A Dynamic Optimization Framework for a Java Just-In-Time Compiler

Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, Toshio Nakatani
IBM Tokyo Research Laboratory
1623-14 Shimoturuma, Yamato-shi, Kanagawa 242-8502, Japan
Phone: +81-46-215-4658
Email: {suganuma, yasue, jl25131, komatsu, nakatani}@jp.ibm.com

ABSTRACT

The high performance implementation of Java Virtual Machines (JVM) and Just-In-Time (JIT) compilers is directed toward adaptive compilation optimizations on the basis of online runtime profile information. This paper describes the design and implementation of a dynamic optimization framework in a production-level Java JIT compiler. Our approach is to employ a mixed mode interpreter and a three level optimizing compiler, supporting quick, full, and special optimization, each of which has a different set of tradeoffs between compilation overhead and execution speed. A lightweight sampling profiler operates continuously during the entire program's execution. When necessary, detailed information on runtime behavior is collected by dynamically generating instrumentation code which can be installed to and uninstalled from the specified recompilation target code. Value profiling with this instrumentation mechanism allows fully automatic code specialization to be performed on the basis of specific parameter values or global data at the highest optimization level. The experimental results show that our approach offers high performance and a low code expansion ratio in both program startup and steady state measurements in comparison to the compile-only approach, and that the code specialization can also contribute modest performance improvements.

1. INTRODUCTION

There has been a significant challenge for the implementation of high performance virtual machines for Java [18] primarily due to the dynamic nature of the language, and many research projects have been devoted to developing efficient dynamic compilers for Java [10, 13, 17, 23, 24, 26, 30, 36, 37, 43]. Since the compilation time overhead of a dynamic compiler, in contrast to that of a conventional static compiler, is included in the program's execution time, it needs to be very selective about which methods it decides to compile and when and how it decides to compile them. More specifically, it should compile methods only if the extra time spent in compilation can be amortized by the performance gain expected from the compiled code. Once program hot regions are detected, the dynamic compiler must be very aggressive in identifying good opportunities for optimizations that can achieve higher total performance. This tradeoff between compilation overhead and its performance benefit is a crucial issue for dynamic compilers.

In the above context, the high performance implementation of Java Virtual Machines (JVM) and Just-In-Time (JIT) compilers is moving toward exploitation of adaptive compilation optimizations on the basis of runtime profile information. Although there is a long history of research on runtime feedback-directed optimizations (FDO), many of these techniques are not directly applicable for use in JVMs because of the requirements for programmer intervention. Jalapeño [3] is the first JVM implementing a fully automatic adaptive compilation framework with feedback-directed method inlining, and it demonstrated a considerable performance improvement benefit. The Intel research compiler, JUDO [13], also employs dynamic optimization through a recompilation mechanism. Both of these systems use the compile-only approach, and it can result in relatively higher costs in compilation time and code size growth.

In this paper, we present a different approach for the dynamic optimization framework implemented in our production-level JIT compiler. We use a combination of an interpreter and a dynamic compiler with three levels of optimization to provide balanced steps for the tradeoff between compilation overhead and compiled code quality. A low-overhead, continuously operating sampling profiler identifies program hot regions for method reoptimization. To decide on the recompilation policy, we use a value profiling technique, which can be dynamically installed into and uninstalled from target code, for collecting detailed runtime information. This technique does not involve target code recompilation, and is reasonably lightweight and effective for the use of collecting a fixed amount of sampled data on program hot regions. In the highest level optimization for program hot methods, we apply code specialization using impact analysis. This is performed fully automatically on the basis of the parameter values or global object data, which exhibit runtime invariant or semi-invariant behavior [11] through the dynamically instrumented value profiling. Our experimental results show that this approach provides significant advantages in terms of performance and memory footprint, compared to the compile-only approach, both at program startup and in steady state runs.

1.1 Contributions

This paper makes the following contributions:

- **System architecture:** We present a system architecture for a simple, but efficient and high-performance dynamic optimization framework in a production-level Java JIT compiler with a mixed mode interpreter. Extensive experimental data is presented for both performance and memory footprint to verify the advantages of our approach.
- **Profiling techniques:** We present a program profiling mechanism combining two different techniques. One is a continuously operating, lightweight sampling profiler for detecting program hot methods, and the other is a dynamically installed and uninstalled instrumenting profiler that collects detailed information for the methods gathered by the first profiler.
- **Code specialization:** The design and implementation of code specialization, an example of FDO, is described, using the dynamically instrumented profiling mechanism for value sampling. This is a fully automated design with no programmer intervention required. The effectiveness of this technique is evaluated using industry standard benchmark programs.

The rest of this paper is organized as follows. The next section summarizes related work, comparing our system to prior systems. Section 3 describes the overall system architecture of our dynamic compilation system, including the multiple levels of the execution model divided between the mixed mode interpreter and recompilation framework. Section 4 discusses recompilation issues, including profiling techniques and instrumentation-based data sampling. The detailed description of code specialization appears in Section 5. Section 6 presents some experimental results using a variety of applications and industry standard benchmarking programs to show the effectiveness of our dynamic compilation system. Finally we conclude in Section 7.

2. RELATED WORK

This section discusses prior dynamic optimization systems for Java and other related work.

2.1 Dynamic Optimization Systems

There have been three major dynamic systems for automatic, profile-driven adaptive compilers for Java. These can be roughly broken into two categories; the Intel research compiler, the JUDO system [13], and the Jalapeño JVM [3, 4], all follow a compile-only approach, while HotSpot™ [30, 37] is provided with an interpreter to allow a mixed execution environment with interpreted and compiled code, as in our system.

The Intel compiler employs dynamic optimization through recompilation, by providing two different compilers: a fast code generator [1] and an optimizing compiler. As a way of triggering recompilation, it inserts counter updating instructions for both method entry point and loop backward branches in the first level compiled code. It incurs a continuous bottom-line performance penalty. The target code has to be recompiled if we want to remove these instructions overhead. Since the recompiled code is not instrumented, further reoptimization is not possible in this system. In contrast, our system uses a low-overhead profiling system for continuous sampling operation throughout the entire program execution, and thus allows for further reoptimizations, such as specialization.

Jalapeño is another research JVM implemented in Java itself. They implemented a multilevel recompilation framework using a baseline and an optimizing compiler with three optimization levels, and they presented good performance improvements in both startup and steady state regimes compared to other non-adaptive configurations or adaptive but single level recompilation configurations. Profile-directed method inlining is also implemented, and considerable performance improvement is obtained thereby for some benchmarks. Their overall system architecture is quite similar to ours, but the major difference lies in its compile-only approach and in how the profiling system works. The compilation-only approach can incur a significant overhead for the system. Although their baseline compiler was designed separately from the optimizing compiler for minimum compilation overhead, the system can result in a large memory footprint. Our system features a mixed mode interpreter for the execution of many infrequently called methods with no cost in compile time or code size growth. This allows the recompilation system to be more flexible and aggressive in its reoptimization policy decision. Their profiling system continuously gathers full runtime profile information on all methods, including information for organizer threads to construct data structures such as dynamic call graphs. Our system employs two separate profiling techniques to reduce the overall profiling overhead. That is, a lightweight sampling profiler focuses on detecting hot methods, and then an instrumenting profiler collects more detailed information only on hot methods.

HotSpot is a JVM product implementing an adaptive optimization system. It runs a program immediately using an interpreter, as in our system, and detects the critical "hot spots" in the program as it runs. It monitors program hot-spot continuously as the program runs so that the system can adapt its performance to changes in the program behavior. However, detailed information about the program monitoring techniques and the system structure for recompilation is not available in the literature.

The notion of mixed execution of interpreted and compiled code was considered as a continuous compiler or smart JIT approach in [31], and the study of three-mode execution using an interpreter, a fast non-optimizing compiler, and a fully optimizing compiler was reported in [2]. In both of these papers, it was proven that there is a performance advantage by using an interpreter in the system for balancing the compilation cost and resulting code quality, but the problem of the generated code size was not discussed.

The Self-93 system [20, 21] pioneered the on-line profile-directed adaptive recompilation systems. The goal of this system is to avoid the long compile pauses and to improve the responsiveness for interactive applications. It is based on a compile-only approach, and for the method recompilation, an invocation counter is provided and updated in the method prologue in the unoptimized code. The counters decay over time for reflecting the invocation frequencies to avoid eventually reaching the invocation limit for many unimportant methods. The recompilation also takes advantage of the type feedback information for receiver class distributions using the profiling in the previous version code.

