Adaptive Optimization in the Jalapeño JVM

Matthew Arnold[*,†] Stephen Fink[†] David Grove[†] Michael Hind[†] Peter F. Sweeney[†]

[†]IBM T.J. Watson Research Center
{sjfink,groved,hindm,pfs}@us.ibm.com

[*]Rutgers University
marnold@cs.rutgers.edu

ABSTRACT

Future high-performance virtual machines will improve performance through sophisticated online feedback-directed optimizations. This paper presents the architecture of the Jalapeño Adaptive Optimization System, a system to support leading-edge virtual machine technology and enable ongoing research on online feedback-directed optimizations. We describe the extensible system architecture, based on a federation of threads with asynchronous communication. We present an implementation of the general architecture that supports adaptive multi-level optimization based purely on statistical sampling. We empirically demonstrate that this profiling technique has low overhead and can improve startup and steady-state performance, even without the presence of online feedback-directed optimizations. The paper also describes and evaluates an online feedback-directed inlining optimization based on statistical edge sampling. The system is written completely in Java, applying the described techniques not only to application code and standard libraries, but also to the virtual machine itself.

1. INTRODUCTION

The dynamic nature of the Java programming language [26] presents both the largest challenge and the greatest opportunity for high-performance Java implementations. Language features such as dynamic class loading and reflection prevent straightforward applications of traditional static compilation and interprocedural optimization. As a result, Java Virtual Machine (JVM) implementors have invested significant effort in developing dynamic compilers for Java. Because dynamic compilation occurs during application execution, dynamic compilers must carefully balance optimization effectiveness with compilation overhead to maximize total system performance. However, dynamic compilers can also exploit runtime information to perform optimizations beyond the scope of a purely static compilation model.

The first wave of virtual machines provided Just-In-Time (JIT) compilation that relied on simple static strategies to choose compilation targets, typically compiling each method with a fixed set of optimizations the first time it was invoked. These virtual machines include early work such as the Smalltalk-80 [24] and Self-91 [16] systems, as well as a number of more recent Java systems [1, 36, 14, 47]. A second wave of more sophisticated virtual machines moved beyond this simple strategy by dynamically selecting a subset of all executed methods for optimization, attempting to focus optimization effort on program hot spots. Systems in this category include Self-93 [32], the HotSpot JVM [34], the IBM Java Just-in-Time compiler (version 3.0) [43], JUDO [18], and the initial stage of the Jalapeño Adaptive Optimization System described in this paper. Some second-wave virtual machines also include limited forms of online feedback-directed optimization (e.g. inlining in Self-93), but do not develop general mechanisms for adaptive online feedback-directed optimization.

A number of research projects have explored more aggressive forms of dynamic compilation [35, 9, 12, 13, 8, 27, 28, 38, 37, 19, 39], using runtime information to tailor the executable to its current environment. Most of these systems were not fully automatic, and so far, few of these techniques have appeared in mainstream JVMs. However, these systems have demonstrated that online feedback-directed optimizations can yield substantial performance improvements. It seems clear that a key component in the upcoming third wave of high-performance virtual machines will be sophisticated adaptive online feedback-directed optimizations.

Jalapeño [2, 3] is a research JVM being developed at the IBM T.J. Watson Research Center. This paper introduces the Jalapeño Adaptive Optimization System, a key component of the Jalapeño JVM. Previous papers [2, 14] briefly describe the adaptive optimization system at a high level. This paper presents the first description of the architectural details, implementation, and performance results.

The main contributions of this paper are

- an extensible adaptive optimization architecture that both supports current leading-edge JVM technology and enables research into online feedback-directed optimization,
- the demonstration of low-overhead sampling techniques to drive adaptive and online feedback-directed optimizations,

- the demonstration of an adaptive optimization system that uses multiple optimization levels to improve performance compared to using only a single level of optimization, and
- an initial implementation and evaluation of our first online feedback-directed optimization: inlining based on low-overhead sampling data. This optimization improves performance by 11% on average, with improvements ranging up to 73%.

The next section provides some background on the Jalapeño JVM. Section 3 describes the architecture of the Jalapeño adaptive optimization system and highlights key issues and design decisions. Section 4 describes the recompilation implementation. Section 5 describes our first online feedback-directed optimization, adaptive inlining. Section 6 presents experimental data demonstrating that sampling is both an effective and low-overhead mechanism for driving adaptive optimization. Section 7 presents a discussion of our experiences. Finally, Section 8 discusses related work and Section 9 offers our conclusions.

2. BACKGROUND

The Jalapeño JVM is a research JVM targeting server applications. A comprehensive description of Jalapeño appears in [2]. In this section we highlight the characteristics of Jalapeño that are most relevant to this work.

Jalapeño is written in Java [3]. In addition to providing a high-level strongly-typed development environment, this design decision allows the techniques described in this paper to apply not only to application code, but also to the JVM itself. That is, we apply adaptive optimization to the JVM subsystems, including the compilers, the thread scheduler, the garbage collector, and the adaptive optimization system itself.

Jalapeño employs a compile-only strategy; it compiles all methods to native code before they execute. Currently, the system includes two fully operational compilers.

- The *baseline* compiler translates bytecodes directly into native code by simulating Java's operand stack. The compiler does not perform register allocation. The baseline compiler provides a reference compiler for testing purposes, and generates native code that performs only slightly better than bytecode interpretation [2].
- The *optimizing* compiler [14] translates bytecodes into an intermediate representation, upon which it performs a variety of optimizations. The compiler uses linear scan register allocation [40], an efficient and effective register allocator. For this paper, we group the compiler's optimizations into several levels:
  - *Level 0* consists mainly of a set of optimizations performed on-the-fly during the translation from bytecodes to the intermediate representation. As these optimizations reduce the size of the generated IR, performing them tends to reduce overall compilation time [45]. Currently, the compiler performs the following optimizations during IR generation: constant, type, non-null, and copy propagation; constant folding and arithmetic simplification; unreachable code elimination; and elimination of redundant nullchecks, checkcasts, and array store checks.
  - *Level 1* augments level 0 with additional local optimizations such as common subexpression elimination, array bounds check elimination, and redundant load elimination. It also adds inlining based on static-size heuristics,[1] global flow-insensitive copy and constant propagation, global flow-insensitive dead assignment elimination, StringBuffer synchronization optimizations, and scalar replacement of aggregates and short arrays.
  - *Level 2* augments level 1 with SSA-based flow-sensitive optimizations. In addition to traditional SSA optimizations on scalar variables [21], the system also uses an extended version of Array SSA form [25] to perform redundant load elimination and array bounds check elimination [11].

Compiler	Bytecode Bytes/Millisecond
Baseline	274.14
Opt Level 0	8.77
Opt Level 1	3.59
Opt Level 2	2.07

Table 1: Average compilation rates on the SPECjvm98 benchmark suite

Table 1 gives the average compilation rate for the compilers used in this paper on the PowerPC machine described in Section 6.

Jalapeño multiplexes Java threads onto JVM *virtual processors*, which are implemented as AIX pthreads. The system supports thread scheduling with a quasi-preemptive mechanism. Each compiler generates *yield points*, which are program points where the running thread checks a dedicated bit in a machine control register to determine if it should yield the virtual processor. Currently, the compilers insert these yield points in method prologues and on loop back edges. Future work includes developing algorithms for more optimal placement of yield points to reduce the dynamic number of yield points executed while still supporting effective quasi-preemptive thread scheduling. Using a timer-interrupt mechanism, an interrupt handler periodically sets a bit on all virtual processors. When a running thread next reaches a yield point, a check of the bit will result in a call to the scheduler. Section 4 discusses how we exploit this mechanism in the current version of our system.

As shown in Figure 1, a Jalapeño compiler can be invoked in three ways. First, when the executing code reaches an

[1] The compiler performs both unguarded inlining of final and static methods and guarded inlining of non-final virtual methods. 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 [23].

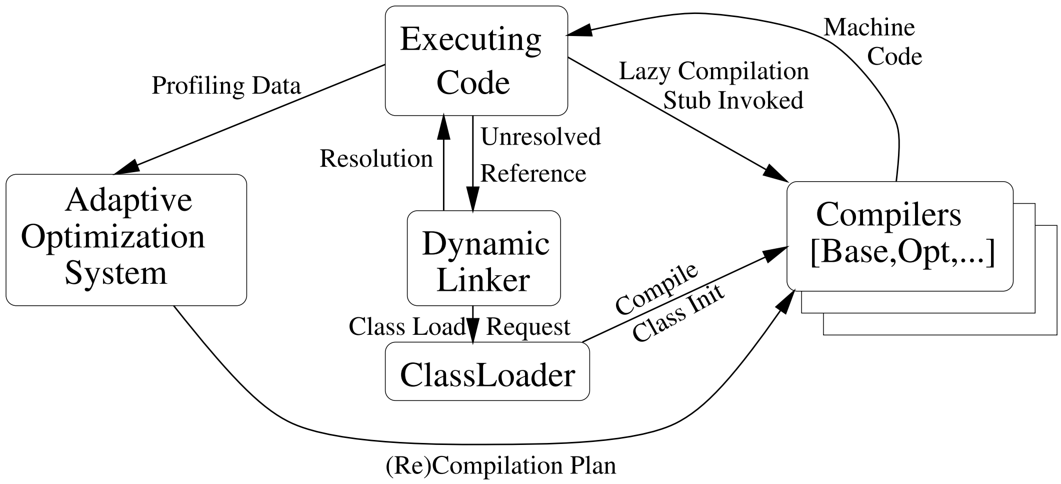

Figure 1: Compilation scenarios in the Jalapeño JVM

unresolved reference, causing a new class to be loaded, the class loader invokes a compiler to compile the class initializer (if one exists). The class loader also initializes the compiled code for all methods to a *lazy compilation stub*. The second compilation scenario occurs whenever the executing code attempts to invoke a method that has not yet been compiled. When this happens, the lazy compilation stub is executed, which leads to the compilation of the method. In these first two scenarios, the application thread that caused the compiler to be invoked will stall 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 Jalapeño JVM begins execution by reading from a *boot image* file, which contains the core services of Jalapeño precompiled to machine code [3]. Jalapeño supports several configurations of the "core". The simplest configuration includes a class loader, an object allocator, and the baseline compiler. This version loads and compiles (with the baseline compiler) all non-core classes, including the optimizing compiler, when the JVM boots. For better performance, a *full configuration* includes the optimizing compiler in the precompiled boot image. Results in this paper use the full configuration, which increases boot-image writing time, but produces a higher performance JVM.

