Adaptive Optimization in the Jalapeño JVM: The Controller's Analytical Model

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

This paper provides details of the component of the Jalapeño adaptive optimization system that determines what methods to optimize. This component, called the *controller*, can choose from one of several optimization levels. In the current implementation, the controller uses a simple cost/benefit analysis to drive adaptive compilation decisions. It has been demonstrated that even this simple analytic model can achieve reasonable performance compared to various JIT compilation scenarios in both startup and steady-state program regimes.

This paper outlines several open questions in developing a more accurate controller model. We present two experiments that study the effects of how the current model predicts future execution from the past, a limited experimental evaluation of stability of the current model across applications, and describe our ongoing efforts to improve the Jalapeño controller.

1. INTRODUCTION

Many virtual machines use two modes of execution to reconcile the tension between responsiveness and aggressive dynamic compilation. The first mode, typically an interpreter or a fast just-in-time (JIT) compiler, focuses on responsive execution; whereas the second mode, typically an optimizing compiler, strives to achieve improved execution performance. Because the second mode may incur significant overhead, it typically optimizes a dynamically selected subset of all executed methods. Examples of such systems include Self-93 [14], the HotSpot JVM [15], the IBM Java Just-in-Time compiler [19], Intel's JUDO [10] system, and the Jalapeño JVM [3] from IBM Research.

Previously [3], we described the current implementation of the adaptive optimization system in Jalapeño. In this paper, we provide details of the key component of the system, the *controller*. The controller determines which methods to optimize, and what optimizations to apply to each method. The previous work showed that a simple analytical model, based on naive cost and benefit estimates, obtained comparable performance to the best JIT strategy for a range of workloads.

FDDO-3, the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization, Monterey, California, December 10, 2000

From our experiences in implementing the system, we have become convinced that making adaptive compilation decisions based on even a limited analytic model leads to more predictable and robust performance than using ad-hoc constants to drive online compilation. This strategy differs from the strategy employed in many current product JVMs. We anticipate that the design of an effective model will become even more important as generated code quality improves, and other Java performance problems are resolved.

This paper examines the Jalapeño controller's analytic model in more detail. We present three experimental studies, which focus on some aspects of the analytic model related to its ability to predict the future based on the past and its stability across several applications. We also provide a discussion of the outstanding research issues for developing an analytic controller model, and describe our ongoing experimental plan to resolve these open questions.

Section 2 provides background on the Jalapeño JVM and its adaptive optimization system. Section 3 describes the current cost/benefit model used by Jalapeño. Section 4 discusses issues related to this model and assesses its strengths and weaknesses. It also illustrates how varying some of its parameters can affect performance. Section 5 discusses related work and Section 6 offers our conclusions.

2. BACKGROUND

A comprehensive description of Jalapeño appears in [1]. In this section we highlight the characteristics of Jalapeño that are most relevant to this work.

Jalapeño [1, 2] is a research JVM being developed at the IBM Watson Research Center. Jalapeño is written in Java [2], enabling adaptive optimizations to be applied to both the application and the JVM. Because Jalapeño is targeted for server applications it employs a compile-only strategy, i.e., it compiles all methods to native code before they execute. The system currently

includes two fully operational compilers.

- The *baseline* compiler translates bytecodes directly into native code by simulating Java's operand stack without performing register allocation. This reference compiler generates native code that performs only slightly better than bytecode interpretation [1].
- The *optimizing* compiler [8] performs a variety of optimizations on an intermediate representation. The compiler uses linear scan register allocation [18], an efficient and effective register allocator. The compiler's optimizations are grouped into several levels
  - *Level 0* consists mainly of on-the-fly optimizations performed during the translation to the intermediate representation. These optimizations include 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* enhances level 0 with additional local optimizations such as common subexpression elimination, array bounds check elimination, and redundant load elimination. It performs 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* enhances level 1 with flow-sensitive optimizations based on SSA form. In addition to traditional SSA optimizations on scalar variables [11], the system also uses an extended version of Array SSA form [13] to perform redundant load elimination and array bounds check elimination [7].