Dynamo [8] uses a unique approach, focusing on native-to-native runtime optimization. It is a fully transparent dynamic compilation system, with no user intervention required, which takes an already compiled native instruction stream as input and reoptimizes it at runtime. The use of the interpreter here is to identify the hot paths for reoptimization rather than to reduce the total compilation cost as in our system. Another profile-driven dynamic recompilation system is described in [9] for Scheme. They use edge-count profile information for basic block reordering in the recompiled code for improved branch prediction and cache locality.

2.2 Instrumentation

Ephemeral instrumentation [34, 39] is, in principle, quite close to our dynamically installed and uninstalled instrumentation technique for value profiling. Their method is to dynamically replace the target addresses of conditional branches in the executing code with the pointer to a general subroutine that updates a frequency counter of the corresponding edge. The collected data is then used off-line for a static compiler. Our profiling system, on the other hand, is not limited to the branch target, but applicable to any point of the program by generating the corresponding code for value sampling. Also the instrumentation system is integrated into the fully automated dynamic compilation system.

A framework for reducing the instrumentation overhead in an online system [5] is prototyped in Jalapeño. This technique introduces a second version of the code, called checking code, to reduce the frequency of executing the instrumented code. This will allow a variety of profiling techniques to be integrated in the framework. The main concern is the space overhead caused by duplicating the whole method for extra versions for both checking and instrumented code, although some space saving techniques are described. Our system dynamically attaches only a small fragment of the code for value sampling at the method entry points, and thus it is more space efficient.

2.3 Code Specialization

There has been much work in the area of dynamic code generation and specialization, most of which require either source language extensions, such as tcc system [32], or programmer annotations such as Tempo [28], the dynamic compiler developed at the University of Washington [6], and its successor system, DyC [19]. In these systems, a static compiler performs the majority of optimization work and prepares for a dynamic compilation process by generating templates, and a dynamic compiler instantiates the templates at runtime.

As a restricted form of specialization, called customization [12], the Self system creates a separate version of a given method for each possible receiver class, relying on the fact that many messages within a method are sent to self object. The selective specialization technique [15] then corrected the problem of both overspecialization by specializing only heavily-used methods for their most beneficial argument classes, and underspecialization by specializing methods on arguments other than the receiver. This system resembles ours in that it combines static analysis (corresponding to our impact analysis) and profile information to identify the most profitable specialization. However, their work was focused on converting dynamic calls to static calls to avoid the large performance overhead caused by dynamic method dispatch. Our specialization allows not only method call optimizations, but also general optimizations, such as type test elimination, strength reduction, and array bound check elimination, on the basis of specific values dynamically collected.

The inlining trials [16] in the Self system is an attempt to predict the benefit of inlining based on type group analysis. Inlining method calls with special parameter values can be considered an extreme case of specialization to a particular call site. Our impact analysis is more general in the sense that it can handle not only method parameters but also global variables such as object instance fields.

An analysis to identify so called glacial variables [7] is proposed to find good candidates for specialization. However, their analysis is static, and the execution frequency is estimated only by loop nesting level, without using the dynamic profile information as in our system.

3. SYSTEM ARCHITECTURE

The goal of our dynamic optimization system is to achieve the best possible performance with a set of currently available optimization capabilities for varying phases of application programs, including program startup, steady state, and phase shifts. It also needs to be robust for continuous operation for long-running applications. The overall architecture of our system is as depicted in Figure 1. We describe each of the major components of the system in the following sections.

3.1 Mixed Mode Interpreter

Most of the methods executed in Java applications are neither frequently called nor loop intensive as shown in the results in Section 6.2, and the approach of compiling all methods is considered inefficient in terms of both compilation time and code space. The mixed mode interpreter (MMI), written in assembler code, allows the efficient mixed execution of interpreted and compiled code by sharing the execution stack and exception handling mechanism between them. It is roughly three times faster than an interpreter written in C.

Initially, all methods are interpreted by the MMI. A counter for method invocation frequencies and loop iterations is provided for each method and initialized with a threshold value. Whenever the method is invoked or loops within the method are iterated, the counter is decremented. When the count reaches zero, it is known that the method has been invoked frequently or is computation intensive, and JIT compilation is triggered for the method. Thus the JIT compilation can be invoked either from the top entry point of the method or from a backward branch within a loop. In the latter case, the control is directly transferred to the JIT compiled code from the currently interpreted code, by dynamically changing the frame structure for JIT use and jumping to specially generated compensation code. The JIT compilation for such methods can be done without sacrificing any optimization features.

If the method includes a loop, it is considered to be very performance sensitive and special handling is provided to initiate compilation sooner. When the interpreter detects a loop's backward branch, it snoops the loop iteration count on the basis of a simple bytecode pattern matching sequence, and then adjusts the amount by which the counter is decremented depending on the loop iteration count. In the case where the iteration count is large enough, the JIT compilation is immediately invoked without waiting until the counter value reaches zero.

The collection of runtime trace information is another benefit of the MMI for use in JIT compilation. For any conditional branches

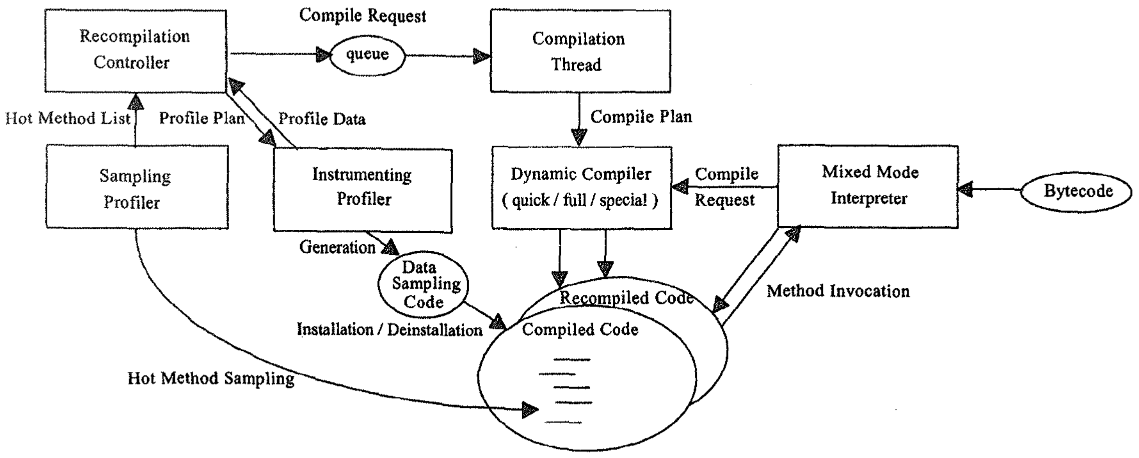


Figure 1. System architecture of our dynamic optimization system.

encountered, the interpreter keeps the information of whether it is taken or not to provide the JIT compiler with a guide for the branch direction at basic block boundaries[1]. The trace information is then used by the JIT compiler for ordering the basic blocks in a straight-line manner according to the actual program behavior, and for guiding branch directions in partial redundancy optimizations [26].

3.2 Dynamic Compiler

The dynamic optimizing compiler has the following optimization levels.

- *Quick (1st level) optimization* employs only a limited set of the optimizations available. Basically, optimizations causing higher costs in compilation time or greater code size expansion are disabled. For example, only those methods whose bytecode size does not exceed the size of a typical method invocation sequence will be inlined. This saves the compilation overhead not only of the method inlining process, but for the later optimization phases which traverse the entire resulting code block. The guarded or unguarded devirtualization of method calls is applied based on the class hierarchy analysis [23, 24]. The maximum number of iterations in the dataflow-based optimizations is also reduced. These optimizations involve iterations over several components, such as copy propagation, array bound check elimination, null pointer check elimination, common subexpression elimination, and dead code elimination.
- *Full (2nd level) optimization* employs all optimizations available. Additional and augmented optimizations at this level include full-fledged method inlining, escape analysis (including stack object allocation, scalar replacement, and synchronization elimination), an additional pass for code generation and code scheduling, and DAG-based loop optimization. The iteration count for dataflow-based optimizations is also increased.
- *Special (3rd level) optimization* applies code specialization, a feedback-directed optimization, in addition to the same set of optimizations as in the previous level. This is described in detail in Section 5.

[1] Since keeping trace information every time can cause additional overhead, the branch instruction is converted to the corresponding quick instruction after being executed a fixed number of times in order to minimize the performance penalty.

The internal representation is common for all optimization levels. The differences in compilation time, generated code size, and generated code's performance quality between the quick optimization and full optimization versions can be found in the experimental results presented in Section 6.1 and 6.2.