3. SYSTEM ARCHITECTURE

The Jalapeño Adaptive Optimization System (AOS) contains three components, each of which encompasses one or more separate threads of control. These subsystems are the *runtime measurements subsystem*, the *controller*, and the *recompilation subsystem*. Figure 2 depicts the internal structure of the Jalapeño adaptive optimization system and the interactions between its components. In addition to the components, the *AOS database* provides a repository that records component decisions and allows components to query these decisions. The next four sections discuss this figure in more detail.

3.1 Runtime Measurements Subsystem

The runtime measurements subsystem gathers information about the executing methods, summarizes the information, and then either passes the summary along to the controller via the organizer event queue or records the information in the AOS database.

Figure 2 shows the structure of the runtime measurements subsystem. Several systems, including instrumentation in the executing code, hardware performance monitors, and VM instrumentation, produce raw profiling data as the program runs. Usually, these systems perform only extremely limited processing of the raw data as it is produced. Instead, separate threads called *organizers* periodically process and analyze the raw data. The design separates the generation of raw profiling data from the data analysis for two reasons. First, this design allows multiple organizers to process the same raw data, possibly in different ways. Second, this separation allows low-level profiling code to execute under strict resource constraints. Recall that we monitor not just application code, but also system services of the VM. So, for example, low-level code that monitors the VM memory allocator must not allocate any objects (it must use preallocated data structures and buffers) and should complete its task in a short time period.

The controller directs the data monitoring and creates organizer threads to process the raw data at specific time intervals. When awoken, each organizer analyzes raw data, and packages the data into a suitable form for consumption by the controller. Additionally, an organizer may add information to the organizer event queue for the controller to process, or may record information in the AOS database for later queries by other AOS components.

This architecture can support a variety of measurement techniques, including hardware performance monitors, call stack sampling [46, 7], and compiler-inserted instrumentation such as invocation counters, basic block edge or path profiles [10], and value profiles [15].

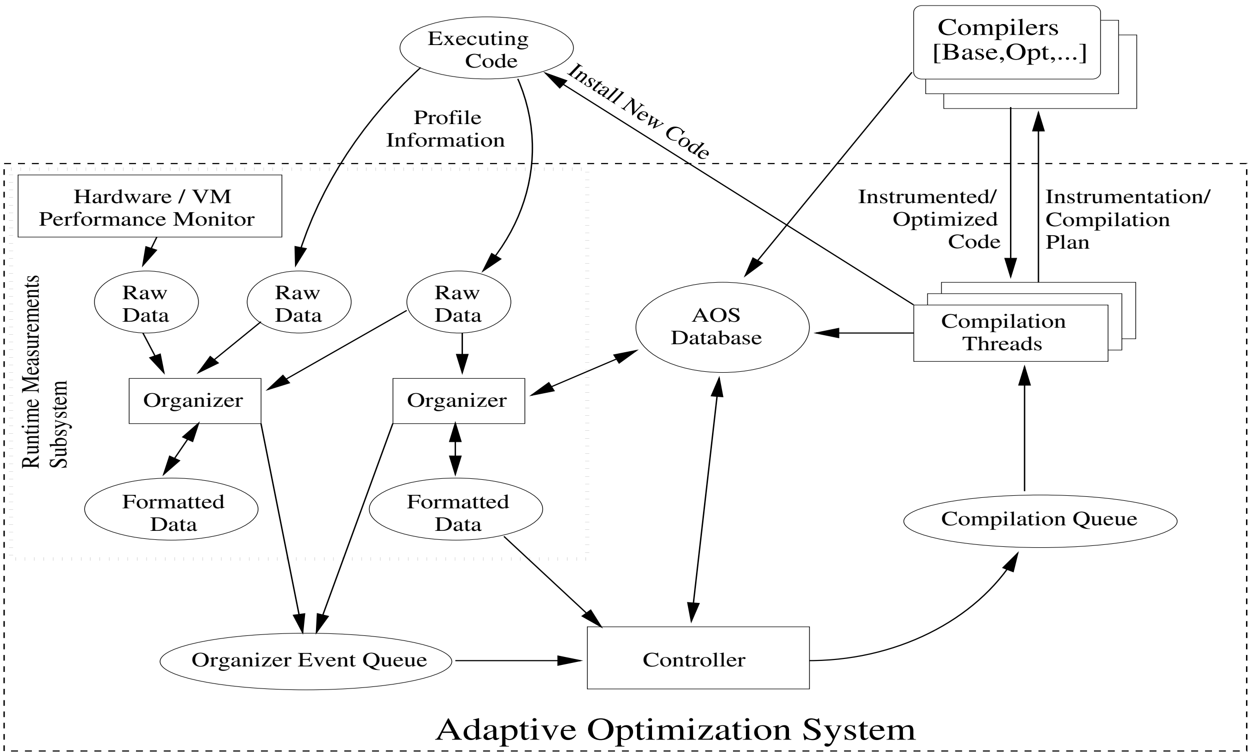

Figure 2: Architecture of the Jalapeño Adaptive Optimization System

3.2 Controller

The controller orchestrates and conducts the other components of the adaptive optimization system. It coordinates the activities of the runtime measurements subsystem and the recompilation subsystem. The controller initiates all runtime measurement subsystem profiling activity by determining what profiling should occur, under what conditions, and for how long. It receives information from the runtime measurement subsystem and AOS database, and uses this information to make compilation decisions. It passes these compilation decisions to the recompilation subsystem, directing the actions of the various compilers.

Based on information from the runtime measurements subsystem and the AOS database, the controller can perform the following actions: 1) it can instruct the runtime measurements subsystem to continue or change its profiling strategy, which could include using the recompilation subsystem to insert intrusive profiling; 2) it can recompile one or more methods using profiling data to improve their performance. The controller makes these decisions based on an analytic model representing the costs and benefits of performing these tasks.

The controller communicates with the other two components using priority queues; it extracts measurement events from a queue that is filled by the runtime measurements subsystem and inserts recompilation decisions into a queue that compilation threads process. When these queues are empty, the dequeuing thread(s) sleep. The various system components also communicate indirectly by reading and writing information in the AOS database.

3.3 Recompilation Subsystem

The recompilation subsystem consists of compilation threads that invoke compilers. The compilation threads extract and execute compilation plans that are inserted into the compilation queue by the controller. Recompilation occurs in separate threads from the application, and thus, can occur in parallel. This differs from the initial (lazy) compilation of a method, which occurs the first time a method is invoked: during lazy compilation, compilation occurs in the application thread that attempted to invoke the uncompiled method.

Each compilation plan consists of three components: an *optimization plan*, *profiling data*, and an *instrumentation plan*. The optimization plan specifies which optimizations the compiler should apply during recompilation. The profiling data, initially gathered by the runtime measurements subsystem, directs the optimizing compiler's feedback-directed optimizations. Instrumentation plans dictate which, if any, intrusive instrumentation the compiler should insert into the generated code.

The compilation threads takes the output of the compiler — a Java object that represents the executable code and associated runtime information (exception table information and garbage collection maps) — and installs it in the JVM, so that all future calls to this method will use the new version. In our current implementation, any previous activations of the method will continue to use the old compiled code for the method until that method's activation completes. Nothing in our design precludes using stack frame rewriting to enable previous activations of the method to use the new compiled version, but this functionality has not yet been implemented. We expect that Jalapeño will eventually rewrite baseline stack frames to optimized stack frames. It is as yet unclear if there is sufficient motivation to support the more difficult transition between two optimized stack frames.

3.4 AOS Database

The AOS database provides a repository where the adaptive optimization system records decisions, events, and static analysis results. The various adaptive system components query these artifacts as needed.

For example, the controller uses the AOS database to record compilation plans and to track the status and history of methods selected for recompilation. As another example, the compilation threads record static analysis and inlining summaries produced by the optimizing compiler. The controller and organizer threads query this information as needed to guide recompilation decisions. More details on the current implementation appear in Section 5.

4. MULTI-LEVEL RECOMPILATION

This section describes the implementation of the adaptive recompilation system. Section 5 describes how this system is extended to support our first online feedback-directed optimization, adaptive inlining. Section 4.1 provides an overview of the implementation. Section 4.2 provides details concerning how profiling information is obtained. Section 4.3 discusses the recompilation model.

4.1 Overview

Figure 3 provides an overview of the implementation. The controller thread is created during JVM boot time. It subsequently creates threads corresponding to the other subsystems: organizer threads to perform sample-based runtime measurements and a single compilation thread[2] to perform recompilation. After these threads are created, the controller sleeps until the runtime measurements subsystem inserts an event in the organizer event queue.

The adaptive optimization system (without adaptive inlining) creates two organizer threads, a *hot methods organizer* and a *decay organizer*. The hot methods organizer processes method samples and inserts hot method events in the organizer event queue to allow the controller to consider the methods for recompilation. Each event contains the method ID and its relative hotness. The decay organizer decays counters contained in the runtime measurements subsystem. Such counters include the hot method counter and the counters associated with call graph edges discussed in Section 5. The decay organizer does not communicate directly with the controller.

[2]Currently, Jalapeño uses a simple coarse-grained locking scheme to control access to a few JVM services. In particular, accessing certain class loader data structures that Jalapeño's compilers need to read during compilation requires that the JVM master lock be held. This locking strategy prevents multiple compilation threads from being effective, and can introduce lock contention between the compilation thread and application threads.

The compilation thread extracts plans from the compilation queue and invokes the optimizing compiler passing in the compilation plan. The compilation thread records the compilation time for the recompiled method in the AOS database to allow for more accurate modeling of future recompilation decisions.

