Design and Evaluation of Dynamic Optimizations for a Java Just-In-Time Compiler

Java Just-In-Time コンパイラにおける動的最適化の設計と評価

2007年 7月

Toshio Suganuma

菅沼 俊夫

Design and Evaluation of Dynamic Optimizations for a Java Just-In-Time Compiler

Toshio Suganuma

Abstract

The Java™ programming language has gained widespread popularity in the industry, offering many benefits to programmers such as dynamic class loading and reflections. Unfortunately the dynamic nature of the language presents a number of challenges for implementing an efficient Java Virtual Machine (JVM), and especially a Just-In-Time (JIT) compiler. However, at the same time this dynamic execution environment presents opportunities for potential performance advantages over the traditional static compilation model, because dynamic optimizations can exploit runtime profile information. Although there are currently several Java implementations that claim to include dynamic optimizations, few of these JVMs have released detailed information about their internal designs. This dissertation presents a comprehensive study of the dynamic optimizations for Java that were designed for and implemented in an industry-leading, high performance IBM Java JIT compiler.

The first contribution of this dissertation is to describe the design and implementation of a dynamic optimization framework for Java. Our system uses a multi-level execution model divided between a mixed-mode interpreter and a recompilation framework. The mixed-mode interpreter allows the efficient mixed execution of interpreted and compiled code, and covers a large number of unimportant methods without incurring any compilation costs. The dynamic compiler supports three levels of optimization to provide a balanced steps for the tradeoff between the compilation overhead and the compiled code quality. We also designed a reliable and lightweight program profiling system, combining three different techniques, depending on the profiler characteristics and target compilation levels; a counter-based profiler, a sampling-based profiler, and an instrumentation-based profiler. We implemented and evaluated our framework in the IBM JVM and JIT. Our results demonstrate that our configuration provides significant advantages in terms of performance and compilation overhead compared to other strategies, including the compile-only approach, both in the program startup and steady state phases.

The second contribution of this dissertation is to describe the design and implementation of two profile-directed optimizations, profile-directed method inlining and dynamic code specialization, both built on top of the dynamic optimization framework described in the above. For method inlining, we rely solely on the runtime profile information on call site distribution and invocation frequencies to decide which methods should be inlined along which call paths. In our experiments, we obtained significant improvements in both performance and compilation

overhead across a variety of benchmarks. For code specialization, which takes advantage of a program's runtime invariant or semi-invariant behavior for optimizations, we employ an impact analysis technique to estimate the benefit of specialization, and collect value profile information only when the optimization on a variable is deemed beneficial. Our experiments show a modest performance improvement with this technique.

The third contribution of this dissertation is to describe the design and implementation of a region-based compilation technique, consisting of region formation, partial inlining, region exit handling, and region-aware optimizations, which are effective in dynamic compilation environments. This implementation is also built on top of the dynamic optimization framework described in this dissertation. Our system employs a dataflow-based intra-method region selection algorithm that uses both static heuristics and a dynamic profile, and then integrates the algorithm in the inlining process to extract effective inter-procedural regions. When the program attempts to exit from a region boundary at runtime, we trigger recompilation and perform on on-stack replacement (OSR) to continue the execution from the corresponding entry point in the recompiled code. Our empirical evaluation demonstrates that our system can improve the performance and reduce the compilation overhead significantly.

Acknowledgements

I am deeply indebted to many people who made this dissertation possible. I would like to start by thanking my advisor, Professor Yoichi Muraoka, for the guidance and support that he gave me for completing the dissertation work. I am thankful for the time that he spent working with me, and his continued support throughout the time I worked on the dissertation.

I would also like to thank the dissertation committee members, Professor Katsuhiko Kakehi, Professor Yoshiaki Fukazawa, and Professor Hironori Kasahara, who spent a lot of their valuable time examining the dissertation, and who gave me valuable comments and advice for improving its quality.

I am grateful to Toshio Nakatani, my manager in the IBM Tokyo Research Laboratory for more than ten years, for his continued support and encouragement for completing this dissertation work. He created a productive and enjoyable research environment for working on the projects of optimizing the Java Just-In-Time compiler, the work which is the basis of this dissertation. Other contributing members in the Systems group (formerly called the Network Computing Platform group), including Hideaki Komatsu, Tamiya Onodera, Kazuaki Ishizaki, Kiyokuni Kawachiya, Takeshi Ogasawara, Osamu Gohda, Mikio Takeuchi, Motohiro Kawahito, Toshiaki Yasue, Kazunori Ogata, Akira Koseki, and Tatsushi Inagaki, worked with me to improve various aspects of the Java performance throughout the entire period of the long-running project. The work described in this dissertation would not have been possible without the state-of-the-art, industry-leading Java Virtual Machine and the Just-In-Time Compiler, which we developed from scratch through the tremendous efforts of all of the group members.

I also thank my colleagues of the IBM Java performance team in Austin, Bill Alexander, Mike Collins, and Weiming Gu, for their helpful discussions and analysis for identifying opportunities to improve the overall performance. I also thank the IBM Toronto Laboratory for continuing the development and support of the product of IBM Java JIT compiler. I am also indebted to the former members of our group, including Hiroshi Dohji, Masashi Doi, Hiroyuki Momose, Arvin Shepherd, Janice Shepherd, Kunio Tabata, Akihiko Togami, and John Whaley, for their respective contributions to improve the JIT compiler.

I would like to express my special appreciation to my parents for bringing me up and for giving me support for my education. And last, but certainly not least, I want to thank my wife Natsue and my son Hiderou for their continual support. They were always there both in good times and hard times, and encouraged me when I needed it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Java™ [51, 8] has gained widespread popularity in the industry since its advent in 1995, and it is now regarded as one of the most important programming languages, on a par with such established languages as C and C++. Java is a very popular choice for creating server side applications, such as Web Services based on the J2EE standard [102]. On the client side, although there have been some stumbling blocks inhibiting the adoption of Java, it is now being used for IDE platform development, as in Eclipse [49], and for some rich client applications [65] as well.

Java offers a number of advantages to programmers, such as lazy class loading and dynamic installation of software components, through such innovative features as dynamic class loading [78] and reflection [103], which are both integral parts of the language. Unfortunately, some of these advantages come at the cost of decreased performance, because it is more difficult or even impossible to perform certain optimizations when an optimizing compiler cannot assume certain knowledge about the whole program. Thus, in order to overcome the challenges due to the dynamic nature of the language, many of today's Java Virtual Machines (JVMs) employ a Just-in-Time (JIT) compiler to dynamically compile and optimize Java bytecode into native code for the underlying running platform at runtime .