The reason that we provide three optimization levels in our dynamic compiler is twofold. First, one level of compilation model is, in our experience, simple and still effective until a certain level of optimization in the presence of MMI. However, as more sophisticated and time-consuming optimizations are added for pursuing higher performance, more of the negative side of the dynamic compilation (that is, the compilation overhead and code size growth problems) starts to appear. Even if more expensive optimizations are implemented, the return is diminishing and the net performance gain becomes marginal. This is considered to be due primarily to the larger gap between the interpreter and the compiler regarding the level of tradeoff between compilation cost and the resulting performance. If we set a lower threshold for triggering compilation, we may have better performing compiled code earlier but more total compilation cost is incurred. If we set a higher threshold value, we may miss some opportunities for gaining performance for some methods due to delayed compilation. There is also a problem with application startup performance degradation with one level of a highly optimizing compiler. It is therefore desirable to provide multiple, reasonable steps in the compilation level with well-balanced tradeoffs between the cost and the expected performance, from which an adequate level of optimization can be selected corresponding to the current execution context.

Secondly, it is not clear whether it would be effective to have more than three levels of optimization in the dynamic compilation system, without knowing the exact relationship between each component of the optimization on performance and compilation cost. Having more levels of optimization would make more choices available for recompilation. However it would complicate the selection process and more informative profiling data would be necessary to make correct decisions, which might add more overhead. Furthermore, the resulting code may or may not be of better quality depending on the target methods. The gradual promotion with finer steps of optimization can result in more code expansion rather than any overall performance benefit. The results shown in Section 6.1, combined with the interpreter available in our system, suggest that the current classification of three levels of optimization can provide an adequate tradeoff for reasonable promotion for recompilation decisions.

3.3 Sampling-Based Profiler

The sampling-based profiler [40] gathers information about the program threads' execution. This profiling collector keeps track of methods where the application threads are using the most CPU time by periodically snooping the program counters of all of the threads, identifying which methods they are currently executing, and incrementing a hotness counter associated with each method.

Since the MMI has its own counter-based profiling mechanism, this sampling profiler only monitors compiled methods for reoptimization. The hot methods identified by the profiler are kept in a linked list, sorted by the hotness counter, for use by the recompilation controller in deciding on method recompilation. To minimize the bottom-line overhead, the profiler doesn't operate by constructing and maintaining a call context tree for every sampling time interval, which would involve traversing the stack to a certain depth. Instead, additional information such as caller-callee relationships is collected by instrumentation code only for methods considered as candidates for recompilation as described in Section 3.5. This two-stage profiling design results in low overhead for the sampling profiler and hence allows continuous operation during the entire program execution with virtually no performance penalty.

3.4 Recompilation Controller

The recompilation controller, which is the brain of the recompilation system, takes as input from the sampling profiler the list of hot methods and makes decisions regarding which methods should be recompiled. The recompilation requests, as the results of these decisions, are put into a queue for a separate compilation thread to pick up and compile asynchronously.

The controller also directs the instrumenting profiler to install instrumentation code for further profile information such as method return addresses for collecting call site distribution for those hot methods. Some parameter values can also be collected depending on the results of impact analysis done in the full optimization compilation phase (described in Section 5). This additional profile data is useful in guiding more effective optimizations, such as method inlining and code specialization.

3.5 Instrumenting Profiler

The instrumenting profiler, according to an instrumentation plan from the recompilation controller, dynamically generates code for collecting specified data from a target method, and installs it into the compiled code. The entry instruction of the target code, after it is copied into the instrumenting code region, is dynamically patched with an unconditional branch instruction in order to direct control to the generated profiling code. The instrumentation code records the caller's return address or values of parameter and object fields in a table and then jumps back to the next instruction after the entry point. The data table or a counter for storing information is allocated separately to be passed back to the controller. After collecting a predetermined number of samples, the generated code automatically uninstalls itself from the target code by restoring the original instruction at the entry point.

The information that is collected and recorded by the instrumentation code can range from a simple counter (such as zero, non-zero, or array type), which just counts the number of executions, to a form of table with values or types of variables and their corresponding frequencies. Unlike instrumentation code found in other systems [13], this technique allows dynamic installation and uninstallation without involving target code recompilation. Since the target method and sampling numbers are controllable, the overhead of the instrumentation is relatively lightweight, unlike systems where the instrumentation code is generated as part of the compiled code which always incurs overhead.

4. RECOMPILATION

The key to our system is to make correct and reasonable decisions to selectively and adaptively choose methods for each level of optimization. From the mixed mode interpreter to the 1st level compilation, the transfer is made on the basis of the dynamic count on invocation frequencies and loop iterations, with additional special treatment for certain types of loops. The request for 2nd level and 3rd level recompilation from lower level compiled code is through the sampling profiler. The compilation and recompilation need to be done from one level to the next, and there is currently no direct path skipping intermediate levels from interpreter mode or compiled code.

The reason we chose two different ways of method promotion comes from the consideration of the advantages and disadvantages of the two profiling mechanisms: sampling-based and counter-based. For the interpreter, the cost of counter updates is not an issue, given the inherently higher overhead of interpreted execution, compared to additional code for counter maintenance. Instead, the accuracy of the profiling information is rather important, because the large gap in performance between interpreted and compiled code means the performance penalty could be large if it misses the optimum point to trigger the 1st level compilation. This tradeoff between efficiency and accuracy can be measured using counter-based profiling. On the other hand, compiled code is very performance sensitive, and inserting counter updating instructions in this compiled code could have quite a large impact on total performance. Lightweight profiling is much better for continuous operation. Since the target method is already in a compiled form, a certain loss of accuracy in identifying program hot regions, which may cause a delay in recompilation, is allowable. Sampling-based profiling is superior for this purpose.

4.1 Recompilation Request

Since the quick optimization compiler generates code with virtually no method inlining, the sampling profiler collects information on the set of hot methods as individual methods. A simple-minded recompilation request for those methods can result in unnecessary recompilations, since some methods included in the list may be inlined into another during the full optimization compilation [21].

This can happen because the hot methods appearing in the list come from sampling during the same stage in the program's execution, and therefore can be closely interrelated. Instead of simply requesting recompilation for each of the methods, the controller first constructs call graphs, structures representing the caller-callee relationships, from the list of hot methods. This requires information about the call sites' distributions for each method. Then only those methods which are roots in one of the graphs are pushed into the compile request queue with appropriate inlining directions for methods appearing in the subgraph below the root.

4.2 Multiple Version Code Management

After the recompilation is done for a method, it is registered by a runtime system called the *code manager*, which controls and manages all the compiled code modules by associating them with their corresponding method structures and with a set of information such as the compiled code optimization level and specialization context. This means all the future invocations to this method through indirect method lookup will be directed to the new version of the code, instead of the existing compiled code. Static and nonvirtual call sites are exceptions, where direct binding call instructions to old version code have been generated. A dynamic code patching technique is used in order to change the target of the direct call. This is done by putting a jump instruction at the entry point of the old code to a runtime routine, which then updates the call site instruction to direct the flow to the new code using the return address available on the stack, so that the direct invocation of the new code will occur from the next call.

For those threads currently executing old version code, the new optimized code will be used from the next invocation. We currently do not have a mechanism for *on-stack replacement* [21], the technique of dynamically rewriting stack frames from one optimization version to another. Also the problem of cleaning up the old version code remains in our system. The major difficulty with old code is how to guarantee that no execution is currently or will be in the future performed using the old compiled code. The apparent solution would be to eagerly patch all the direct bound call sites, rather than to patch lazily as in our current implementation, and then to traverse all the stack frames to ensure no activation record exists for this old code. This traversal would be done at some appropriate time (like garbage collection time).

5. CODE SPECIALIZATION

In this section, we describe the design and implementation of *code specialization*, as an interesting feedback-directed optimization technique. This optimization is applied for the methods already compiled with the full optimization level. Figure 2 shows the flow of control regarding how the decision on code specialization will be made. Currently code specialization is applied for an entire method, not for a smaller region in the method such as a loop.