4.2 Sampling

The sampling implementation takes advantage of existing mechanisms in the Jalapeño JVM. Before switching threads, a counter associated with the current method is incremented. The system attributes a sample taken on a back edge to the current method. A sample taken in a method prologue is credited to both the calling and current method, capturing the fact that control is in transition between both methods. We have experimented with other sampling attribution strategies and have found this one to be most effective. The accuracy of this sampling technique is investigated in more detail in [6].

This sampling technique provides a basic mechanism to estimate the time spent in execution of each method. In the adaptive optimization system, two organizer threads periodically process the raw data.

The first organizer thread (the *hot method organizer*), created during system startup, installs a sampling object (the *method listener*) to record raw data regarding the execution profile. During a thread switch, the VM invokes the `update` method of this listener, which records the currently active method in a raw data buffer. This activity costs only a few additional cycles during each thread switch, and the performance impact does not stand out from noise from one run to the next. After collecting the number of samples specified by its current sample size, the method listener wakes the hot methods organizer thread.

When awoken, the hot methods organizer scans the method counter raw data to identify methods where the application spends most of its time. The organizer deems a method to be "hot" if the percentage of samples attributed to that method exceeds a controller-directed threshold and the method is not already compiled at the maximum optimization level available. For each hot method it discovers, the hot methods organizer enqueues an event in the organizer event queue that contains the method and the percentage of samples attributed to the method.

The controller dynamically adjusts the listener's sample size and the organizer's hotness threshold to adapt to the current behavior of the application. By adjusting the sample size within specified bounds, the controller attempts to reduce the overhead of the hot methods organizer when the system is in a steady state (the set of hot methods is stable), while still being able to respond quickly to application phase shifts (indicated by changes in the set of hot methods). Similarly, the controller dynamically adjusts the hotness threshold to approximately control the number of hot methods reported by the hot methods organizer. If after several sampling periods, "not enough" hot methods are being returned, then the controller can decrease the hotness threshold. On the other hand, if "too many" hot methods are being returned for several sampling periods, then the controller can increase

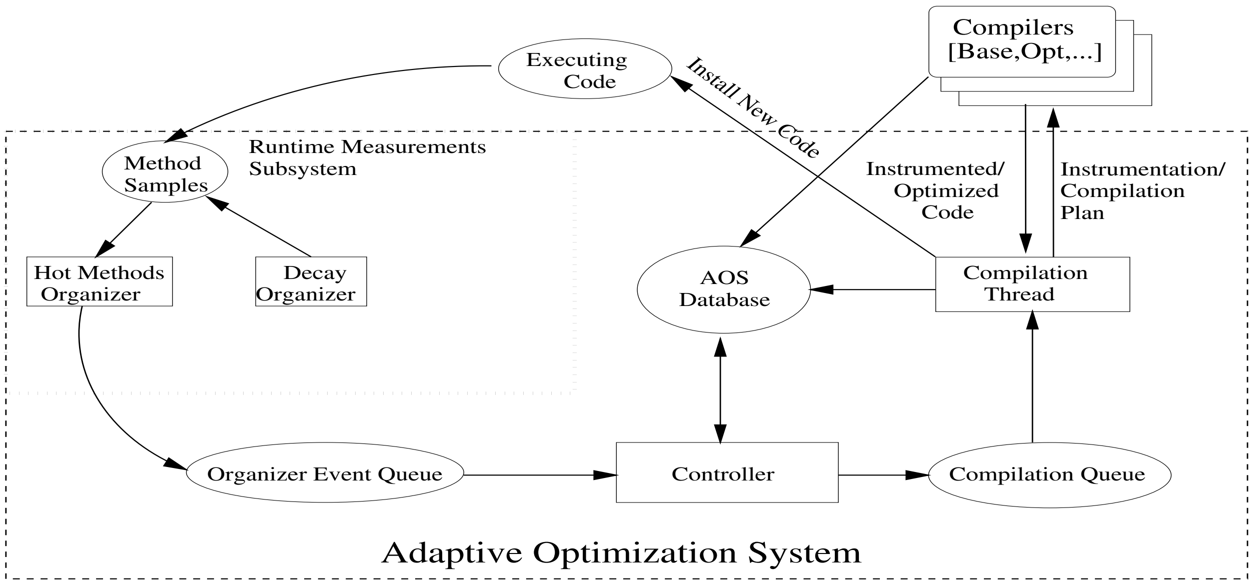

Figure 3: Implementation of adaptive recompilation in the Jalapeño Adaptive Optimization System

the threshold.

A second organizer, the decay organizer, periodically decays the method counters. By decaying the counters, the system gives more weight to recent samples when determining method hotness.

4.3 Recompilation

Given a hot method from the organizer event queue, the controller must decide whether it is profitable to recompile the method with additional optimizations. The controller uses a cost-benefit analysis to make this calculation.

For this discussion, we number the optimization levels available to the controller from 0 to N.[3] For a method m currently compiled at level i, the controller estimates the following quantities:

- T_i, the expected time the program will spend executing method m, if m is not recompiled.
- C_j, the cost of recompiling method m at optimization level j, for $i \leq j \leq N$.[4]
- T_j, the expected time the program will spend executing method m in the future, if m is recompiled at level j.

Using these estimated values, the controller identifies the recompilation level j that minimizes the expected future running time of a recompiled version of m; i.e., it chooses the j that minimizes the quantity $C_j + T_j$. If $C_j + T_j < T_i$, the controller decides to recompile m at level j; otherwise it decides to not recompile.

Clearly, the factors in this model are unknowable in practice. The process of estimating future costs and benefits is an ongoing open research problem. The current controller implementation is based on the fairly simple estimates described below.

First, the controller assumes the program will execute for twice its current duration. So, if the application has run for 5 seconds, the controller assumes it will run for 5 more seconds; if it has run for 2 hours, then it will run for 2 more hours. Define T_f to be the future expected running time of the program.

The system keeps track of where the application spends time as it runs, using the sampling techniques described previously. The system uses a weighted average of these samples to estimate the percentage of future time (P_m) in each method, barring recompilation. From this percentage estimate and the future time estimate, the controller predicts the future time spent in each method. That is,

$$T_i = T_f * P_m \tag{1}$$

For example, if the weighted samples indicate that the application will spend 10% of its time in method m and the code has run for 10 seconds, the controller will estimate the future execution time of m to be 1 second.

The weight of each sample starts at one and decays periodically. Thus, the execution behavior of the recent past exerts the most influence on the estimates of future program behavior. When the controller recompiles methods, it adjusts the future estimates to account for the new optimization level, and expected speedup due to recompilation.

[3]For this discussion, the compilers in our current implementation (baseline, Opt level 0, Opt level 1, Opt level 2) would map into this function as level 0, 1, 2, 3.

[4]The model considers recompilation at the same level because new profiling information may enable additional speedups over the previous version compiled at level i. This is encoded by a feedback-directed optimization boost factor that is used in the calculation of T_j.

The system estimates the effectiveness of each optimization level as a constant based on offline measurements. Let S_k be the speedup estimate for code at level k compared to level 0. Then, if method m is at level i, the future expected running time if we recompile at level j is given by

$$T_j = T_i * S_i / S_j \tag{2}$$

To complete the cost-benefit analysis, the controller needs to estimate the cost of recompilation. It currently use a linear model of the compilation speed for each optimization level, as a function of method size. This model is calibrated offline.

We have described and implemented a simple controller model that neglects many aspects of program behavior. The performance results show that even this simple model functions effectively in practice. However, we will continue to explore model refinements as our system evolves.

5. FEEDBACK-DIRECTED INLINING

This section describes an extension to the adaptive optimization system to support online feedback-directed inlining. At a high level, the system takes a statistical sample of the method calls in the running application and maintains an approximation to 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 already optimized methods to inline hot call edges. Figure 4 shows the structure of the implementation using the architectural framework of Section 3.

When a thread switch occurs in a method prologue, the system calls the `update` method of an *edge listener* (as well as the method listener as discussed in Section 4.2). This edge listener walks the thread's stack to determine the call site that originated the call. The edge listener creates a tuple identifying the calling edge (specified by the caller, call site, and callee) and inserts this tuple into a buffer.

When the buffer becomes full, the edge listener is temporarily deactivated (its update method will not be called again at a prologue thread switch) and it notifies the *dynamic call graph (DCG) organizer* to wake up and process the buffer. The DCG organizer maintains a dynamic call graph, where each edge corresponds to a tuple value in the buffer. After updating the weights in the dynamic call graph, the DCG organizer clears the buffer, and reactivates the edge listener. The decay organizer, a separate thread, periodically decays the edge weights in the dynamic call graph.

Periodically, the DCG organizer invokes the *adaptive inlining organizer* to recompute adaptive inlining decisions. The adaptive inlining organizer performs two functions. First, it identifies edges in the dynamic call graph whose percentage of samples exceed an edge hotness threshold. These edges are added to an *inlining rules* data structure, which is consulted by the controller when it formulates compilation plans. Any edge in this data structure will be inlined if the calling method is subsequently recompiled, subject to generous size constraints. The system sets the initial edge hotness threshold fairly high, but periodically reduces it until reaching a fixed minimal value. Effectively, this forces inlining to be more conservative during program startup, but allows it to become progressively more aggressive as profiling data accumulates.

The second function of the adaptive inlining organizer is to identify methods that are candidates for further recompilation to enable inlining of hot call edges. To be identified as a recompilation candidate by the inlining organizer, a method must satisfy two criteria. First, the method must be hot, as defined by the hotness threshold used by the hot method organizer. Second, recompiling the method must force a new inlining action, as dictated by the inlining rules data structure.

When the adaptive inlining organizer identifies a method for recompilation, it enqueues an event representing the method for consideration by the controller. The organizer estimates a *boost factor*, an estimate of the greater efficacy of optimization on the method, due to the adaptive inlining rules. The controller incorporates this boost factor into its cost/benefit model described in Section 4.