There have been many research projects devoted to developing efficient dynamic compilers for Java. 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, dynamic compilation systems have to reconcile the conflicting requirements between fast compilation speed and fast execution performance. The system needs to generate highly efficient code for high performance, but at the same time, the compilations should be lightweight enough to avoid startup delays or intermittent execution pauses that may occur due to the runtime overhead of the dynamic compilations. This tradeoff between the compilation overhead and the performance benefit is a crucial issue for dynamic compilation systems.

However, at the same time this dynamic compilation environment offers potential performance advantages over the traditional static compilation model, since dynamic

compilers can exploit the runtime profile information from the current execution of a program. This allows them to seek opportunities for higher performance, and is a significant advantage over static compilers.

The uses of profile information can be divided into two categories. One is for detecting the programs' hot spots for applying optimizations (to decide *what* to optimize), and the other is for determining a strategy for better executing optimizations from current program behaviors (to decide *how* to optimize). The first type of profile usage allows us to avoid the overhead of optimizing all methods, and thus is particularly beneficial for focusing the compilation resources only on a limited number of performance critical methods. Many state-of-the-art JVMs [10, 38, 87] indeed employ this *selective optimization* mechanism. The second type of profile usage, however, requires more advanced mechanisms for runtime profiling and optimizing programs, and thus only one Java implementation, Jikes RVM, has explored the challenges to date [14].

This dissertation presents a comprehensive study of dynamic optimizations for Java, and our implementations of both what-to-optimize and how-to-optimize features with runtime profile information. Our system has been designed and implemented in the industry-leading high performance IBM Java JIT compiler.

We first construct a simple, but efficient and high-performance dynamic optimization framework, which provides an infrastructure for later profile-directed optimizations. Our approach is to employ a multi-level execution model divided between a mixed mode interpreter and a recompilation framework. The mixed mode interpreter allows the efficient mixed execution of interpreted and compiled code by supporting a lightweight mechanism of calling and being called by dynamically compiled code. It can cover a large number of methods that are not frequently invoked or computationally intensive in the target application programs without incurring any compilation costs. The dynamic compiler supports three levels of optimization to provide balanced steps for the tradeoff between the compilation overhead and the compiled code quality. We combine two different techniques for the program profiling mechanisms in the compiled code. One is a continuously operating, lightweight, sampling-based profiler to detect the hot methods of a given program, and the other is an instrumentation-based profiler that can dynamically install profiling code into selected target code to collect detailed runtime value information. This code-instrumentation technique does not involve target code recompilation, and is reasonably lightweight and effective for collecting a fixed amount of sampled data for a program's hot regions.

We then describe several optimizations designed and implemented on top of the dynamic optimization framework: profile-directed method inlining, dynamic code specialization, and region-based compilation. All of these are fully automated with no programmer intervention required for profiling and performing the optimizations.

Method inlining, a well-known and very important technique in optimizing compilers, expands the target procedure body at the method invocation call sites, and it defines the scope of the compilation boundary. Thus it is one of the optimizations that have

a significant impact on both system performance and total compilation overhead. The dynamic compilation systems have the advantage of being able to apply this expensive but effective optimization selectively, rather than statically, based on the dynamic program execution behavior, and therefore only at the program locations that are demonstrably shown to provide performance benefits. We can rely on the online profile information about call site distribution and invocation frequencies to decide as to which methods should be inlined along which call paths. Among the many existing static heuristics for method inlining, the only one we use is to always inline very small methods.

Code specialization is another example of profile-directed optimization implemented in our system. This is a technique to take advantage of the program's runtime invariant or semi-invariant behavior [25] for exploiting further optimizations. We first employ an impact analysis at a lower optimization level to evaluate the optimization opportunities and to estimate the benefit of specialization regarding how much better code we would be able to generate if we knew a specific value or the type of the variables. When the method is reoptimized at the highest optimization level, the system collects runtime information for those variables deemed beneficial in the impact analysis, and then generates code for a specialized version if the values are sufficiently biased and the result seems profitable.

Region-based compilation is a speculative optimization technique using profile information. With this optimization, we no longer treat methods as the unit of compilation, as in traditional method-based or function-based compilation. Instead, we select only those portions that are identified as non-rare paths. The term *region* refers to a new compilation unit, which results from collecting code from several methods of the original program but excludes the evidently rarely executed portions of these methods. This technique is especially useful for a dynamic compiler, because 1) we can use profile information from currently executing code for the region selection, 2) we can expect significant reductions of compilation overhead, and 3) we can defer code generation for unselected regions until the code is actually executed at runtime. We explored several strategies to find those that are most effective for the dynamic compilation environment, and designed three key components to support the technique: region selection, partial inlining, and the region exit handler.

1.1 Thesis Contributions

The specific contributions of this dissertation can be divided into the following three areas.

1.1.1 Dynamic Optimization Framework

The first contribution of this dissertation is to describe the design and implementation of a dynamic optimization framework for Java. Our system uses a multi-level execution model divided between a mixed mode interpreter and a recompilation framework. The mixed mode interpreter allows the efficient mixed execution of interpreted and compiled code,

and covers a large number of unimportant methods without incurring any compilation cost. The dynamic compiler supports three levels of optimization to provide a balanced steps for the tradeoff between the compilation overhead and the compiled code quality. We also designed a reliable and lightweight program profiling system, combining three different techniques, depending on profiler characteristics and target compilation levels; a counter-based profiler, a sampling-based profiler, and an instrumentation-based profiler. We implemented and evaluated our framework with the IBM JVM and JIT compiler. The results demonstrate that our configuration provides significant advantages in terms of performance and compilation overhead compared to other strategies, including the compile-only approach, both in the program startup and steady state phases.

1.1.2 Profile-Directed Optimizations

The second contribution of this dissertation is to describe the design and implementation of two profile-directed optimizations built on top of the dynamic optimization framework described in the above: profile-directed method inlining and dynamic code specialization. For method inlining, we solely rely on the runtime profile information on call site distribution and invocation frequencies in order to decide which methods should be inlined along which call paths. In our experiments, we obtained significant improvements in both performance and compilation overhead across a variety of benchmarks. For code specialization, which takes advantage of program's runtime invariant or semi-invariant behavior for optimizations, we employ an impact analysis technique to estimate the benefit of specialization, and collect a variable's profile information only when an optimization on that variable is deemed likely to be beneficial. Our experiment shows a modest performance improvement with this technique.