5.1 Impact Analysis

Since *overspecialization* can cause significant overhead in terms of both time and space, it is important to anticipate before its application how much benefit it can bring to the system and for what values. Impact analysis, a dataflow-based routine that detects opportunities and estimates the benefit of code specialization, is used to make a prediction as to how much better code we would

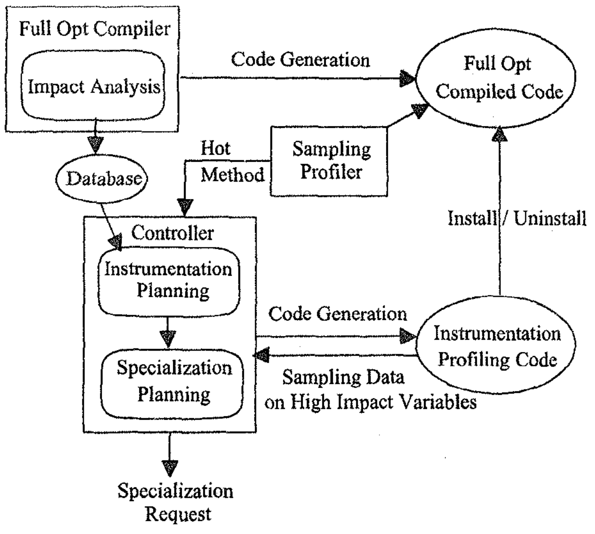


Figure 2. Flow of specialization decisions.

be able to generate if we knew a specific value or type for some variables. The impact analysis is done during the 2nd level compilation and the result of the analysis is stored into a persistent database, so that the controller can make use of it for the next round of recompilation plans.

The specialization targets can be both method parameters and non-volatile global variables[2], such as static fields and object instance fields. The set of specialization targets for global variables within a method can be computed from *In (n)* and *Kill (n)* for each basic block *n* after solving the forward dataflow equations given below:

In (entry_bb): All non-volatile global variables within the method.

Kill (n): The set of global variables that can be changed by instructions in basic block *n*.

$$Out(n) = In(n) - Kill(n)$$

$$In(n) = \bigcap_{m \in Pred(n)} Out(m) \qquad \text{(for } n \neq \text{entry basic block)}$$

This means that *In (n)* is the set of global variables referenced within the method and guaranteed not be updated along any paths reachable from the top of the method to the entry of the basic block *n*. Each global variable reference within the basic block can be checked as to whether it can be included in the specialization target from the *Kill* set for each instruction. That is, the above equation computes all the global variables that are safe for *privatization* at the entry point for each method. The set of specialization targets for the global data, together with the argument list, is then fed to the impact analysis. The pseudocode of the impact analysis is shown in Figure 3.

Each specialization target can be expressed by a triple (*L*, *S*, *V*), where *L* denotes the defined local variable from a parameter or a

[2] For a variable declared volatile, a thread must reconcile its working copy of the field with the master copy every time it accesses the variable [18].

```
Specialization Target (L, S, V)
L: defined local variable from a parameter or a global variable
S: statement where L is defined
V: parameter or global variable for specialization

for each element (L, S, V) in Specialization Target List
{
   Derived [ V ] = { (L, S) };
   Weight [ V, * ] = 0;
   for each variable (L, S) in Derived [ V ]
   {
      for each operation Op which uses L and is reachable from S
      {
         if (Op can be simplified or eliminated by a specialization
            type T )
         {
            Weight [ V,T ] += Impact of T on Op;
            if ( Op is converted to a constant assignment )
            {
               Derived [ V ] ∪= (LHS var of Op, location of Op)
            }
         }
      }
   }
}
```

Figure 3. Pseudocode for impact analysis.

global variable, S denotes the statement in the method in which the variable L is defined, and V denotes the parameter or global variable for specialization. The algorithm traverses the dataflow through a def-use chain for each use of the variable L and its derived variable, tracking any possible impact on each operation in V. The impact of the specialization type T on operation Op appearing in the pseudocode can be expressed by

$$Impact(Op, T) = saved\ cost(Op, T) / SST(V, T) * f(loop\ nest\ level)$$

The baseline of the impact is the execution cost of the *Specialization Safety Test (SST)*, which is the guard to be generated at the entry of the specialized method. This can vary from a simple compare and jump instruction to a set of multiple instructions depending on the variable V and type T. The impact is then the relative cost that can be saved on operation Op with the specialization type T against its SST cost. If the operation is located within a loop, then the impact is scaled with a factor representing the loop iterations to reflect the greater benefit that can be expected.

The cost saving can be quite different for each of the operations. The elimination of checkcast or instanceof operations can have a large effect, while it would be much smaller when getting a constant operand in a binary operation. The final result of impact analysis for the estimated specialization benefit is the *Specialization Candidate SC*;

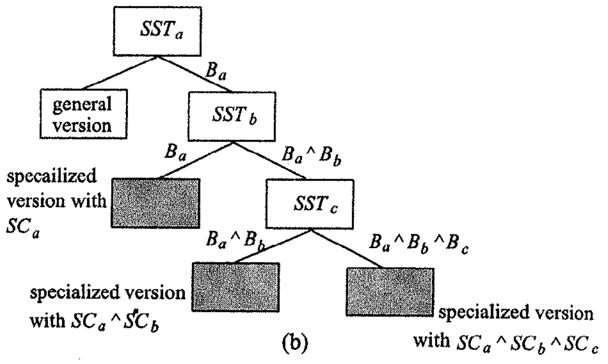
$$SC = \{ V \mid Weight(V, T) > \text{minimum threshold for any type } T \}$$

The factors currently considered in the impact analysis include the following:

- A constant value of a primitive type, which can lead to additional opportunities for constant folding, strength reduction, the replacement of floating point transcendental operations with a computed value, and the elimination of the body of a conditional branch.
- An exact object type, which allows removal of some unnecessary type checking operations and leads to more opportunities for class-hierarchy-based devirtualization[3].
- The length of an array object, which allows us to eliminate array bound checking code. This can also contribute to loop transformations, such as loop unrolling and simplification, if the loop termination condition becomes a constant.
- The type of an object such as null, non-null, normal object, or array object, which can be used for removing some unnecessary null checks and for improving the code sequence for some instructions (e.g. invokevirtualobject, checkcast).
- Equality of two parameter objects, allowing method bodies to be significantly simplified.
- A thread local object, which allows us to remove unnecessary synchronizations.

Candidate	Sample Bias	Weight	Expected Benefit
SC_a	R_a	W_a	B_a
SC_b	R_b	W_b	B_b
SC_c	R_c	W_c	B_c

(a)



SST_a
B_a
general version
SST_b
B_a
$B_a \wedge B_b$
specailized version with SC_a
SST_c
$B_a \wedge B_b$
$B_a \wedge B_b \wedge B_c$
specialized version with $SC_a \wedge SC_b$
(b)
specialized version with $SC_a \wedge SC_b \wedge SC_c$

Figure 4. (a) Multiple specialization candidates available, (b) Construction of decision tree on generating three specialized versions.

5.2 Specialization Decision

When a hot method has been identified in the 2nd level compiled code, the controller checks the results from the impact analysis stored in the code manager database. If a candidate in SC for the method looks promising as a justification for performing specialization, then the controller dynamically installs the instrumentation code into the target native code to decide whether it is indeed worth specializing with the specified type. This is based on the same mechanism described in Section 3.5. Currently a minimum threshold is used to allow all candidates to be selected as an instrumentation installation target.

Upon the completion of the value sampling, the controller then makes a final decision regarding whether it is profitable to specialize with respect to a specialization candidate in SC. The metric

[3] With the preexistence optimization [17], some of this opportunity for parameter variables may be disappeared.

Table 1. List of benchmarks and applications used in the evaluation.

	Program	Description	Measurement Condition
Startup Runs	SwingSet	GUI component, version 1.1	Run the demo program as an application to bring up the initial window.
	Java2D	2D graphics library	Run the demo program as an application with options -runs=1 -delay=0
	ICE Browser [42]	Simple internet browser, version 5.05	Run the browser application to bring up the welcome window.
	HotJava [38]	HotJava browser, version 1.1.5	
	IchitaroArk [25]	Japanese word processor	Run the application to bring up the initial input window
	WebSphere [22]	Web application server, version 3.5.3	Attach administration console after starting administration server
Steady State Runs	_227_mtrt	Multi-threaded image rendering	Run SPECjvm98 benchmark harness from appletviewer with the following settings (the order of these tests is specified in SpecApplet.html). - initial heap size 96m and max heap size 96m. - run individual benchmarks in the experiments from Section 6.1 to 6.3, or run complete benchmarks with a single JVM in the experiment for Section 6.4. - select input size=100, then run with autorun, and test mode (not SPEC compliant mode). - the number of executions in each autorun sequence is 10.
	_202_jess	Java expert system shell	
	_201_compress	LZW compression and decompression	
	_209_db	Database function execution	
	_222_mpegaudio	Decompression of MP3 audio file	
	_228_jack	Java parser generator	
	_213_javac	Java source to bytecode compiler in JDK1.0.2	
	SPECjbb2000-1.0	Transaction processing benchmark	Run with warehouse 1 only, with initial and max heap size 256m.

we use for this decision can be expressed as follows:

f(Weight, Sample Ratio, Code Size, Hotness Count)

This function indicates that the impact analysis result, the ratio of bias in the corresponding sample data, the size of the recompile target code, and the method hotness count are all considered for the final specialization decision. The code size affects the maximum number of versions that can be produced for specialization, since the larger the code size for recompilation, the more costly it would be to generate multiple versions. The method hotness count is used for adjusting the number of versions allowed for some important methods. The construction of the specialization plan then proceeds as follows.