Many factors can contribute to the expected boost factor, including the elimination of call/return overhead and additional optimizations enabled by inlining. Our current implementation estimates the boost factor based on the fraction of dynamic calls attributed to the call edge in the dynamic call graph and an estimate, based on a previous study of offline profile directed inlining [5], of the benefit of eliminating virtually all calls from the program. In future work, we will examine more sophisticated heuristics involving more detailed analysis of inline candidates or techniques such as Dean and Chambers's inlining trials [22].

As described in more detail in the context of the Self-93 implementation [31], the system should take care when recompiling a method previously compiled with feedback-directed inlining 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.

6. PERFORMANCE EVALUATION

This section experimentally assesses the performance of the current implementation of the Jalapeño Adaptive Optimization System. Section 6.1 describes the experimental setup, the benchmark suite, and the adaptive optimization system configuration parameters used in the experiments. Section 6.2 focuses on the effectiveness of multi-level recompilation described in Section 4. The performance of the feedback-directed inlining subsystem introduced in Section 5 is assessed in Section 6.3. Section 6.4 presents data on adaptive optimization system overhead and Section 6.5 presents data on recompilation behavior.

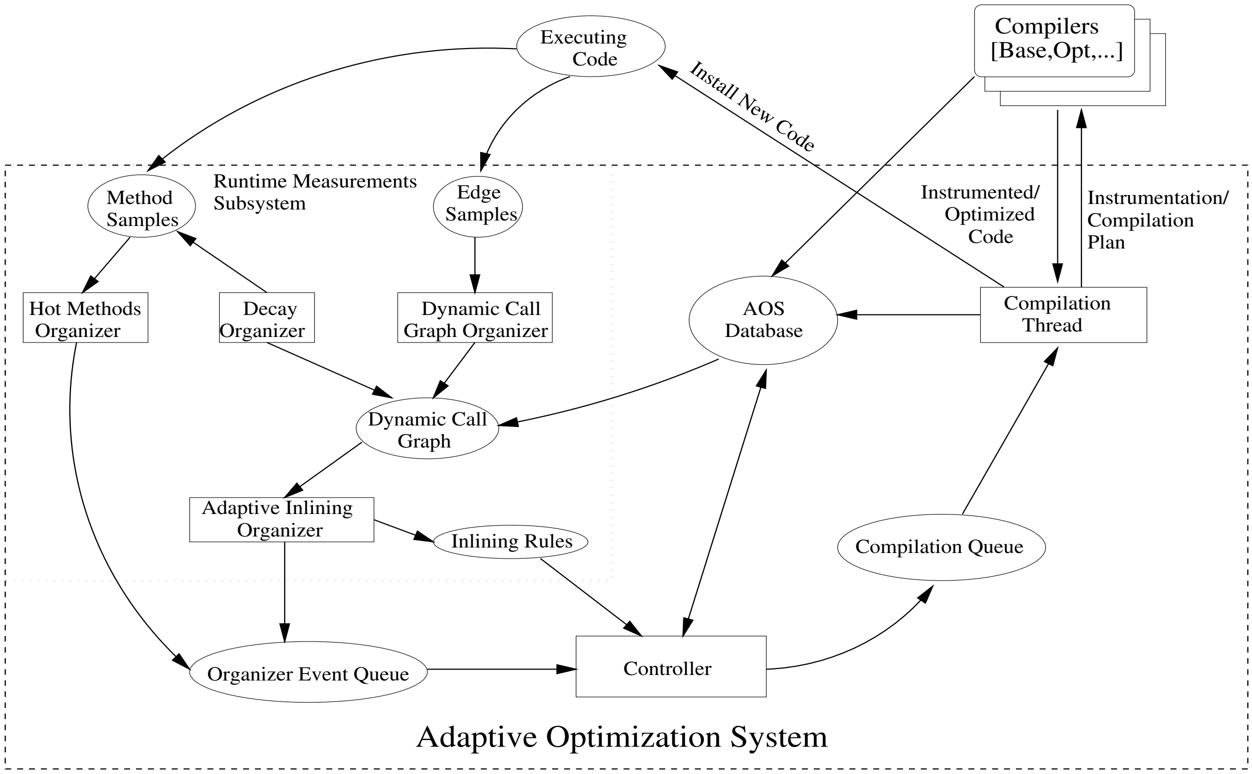

Figure 4: Implementation of feedback-directed inlining in the Jalapeño Adaptive Optimization System

6.1 Experimental Methodology

The performance results in this section were obtained on an IBM F50 Model 7025 with two 333MHz PPC604e processors running AIX v4.3. The system has 1GB of main memory.

All experiments were performed using Jalapeño's nongenerational copying garbage collector. The Jalapeño boot image was compiled using the optimizing compiler at level 2; the optimizing compiler and the adaptive optimization system were included in the boot image. The following controller configuration parameters were used in the experiments:

- Timer interrupts (Section 4.2) were generated every 10 milliseconds.
- The initial recompilation hotness threshold (Section 4.2) was set to 1% and was allowed to vary between 1% and 0.25%.
- The initial recompilation sample size (Section 4.2) was set such that the recompilation organizer would run 2 times a second. The sample size was allowed to vary such that the recompilation organizer could run between 2 times a second and once every 4 seconds.
- The initial inlining edge hotness threshold (Section 5) was set to 1% and was periodically reduced until it reached 0.2%.
- The inlining sample size (Section 5) was set such that the dynamic call graph and adaptive inlining organizers would run once every 2.5 seconds.
- The half-life for method samples is 1.7 seconds. The half-life of edge weights in the dynamic call graph is 7.3 seconds.

We evaluate the system using the SPECjvm98 [20] benchmarks, the Jalapeño optimizing compiler [14], and the Volano benchmark [44]. Table 2 provides a description of each benchmark, the number of classes that comprise the benchmark, and the size, in bytes, of its class files. Previous work [2] has shown that Jalapeño performance on these benchmarks without adaptive compilation roughly matches that of the industry-leading IBM product virtual machine.

We focus on two interesting regimes for adaptive compilation: *program startup* and *steady state*. During program startup, program behavior typically changes rapidly as it dynamically loads classes and initializes data structures. After a while, the program reaches a steady state. We evaluate the startup regime by timing the first run of the SPECjvm98 benchmarks with the size 10 (medium) inputs, and by using the Jalapeño optimizing compiler to compile a `HelloWorld` program. We report the minimum time obtained from five runs of each benchmark (a new JVM is started for each run). For the `Volano` benchmark, we use a configuration that passes 120,000 messages, and runs for roughly 30 seconds on our system.

To measure steady-state performance, on the SPECjvm98 benchmarks, we report the best elapsed time from five runs, *all run during a single JVM execution*, with the size 100 (large) inputs. For the Jalapeño optimizing compiler bench-

Benchmarks	Description	Number of Classes	Size of Class Files (in bytes)
`compress`	An implementation of the Lempel-Ziv compression algorithm	12	17,821
`jess`	Java expert shell system	150	396,536
`db`	Execution of database functions on memory resident data	3	10,156
`javac`	JDK 1.0.2 Java compiler	175	561,463
`mpegaudio`	Decompression of audio files	54	120,182
`mtrt`	Variant of a two-thread raytracing algorithm	25	57,859
`jack`	Java parser generator	55	130,889
`opt-compiler`	Jalapeño optimizing compiler	393	1,378,292
`Volano`	Multithreaded server application to simulate chat rooms	69	209,891

Table 2: The set of benchmarks used to evaluate the Jalapeño Adaptive Optimization System. The first seven rows comprise the suite of SPECjvm98 benchmarks.

mark, we report the best time from five runs of compiling the entire optimizing compiler, consisting of roughly 75,000 lines of Java source code. For the `Volano` benchmark, we report performance in terms of message throughput over a run that passes 1.2 million messages.

6.2 Multi-Level Recompilation

This section evaluates the effectiveness of the adaptive multi-level recompilation system described in Section 4 by comparing its performance to both JIT and simple adaptive single-level configurations of Jalapeño. In the adaptive single-level configurations, the controller compiles all hot methods with the optimizing compiler using a single fixed optimization level.

To allow the experiments to focus on the impact of recompilation decisions, none of the configurations perform any feedback-directed optimizations (i.e., they do not use profiling data to guide specific optimizations). Thus, when the adaptive system chooses to recompile a method at an optimization level, it will compile it the same way a JIT configuration would. For each benchmark we ran the following Jalapeño configurations:

- the baseline compiler as a JIT;
- the optimizing compiler at level 0 as a JIT;
- the optimizing compiler at level 1 as a JIT;
- the optimizing compiler at level 2 as a JIT;
- the adaptive single-level configuration using the optimizing compiler at level 0;
- the adaptive single-level configuration using the optimizing compiler at level 1;
- the adaptive single-level configuration using the optimizing compiler at level 2;
- the adaptive multi-level system using the optimizing compiler at any of its three levels.

The JIT configurations compile each method the first time it executes and never recompile a method. Thus, these configurations only incur the overhead of compilation the first time a method is called. In the adaptive configurations, the baseline compiler compiles each method the first time it executes. However, as the application executes, the adaptive optimization system continuously identifies and recompiles hot methods at higher optimization levels.

Figure 5 shows performance in the startup regime for each benchmark. The graph shows speed relative to the baseline JIT configuration, thus taller bars represent better performance. Execution times for each configuration can be found in the appendix.

The results show that in the startup regime, adaptive recompilation clearly delivers better performance than any of the JIT configurations. The worst adaptive configuration, level 2, improved performance by a mean of 23% compared to the best JIT configuration, at optimization level 0. For four benchmarks, even optimization level 0 is too expensive in compile-time, degrading performance compared to the baseline compiler. These results show that in the startup regime, compile-time overhead plays a large role. For all benchmarks, increasing the optimization level in the JIT configuration causes startup performance to suffer. This property does not hold for the adaptive configurations, where selective compilation allows effective use of higher optimization on several benchmarks. However, the best single level of adaptive optimization varies among the benchmarks between level 0 and level 1. For this reason, overall, the multi-level optimization strategy delivers the best performance of all configurations.

Figure 6 shows performance in the steady-state regime for each benchmark. Again, the graph depicts speed relative to the baseline JIT configuration and the execution times can be found in the appendix.