1.1.3 Region-Based Compilation

The third contribution of this dissertation is to describe the design and implementation of a region-based compilation technique effective for dynamic compilation environments, which consists of region formation, partial inlining, region exit handling, and region-aware optimizations. This implementation is also built on top of the dynamic optimization framework described in this dissertation. Our system employs a dataflow-based intra-method region selection algorithm that uses both static heuristics and dynamic profiles, and then integrates the algorithm into the inlining process to extract effective inter-procedural regions. Our empirical evaluation demonstrates that our system can improve the performance and reduce the compilation overhead significantly.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 gives a brief overview of the system characteristics and features of our Java JIT compiler that are closely related to this dissertation. Since method inlining is one of the key optimizations in the following chapters, it gives a detailed description of the inlining decision heuristics. Chapter 3 describes in detail the design and implementation of the system architecture of the dynamic optimization framework, presents some problems that required special consideration, and gives a detailed evaluation of the system in terms of both performance and compilation overhead, using a variety of applications and industry-standard benchmarking programs.

The following two chapters describe our optimizations designed and implemented on top of the dynamic optimization framework. Chapter 4 describes our design and implementation of the profile-directed optimizations, that is, the dynamic instrumentation profiling mechanism followed by the detailed description of the profile-directed method inlining and the dynamic code specialization. The impact on both performance and compilation overhead when applying these techniques is also presented, for each technique alone and for their combination. Chapter 5 presents the region-based compilation technique, describing the detailed design decisions on region formation, partial inlining, and region exit handling. It also presents extensive experimental results on the impact of the technique for both performance and compilation overhead.

Chapter 6 summarizes the related work for the topics of dynamic optimizations, instrumentation, profile-directed optimizations, and code specialization, for both Java and non-Java systems. Finally, Chapter 7 presents our conclusions, and outlines future research directions.

Chapter 2

Background: Structure of Our JIT Compiler

This chapter gives a brief overview of the underlying Java JIT compilation system, and provides a detailed description of those components that are directly relevant to this dissertation.

The overall structure of the JIT compiler is shown in Figure 2.1. We employ three different intermediate representations (IRs) to perform a variety of optimizations. At the beginning, the given bytecode sequence is converted to the first IR, called *extended bytecode* (EBC).[1] The EBC is stack-based and very similar to the original bytecode, but it annotates additional type information for the destination operand in each instruction. Most EBC instructions have a one-to-one mapping with the corresponding bytecode instructions, but there are additional operators to explicitly express some operations resulting from method inlining and devirtualization. For example, if we inline a synchronized method, we insert `OPC_SYNC_ENTER` and `OPC_SYNC_EXIT` instructions at the entry and exit of the inlined code, respectively. This allows us to easily identify any opportunity to optimize redundant synchronization operations exposed by several stages of inlining synchronized methods. The instruction `OPC_CHA_PATCH` is used to indicate the code location for the runtime patch when unguarded devirtualized code is invalidated due to dynamic class loading [66].

Method devirtualization and method inlining are the two most important optimizations applied on this IR. The class hierarchy is constructed and used, together with the results of the object typeflow analysis, to identify the compile-time monomorphic virtual call sites. The devirtualized call sites and static/nonvirtual call sites are then considered for inlining. Since EBC is a compact representation, method inlining is performed on this IR. It expands the target procedure body at the corresponding call sites, and defines the scope of each compilation target. A detailed description of the inlining policy is provided later in this chapter. All of the optimizations on the EBC following the method inlining

[1] EBC was used in an earlier version of our compiler as the sole IR.

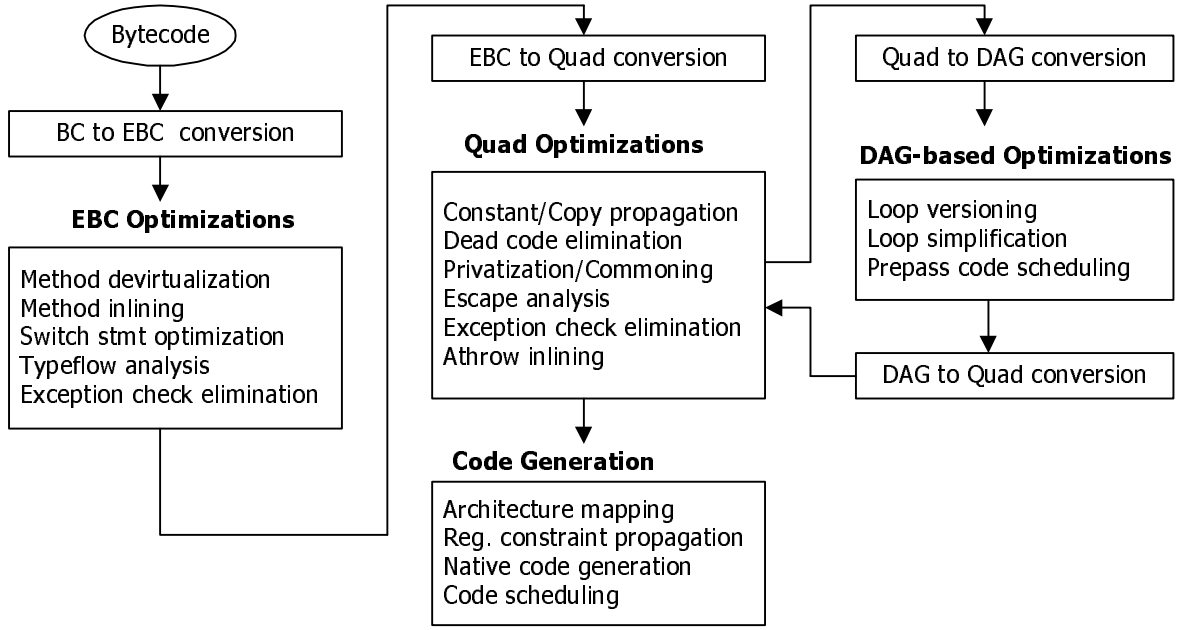


Figure 2.1: Structure of the JIT compiler. Three IRs, EBC, Quad, and DAG, are employed to perform a variety of optimizations.

are then performed to simplify the control flow or to eliminate redundant code to reduce the target code size before converting the code to the second IR.

The EBC is then translated to the second IR, called *quadruples*. This is a register semantic IR. The quadruples are n-tuple representations with an operator and zero or more operands. We have a set of fine-grain operators in quadruples for subsequent optimizations. For example, the exception checking operations implicitly assumed in some bytecode instructions are explicitly expressed in this IR for optimizers to easily and effectively identify redundant exception checking operations. Similarly, the class initialization checking operation is also explicitly expressed, in case a target class is resolved but not yet initialized for the relevant bytecode instructions. Another example is the virtual method invocation. The single bytecode instruction (`OPC_INVOKEVIRTUAL`) is separated into several operations: the operation for setting each argument, the null pointer check of the receiver object, the method table load, the method block load, and finally the method invocation itself. This will increase the opportunities for performing the optimization of common subexpression elimination for some of these operations between successive virtual method invocations.

