(0.29)	11.23(0.12)	11.36(0.68)	11.19(0.34)	11.32(6.13)	11.16(0.44)	10.99(0.33)
javac-long	15.66(0.19)	14.68(0.35)	15.81(0.25)	14.97(0.15)	15.72(1.72)	14.81(0.41)	14.82(0.12)
jess-long	11.53(0.06)	9.15(0.07)	8.88(0.06)	8.88(0.46)	7.44(0.41)	8.81(0.09)	7.38(0.30)
mtrt-long	15.22(2.62)	7.07(0.18)	7.25(0.06)	7.25(0.21)	6.70(0.08)	7.13(0.14)	6.57(0.21)
db-long	28.80(1.03)	26.08(1.05)	25.92(0.84)	25.88(1.02)	22.14(0.88)	26.08(1.14)	22.12(1.60)
ipsixql	18.28(0.09)	16.33(0.19)	16.14(0.19)	16.09(0.27)	13.91(0.50)	16.08(0.82)	13.72(0.17)
compress-long	23.26(0.09)	22.37(0.09)	16.19(0.01)	16.27(0.07)	15.02(0.13)	16.17(0.08)	14.95(0.20)
mpegaudio-long	23.49(0.28)	21.67(0.41)	20.06(0.10)	20.06(0.14)	19.02(0.19)	20.10(0.36)	19.06(0.16)

Table VIII. Percentage of time spent in various Jikes RVM threads.

Total-Time

Benchmark	Application	GC	Recompilation	AOS
db-medium	90.03	0.00	8.53	1.44
xalan	98.69	0.00	0.72	0.59
javac-medium	68.15	25.98	5.30	0.56
jess-medium	94.53	0.00	4.59	0.88
jack-medium	92.32	0.00	7.01	0.66
kawa-medium	95.18	0.00	4.27	0.55
mtrt-medium	82.30	0.00	17.25	0.45
ipsixql-medium	89.82	0.00	9.65	0.54
pseudojbb-medium	87.57	0.00	12.16	0.27
compress-medium	95.70	0.00	3.89	0.40
xerces-medium	83.15	0.00	16.40	0.45
mpegaudio-medium	76.12	0.00	23.41	0.47
daikon-medium	87.71	0.00	12.00	0.29
soot-medium	95.91	0.00	3.90	0.19
kawa-large	94.37	0.00	5.20	0.42
jack-large	87.51	3.38	8.75	0.36
xerces-large	84.64	0.00	14.93	0.43
javac-large	80.37	10.98	8.29	0.36
jess-large	82.48	4.26	12.93	0.32
saber-medium	96.39	0.00	3.42	0.19
mtrt-large	89.64	0.00	9.95	0.42
db-large	98.11	0.00	1.77	0.12
ipsixql-large	85.89	6.11	7.72	0.28
saber-large	95.26	0.00	4.48	0.26
soot-large	79.74	15.00	5.07	0.19
pseudojbb-large	86.15	4.23	9.32	0.29
compress-large	94.07	0.00	5.80	0.13
daikon-large	84.55	6.00	9.16	0.29
mpegaudio-large	85.79	0.00	13.96	0.25
Arith. Mean	88.35	2.62	8.61	0.42

Steady-State

Benchmark	Application	GC	Recompilation	AOS
ipsixql	89.47	9.32	1.12	0.10
SPECjbb2000	92.44	4.49	2.84	0.24
jack	89.83	6.19	3.84	0.15
javac	73.76	19.47	6.52	0.25
jess	90.06	8.26	1.57	0.11
mtrt	91.50	5.30	3.04	0.16
db	98.76	0.95	0.26	0.03
compress	98.46	1.25	0.25	0.03
mpegaudio	97.69	0.00	2.23	0.08
Arith. Mean	91.33	6.14	2.41	0.13