The results show that the performance of each adaptive configuration is competitive with its JIT configuration counterpart. The multi-level adaptive system delivers the best performance of the adaptive configurations, with overall performance within 2% of the best JIT configuration at optimization level 2. This result is encouraging, since the JIT configuration performs no compilation, profiling, or decision making during the runtime at steady state. The adaptive system also benefits from delaying optimization. During the delay, more of the program loads dynamically, so later op-

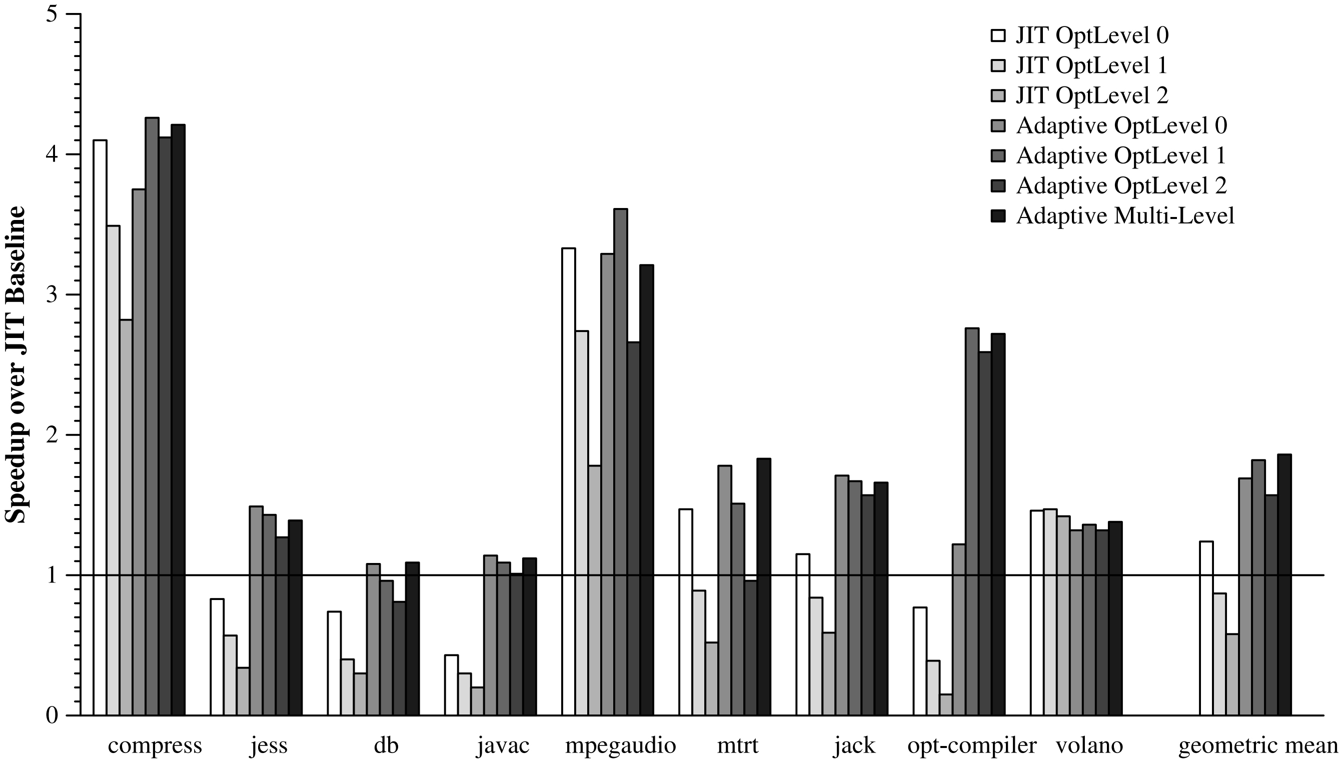

Figure 5: Startup performance

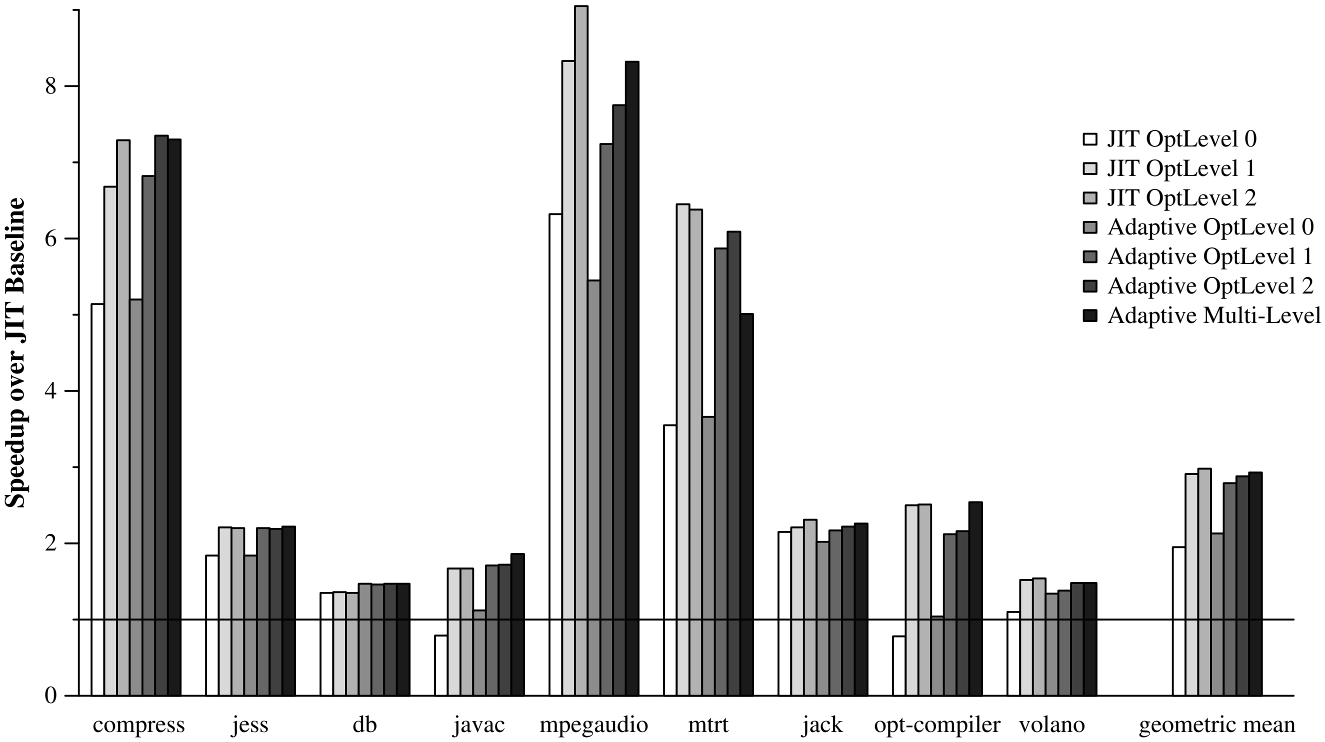

Figure 6: Steady-state performance

timization takes advantage of a greater view of the whole program. In our current compiler, this results in more effective devirtualization and inlining, and less dynamic linking, leading to improved performance over all JIT configurations on some benchmarks.

Both Figures 5 and 6 illustrate that any one fixed strategy does not suit a workload with programs that execute for different lengths of time. For long-running programs, the highest optimization level delivers the best performance for JIT and single-level adaptive configurations. However, for short-running programs, the highest optimization level delivers the *worst* performance. The adaptive multi-level system applies optimizations judiciously, attaining high performance in both the startup and the steady-state regimes.

6.3 Feedback-Directed Inlining

Figure 7 shows the performance impact of feedback-directed inlining in an adaptive multi-level system for both the startup and steady-state regimes. For each benchmark we show the speed relative to the adaptive multi-level system from the previous section. Larger bars represent better performance. Values greater than 1.0 indicate that feedback-directed inlining improved performance over the adaptive multi-level system; values less than 1.0 indicate a performance degradation.

For the startup regime, both `jess` and `javac` significantly improve with feedback-directed inlining, with a 10% and 4.7% performance improvement, respectively. Only `mtrt`'s performance significantly degrades with a 6.7% degradation. Overall, feedback-directed inlining improves performance of the short running programs by 1%. We did not expect to see much performance impact in this regime because of the programs' short execution times. A short execution time does not allow many methods to be compiled at high optimization levels (where inlining occurs), or to be recompiled due to new hot call sites.

For the steady-state regime, feedback-directed inlining consistently improves performance. Feedback-directed inlining improved `jess` by 73% and `mpegaudio` by 22%. The `mtrt` benchmark once again degrades with a 9% degradation. Overall, feedback-directed inlining improves performance by an average of 11%.

6.4 Adaptive System Overhead

Figure 8 illustrates where execution time is spent in the various components of Jalapeño when using the multi-level adaptive system with feedback-directed inlining. The figure characterizes the overhead for two execution regimes: program startup and long-running programs. For this figure, program startup corresponds to first run of the SPECjvm98 benchmarks with input size 10, and long-running corresponds to cumulative timings for five runs of the same benchmarks with input size 100. This differs slightly from the steady-state data, which reports the best time of five runs. The data represents the cumulative time spent executing system components on both processors of the SMP. The fraction of time spent in each thread was collected for all seven SPECjvm98 benchmarks and for both regimes. The average of these fractions was then computed separately for each regime.

The top two pie charts illustrate a coarse-grain break down of total AOS system overhead. These two pie charts demonstrate that the total time spent in the AOS threads is relatively small, averaging 8.6% for program startup and 6.0% for long-running. The similarity in overhead for the two regimes can be attributed to the cost-benefit model of the controller (described in Section 4.3). By estimating future execution time, the model avoids performing too much work at startup, yet allows more time for optimization as program execution continues. Previous versions of the adaptive system that did not use a cost-benefit model tended to spend a larger percentage of time performing optimization in short-running programs. In addition, baseline compilation and garbage collection comprise a small percentage of execution time in both regimes.