The translation from EBC to quadruples is based on an abstract interpretation of the stack operations. In this process, we treat both local variables and stack variables in the same way to convert to symbolic registers. Figure 2.2 shows a simple example of the translation. The direct translation of stack operations produces many computationally redundant copy operations as shown in Figure 2.2(b). We apply copy propagation and

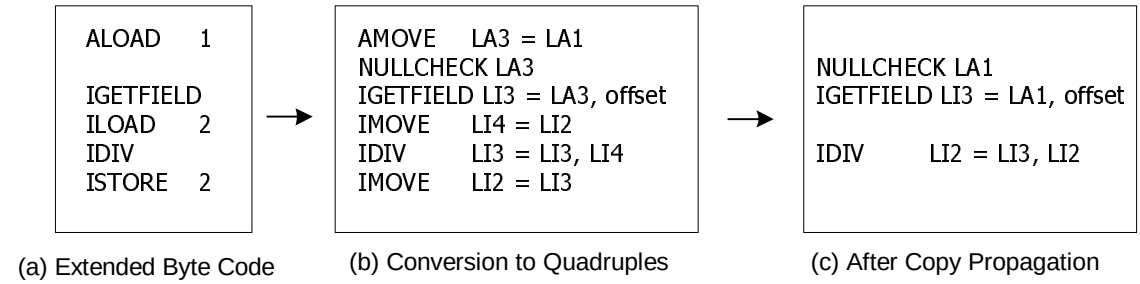


Figure 2.2: An example of translation from EBC to Quadruples: (a) Extended bytecode; (b) conversion to Quadruples; (c) after copy propagation.

dead code elimination immediately after the translation, and most of the redundancies that result from the direct translation of the stack operations can be eliminated, as shown in Figure 2.2(c).

We apply a variety of dataflow-based optimizations to the quadruples. Some dataflow analyses, exception check elimination, common subexpression elimination, and privatization of memory accesses are iterated to take advantage of the fact that the application of one optimization creates new opportunities for the other optimizations. The maximum number of iterations is limited by consideration of its impact on the compilation overhead (with different threshold values used between the methods containing loops and those without loops). The other optimizations on this IR include escape analysis, synchronization optimization, and athrow inlining.

The third IR is called a *directed acyclic graph* (DAG). This is also a register-based representation and is in the form of *static single assignment* (SSA) [41]. It consists of nodes corresponding to quadruple instructions and edges indicating both data dependencies and exception dependencies. Figure 2.3 shows an example of constructing the DAG for a method containing a loop. This simply indicates how the DAG representation looks and ignores all of the exception checks necessary for the array accesses within the loop. The actual DAG includes the nodes for exception checking instructions and the edges representing exception dependencies.

This IR is designed to perform optimizations, such as loop versioning and prepass code scheduling, that are more expensive but sometimes quite effective. The IR is converted back to quadruples before entering the code generation phase. Thus the conversion to DAG and applying the DAG-based optimization is a completely optional phase, and we need to apply the set of these optimizations judiciously and only to those methods that can benefit from the transformation.

All of the instructions of these three IRs are grouped into basic blocks (BBs). Our BBs are extended in the sense that they are not terminated by instructions which constitute potential exceptions, as in the factored control flow graph [35]. The BBs are ordered using the branch frequencies collected during the mixed mode interpreter (MMI) execution. Since the branch history information is limited to the first few executions in the MMI,

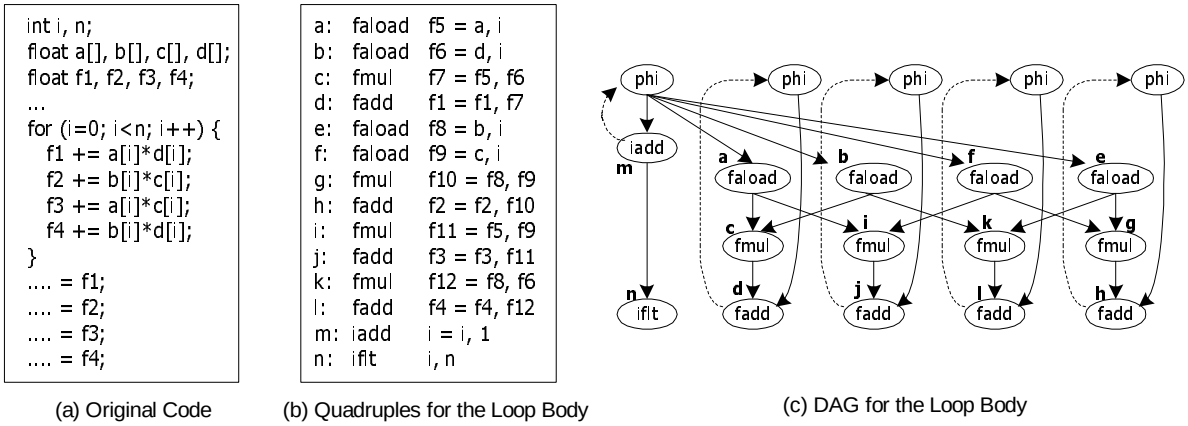


Figure 2.3: Example of constructing a DAG for a loop-containing method.

we do not use the technique of code positioning guided solely by profile information [88]. Instead, we combine the use of this information with some heuristics so that we can place backup blocks generated by versioning optimizations at the bottom of the code. We also use the profile information to select the depth first order for the *if-then-else* blocks. This separates the BBs of frequently and infrequently executed paths.

The compiler is designed to be very flexible so that each optimization can be enabled or disabled for a given method. At a minimum, we need to perform bytecode-to-EBC conversion, EBC-to-quadruple translation, and native code generation from the quadruples, even if all of the optimizations are skipped. Method inlining can be applied either for tiny methods only or based on more aggressive static heuristics. The number of iterations of the dataflow analyses can be adjusted. The generation of a DAG representation and the optimizations on the DAG are optional as mentioned above. We exploit these capabilities in the design of the dynamic optimization framework as described in the next chapter.

Method Inlining

Object-oriented languages encourage data encapsulation through the use of methods, resulting in frequent method invocations. In addition, object-oriented languages support dynamically dispatched (virtual) calls, where the method called depends on the runtime type of the receiver object. Efficient implementations of virtual dispatch [60] help reduce the direct overhead of virtual method invocation, but method inlining remains an important optimization for effective implementations of object-oriented languages.