Section 3 quantifies the compilation costs and performance improvements for these compilers.

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. 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 2.1 discusses how we exploit this mechanism in the current version of our system.

[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 [12].

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 1 depicts the internal structure of the adaptive optimization system and the interactions between its components. The next sections discuss this figure in more detail.

2.1 Runtime Measurements Subsystem

The runtime measurements subsystem gathers information about the executing methods, summarizes the information, and then passes the summary along to the controller. Figure 1 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 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 for later queries by other AOS components.

This architecture can support a variety of measurement techniques, including hardware performance monitors, call stack sampling [20, 4], and compiler-inserted instrumentation such as invocation counters, basic block edge or path profiles [6], and value profiles [9].

2.1.1 The Current System

The current system employs a sampling technique that leverages existing mechanisms in the Jalapeño JVM. Namely, whenever the scheduler is entered because a timer interrupt has occurred and the executing thread has reached a compiler-generated yield point, instrumentation in the JVM code records the current method before it switches to a new thread. Currently these timer interrupts occur every 10 milliseconds, resulting in roughly 100 samples/second.

Two organizers, implemented as Java threads, periodically process these samples. The *hot method orga-*

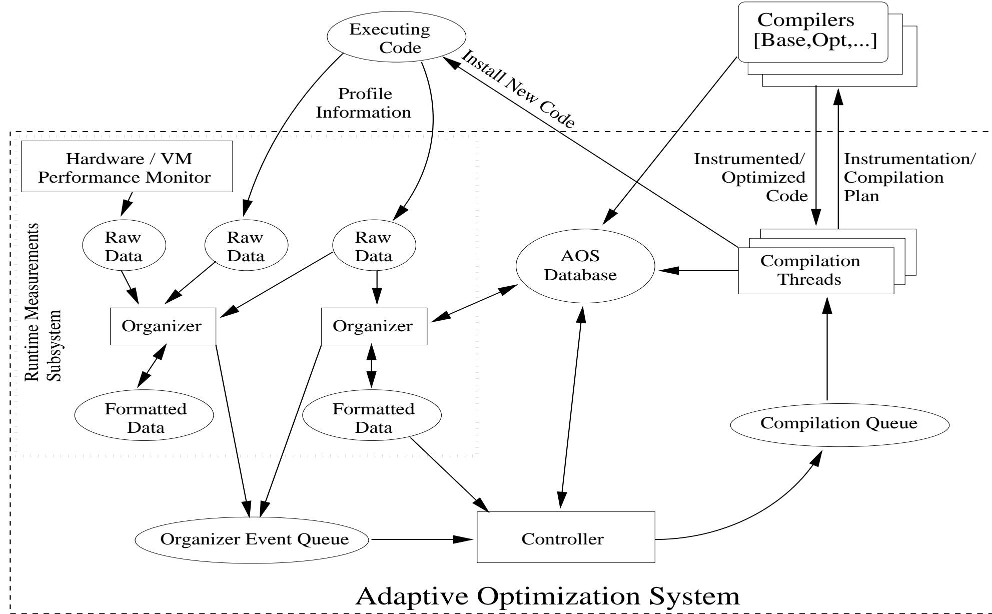


Figure 1: Architecture of the Jalapeño Adaptive Optimization System

nizer searches for methods with a percentage of samples above a certain threshold. The current system adaptively varies this threshold between 0.25% to 1% depending on the amount of recompilations the controller schedules. The initial value is 1%. The hot method organizer runs between twice a second and once every four seconds. The frequency is adaptively varied by the controller.

The second organizer is the *adaptive inlining organizer*, which is used to guide inlining decisions. More specifically, this organizer searches for hot call edges, i.e., a caller-callee method pair where there is a high frequency of samples attributed to the entry of the callee. Currently, this hotness threshold is initialized to 1% and is periodically reduced by the controller until it reaches 0.2%. The adaptive inlining organizer runs once every 2.5 seconds.

For both method and call edge samples, we currently use a decay mechanism to weight recent behavior more heavily, assuming that the recent program behavior is a better indication of future execution. Currently, the method and call edge samples are decayed every 100 samples at a rate of 1.5 and 1.1, respectively. This results in a half-life of 1.7 seconds for method samples and 7.3 seconds for call edge samples.