The bottom pair of pie charts shows the relative time spent in each AOS thread. The percentage of time spent compiling in both regimes is similar, approximately 50%, again due to the controller model. The organizer threads incur slightly higher overhead for the long-running programs because each program spends more time in a steady state. During this time, the organizer threads continue to process runtime measurements, but insert fewer events in the organizer event queue, thus reducing the controller's computation load.

For both program startup and steady state the AOS overhead (the execution time of the controller and organizer threads) is less than 3.7% of the total execution time (only 2.9% for the long-running regime). This confirms experimental observations of previous research [35, 46].

6.5 Recompilation Decisions

Figure 9 depicts the recompilation decisions made by the controller in the adaptive multi-level system during the steady-state regime. For each benchmark, four bars are shown: each bar represents the percentage of all methods that finished at the given optimization level. The baseline bar represents methods that were never recompiled by the adaptive optimization system. The level 1 and level 2 bars are further subdivided to show how methods reached their final compilation state. In the adaptive multi-level system there are two paths a method can follow to level 1: either the method was directly recompiled at level 1, or the method was first recompiled at level 0 and then recompiled again at level 1. Similarly, there are four possible paths that result in a method being compiled at opt level 2. However, for our benchmarks no method actually took the path that entailed being recompiled at all three of the optimization levels.

The adaptive system recompiled between 12% (`Volano`) and 45% (`javac`) of all methods dynamically compiled. The most commonly selected optimization level was level 1, which is consistent with the model parameters that define the expected benefits and costs of each optimization level. Most of the methods that reached the higher optimization levels (1 and 2) passed through an intermediate stage of optimization before they were recompiled at their ultimate level.

Figure 10 depicts the recompilation activity of the adaptive multi-level system with feedback-directed inlining during a long-running program with phase shifts. In this experiment,

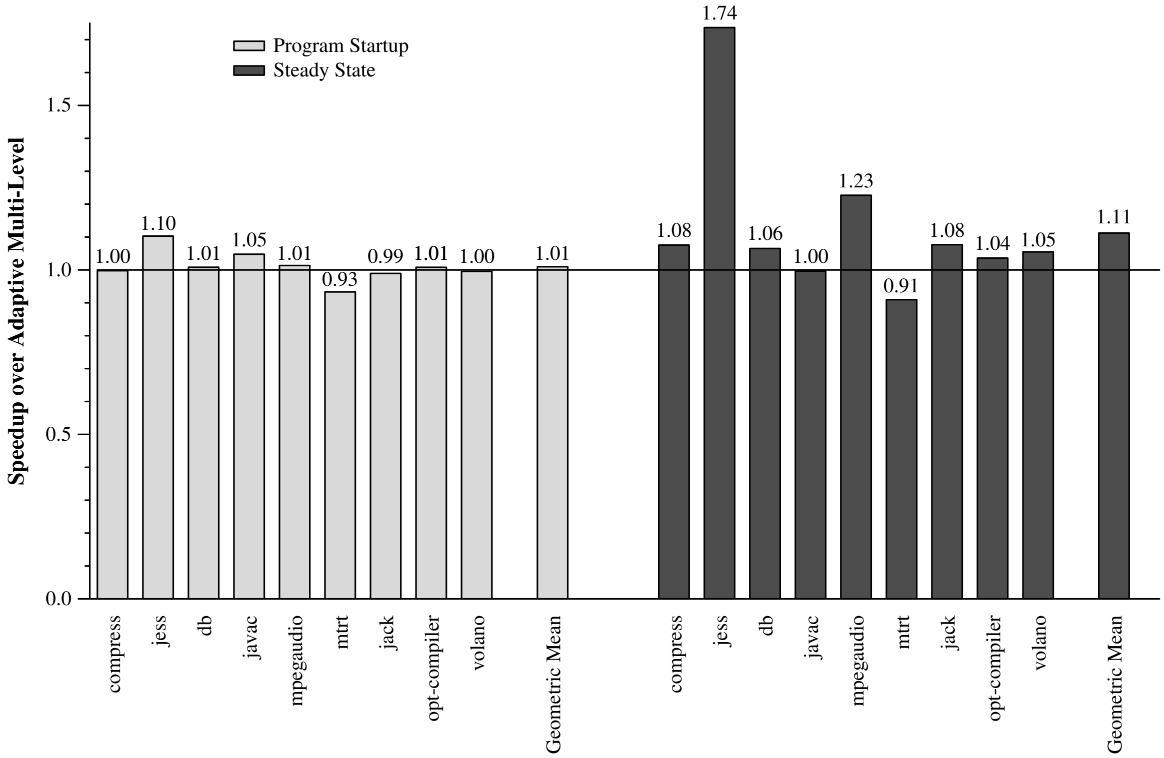

Figure 7: Impact of online feedback-directed inlining

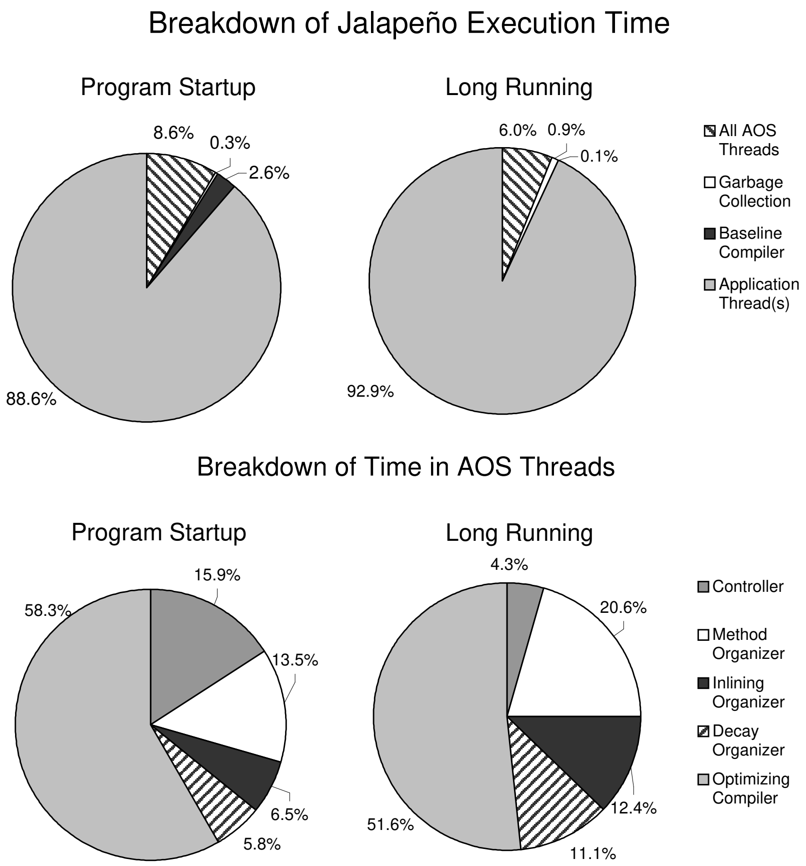

Figure 8: Breakdown of time spent in various Jalapeño threads

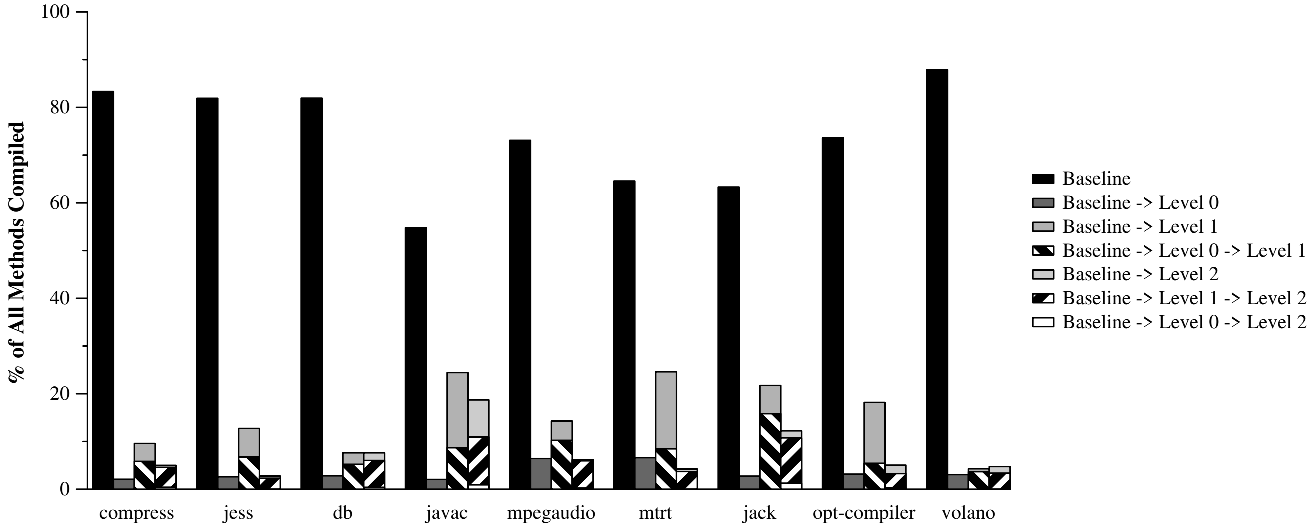

Figure 9: Controller recompilation decisions for the adaptive multi-level system with feedback-directed inlining during the steady-state regime