Our dynamic compiler is able to inline any method regardless of the contexts of the call sites and the types of the caller or callee methods. For example, there is no restriction in inlining synchronized methods or methods with try-catch blocks (exception tables), nor against inlining methods into call sites within a synchronized method or a synchronized block. For the runtime handling of exceptions and synchronizations, we create a data structure indicating the inline context tree within the method, and map each potentially

excepting instruction or call site address to a corresponding node in the tree structure. Using this information, the runtime system can identify the associated try region identification or any necessary object-unlocking actions based on the current program counter. Thus the inlining decision can be made purely from the cost-benefit estimate for the method being compiled.

There are many methods in Java that simply result in a few instruction codes when compiled, such as accessor methods. We call these *tiny methods*. A tiny method is one whose estimated compiled code is equal to or less than the corresponding call site code sequence (argument setting, volatile registers saving, and the call itself). That is, the entire body of the method is expected to fit into the space required for the method invocation. Our implementation identifies these methods on the basis of the estimated compiled code size.

Since the invocation and frame allocation costs outweigh the execution costs of the method bodies for these methods, inlining them is considered completely beneficial and unable to cause any harmful effects in either compilation time or code size expansion. In fact, an empirical study showed that the tiny-only inlining policy has very little effect on compilation time, while it produces significant performance improvements over the no-inline case [99]. Thus, these methods are always inlined without any qualification. For non-tiny methods, we employ static heuristics to perform method inlining in an aggressive way while keeping the code expansion within a reasonable limit.

The inliner first builds a possibly large call tree of inlined scopes based on allowable call tree depths and callee method sizes using optimistic assumptions, and then looks at the total cost by checking each individual decision to come up with a pruned final tree. Whenever it performs inlining on a method, the inliner updates the call tree to encompass the new possible inlining targets within the inlined code. When looking at the total cost, we manage two separate budgets proportional to the original size of the method: one for tiny methods, and the other for any type of method. The inliner tries to greedily incorporate as many methods as possible from the given tree using static heuristics until the predetermined budget is used up. Currently the static heuristics consist of the following rules:

- If the total number of local variables and stack variables for the method being compiled (both caller and callees) exceeds a threshold, reject the method for inlining.
- If the total estimated size of the compiled code for the methods being compiled (both caller and callees) exceeds a threshold, reject the method for inlining.
- If the estimated size of the compiled code for the target method being inlined (callee only) exceeds a threshold, reject the method for inlining. This is to prevent wasting the total inlining budget due to a single excessively large method.
- If the call site is within a basic block that has not yet been executed at the time of the compilation, then it is considered a cold block of the method and the inlining is

not performed. On the other hand, if the call site is within a loop, it is considered a hot block, and extra efforts are made to inline the deeply nested call chains.

Throughout the process, the inliner devirtualizes dynamically dispatched call sites using both class hierarchy analysis (CHA) [46] and typeflow analysis. It produces either guarded code (via a class test or a method test [47]) or unguarded code (via code patching on invalidation [66]), depending on whether the call site can be assumed to be monomorphic or a single implementation. When more than two target methods are found, it performs devirtualization only when their profiles are available and the most beneficial target can be selected. A backup path is generated for both the guarded and unguarded devirtualization cases to execute when the compile-time assumption is invalidated at runtime. Thus it produces a diamond-shaped control flow with the nonvirtual call on the fast path and the backup path on the other. The nonvirtual call sites in the fast path then may or may not be inlined according to the static rules described above.

As a result of method inlining, a data structure called an *inlined method frame* (IMF) is produced for each call site to provide information about the inlining context for the runtime system, which needs to know the exact call stack and call context prior to inlining. For example, the security manager requires the correct depth of the current call-chain on the stack, or a runtime exception is raised. The exception handler also requires the inlining context of the call sites in order to keep track of the handlers for catching exceptions from the current context.

Chapter 3

Dynamic Optimization Framework

3.1 Introduction

Some of the very early Java Virtual Machines (JVM) provided Just-In-Time (JIT) compilation in a single-level execution model, employing a simple strategy of compiling every method with a fixed set of optimizations the first time it was invoked. The JIT compilers at this level were in general immature and implemented relatively simple and inexpensive optimizations only, and therefore they typically resulted in a limited level of performance. The compilation time was not an issue given the level of optimizations applied, but they could suffer from problems with large amounts of compiled code because of the simple all-methods compilation strategy. The JVMs in this category include Intel's early research JVM [2], Cacao [75], LaTTe [118], and the early version of the IBM JIT compilation system.

As more effective but time consuming optimizations were added to JIT compilers to improve the Java execution performance, the negative aspects of the dynamic compilation started to become problematic. Compilation overhead, typically in the form of application startup delays and unacceptable code size growth could no longer be ignored. Thus, the next generation of JVMs supported a two-level execution model, consisting of an interpreter and a single level of compilation or two different optimization levels of compilation. This type of JVM can bring higher performance in both application startup and steady state runs over the earlier generation of JVMs, while mitigating the compilation overhead by focusing the optimization efforts on a subset of the executed methods of the given program. Examples of JVMs in this category are the Intel JUDO system [38], version 3.0 of the IBM JIT compiler [96], and Sun's JDK for version 1.1.x.

The two-level execution model, however, still imposes some limitations. As we add yet more expensive optimizations, sometimes comparable to those employed by static compilers, the system cannot manage to balance the optimization effectiveness with the compilation overhead due to the increasing gap in the trade-off level between the two execution modes. Also, as the level of optimizations in the JIT compilation technology

matures, the runtime profile information is being exploited for new optimization opportunities. Thus a sophisticated framework for online profile collection and a feedback system has become required. This is one of the advantages of the dynamic compilation system that cannot be used by conventional static compilers. In this context, the high performance implementations of recent JVMs and JIT compilers are moving toward the exploitation of adaptive compilation techniques on the basis of online runtime profile information [10, 87, 97, 98].

Typically these systems have multiple execution modes. They begin the program execution using an interpreter or a baseline compiler as the first execution mode. When the program's frequently executed methods or critical hot spots are detected, they use the optimizing compiler for those selected methods to run in the next execution mode for higher performance. Most systems of this type have several different optimization levels to select from based on the invocation frequency or relative importance of the target methods. The higher-level optimizations may exploit some forms of profile information collected in the lower execution mode to generate better code.

In this chapter, we present detailed discussions of the design and evaluation of our dynamic optimization framework as implemented in a production-level Java JIT compiler. Our approach is to employ a mixed-mode interpreter and a three-level optimizing compiler, each of which has a different set of tradeoffs between compilation overhead and execution speed. A lightweight sampling profiler operates continuously during the entire period while applications are running to monitor the programs' hot spots. Detailed information on runtime behavior can be collected by an instrumentation profiler that dynamically installs and uninstalls the instrumentation code into and out of the specified target code. This provides value profiling mechanism available to follow-on profile-directed optimizations at a minimum runtime overhead. The experimental result shows that this configuration can provide significant advantages in terms of performance and compilation overhead compared to other strategies, including the compile-only approach, both at program startup and in steady state phases.