Table IX. Percentage summary of final optimization level for all methods

Total-Time

Benchmark	Baseline	Total Opt	Opt 0	Opt 1	Opt 2
db-medium	97.62 (246)	2.38 (6)	2.38 (6)	0.00 (0)	0.00 (0)
xalan	99.81 (1579)	0.19 (3)	0.19 (3)	0.00 (0)	0.00 (0)
javac-medium	99.04 (925)	0.96 (9)	0.75 (7)	0.21 (2)	0.00 (0)
jess-medium	98.62 (645)	1.38 (9)	0.61 (4)	0.76 (5)	0.00 (0)
jack-medium	94.27 (444)	5.73 (27)	4.25 (20)	1.49 (7)	0.00 (0)
kawa-medium	98.89 (1774)	1.11 (20)	0.89 (16)	0.22 (4)	0.00 (0)
mtrt-medium	91.99 (333)	8.01 (29)	4.70 (17)	3.31 (12)	0.00 (0)
ipsixql-medium	95.42 (438)	4.58 (21)	3.70 (17)	0.87 (4)	0.00 (0)
pseudojbb-medium	85.65 (555)	14.35 (93)	8.95 (58)	4.78 (31)	0.62 (4)
compress-medium	95.80 (228)	4.20 (10)	2.10 (5)	2.10 (5)	0.00 (0)
xerces-medium	91.93 (661)	8.07 (58)	6.54 (47)	1.53 (11)	0.00 (0)
mpegaudio-medium	90.95 (372)	9.05 (37)	6.11 (25)	2.93 (12)	0.00 (0)
daikon-medium	94.97 (1586)	5.03 (84)	3.53 (59)	1.50 (25)	0.00 (0)
soot-medium	98.11 (1193)	1.89 (23)	1.23 (15)	0.66 (8)	0.00 (0)
kawa-large	97.14 (3396)	2.86 (100)	2.37 (83)	0.46 (16)	0.03 (1)
jack-large	89.22 (422)	10.78 (51)	4.86 (23)	5.71 (27)	0.21 (1)
xerces-large	87.23 (717)	12.77 (105)	7.91 (65)	4.74 (39)	0.12 (1)
javac-large	81.29 (782)	18.71 (180)	13.31 (128)	5.30 (51)	0.10 (1)
jess-large	94.01 (628)	5.99 (40)	2.10 (14)	2.84 (19)	1.05 (7)
saber-medium	99.34 (1956)	0.66 (13)	0.41 (8)	0.20 (4)	0.05 (1)
mtrt-large	81.82 (297)	18.18 (66)	5.79 (21)	11.57 (42)	0.83 (3)
db-large	98.05 (251)	1.95 (5)	0.78 (2)	0.00 (0)	1.17 (3)
ipsixql-large	91.63 (449)	8.37 (41)	4.29 (21)	3.88 (19)	0.20 (1)
saber-large	99.04 (4331)	0.96 (42)	0.73 (32)	0.21 (9)	0.02 (1)
soot-large	89.84 (1557)	10.16 (176)	6.23 (108)	3.58 (62)	0.35 (6)
pseudojbb-large	66.98 (434)	33.02 (214)	15.12 (98)	14.97 (97)	2.93 (19)
compress-large	95.38 (227)	4.62 (11)	0.42 (1)	2.52 (6)	1.68 (4)
daikon-large	90.38 (1512)	9.62 (161)	5.14 (86)	3.53 (59)	0.96 (16)
mpegaudio-large	83.82 (342)	16.18 (66)	6.13 (25)	9.31 (38)	0.74 (3)
Arith. Mean	92.35 (975)	7.65 (58)	4.19 (34)	3.08 (21)	0.38 (2)

Steady-State

Benchmark	Baseline	Total Opt	Opt 0	Opt 1	Opt 2
ipsixql	86.94 (426)	13.06 (64)	3.27 (16)	4.90 (24)	4.90 (24)
SPECjbb2000	71.69 (580)	28.31 (229)	4.45 (36)	12.73 (103)	11.12 (90)
jack	66.17 (313)	33.83 (160)	11.21 (53)	13.95 (66)	8.67 (41)
javac	50.21 (483)	49.79 (479)	18.71 (180)	22.56 (217)	8.52 (82)
jess	81.94 (549)	18.06 (121)	8.36 (56)	5.82 (39)	3.88 (26)
mtrt	72.53 (264)	27.47 (100)	4.67 (17)	15.38 (56)	7.42 (27)
db	94.57 (244)	5.43 (14)	1.94 (5)	1.55 (4)	1.94 (5)
compress	91.60 (218)	8.40 (20)	3.36 (8)	2.94 (7)	2.10 (5)
mpegaudio	72.13 (295)	27.87 (114)	6.11 (25)	12.71 (52)	9.05 (37)
Arith. Mean	76.42 (374)	23.58 (144)	6.90 (44)	10.28 (63)	6.40 (37)