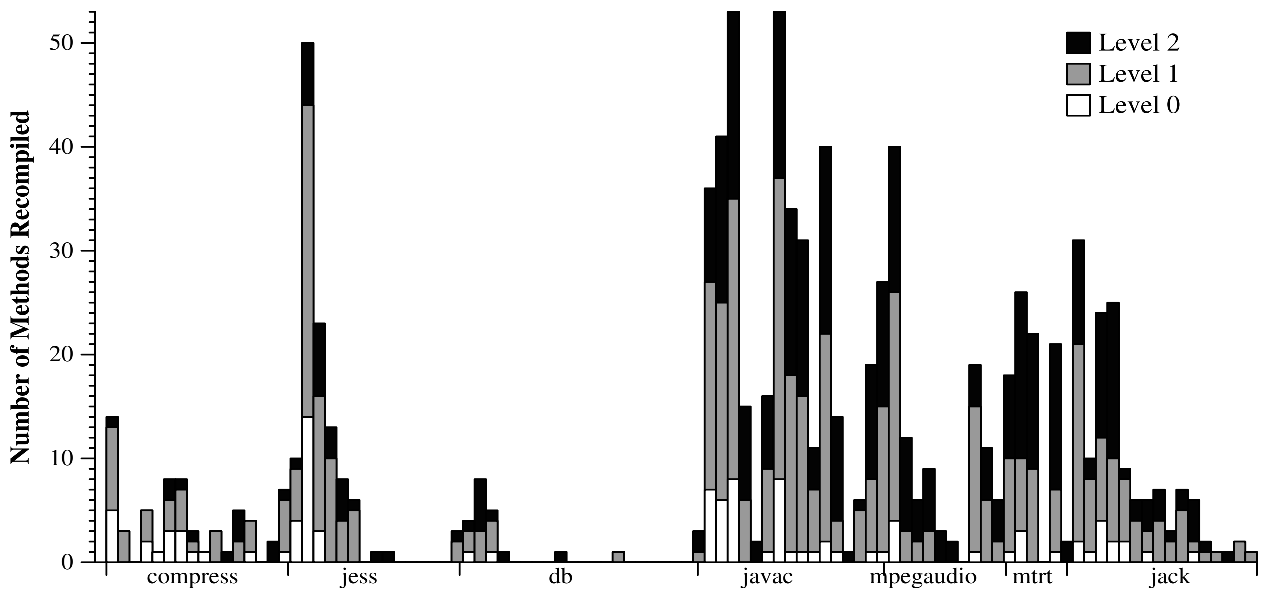

Figure 10: Recompilation activity while running the seven SPECjvm98 benchmarks in the same JVM. Each benchmark is run once using the size 100 inputs. The x-axis represents time partitioned into 100 fixed-size intervals.

each of the seven SPECjvm98 benchmarks was run once with size 100 input in a single JVM. This differs from the previous performance results, which run each benchmark in a fresh JVM. The x-axis of the figure represents time, from system boot to program exit, partitioned into 100 fixed-size intervals. The x-axis is marked to show approximately when each benchmark begins and ends its execution. The y-axis gives the number of recompilations that occurred in each interval. Each bar is subdivided to show the number of recompilations at each optimization level.

As expected, when each new benchmark begins its execution, the set of hot methods changes dramatically, generating a new set of recompilation candidates. This results in the increase in the number of level 0 recompilations at the beginning of each benchmark's execution. As execution of each phase continues, many methods that remain hot graduate to higher optimization levels. Another interesting trend is that later phases initiate more recompilation activity than earlier phases. Two factors cause this increase in recompilation activity. First, `javac` and `jack` have larger working sets of hot methods than do `compress`, `jess`, and `db`. Second, as explained in Section 4.3, the controller assumes that the program will execute for twice as long as it has currently run. Therefore, in later stages of this run the controller selects methods for recompilation more aggressively because it expects to enjoy a longer period of time to recover its compilation costs.

7. DISCUSSION

We have presented the design and implementation of an extensible and high-performance adaptive optimization system. We now present some subjective observations on the system.

The system must manage a substantial volume of data efficiently. We believe that our distributed, asynchronous, object-oriented design serves this purpose in two ways. First, as data passes through the pipeline from raw data to compilation decisions, each successive pipeline phase performs increasingly sophisticated analysis on a decreasing volume of data. Thus, the design helps structure a system that performs sophisticated analysis with reasonably low overhead. Second, by separating functionality into modules, the design separates concerns and allows an extensible architecture, as is common in well-designed object-oriented systems.

The sampling-based online profiling also helps control runtime overhead of the adaptive optimization system. By changing sampling frequencies dynamically, the system can adaptively throttle its own overhead. This behavior would likely be more difficult with compiler-inserted intrusive instrumentation. However, in future work, we will let the controller insert more expensive intrusive instrumentation for limited periods of time, in order to collect more precise information.

Many times, we faced implementation decisions regarding whether to make controller decisions based on an analytic model of program behavior, or whether to introduce ad-hoc tuning parameters to guide decisions. Invariably, tuning the parameters proved more difficult than expected, due to unforseen differences in application behavior. As work on the system progresses, we will move increasingly toward analytic decisions based on first principles, and excise all ad hoc parameters from the implementation. Developing an effective model is a main area for future work, especially in the presence of inlining and multithreading.

We feel that Java served as an effective and productive language for implementing the system. Java's safety properties and memory management eased debugging throughout development. We exploited Java threads and synchronization operations to build the asynchronous system components. However, race conditions remain difficult, and sound concurrent system design is vital.

Some previous systems (e.g. [28]) have relied on limited, but fast compiler stubs to perform runtime optimization. In contrast, we chose a different design point with a full, general optimizing compiler that can recompile any part of the application code, libraries, or the VM itself. While our approach potentially introduces more runtime overhead, we have demonstrated techniques that limit the overhead and achieve good performance in a full-blown Java Virtual Machine.

Finally, we note an oversimplification in the analytic model of Section 4.3. This model assumes 100% CPU utilization, and decides whether to allocate CPU cycles to the application or to the compiler. However, in hindsight, we realize that this assumption is not valid for the subset of our experiments that use single-threaded codes on a multiprocessor. To correct this, we plan to enhance the analytic model to reduce the expected cost of compilation if the system detects idle processors. We do not expect the cost of compilation to be zero, due to memory system contention, but clearly the compiler can use idle cycles more aggressively than our current system. We expect this change will improve our performance for scenarios with idle cycles, as the controller will be significantly more aggressive in scheduling compilation.

8. RELATED WORK

Previous adaptive virtual machines have used method invocation counters to identify and optimize online program hot spots. Hölzle and Unger [32] describe the SELF-93 system, an adaptive optimization system for SELF. The goal of the system is to avoid long pauses in interactive applications by optimizing only the performance-critical parts of the application. Method invocation counters with an exponential decay mechanism are used to identify candidates for optimization. In addition, Self-93 used polymorphic inline caches (PICs) to gather context-sensitive receiver class distributions to guide class prediction and inlining. In SELF-93, optimized methods do not contain invocation counters. Therefore, although SELF-93 could reoptimize an already optimized method, [5] its counter-based profiling mechanism would be less effective for identifying hot regions of already optimized code, thus making it more difficult to implement

[5] Recompilation of an optimized method could be triggered when it calls an unoptimized method whose invocation counter exceeds the threshold on the call. However, this situation would typically arise only after an application phase shift, since if the call edge between the two methods had been hot when the caller was optimized, the callee would have either been inlined into the caller or optimized itself.

an effective multi-level optimization strategy. In contrast, our sampling technique allows optimized, as well as unoptimized, methods to be sampled continuously and fairly.

The HotSpot JVM [34] and the IBM Java Just-in-Time compiler (version 3.0) [43] are adaptive systems for Java. Both systems initially interpret an application and later compile performance-critical code. The IBM JIT uses method invocation counters augmented to accommodate loops in methods to trigger compilation. Details of the HotSpot compilation system are not provided.

Bala et al. [9] describe Dynamo, a transparent dynamic optimizer that performs optimizations at runtime on a native binary. Dynamo initially interprets the program, keeping counters to identify sections of code called hot *traces*. These sections are then optimized and written as an executable.

Other dynamic optimization research have used non-sample-based profiling techniques to identify and optimize online program hot spots. Burger and Dybvig [12, 13] explore profile-driven dynamic recompilation in Scheme, with an implementation of a basic block reordering optimization. Unfortunately, the overhead for basic block edge profiling, even after applying basic block reordering, is 27% of the execution time. Furthermore, they require programmer intervention to determine how to throttle profiling to reduce its overhead. In contrast, our sampling technique's overhead is low enough (Section 6) that sampling can be continuous, and our architecture for adaptive optimization does not require any programmer intervention.

Hansen [30] describes an adaptive FORTRAN system that makes automatic optimization decisions. When a basic block counter reaches a threshold, the basic block is reoptimized and moved to the next *optimization state*, where more aggressive optimizations are performed.

Perhaps the work that is most similar to our work is that of Kistler [35]. Kistler presents a continuous program optimization architecture for Oberon that allows "up-to-the-minute" profiling information to be used in program reoptimization. Kistler concludes that continuous optimization can produce better code than can be achieved with offline compilation, regardless of whether profiling information is used in the latter. There are some interesting differences in the overall architectural design of his and our systems. Consider, for example, the interactions between the controller (manager) and runtime measurements (profiler). In his design, this interaction is based on a message protocol in which the "individual profiling components are autonomous." In our design, an organizer thread processes the profiling data for consumption by the controller. Adding a new organizer thread allows alternative processing of raw profiling data without requiring a new profiler. Although Kistler evaluates the time overhead of periodically sampling the program counter, it is not obvious what optimizations are driven by this sampling technique, and therefore he does not evaluate the technique's usefulness. In contrast, we are able to demonstrate the effect on performance of optimizations that are driven by our sample-based profiling. Finally, it is hard to compare the results of the two systems due to the difference in languages, the maturity of the systems, the different target optimizations, and the selection of benchmarks. For example, Kistler's work does not measure the difference in a program's behavior between its startup and steady states.

Whaley [46] implemented sample-based calling-context-sensitive profiling in a production JIT compiler. He demonstrated that the overhead of this sample-based approach was low enough to run continuously, and that this sampling technique is stable over repeated runs of the same benchmark with the same set of inputs. However, no optimizations are driven by the sampled data and he does not, therefore, evaluate the sampling technique's usefulness.

Another area of research considers runtime optimizations that exploit invariant runtime values. Because such values are not statically determinable, these systems provide optimization opportunities not available with static compilation. Some systems include *DyC* [8, 27, 28], *Tempo* [38] (based on C), Fabius [37] (based on ML), and Consel and Noel's work [19] which takes a partial evaluation approach. The *tcc* system [39] provides a mechanism to specify and compose arbitrary expressions and statements at runtime. The main disadvantage of these techniques is that they rely on programmer directives to identify the regions of code to be optimized.