Section 3.2 describes the design and implementation of the system architecture of our dynamic optimization system in detail, and Section 3.3 presents some problems that required special consideration. Section 3.4 gives the detailed evaluation of the dynamic optimization system in terms of both the performance and compilation overhead, using a variety of applications and industry-standard benchmarking programs. Finally, Section 3.5 gives the summary of this chapter.

3.2 Design of the Dynamic Optimization Framework

The goals of our dynamic optimization system are twofold:

- To achieve the best possible performance using the same set of existing optimization capabilities for varying phases of application programs, including program startup,

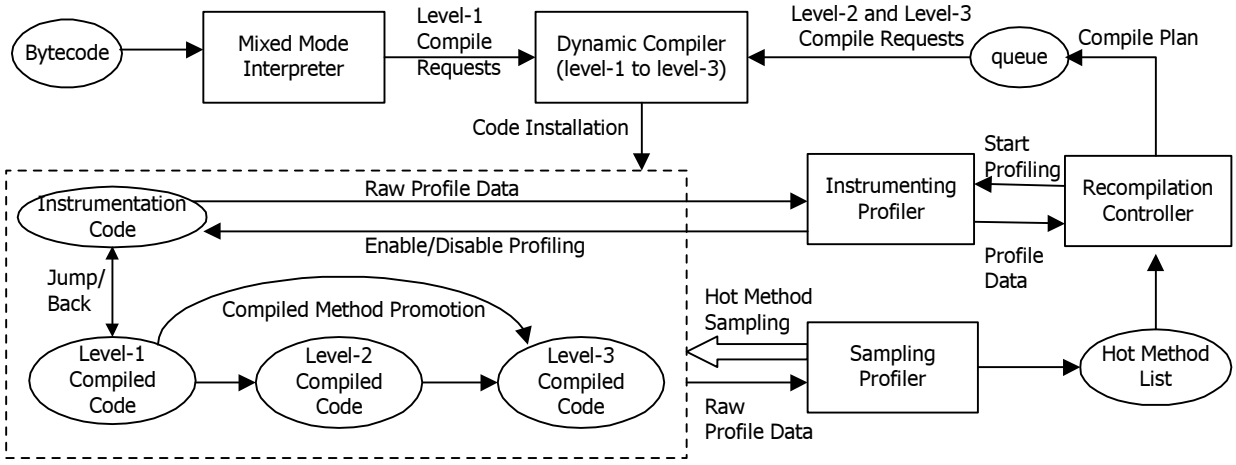


Figure 3.1: System architecture of our dynamic optimization system.

steady state, and phase shifts, while minimizing the compilation overhead.

- To provide a base framework that is highly extensible to allow adding various profile-directed optimizations, by employing practical and effective means of online profile information collection and a feedback system.

In order to achieve these goals, we need the system to be very selective about which methods it decides to compile and when and how it decides to compile them. The system 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, however, the system should respond aggressively to identify good opportunities for optimizations that can bring higher total performance. The two key elements for this requirement are: 1) a well-balanced multi-level recompilation system, and 2) a reliable and low overhead profiling system for method promotion. We addressed these points in the design of our system.

The overall architecture of our dynamic optimization system is as depicted in Figure 3.1. This is a multi-level compilation system, with a mixed mode interpreter (MMI) and three optimization levels (level-1 to level-3). In this section, we first describe each of the major components of the system in the following five subsections, and then discuss some of the design points we gave special consideration to in order to achieve our goals.

3.2.1 Mixed Mode Interpreter

Most of the methods executed in Java applications are neither frequently called nor loop intensive as shown by the measurements in Section 3.4.4, and the approach of compiling all methods is considered inefficient in terms of both compilation time and code size. 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 the two execution modes. It is roughly three times faster than an interpreter written in C.[1]

Initially, all methods are interpreted by the MMI. A counter for accumulating both method invocation frequencies and loop iterations is provided for each method and initialized with a threshold value. The counter is decremented whenever the method is invoked or loops within the method are iterated. When the counter reaches zero, we recognize that the method has been invoked frequently or is computation intensive, and the first 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 backward-branch case, the control is directly transferred to the JIT compiled code from the partially 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. If the iteration count is large enough, the JIT compilation is immediately invoked without waiting for the counter value to reach zero.

The collection of runtime trace information is another benefit of the MMI for use in JIT compilation. For any conditional branches encountered, the interpreter keeps track of whether or not it is taken to provide the JIT compiler with a guide for the branch direction at basic block boundaries. 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 performing partial redundancy optimizations [71].

3.2.2 Dynamic Compiler

The dynamic optimizing compiler in our system has the following three optimization levels. The compilation with level-1 optimization is invoked from the MMI and is executed as an application thread, while the level-2 and level-3 optimizations are performed by a separate compilation thread in the background.

1. *Level-1 (L1) optimization* employs only a limited set of the optimizations available in our system. Basically, it considers only those optimizations having very low

[1]This is based on comparisons against an interpreter written in C and compiled with a normal C compiler. Using GCC's label variables as specifically designed for interpreters should make the performance advantage smaller.

compilation overhead in terms of both compilation time and code size expansion. For example, it performs method inlining only for tiny methods. This can reduce the total compilation overhead significantly, as shown in a later section. It performs the devirtualization of dynamically dispatched call sites based on the class hierarchy analysis (CHA), and produces either guarded code or unguarded code [66]. The resulting devirtualized call sites are inlined only when the target method is tiny. Also, of the many dataflow-based optimizations, this level of optimization only uses very basic copy and constant propagation.

2. *Level-2 (L2) optimization* enhances level-1 by employing additional optimizations, including more aggressive full-fledged method inlining based on static heuristics (as described in Chapter 2), a wider set of dataflow optimizations, and an additional pass for code generation for improved code quality. It performs type-flow and pre-existence analysis [47] to safely remove guard code and back-up paths that resulted from devirtualization without requiring an on-stack replacement mechanism. The maximum number of iterations in the dataflow-based optimizations is still limited. 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.

3. *Level-3 (L3) optimization* is augmented with all of the remaining optimizations available in our system. Additional optimizations enabled at this optimization level include escape analysis (including stack object allocation, scalar replacement, and synchronization elimination), code scheduling, and DAG-based optimizations, such as pre-pass code scheduling and loop versioning. The maximum iteration count allowed for dataflow-based optimizations is increased.