Suppose there are three specialization candidates SC_i ($\in SC$ for $i = a, b, c$) available for a method as in Figure 4 (a). The expected benefit for each candidate using specialized code, based on the probability of the specialized code hit ratio, is computed as $B_i = R_i * W_i$. The plan on how to organize the specialized code can then be viewed as constructing a decision tree. That is, each internal node of the tree represents the specialization safety test SST_i guarding each specialization. The right subtree is for the version where the benefit B_i is applied, and the left subtree tracks the version where it is not used. The number of leaf nodes is two raised to the number of candidates. The specialization is then organized by selecting leaf nodes, from right to left, for as many as the number of versions allowed as calculated from the code size and method hotness count. All the nodes that are not selected for specialization are contracted to a single node that represents a general version of the code (that is, the original 2nd level compiled code).

Two strategies can be considered for the tree construction: benefit ordered and sample ratio ordered. In the benefit-ordered construction, a specialization candidate having a larger value of B_i moves to a higher level internal node, reflecting a greater expected benefit when the condition holds true. In the sample-ratio-ordered, a value of R_i is regarded as a more important factor with expectation of a higher rate of executing the specialized versions. In Figure 4 (b), assuming that SC_i is in the order of *a*, *b*, *c* in either criteria, the decision-tree is constructed with the number of specialized versions limited to three.

The recompilation with code specialization can introduce additional opportunities for many other optimizations, such as constant folding, null-check and array-bound-check elimination, type check elimination or simplification, and virtual method inlining. In the case of *SST* failure upon specialized code entry, the control is directed to the general version of the code. This can generally occur as the result of changes in the execution phase of the program, so that the specialized methods can be called with a different set of parameters. In this situation, the general version of the code may be identified as hot again, and the next round of recompilation can be triggered for this method, possibly with specialization using different values. Thus the new version of specialized code can then become active, by replacing the previous code. The maximum number of versions for specialized code for a method is limited to N, a number which is based on the target code size, set by the recompilation controller to avoid excessive code growth.

6. EXPERIMENTAL RESULTS

This section presents some experimental results showing the effectiveness of our dynamic optimization system. We outline our experimental methodology first, describe the benchmarks and applications used for the evaluation, and then present and discuss our performance results.

6.1 Benchmarking Methodology

All the performance results presented in this section were obtained on an IBM IntelliStation (Pentium III 600 MHz uni-processor with 512 MB memory), running Windows 2000 SP 1, and using the JVM of the IBM Developer Kit for Windows, Java Technology

Table 2. **Comparison of compilation time with three different optimization levels on SPECjvm98 benchmark programs. All results are in seconds.**

		mtrt	jess	compress	db	mpegaudio	jack	javac
no opt	first run	22.14	16.02	29.52	36.81	28.36	17.22	22.58
	best run	21.25	14.31	29.08	36.52	26.97	15.94	19.81
	compile time	0.89 (4.0%)	1.71 (11.0%)	0.44 (1.5%)	0.29 (0.8%)	1.39 (4.9%)	1.28 (7.4%)	2.77 (12.3%)
quick opt	first run	8.38	10.5	17.82	29.63	14.01	10.47	17.92
	best run	6.53	7.83	17.39	29.14	11.89	8.31	12.18
	compile time	1.85 (22.1%)	2.67 (25.5%)	0.43 (2.4%)	0.49 (1.7%)	2.12 (15.1%)	2.16 (20.6%)	5.74 (32.0%)
full opt	first run	11.34	20.91	16.81	30.22	14.17	15.95	37.25
	best run	5.03	7.06	15.84	28.83	9.91	7.88	12.22
	compile time	6.31 (55.6%)	13.85 (66.2%)	0.97 (5.7%)	1.39 (4.6%)	4.26 (30.1%)	8.07 (50.6%)	25.03 (67.2%)
special opt	first run	13.21	21.88	16.88	30.81	14.56	16.56	40.41
	best run	4.89	6.51	15.83	28.51	9.47	7.46	12.08
	compile time	8.32 (63.0%)	15.37 (70.2%)	1.05 (6.2%)	2.30 (7.5%)	5.09 (35.0%)	9.10 (55.0%)	28.33 (70.1%)

Edition, Version 1.3.1 prototype build. The benchmarks we chose for evaluating our dynamic optimization system are shown in Table 1 (the size of each benchmark is indicated in the MMI-only row in Tables 3 and 4). We conducted two sets of measurements: startup runs and steady state runs. For the startup performance evaluation, we selected a variety of real-world applications, ranging from a simple Internet browser to a complex Web application server. For evaluating the steady state performance, we use SPECjvm98 and SPECjbb2000 [35], two industry standard benchmark programs for Java.

Table 2 shows the compilation times for the three different optimization levels used in our system. The numbers for no-opt compilation are additionally presented to evaluate our system against the compile-only approach. These measurements were done by running each level of optimization without MMI. We assume that the difference between the first run and the best run of SPECjvm98 can be regarded as the compilation time. We ignored the cache and file system effects, which we tried to minimize by executing some warm-up runs before the measurements. The percentages shown in the compilation time fields are the ratios of compilation time over the time of the first run.

The following configuration and parameters were used throughout the experiments.

- The threshold in the mixed mode interpreter to initiate dynamic compilation (quick optimization) was set to 2,000.
- The timer interval for the sampling profiler for detecting hot methods was 10 milliseconds. With this interval, the overhead introduced is below the noise and the accuracy is considered adequate for identifying recompilation candidates [40].
- The sampling profiler was operated continuously to build a list of hot methods, while the controller examined and purged the data every 100 sampling ticks for recompilation decisions.
- For instrumentation-based sampling for hot methods, a maximum of 512 values were collected for each of the target parameters, global variables, or return addresses. The maximum number of data variations recorded was 8.
- The priority of the sampling profiler thread and the compilation thread was set to above and equal to that of the application threads, respectively.
- The number of code duplications allowed for specialization was set to one, regardless of the target method code size. In the measurement for Section 6.3, the decision-tree construction was based on the benefit-ordered strategy.
- Exception-directed optimization (EDO) [29] was enabled. The recompilation request from EDO profiler was processed with quick optimization with special treatment for method inlining for the specified hot exception paths.

6.2 Evaluation of Recompilation System

There are different requirements for the best performance between the two phases of application execution, program startup and steady state. During the startup time, many classes are loaded and initialized, but typically these methods are not heavily executed. When the program enters a steady state, a working set of hot methods will appear. In our experiment, we evaluated the startup performance by running each of the listed applications individually and measuring the timing from the issuing of the command until the time the initial window appeared on the screen. For Java2D with the options indicated in the table, each tab on the window is automatically selected and the execution proceeds until the whole component selection is done. That is, we measured the timing from the issuing of the command until the program terminates. For WebSphere, we first started the server process, and then measured the time to bring up the administration console window which runs on the same machine. For the steady state measurement, we took the best time from 10 repetitive autoruns for each test in SPECjvm98. For SPECjbb2000, we chose the

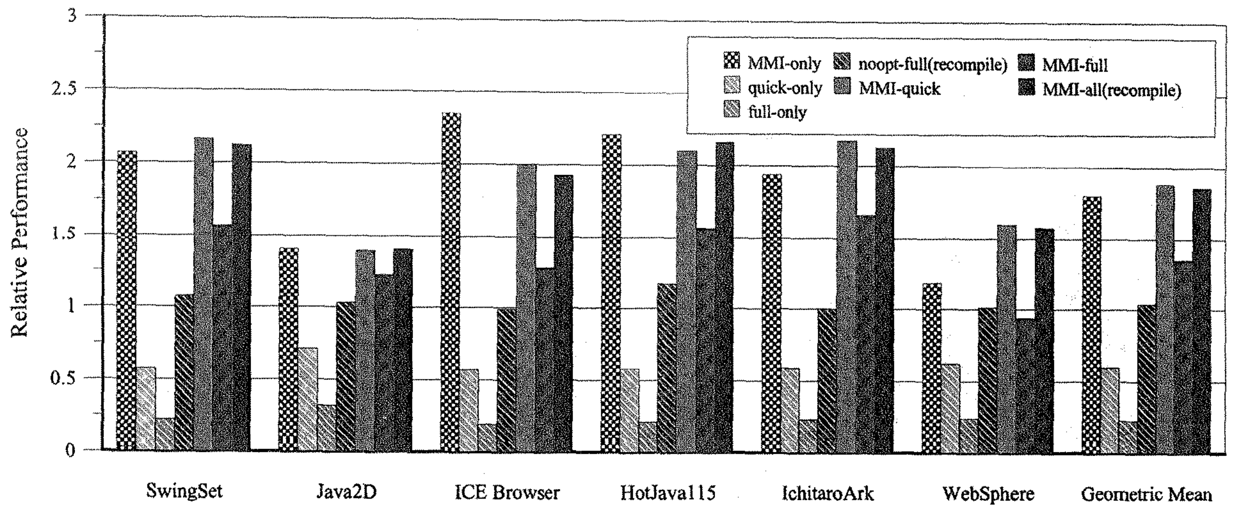


Figure 5. Startup performance comparison. Each bar indicates the total execution speed relative to no opt compiler without MMI. Therefore higher bar shows better performance.