Other work [17, 29] has explored offline profile-directed compilation schemes that use one or more profiles from previous runs of an application as feedback into a compiler to make better optimization decisions for future executions. In addition, there are fully automated profiling systems that use transparent low overhead profiling to improve performance of future executions. Such systems include Digital FX!32 [33], *Morph* [48], and *DCPI* [4]. In contrast, this paper focuses on using profiling information of an application's execution to help optimize that same execution of the application. Nevertheless, our architecture can be extended to handle offline profile-directed compilation schemes; for example, the controller could use compilation plans and profiling data from previous executions of an application.

There are also nonadaptive systems that perform compilation/optimization at runtime to avoid the cost of interpretation. This includes early work such as the Smalltalk-80 [24] and Self-91 [16] systems, as well many of today's JIT Java compilers [1, 36, 47].

Previous studies have evaluated the viability of selective optimization. Arnold et al. [6] used the Jalapeño JVM to quantify the performance potential of selective optimization. Their results confirmed that selective optimization has the potential to significantly outperform a fixed JIT strategy and can even approach the performance of static compilation for longer-running benchmarks. Radhakrishnan et al. [41] used the Kaffe Virtual Machine to establish the maximum performance improvement possible by interpreting, rather than compiling, cold methods for the SPECjvm98 with input size 1.

Serrano et al. [42] describe a *quasi-static* compilation approach using Jalapeño that 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 JVM terminates. When a compiled method is needed during subsequent executions of the application, the dynamic compiler is not automatically invoked as is the case in our adaptive system. The system checks to see if a precompiled image of the method exists. If an image exists, the system relocates and links the code into the new JVM environment. If no image exists, the dynamic compiler compiles the method.

9. CONCLUSIONS

We have described the design and implementation of the Jalapeño Adaptive Optimization System, a general and extensible architecture to support online feedback-directed optimization. The system is written completely in Java, allowing the techniques described to apply not only to the application, but also to the virtual machine and its components. We have presented the current instantiation of this framework, relying on sample-based online profiling, multiple optimization levels, and sample-driven online feedback-directed inlining.

Performance results demonstrate that low-overhead sampling techniques can effectively drive profile-directed online recompilation of Java programs. The adaptive system, including recompilation, on average introduces less than 10% overhead and delivers total system performance competitive with the best alternative strategies considered. We have further demonstrated that a multi-level optimization strategy can deliver robust performance in both startup and steady-state program regimes, competitive with the best alternative in each regime. Finally, we have demonstrated that sampling techniques can effectively drive feedback-directed inlining, delivering performance improvements of 11% on average and up to 73%.

This system provides a flexible infrastructure for future research on online optimization. Future research topics include automatic specialization, profile-directed memory layout optimizations, refinements to the recompilation analytic model, and consideration of larger server codes based on IBM middleware products. We anticipate that the Jalapeño Adaptive Optimization System will play a key role in our efforts to improve Java server performance.

Acknowledgments

We thank the Jalapeño team members [2] for their work in developing the system used to conduct this research. We thank Vivek Sarkar for his support of this work. We also thank Barbara Ryder for her support of Matthew Arnold's involvement in this work. Michael Burke, Laureen Treacy, and the anonymous OOPSLA reviewers provided valuable suggestions and feedback on the presentation of this work.

10. REFERENCES

[1] A.-R. Adl-Tabatabai, M. Cierniak, C.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a Just-in-Time Java compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 280–290, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), May 1998.

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

[3] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, D. Lieber, S. Smith, and T. Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, 1999.

[4] J. M. Andersen, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? Technical Note 1997-016a, Digital Systems Research Center, www.research.digital.com/SRC, Sept. 1997.

[5] M. Arnold, S. Fink, V. Sarkar, and P. Sweeney. A comparative study of static and dynamic heuristics for inlining. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000.

[6] M. Arnold, M. Hind, and B. G. Ryder. An empirical study of selective optimization. In *13th International Workshop on Languages and Compilers for Parallel Computing*, Aug. 2000.

[7] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical Report RC 21789, IBM T.J. Watson Research Center, July 2000.

[8] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, 21–24 May 1996. *SIGPLAN Notices* 31(5), May 1996.

[9] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[10] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, pages 300–313, Albuquerque, New Mexico, 23–25 June 1993. *SIGPLAN Notices* 28(6), June 1993.

[11] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[12] R. G. Burger. *Efficient Compilation and Profile-Driven Dynamic Recompilation in Scheme.* PhD thesis, Indiana University, 1997.

[13] R. G. Burger and R. K. Dybvig. An infrastructure for profile-driven dynamic recompilation. In *ICCL'98, the IEEE Computer Society International Conference on Computer Languages*, May 1998.

[14] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[15] B. Calder, P. Feller, and A. Eustace. Value profiling. In *the 30th International Symposium on Microarchitecture*, pages 259–269, Dec. 1997.

[16] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, Nov. 1991. *SIGPLAN Notices* 26(11).

[17] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software – Practice and Experience*, 22(5):349–369, May 1992.

[18] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[19] C. Consel and F. Noël. 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*, pages 145–156, Jan. 1996.

[20] T. S. P. E. Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method for computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[22] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the ACM Conference on LISP and Functional Programming '94*, pages 273–282, Orlando, FL, June 1994.

[23] D. Detlefs and O. Agesen. Inlining of virtual methods. In *the 13th European Conference on Object-Oriented Programming*, 1999.

[24] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Annual ACM Symposium on the Principles of Programming Languages*, pages 297–302, Jan. 1984.

[25] S. Fink, K. Knobe, and V. Sarkar. Unified Analysis of Array and Object References in Strongly Typed Languages. In *Seventh International Static Analysis Symposium (2000)*, June 2000.

[26] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[27] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report TR-97-03-03, University of Washington, Department of Computer Science and Engineering, Mar. 1997.

[28] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 293–304, 1999.

[29] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, Oct. 1995.

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

[31] U. Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* PhD thesis, Stanford University, Aug. 1994.

[32] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.

[33] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary transslation. *Digital Technical Journal*, 9(1):3–12, Jan. 1997.

[34] The Java Hotspot performance engine architecture. White paper available at http://java.sun.com/products/hotspot/whitepaper.html, Apr. 1999.

[35] T. P. Kistler. *Continuous Program Optimization.* PhD thesis, University of California, Irvine, 1999.

[36] A. Krall. Efficient JavaVM Just-in-Time compilation. In J.-L. Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Oct. 1998.

[37] M. Leone and P. Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):1–5, Sept. 1998. Article 23.

[38] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 281–292, 1999.

[39] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 109–121, Las Vegas, Nevada, 15–18 June 1997. *SIGPLAN Notices* 32(5), May 1997.

[40] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.

[41] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam. Architectural issues in Java runtime systems. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture (HPCA-6)*, pages 387–398, Toulouse, France, Jan. 2000.

[42] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quasi-static compilation in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.

[43] T. Suganama, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systems Journal*, 39(1), 2000.

[44] VolanoMark 2.1. `http://www.volano.com/benchmarks.html`.

[45] J. Whaley. Dynamic optimization through the use of automatic runtime specialization. M.eng., Massachussetts Institute of Technology, May 1999.

[46] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *ACM 2000 Java Grande Conference*, June 2000.

[47] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation. In *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[48] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automated profiling and optimization. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, Operating Systems Review, 31(5), pages 15–26, Oct. 5–8 1997.

APPENDIX

Tables 3 and 4 give the absolute execution data for the startup and steady-state regimes for the bar charts presented in Sections 6.2 and 6.3.

Benchmarks	Base Line	JIT			Adaptive				
		OPT 0	OPT 1	OPT 2	OPT 0	OPT 1	OPT 2	multi	multi+AI
compress	24.987	6.096	7.156	8.868	6.660	5.860	6.071	5.939	5.953
jess	7.398	8.955	12.957	21.473	4.960	5.158	5.824	5.305	4.810
db	1.907	2.582	4.788	6.282	1.761	1.982	2.355	1.745	1.731
javac	6.331	14.627	21.067	31.844	5.533	5.823	6.295	5.670	5.412
mpegaudio	25.794	7.736	9.413	14.472	7.838	7.153	9.714	8.028	7.922
mtrt	6.444	4.380	7.221	12.357	3.621	4.266	6.727	3.524	3.776
jack	11.879	10.318	14.063	20.015	6.933	7.092	7.554	7.139	7.216
opt-compiler	12.808	16.740	33.014	83.084	10.466	4.639	4.937	4.709	4.672
Volano	3045	4429	4468	4312	4007	4155	4019	4185	4204

Table 3: Performance of the benchmarks in the startup regime. All results are in seconds, except `Volano`, with units of messages per second (larger is better).

Benchmarks	Base Line	JIT			Adaptive				
		OPT 0	OPT 1	OPT 2	OPT 0	OPT 1	OPT 2	multi	multi+AI
compress	276.586	53.789	41.422	37.945	53.235	40.582	37.646	37.882	35.221
jess	63.989	34.730	29.018	29.023	34.708	29.040	29.185	28.813	16.591
db	102.863	76.235	75.790	75.964	70.090	70.663	69.885	70.193	65.898
javac	73.461	92.905	44.004	43.908	65.687	42.869	42.649	39.425	39.558
mpegaudio	235.265	37.213	28.236	25.997	43.175	32.517	30.357	28.271	23.043
mtrt	57.017	16.050	8.839	8.940	15.572	9.719	9.355	11.379	12.511
jack	87.970	40.951	39.726	38.036	43.526	40.603	39.658	38.877	36.107
opt-compiler	267.822	345.557	107.004	106.745	257.773	126.083	124.270	105.292	101.649
Volano	3375	3697	5134	5194	4535	4652	4994	5272	5000

Table 4: Performance of the benchmarks in the steady-state regime. All results are in seconds, except `Volano`, with units of messages per second (larger is better).