The register allocation in our system is based on on-the-fly local allocation that works in parallel with code generation, not an independent single pass used for global allocation, as described in [96, 95]. We use a simple and fast algorithm for assigning registers. The same policy is used for all three optimization levels.

The internal representation is the same for all of these optimization levels. The relative differences in compilation time, generated code size, and the generated code's performance quality for each optimization level are shown in Section 3.3.

3.2.3 Sampling Profiler

The sampling-based profiler [113] gathers information about the program threads' execution. This profiling collector keeps track of the 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 count* associated with each method. The priority of the sampling profiling thread

is set higher than normal application threads to ensure periodic sampling at regular intervals.

Since the MMI has its own counter-based profiling mechanism, this sampling profiler only monitors compiled methods for reoptimizations. The profiler keeps the current hot methods in a linked list, sorted by the hotness counter values, and then groups them together and gives them to the recompilation controller at every fixed time interval to consider upgrade method recompilations. The sampling profiler operates continuously during the entire period of program execution to adapt effectively to the dynamic changes of the program's behavior. To minimize the bottom-line overhead, this profiler focuses only on detecting hot methods and does not collect other information such as the call context of the given method. Instead, additional information is collected by a different profiler for only selected hot methods, as described in the next section.

3.2.4 Instrumenting Profiler

There is another profiler, an instrumenting profiler, available when detailed information needs to be collected from a target method. The instrumenting profiler, according to an instrumentation plan from the recompilation controller, dynamically generates code for collecting specified data from the target method, and installs it into the compiled code by rewriting the entry instruction of the target. In order to minimize the performance impact, after collecting a predetermined amount of profile data, the generated instrumentation code automatically uninstalls itself from the target code by restoring the original instruction at the entry point.

Unlike the instrumentation sampling framework [15] used in Jikes RVM, our instrumenting profiler records specified data in a bursty manner for a certain interval of the program's execution. This mechanism allows the profiler to collect the information fairy quickly, and is therefore suitable to use when the recompilation controller identifies the hot methods and specifies the profile targets before recompiling those methods.

Thus the compiler can take advantage of the online profile information dynamically collected at runtime for the higher-level optimizations. This dynamic value profiling mechanism allows the intensive monitoring of the runtime behavior for a certain interval of the program's execution, and the dynamic installation and uninstallation of the instrumentation code is done without recompilation of the target code. This instrumenting profiler is used for collecting profile information on call site distributions, parameter values, and basic block execution frequencies to drive profile-directed optimizations and region-based compilation, as described in detail in Chapters 4 and 5.

3.2.5 Recompilation Controller

The key to our system is to make correct and reasonable decisions to selectively and adaptively choose methods for each level of optimization. The recompilation controller, which

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

The upgrade recompilation from level-1 compiled code to higher-level optimized code is triggered on the basis of the hotness count of the compiled method as detected by the timer-based sampling profiler. Depending on the relative hotness level, the controller promotes the method from level-1 compiled code to either level-2 or directly to level-3 optimized code. This decision is made based on different thresholds of the hotness level for each level-2 and level-3 method promotion.

The controller utilizes a decaying mechanism for the methods' hotness counters so that the system can check the application behavior most effectively. The decay parameter, which ranges from zero to one, controls how much of the hotness data should be carried from the previous interval to the next. A value of zero means the system purges all the previous data, and this should result in quick responses to changes in the application's behavior. A value of one means the system accumulates all of the previous hotness counts, and this is expected to detect hot methods more effectively for stable running programs. We set an appropriate fixed value for this decay parameter based on a wide range of experimental results.

The controller can also direct the instrumenting profiler to install instrumentation code for collecting further profile information such as parameter values and call site distribution for those hot methods. The additional profile data can be used for recompilations to guide more effective optimizations at the higher optimization levels. This function is enabled when the profile-directed optimization is turned on.

3.3 Discussion

Figure 3.2 shows a summary of the four execution modes in our system divided into the interpreter execution and compiled code with the three optimization levels. A different profiling system is used for each of the execution modes for promoting methods and for collecting additional information. In what follows, we describe some design issues and special considerations that affect several important points in our overall system architecture.

3.3.1 Tiny Methods

Method inlining is one of the optimizations that can significantly impact both compilation overhead and performance. In level-1 optimization, we consider method inlining only for extremely small target methods. This is based on the observation that there are a number of methods in Java that simply result in a single instruction of code when compiled, such

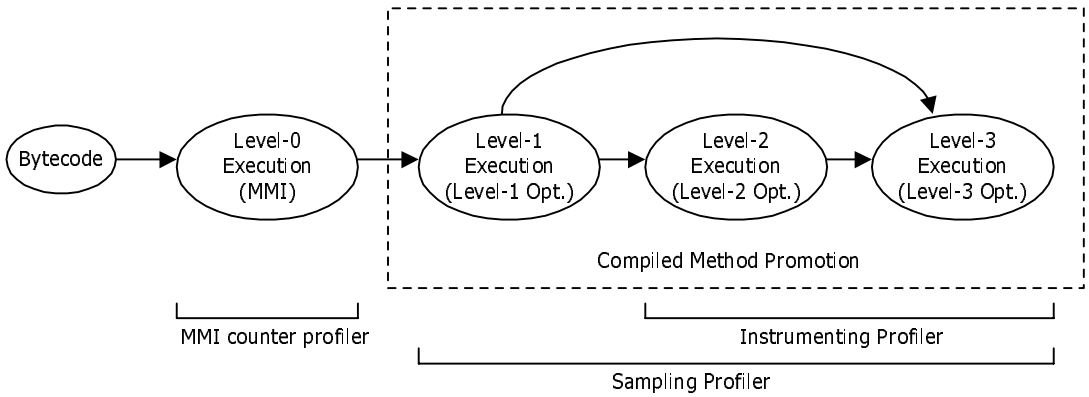


Figure 3.2: Four execution modes divided into the interpreter execution and three optimization levels of compiled code. A different profiling mechanism is used depending on the execution mode for promoting methods and for collecting additional information.

as just returning an object field value, or other methods that may turn out to be a very small number of instructions for their procedure body after compilation.

We identify tiny methods based on the estimated compiled code size.[2] Tiny methods are defined as those for which the estimated compiled code is equal or less than the corresponding call site code sequence (argument setting, volatile registers saving, and the call instruction itself). Therefore, the entire body of the method is expected to fit into the space required for the method invocation. This can save compilation overhead not only for the method inlining process, but also for the later optimization phases that traverse the entire resulting code block.