2.2 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 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 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. This model is described separately in Section 3.

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.

In the current implementation the recompilation subsystem is used only to improve method performance, i.e., it is not yet used to insert instrumentation. Furthermore, only one online feedback-directed optimiza-

tion is present, the call edge samples described in Section 2.1 are used to tailor inlining decisions.

2.3 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.

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.

3. CURRENT CONTROLLER MODEL

The central role of the controller is to determine if it is profitable to recompile a hot method with additional optimizations, and if so, which optimization level to use.[2] The controller make this decision using a cost/benefit analysis. This section describes the current analysis.

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.

[2]A hot method is a method that exceeded the hotness threshold described in Section 2.1.

[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.

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 not to recompile.

Clearly, the factors in this model are unknowable in practice. The process of estimating future costs and benefits is an 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 n seconds, the controller assumes it will run for n more seconds. Define T_f to be the future expected running time of the program.

As described in Section 2.1 the system uses a weighted average of 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,

$$T_i = T_f * P_m \quad (1)$$

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.

The system estimates the effectiveness of each optimization level as a parameterized constant specified at boot time. 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

$$T_j = T_i * S_i / S_j \quad (2)$$

Like optimization effectiveness, the system uses a linear model of the compilation speed for each optimization level, as a function of method size. These values are also specified at boot time.

Our simple analytical model takes two parameters for a particular optimization level: what was the cost to compile, C, and what was the expected speedup, S. Table 1 presents the compilation rate (bytecodes/millisec) that is used by C, and speedup, S, of the SPECjvm98 benchmarks run with input size 100. The numbers in this table are gathered using a configuration that compiles (with the designated compiler) all invoked meth-

Table 1: Compilation rate and speedup of the SPECjvm98 benchmarks run with input size 100. This configuration is not adaptive; it compiles all invoked methods and no profiling or recompilation occurs. Speedup is measured as the best of five runs relative to a baseline-compiled system.

	Compilation Rate (bc/msec)				Speedup over Baseline			
Benchmark	Baseline	Opt 0	Opt 1	Opt 2	Baseline	Opt 0	Opt 1	Opt 2
compress	318.76	9.59	3.16	1.69	1.00	5.42	6.92	7.50
jess	287.16	9.16	3.56	1.73	1.00	3.10	5.14	5.33
db	350.00	9.55	3.20	1.62	1.00	2.54	2.69	2.90
javac	327.00	10.00	4.42	1.87	1.00	1.21	3.14	3.46
mpeg	479.16	10.12	4.00	2.08	1.00	7.00	10.34	11.69
mtrt	336.38	9.33	3.43	1.70	1.00	3.87	6.57	6.68
jack	369.09	9.23	3.98	1.81	1.00	3.43	4.22	4.73
Geo Mean	348.45	9.56	3.65	1.78	1.00	3.36	5.07	5.48

ods with no profiling or recompilation occurring, i.e., a non-adaptive system. Speedup is reported as the best of five runs relative to a baseline-compiled system. For example, on average, the baseline compiler compiles methods at a rate of 348.45 bytecodes/millisecond, whereas the optimizing compiler at opt level 1 is about 100 times slower (3.65 bytecodes/millisecond). On average, this level of the optimizing compiler, level 1, provides a speedup of 5.07 over baseline compiled code.

The average values, given in the last line, are used as the default parameters to the analytical model. Thus, by default, the system uses rather coarse values to estimate the cost and benefits of optimizing a method. It uses an average over the seven benchmarks, which are averaged over the methods of the benchmark.

4. TOWARDS A BETTER MODEL

As described above, the current implementation of the controller model rests on many simplifying questions. In ongoing and future work, we are attempting to evaluate the analytical model, and if needed develop a less abstract model that delivers better performance.

There are many potential topics to explore in this area, and it may be difficult to decide when the model is "good enough." In this section, we outline some topics in model development that we are currently exploring and present experimental evaluations that address these topics.