Table 3. Comparison of number of compiled methods and generated code size (Kbytes) in program startup. The MMI-only row indicates the number of executed methods and bytecode size. Native methods are not counted.

		SwingSet		Java2D		ICE Browser		HotJava		IchitaroArk		WebSphere	
		Method	Size	Method	Size	Method	Size	Method	Size	Method	Size	Method	Size
MMI-only		7,273	440.5	7,327	531.5	3,410	275.7	4,110	297.5	7,282	473.7	9,029	615.7
NO MMI	no-opt	7,273	3,218.5	7,327	3,884.3	3,410	1,791.7	4,110	2,110.5	7,282	3,581.1	9,029	4,511.3
	quick-opt	6,444	3,096.7	6,511	3,473.6	2,905	1,556.2	3,599	1,945.8	6,193	3,307.3	7,951	4,190.2
	full-opt	3,701	4,518.6	3,985	5,003.8	1,848	2,438.6	2,191	3,019.1	3,910	4,826.3	4,898	6,165.4
MMI	quick-opt	439	233.8	523	281.1	75	78.1	91	65.6	205	124.7	321	165.7
	full-opt	416	347.6	463	416.2	69	131.5	85	128.9	194	217.7	301	350.9

configuration of one warehouse with 60 seconds of rampup time, longer than the standard SPEC rule, to give the system enough warm-up time.

We compared the following sets of compilation schemes.

1. MMI only
2. no optimization compilation with no MMI (noopt-only)
3. quick optimization compilation with no MMI (quick-only)
4. full optimization compilation with no MMI (full-only)
5. no optimization compilation with no MMI and recompilation using full optimization compilation (noopt-full)
6. quick optimization compilation with MMI (MMI-quick)
7. full optimization compilation with MMI (MMI-full)
8. all levels of compilation with MMI for adaptive recompilation (MMI-all)

The fifth case is provided as a comparison of our system to the corresponding recompilation system with the compile-only approach as in other systems [3, 13]. In this case, the MMI was not executed and all methods were first compiled by the compiler with no optimization applied (level 0), and then some hot methods identified by the sampling profiler were reoptimized with the full optimization compilation. Our no-opt compilation may have different characteristics in terms of both compilation overhead and code quality from the baseline compiler [3] or the fast code generator [1, 13], especially because our no-opt compilation system is not a separate compiler, while theirs are designed and implemented differently from the optimizing compilers. Nevertheless, we think the comparison with this configuration can be an indication as to how well our system can compete against a compile-only system.

Figures 5 shows the comparisons of program startup performance. The base line in this graph is the second case above, no optimization compilation with no MMI (noopt-only). The chart indicates that the performance of our dynamic optimization system, MMI-all, is almost comparable to that of the lightweight configuration of the MMI-quick. On the other hand, no-MMI configurations show poor performance in all the programs, probably due to the high cost of compiling all executed methods. The recompilation system with the compile-only approach, noopt-full, shows better performance than the other no-MMI configurations, but it still cannot compete against other top performing configurations. The fact that the performance of MMI-only is nearly two times faster in average than that of noopt-only (base line of the graph)

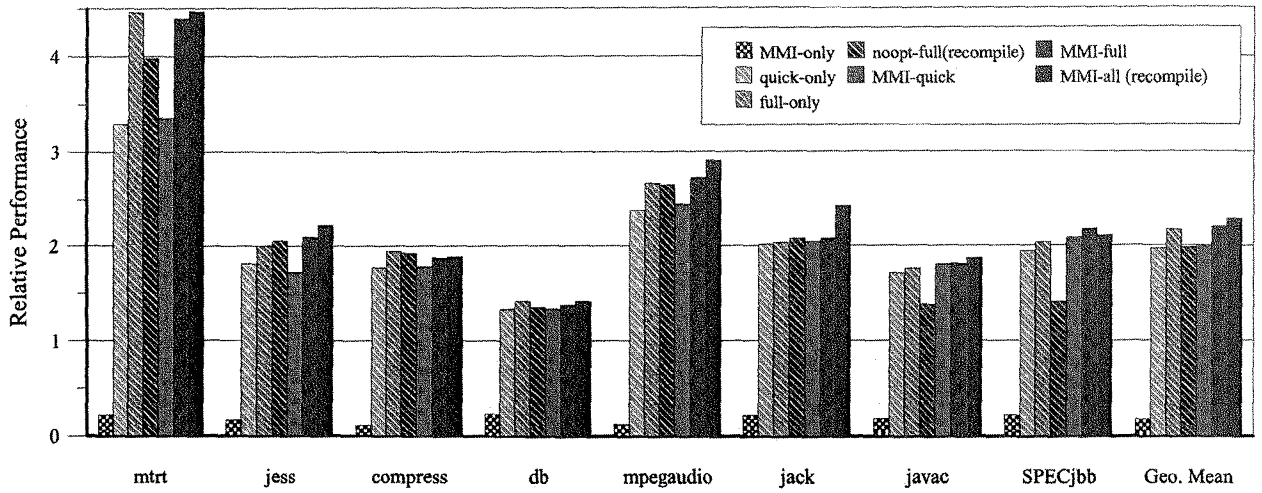


Figure 6. Steady state performance comparison. Each bar indicates the total execution speed relative to no opt compiler without MMI. Therefore higher bar shows better performance.

Table 4. Comparison of number of compiled methods and generated code size (Kbytes) in steady state. The MMI-only row indicates the number of executed methods and bytecode size. Native methods are not counted.

			mtrt		jess		compress		db		mpegaudio		jack		javac		SPECjbb	
			Method	Size	Method	Size	Method	Size	Method	Size	Method	Size	Method	Size	Method	Size	Method	Size
MMI-only			451	35.7	724	44.9	317	25.7	312	25.5	496	71.8	557	56.2	1,079	96.1	2,781	235.2
NO MMI	no-opt		451	232.3	724	289.5	317	160.7	312	162.7	496	492.3	557	351.7	1,079	571.8	2,781	1,685.3
	quick-opt		295	231.6	611	293.3	232	150.7	232	157.9	392	354.9	374	340.1	940	531.9	2,307	1,463.6
	full-opt		264	498.9	553	912.3	220	330.1	219	343.7	324	497.4	342	742.1	808	1,157.4	1,623	2,109.3
	noopt-full (recompile)	1st	445	233.1	726	293.7	317	161.9	312	164.1	487	475.7	564	357.5	1,079	578.6	2,783	1,692.1
		2nd	26	90.2	29	120.4	7	6.1	8	26.1	35	33.4	66	80.3	75	277.9	40	153.7
with MMI	quick-opt		128	95.6	205	112.2	54	19.3	66	34.9	147	53.5	233	176.2	716	386.1	369	267.5
	full-opt		118	204.4	183	438.5	52	34.5	56	77.1	137	104.2	229	437.9	661	861.9	330	447.4
	quick-full-special (recompile)	1st	128	95.5	203	112.8	56	19.8	65	33.7	147	53.1	272	241.3	710	372.6	368	269.9
		2nd	14	74.2	19	26.1	6	4.7	6	17.6	28	26.9	27	57.1	37	100.5	37	129.5
		3rd	6	32.3	8	10.7	2	1.7	3	7.1	15	18.9	9	25.8	13	55.4	13	34.7
		Inst	6	2.5	8	3.1	2	1.1	3	1.1	18	8.8	12	3.9	18	6.5	16	5.9

indicates that our MMI is reasonably fast and is comparable to no-opt compiled code.