4.1 Experimental Setup

The experimental evaluations presented in this section were performed 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 non-generational 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.

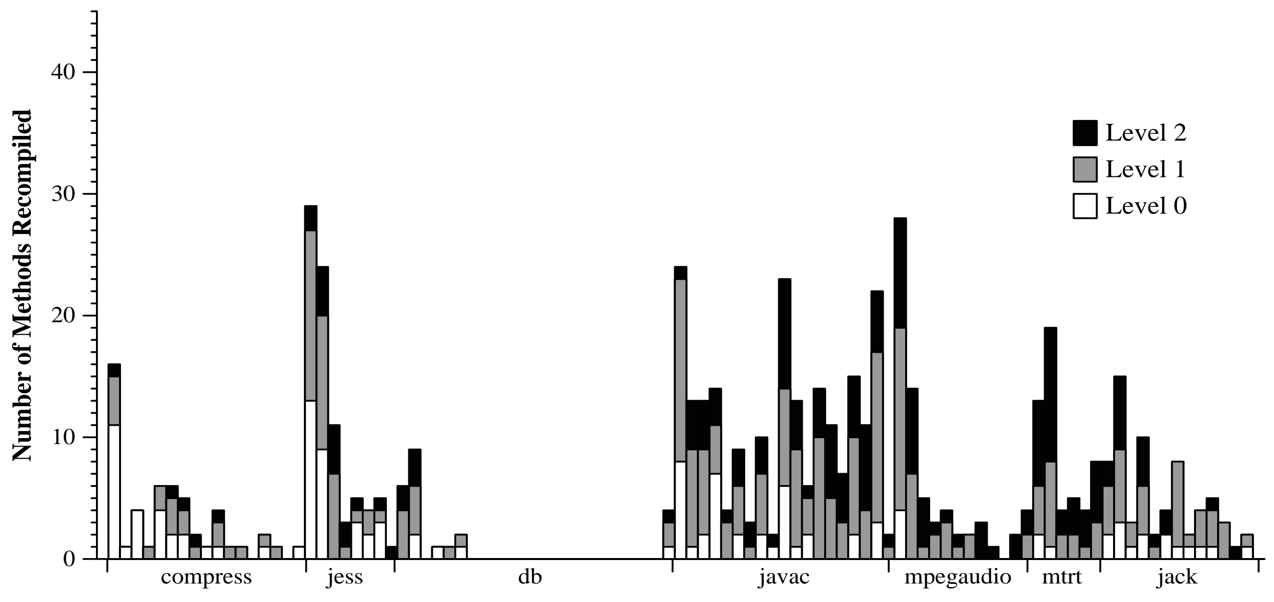
4.2 Predicting the future from the past

The current model implementation assumes that the future running time of a program is twice the current running time. To understand how effective this heuristic is in modeling the behavior of software execution, we ran two experiments.

The first experiment, presented in Figure 2, depicts two instances of recompilation activity of the adaptive multi-level system during a long-running program with phase shifts. In this experiment, each of the seven SPECjvm98 benchmarks was run once with size 100 input in a single JVM. The graphs differ only in the order in which the benchmarks were run. The first graph represents the "original" order of the benchmarks, while the second graph represents a "swapped" order. 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.

Although both executions took approximately the same time to complete (what the original ordered execution gained in run time, it lost in compilation time), the compilation activity for individual benchmarks changed significantly. Consider `javac`. When `javac` is the first benchmark to execute, far less methods are compiled at level 2 than when it is the fourth benchmark. When `javac` is executed later, the controller assumes that it will execute longer because the heuristic assumes that the cumulative execution, i.e., `javac` and the three benchmarks that preceeded it, will continue for twice as long as it has currently executed. Because `javac` is expected to execute longer, the controller is more aggressive in optimizing the code.[5] This results in higher

[5]Demarcating the beginning of a benchmark is currently an imprecise art. It requires inspecting a log file for the first time that a method in a benchmark becomes hot. This imprecision may account for irregularities in benchmark execution times.



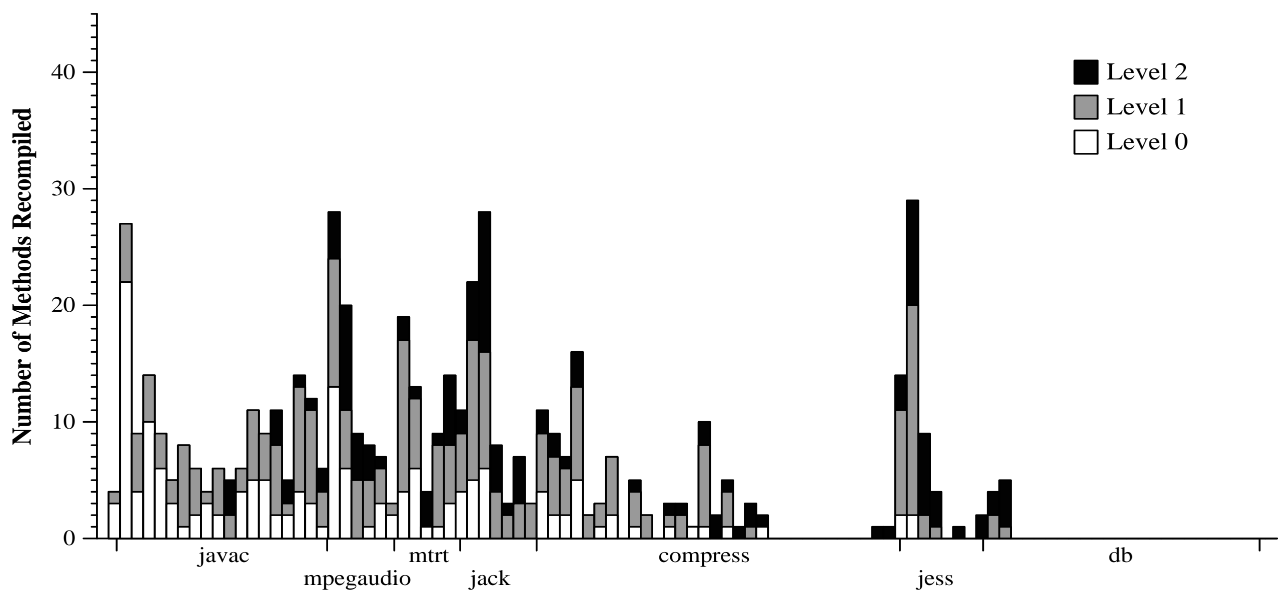


Figure 2: Recompilation activity while running the seven SPECjvm98 benchmarks in the same JVM. Each benchmark is run once using the size 100 inputs. The x-axis represents time partitioned into 100 fixed-size intervals. The second graph differs from the first only in the order in which the benchmarks were run.

javac configurations	Total Time	Threads			Compiled				
		main	compile	gc	Total	Methods	Opt 0	Opt 1	Opt 2
`heuristic`	57.67	49.94	4.98	2.75	266	205	93	99:52	13:9
`exact`	57.15	49.42	4.90	2.83	225	188	78	94:29	16:8
`large`	67.53	51.93	11.55	4.05	66	66	0	0	66:0

Table 2: This table presents the runtime behavior of `javac` when the application's predicted execution time varies.

jack configurations	Total Time	Threads			Compiled				
		main	compile	gc	Total	Methods	Opt 0	Opt 1	Opt 2
`heuristic`	43.85	40.82	1.83	1.20	159	123	76	44:33	3:3
`exact`	42.79	39.80	1.79	1.20	121	98	47	45:21	6:2
`large`	46.01	42.35	2.39	1.27	23	23	0	0	23:0

Table 3: This table presents the runtime behavior of `jack` when the application's predicted execution time varies.

optimization levels and longer compilation times. This behavior illustrates a weakness with the heuristic: the heuristic does not accurately account for phase shifts during the lifetime of the VM. In this experiment, the start of each benchmark represents a phase shift.

For programs with phase shifts, such as VMs that serve as web servers and thus have a variable mix of code that they execute over time, it may make more sense to compute the future expected running time of the program as twice the current running time since the last phase shift. We are exploring techniques to identify and predict phase shifts, while preserving worst-case bounds on lost efficiency.