Table 3 shows both the number of compiled methods and the code size in these program startup runs. In this table, the numbers for recompilation system, both noopt-full and MMI-all, are not presented, since no recompilation activity occurred in the program startup phase and therefore the numbers are mostly similar to those of their corresponding baseline configurations, noopt-only and MMI-quick, respectively. The differences in the number of compiled methods among no MMI cases are primarily caused by the varying degrees of applying method inlining at each optimization level. The table shows the significant differences in the number of compiled methods between no-MMI and with-MMI configurations. With MMI, only 2-7% of the executed methods are compiled, even considering the effects of method inlining. As for the generated code size, it is an order of magnitude larger with no-MMI configurations than with MMI configurations. The code expansion factor from the bytecode size can be up to 10x without MMI, while it is less than 1x for with-MMI configurations.

Figure 6 is the corresponding performance chart for the steady state program runs, and Table 4 shows the numbers for compiled methods and code size at that time. In Table 4, the two configurations that involve recompilation are shown with the numbers for each level of compilation separately. Also MMI-all configuration is indicated with the number of methods targeted by the instrumenting profiler and its code and table space for specialization

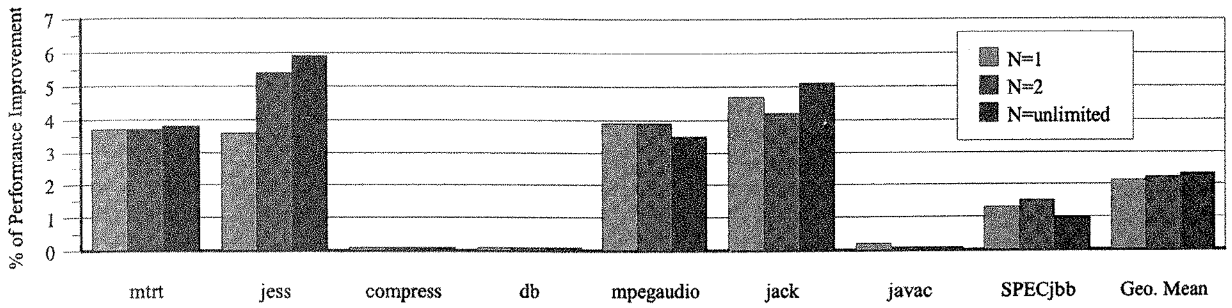


Figure 7. Performance improvement by code specialization for three different N (max. number of specialized versions for a method). Each bar indicates the percentage of improvement over the system with code specialization disabled.

Table 5. Statistics of code specialization on SPECjvm98 and SPECjbb2000.

		mtrt	jess	compress	db	mpegaudio	jack	javac	SPECjbb
N = 1	# of specialized versions (methods)	6 (6)	8 (8)	2 (2)	3 (3)	15 (15)	9 (9)	13 (13)	13 (13)
	% of code size increase	18.9	7.7	6.9	13.8	23.6	8.6	11.7	9.4
	% of hit ratio for specialized versions	100	80.5	60.2	100	99.7	95.7	51.8	90.3
N = 2	# of specialized versions (methods)	10 (6)	12 (8)	3 (2)	5 (3)	29 (15)	13 (9)	16 (12)	18 (13)
	% of code size increase	21.3	12.7	9.7	18.6	33.4	12.6	23.3	10.3
	% of hit ratio for specialized versions	100	92.7	99.9	100	99.7	96.8	79.2	93.5
N = unlimited	# of specialized versions (methods)	20 (6)	18 (8)	4 (2)	7 (3)	63 (15)	17 (9)	22 (12)	22 (13)
	% of code size increase	40.4	19.3	14.2	27.8	79.4	16.7	26.2	13.4
	% of hit ratio for specialized versions	100	98.6	100	100	99.9	97.5	89.8	97.8

value profiling. From Figure 6, three configurations, full-only, MMI-full and MMI-all, are top performers in this category. The recompilation system with compile-only approach, noopt-full, also works quite well for many of the tests here by applying full optimization on performance-critical methods. Currently the same parameters for recompilation decision are used in this configuration as those in MMI-all, and thus we think the performance can be further improved to the level of other full optimization configurations by adjusting the parameters more appropriately.

The following observations can be made from Table 4 for our MMI-enabled dynamic recompilation system. First the number of methods compiled with quick optimization is, except for jack and javac, at most 30% of the total number of methods, among which the number of recompiled methods is roughly 10 to 15%. Therefore we can achieve the high performance attained by the full-only configuration by focusing on merely 3 to 4% of all methods. As described in [2], javac has a flat profile, involving many methods that are executed, and thus poses a challenge for the recompilation system. This characteristic caused a relatively higher number of quick optimizations in our system, 70% of the total methods executed. However, the number of fully optimized methods is around 4%, similar to the other test cases, showing that our recompilation decision process works quite well. As for Jack, the higher count of methods with quick optimization is caused by the additional recompilation request from EDO, after detecting some hot exception paths as inlining candidates, and this results in the better performance as shown in Figure 6.

Second, the increase in the size of the compiled code is small in comparison to that of the MMI-quick configuration, and the total size of all levels of compilations and the instrumenting profiler combined is well below that of the MMI-full configuration. The largest size for the recompiled code (for 2nd and 3rd) is for SPECjbb2000, but it is still much less than that of the bytecode. Again the compile-only approach shows a problem with the large size of the compiled code. The expansion factor from the bytecode size with no MMI configurations is from 7x to 10x, while it is around 3.5x on average, including the space overhead by the instrumenting profiler, with our MMI enabled recompilation system.

Overall our dynamic optimization system adapts very well to the requirements of both program startup and steady state performance, and also has strong advantages in terms of the system memory footprint.

6.3 Code Specialization

Figure 7 shows the percentage performance improvement from the code specialization over the system with 3rd level optimization disabled. The measurement was done with the same conditions as in the steady state performance runs in the previous subsection. Three cases are shown for the specialization parameter N, that is the maximum number of specialized versions per method was set to one, two and unlimited. Table 5 shows the number of specialized versions produced, the percentage increases in code size, and

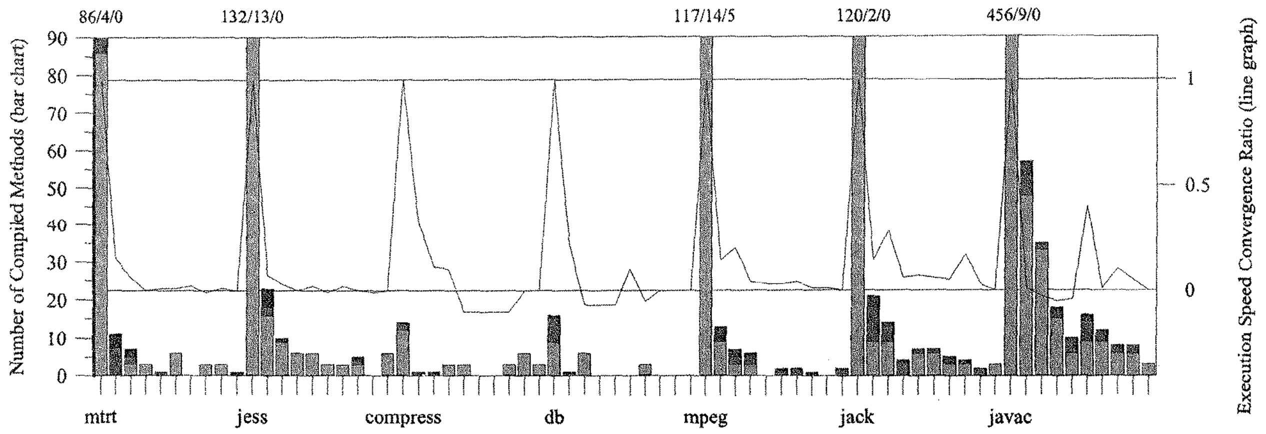


Figure 8. Change of compilation activity and the program execution time as program shifts its execution phase. Bar chart (left Y axis) shows the number of compiled methods for quick, full, and special optimizations, and line graph (right Y axis) shows execution time ratio from 1st to 10th runs

▨ quick-opt
■ full-opt
■ special-opt

the ratio showing the frequency of specialized code entry test, *SST*, succeeded, for all of the three cases.