Our second experiment explores the effects of predicting the future execution time of an application. We modified our system to take as a command line argument the expected execution time of an application, and then varied this argument's value to observe an application's runtime behavior with respect to our heuristic.

Tables 2 and 3 show the results for `javac` and `jack`. The first row illustrates the runtime behavior of the application when the heuristic is used. The second row, denoted `exact`, illustrates the runtime behavior when the application's execution time is known in advance. The third row, denoted `large`, illustrates the runtime behavior when the application's execution time is three orders of magnitude larger than its actual execution time. The `Total Time` column is the application's execution time. The three columns under the heading of `Threads` illustrate how much time is spent in the threads that contribute significantly to time. The three threads are `main` (the application itself),[6] `compile` (the optimizing compiler), and `gc` (the garbage collector).[7] The columns labeled by `Compiled` illustrate how many times the optimizing compiler was invoked, at what level of optimization, and for how many methods. The column labeled `Total` is the total number of times the optimizing compiler is executed. The column labeled `Methods` is the total number of methods that are optimized compiled. The columns labeled `Opt 0`, `Opt 1`, and `Opt 2` show how many methods are compiled at that level of optimization. The second number in the last two columns represents the number of methods that are reoptimized multiple times before obtaining this level of optimization. No method is ever reoptimized more than twice.

The general observations from Tables 2 and 3 are that the exact prediction of execution time is slightly better than the heuristic, and that grossly mispredicting execution time has significant performance implications. The fact that predicting execution time exactly is only slightly better than the heuristic indicates that this heuristic performs well in practice, at least for these benchmarks. Furthermore, when execution time is grossly over-predicted, the controller is too aggressive with compilation. This behavior of the controller is as expected because the controller assumes that the compilation time will be absorbed by the faster execution time of the application if the application runs long enough. In addition, the increase in compilation time increases the pressure on memory and results in more time spent in garbage collection.

4.3 Stability of model constants across different applications

Our simple analytical model relies on two sets of fixed constants, compilation cost and speedup, for each opti-

[6]This time includes the time to baseline compile every method executed. Previously we reported [3] that the baseline compile time is minimal.

[7]The time spent in the controller, organizer, and decay threads was insignificant, and therefore we do not report them.

Table 4: Performance of the adaptive system for SPECjvm98 with input size 100.

Benchmark	Parameters	
	Specialized	Average
compress	48.918	48.709
jess	27.302	25.933
db	70.326	69.855
javac	57.525	57.154
mpeg	40.443	39.613
mtrt	21.487	23.230
jack	40.352	44.374

mization level. The current implementation estimates the set of constants with off-line experiments. For example, we currently use constants derived from averages across the seven SPECjvm98 benchmarks.

However, as Table 1 shows, the values of these constants vary significantly across the benchmarks. So, we would expect our system to perform better on a particular benchmark if we use the constants derived only from that benchmark. If so, then perhaps the current model would perform better if we derive the constants online during each program's execution.

Table 4 presents the results of this experiment. The second column gives the runtime of the first run using the cost/benefit parameters for the particular benchmark. The third column gives the same value using the average cost/benefit parameters. In two programs, `mtrt` and `jack`, the specialized parameters offer a clear improvement over the average parameters. However, in the other five programs there is either no improvement or a degradation. The programs with the greatest variance from the speedup average, `javac` and `mpeg`, did not show improvements by using the specialized parameters. Thus, it appears that specializing for all methods on a particular benchmark is not more beneficial in the current model than using an average over a set of benchmarks.

We were surprised by this result and are currently investigating the causes. Perhaps the constants derived from all the methods in a particular benchmark differ significantly from the values for the "hot" methods in a benchmark. Or, perhaps the overall performance with the current model is relatively insensitive to variations in the constants of the model.

4.4 Incorporating method characteristics

> *All methods are equal, but some are more equal than others.*

Another straightforward extension would be to estimate the costs and benefits for each method based on more characteristics of the method. Our current model assumes a compile-time cost that is linear in the size of a method, and a benefit that is constant depending on the optimization level. Clearly, some optimizations do not run in linear time (although the most expensive optimization currently implemented [13] does run in almost-linear time). Perhaps more importantly, the benefit of optimization on a particular method depends on numerous factors. Perhaps we can better estimate the effects of optimization on a particular method based on simple static characteristics, such as loop structure, exceptional control flow, the static mix of bytecode instructions, load and store characteristics, etc. In future work, we will attempt to produce a more accurate cost/benefit model. It remains an open question whether this extension can significantly improve overall performance.

A related question concerns inlining. In our system, inlining drastically impacts both costs and benefits of compilation. The controller currently orchestrates feedback-directed inlining at a very high level, distilling information from a call graph profile and passing it to the compiler. However, the controller does not try to account for the impact of inlining for a specific method. Perhaps the controller should "micro-manage" inlining decisions, in order to obtain better estimates of costs and benefits for each method. This technique may also apply to other optimizations; however, in our system, we suspect inlining will exert the most influence.

4.5 Using idle resources

The current analytical model assumes 100% CPU utilization when considering costs and benefits. If there are idle cycles, the model should use these idle cycles more aggressively for compilation. For example, on an SMP, some processors may idle if the system is not driven to capacity.

We are currently building functionality to exploit hardware and system performance monitors to detect idle CPU cycles and load on other system resources. We expect at the very least to improve bottom-line performance on single-threaded benchmarks run on SMPs, by changing the model to account for idle cycles. We also plan to enhance our compilation infrastructure to allow parallel compilation activities.

A related issue is that the analytical model does not consider the impact of compilation on other system resources. For example, compilation threads will consume memory bandwidth that might be better used by application threads. Also, as our system is implemented entirely in Java, the compiler generates temporary objects that increase load on the garbage collector. We need to determine if these effects have a substantial impact on system performance, and if so, to incorporate them into the controller model when appropriate.

5. RELATED WORK

Other adaptive systems [14, 15, 19, 10, 5] typically rely on counters to record events, such as method invocations and loop iterations. In these systems once a counter exceeds a threshold, a method, typically the one that resulted in the threshold being surpassed, is optimized. Thus, these systems do not employ a cost/benefit model for determining if a hot method should be optimized.

One exception is the work of Kistler [16], which describes a continuous program optimization architecture for Oberon. Periodically, the system optimizes a collection of hot methods, which are determined by a sampling-based profiling technique. Each optimization phase in the system provides an expected benefit, which is derived from hard-coded benefit estimates and actual observed benefits for a particular optimization. These estimated benefits are used to determine the relative order in which to optimize the hot methods. The cost of optimization is not considered.

Plezbert and Cytron [17] propose several adaptive implementation strategies and simulate their effectiveness on C programs using a binary rewriting tool.

6. CONCLUSIONS

This paper provides details of the controller component of the Jalapeño adaptive optimization system. Previously we have demonstrated that a simple analytical model that determines which methods to optimize is sufficient to obtain comparable performance to the best JIT for a range of workloads. This paper provides insight into the performance of this simple model.

We have discovered that predicting the future execution time of a method based on how long the JVM has been running can dramatically effect the recompilation activity for an application, if the order in which the application is run is varied. Ignoring this limitation, i.e., holding the order of applications constant, we have discovered that the current heuristic of assuming a method will execution for twice the current duration is an effective predictor. We have discovered that specializing some of the model's parameters to be benchmark-specific is not beneficial in our current implementation. Finally, we have posed several open questions relevant to the design and evaluation of an enhanced controller model.

Acknowledgments

We thank the Jalapeño team members [1] for their work in developing the system used to conduct this research. We thank David Ungar for suggesting the experiment described in Tables 2 and 3. We acknowledge Jeanne Ferrante and Barbara Ryder for related discussions to this work. Laureen Treacy and the anonymous reviewers provided useful feedback on the presentation of this work.

7. REFERENCES

[1] 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.

[2] 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.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

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

[17] M. P. Plezbert and R. K. Cytron. Does "just in time" = "better late than never"? In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 120–131, Jan. 1997.

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

[19] 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.

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