A modest performance improvement, from 3 to 6%, can be observed for four benchmarks, while others do not show any significant difference. For those benchmarks which are sensitive to this specialization, approximately half of the 2nd level compiled code was specialized, and a 7 to 30% code size growth was observed for the cases of small number of specialization parameter N. The increased code size for mpegaudio, db, and javac seems to be excessively high relative to the resulting performance gain. One of the reasons for this problem is that the target of specialization in our current implementation is the whole method, rather than a smaller region in the method. When specialization is applied for a part of the method, a technique called *method outlining* (in contrast to method inlining) needs to be explored to allow multiple versions of specialized code to be generated for that part of the method.

The hit ratio of specialized versions code is quite high overall, considering the fact that only a limited amount of data sampling is performed in our instrumentation-based value profiling. This is because the variation of data for parameters or global variables is relatively small within a single benchmark. Jess is the only exception, that shows significant performance gain by producing multiple specialized versions from a single method.

Three benchmarks do not show any performance improvement from code specialization. Two of them, compress and db, have spiky profiles and only a few methods are heavily executed. But our impact analysis could not find any good candidates for specialization among the hot methods. On the other hand, javac has many equally important methods, and specializing only a few of them does not seem to provide any additional speedup.

6.4 Compilation Activity

In Figure 8, we show how the system reacts to changes in the program behavior with our dynamic optimization system. This was measured by running all the tests included in the SPECjvm98 with autorun mode, ten times each with a single JVM. The horizontal axis of the graph is equally partitioned by each run of the tests. The bar chart indicates the number of compiled methods with each level of optimization, and the line graph indicates the changes of the execution time from the first to the tenth run normalized by the time differences[4]. That is 1 corresponds to the first run and 0 corresponds to the tenth run. In the case of compress and db, the best timing appeared in the earlier runs, however, the irregularities in the graph after the best runs can be considered noise, since no compilation activity occurred.

The graph shows that the system tracks and adapts to the changes in the application program behavior quite well. At the beginning of each test, a new set of classes is loaded and the system uses the quick optimization compilation for a fair number of methods. As the program executes several runs in the same test, the system identifies a working set of hot methods, and promotes some of them to full or special optimization compilation. The execution time, on the other hand, is consistently improved after the first run for many of the tests by successful method promotions from the interpreted code to the 1st level, and then to the 2nd and 3rd level compiled code. In two of the tests, jack and javac, the cost of recompilation seems to appear in the execution time. This is partly because we performed the measurement on a uni-processor machine and cannot hide the background compilation cost completely. No significant overhead can be observed in other tests, since the execution time usually decreases steadily. When one test program terminates and another test begins, the system reacts quickly to drive compilation for a new set of methods.

7. CONCLUSION

We have described the design and implementation of our dynamic optimization framework, that consists of a mixed mode interpreter, a dynamic compiler having three levels of optimization, a sampling profiler, a recompilation controller, and an instrumenting profiler. Performance results show that the system can effectively work for initiating each level of compilation, and can achieve high

[4] This looks similar to the corresponding figure in [3], but note that the horizontal axis here is the number of runs, while it represents time partitioned into fixed-size intervals in [3].

performance and a low code expansion ratio in both program startup and steady state measurements in comparison to the compile-only approach. Owing to its zero compilation cost, the MMI allowed us to achieve an efficient recompilation system by setting appropriate tradeoff levels for each level of optimizations. We also described the design and implementation of automatic code specialization, which is used for the highest level of optimization compilation. This exploits the impact analysis and the dynamically-generated instrumentation mechanism for runtime parameter and global variable value sampling. The experiment shows that the technique can make a modest performance improvement for some benchmark programs.

In the future, we plan to further refine the system to improve the total performance by employing feedback-directed optimizations including more effective specialization, context sensitive method inlining using runtime profile data, and some optimizations based on runtime exception profiling.

8. ACKNOWLEDGMENTS

We would like to thank all the members of the Network Computing Platform group in IBM Tokyo Research Laboratory for helpful discussions and comments on an earlier draft of this paper. We are also grateful to John Whaley for prototyping the sampling profiler system. Finally the anonymous reviewers provided many valuable suggestions and comments on the presentation of the paper.

9. 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*, pp. 280-290, Jun. 1998.

[2] O. Agesen and D. Detlefs. Mixed-mode Bytecode Execution. Technical Report SMLI TR-2000-87, Sun Microsystems, 2000.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimizations in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications, OOPSLA '00*, Oct. 2000.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimizations in the Jalapeño JVM: The Controller's Analytical Model. In *Proceedings of the ACM SIGPLAN Workshop on Feedback-Directed and Dynamic Optimization, FDDO-3*, Dec. 2000.

[5] M. Arnold, B.G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN '01 Conference on Program Language Design and Implementation*, pp. 168-179, Jun. 2001.

[6] 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*, pp. 149-158, May 1996.

[7] T. Autrey and M. Wolfe. Initial Results for Glacial Variable Analysis. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, Aug. 1996.

[8] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp. 1-12, Jun. 2000.

[9] 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.

[10] M.G. Burke, J.D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java, In *Proceedings of the ACM SIGPLAN Java Grande Conference*, pp. 129-141, Jun. 1999

[11] B. Calder, P. Feller, and A. Eustace. Value Profiling. In *30th International Conference on Microarchitecture*, pp. 259-269, Dec. 1997.

[12] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Languages. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 146-160, Jul. 1989.

[13] M. Cierniak, G.Y. Lueh, and J.M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp. 13-26, Jun. 2000.

[14] C. Consel and F. Noel. 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*, pp. 145-156, Jan. 1996

[15] J. Dean, C. Chambers, and D. Grove. Selective Specialization for Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 93-102, Jun. 1995.

[16] J. Dean and C. Chambers. Towards Better Inlining Decisions Using Inlining Trials. In *Proceedings of the ACM SIGPLAN '94 Conference on LISP and Functional Programming*, pp. 273-282, Jun. 1994.

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

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

[19] 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*, pp. 293-304, May 1999.

[20] U. Hölzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Ph.D. Thesis, Stanford University, CS-TR-94-1520, Aug. 1994.

[21] 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, Jul. 1996.

[22] IBM Corporation Inc. "WebSphere Software Platform", documentation available at http://www.ibm.com/websphere 2000.

[23] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, Y. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler. In *Proceedings of ACM SIGPLAN*

Java Grande Conference, pp. 119-128, Jun. 1999.

[24] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications, OOPSLA '00*, pp. 294-310, Oct. 2000.

[25] Just System Corp. "IchitaroArk for Java", available at http://www.justsystem.com/ark/index.html, 1998.

[26] M. Kawahito, H. Komatsu, and T. Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap. In *Proceedings of the 9th International Conference on Architectural Support on Programming Languages and Operating Systems*, Nov. 2000.

[27] A. Krall. Efficient JavaVM Just-in-Time Compilation. In *Proceedings of International Conference on Parallel Architecture and Compilation Technique*, Oct. 1998.

[28] 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*, pp. 281-292, Jun.1999.

[29] T. Ogasawara, H. Komatsu, and T. Nakatani. A Study of Exception Handling and its Dynamic Optimization for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications, OOPSLA '01*, Oct. 2001.

[30] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pp. 1-12, Apr. 2001.

[31] 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*, pp. 120-131, Jan. 1997.

[32] M. Poletto, D. 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*, pp. 109-121, Jun. 1997.

[33] V.C. Sreedhar, M. Burke, and J.D. Choi. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proceedings of the ACM SIGPLAN '00 Conference of Program Language Design and Implementation*, pp. 196-207, Jun. 2000.

[34] M.D. Smith. Overcoming the Challenges to Feedback-Directed Optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00)*, pp. 1-11, Jan. 2000.

[35] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks, available at http://www.spec.org/osg/jvm98 and SPECjbb-2000 available at http://www.spec.org/osg/jbb2000.

[36] T. Suganuma, 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.

[37] Sun Microsystems. The Java Hotspot Performance Engine Architecture. White paper available at http://java.sun.com/products/hotspot/index.html, May. 2001.

[38] Sun Microsystems. HotJava^{TM} Browser available at http://java.sun.com/products/hotjava/index.html 1997.

[39] O. Traub, S. Schechter, and M.D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. Technical Report, Harvard University, 1999.

[40] J. Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of the ACM SIGPLAN Java Grande Conference*, Jun. 2000.

[41] J. Whaley. Dynamic Optimization through the Use of Automatic Runtime Specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[42] Wind River Systems Inc. "IceStorm Browser 5", available at http://www.icesoft.no/icebrowser5/index.html 2000.

[43] 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 *Proceedings of International Conference on Parallel Architecture and Compilation Technique*, Oct. 1999.