Figure 3.3 shows both the compilation overhead and performance impact with three different policies of method inlining when running two industry standard benchmarks. The data was collected based on our system configuration described in the above and using the methodology described in Section 3.4.1. In these charts, *full-inline* indicates using our static heuristics for inlining in level-2 and level-3 compilation, *tiny-only* means inlining only tiny methods in level-2 and level-3 as well as level-1 compilation, and *no-inline* performs no method inlining at any level of compilation regardless of the size of the target methods. Note that devirtualization is still enabled for all three cases.

In the top left chart, each bar gives a breakdown of where in the optimization phase time is spent for compilation. *Flow-graph* indicates the time for basic block generation, inlining analysis to determine the scope of the compilation boundary, and flow graph construction for the expanded inlined code. *Dataflow-opt* is the time for a variety of dataflow-based optimizations, such as constant and copy propagation, and redundant null-pointer and array-bound check elimination. *DAG-opt* is for DAG-based loop optimization and pre-pass code scheduling. *Code-gen* includes the time for register assignment, code

[2]This estimate excludes the prologue and epilogue code. The compiled code size is estimated based on the sequence of bytecodes, each of which is assigned an approximate number of instructions generated.

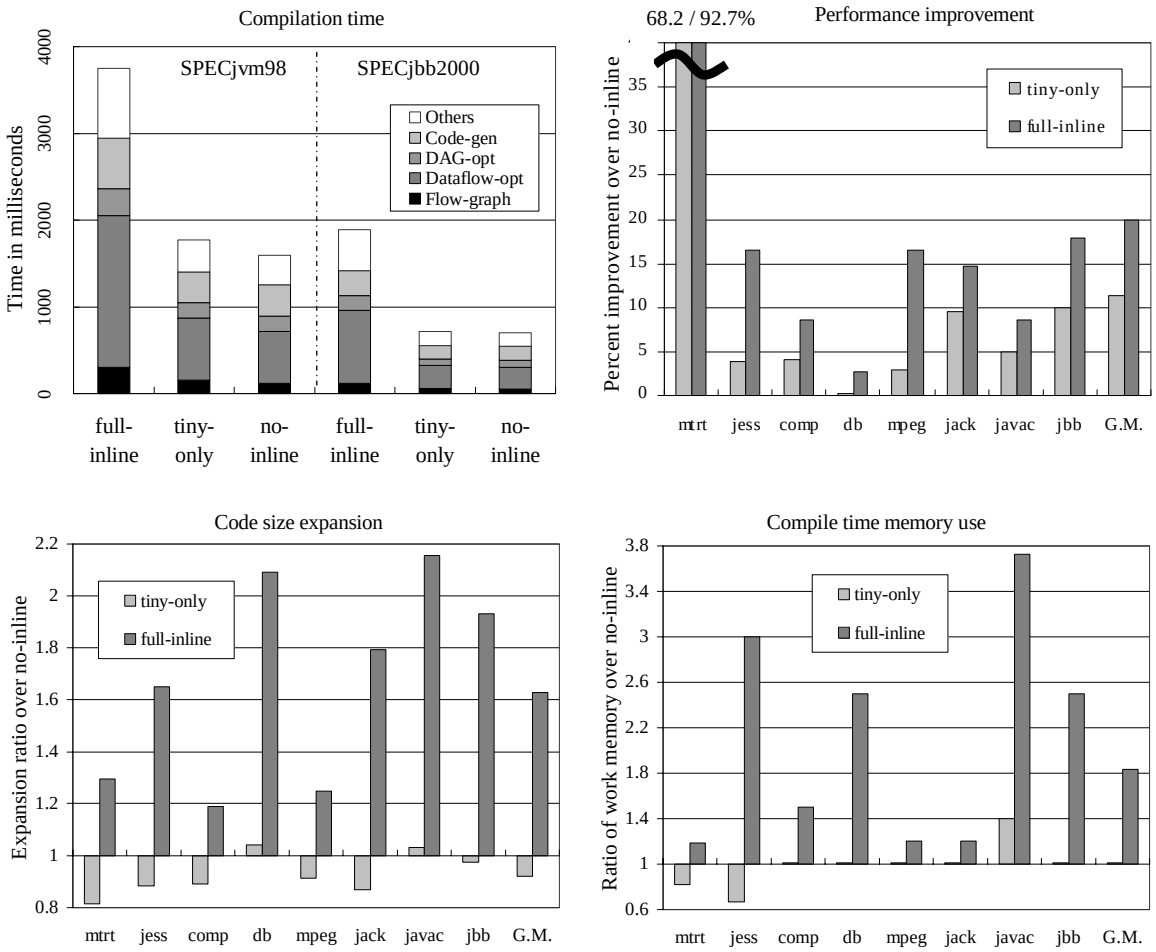


Figure 3.3: Compilation time and performance impact for SPECjvm98 and SPECjbb2000 benchmarks with three different inlining policies: full-inline, tiny-only, and no-inline. Top left chart indicates the compilation time breakdown and top right chart is the peak performance difference. Two charts in the bottom show the ratios of the code size expansion and the compile time memory usage. The data was collected using the methodology described in Section 3.4.1.

generation, and code scheduling. *Others* denotes the memory management costs and any other code transformations, including live analysis and peephole optimizations, each costing less than 5% of the total compilation time.

The top right chart shows the peak performance differences. This is based on the best time for each of the three inlining policies with minimum effect on compilation times. The two charts in the bottom show the ratios of the code size expansion and the compile time peak work memory usage for both the full-inline and the tiny-only policies over the no-inline policy.

As we can see in these charts in Figure 3.3, method inlining has significant impact on both compilation time and performance. With the full-inline policy, the time spent for

inlining analysis and flow graph construction itself is not a big part of the total overhead, but optimizations in later phases, the dataflow optimizations in particular, cause a major increase of the compilation time due to the greatly expanded code that has to be traversed. The code size expansion and the peak compiler memory use also have similar increases with this inlining policy.

The tiny-only inlining policy, on the other hand, has very little impact on compilation time, as expected, while it produces significant performance improvements over the no-inline case. For some benchmarks, it provides a majority of the performance gain that can be obtained using the full-inline policy. On average, it contributes a little over half of the performance improvement compared to the full-inline policy. From these two charts, our current policy of always inlining tiny methods is clearly justified.

3.3.2 Optimization Levels

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 an 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 dominate. Even if more expensive optimizations are implemented, the return for invested resources 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 compilation costs. 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.

Second, it is not clear whether it would be effective to have a larger number of optimization levels 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 in general complicate the selection process and more informative profiling data would be necessary to make correct decisions, which might add more overhead.

Thus, the actual number of optimization levels within a given system should be determined by experiments. A number of factors, such as optimization capabilities within the system, the tradeoff between the performance and the compilation overhead for each op-

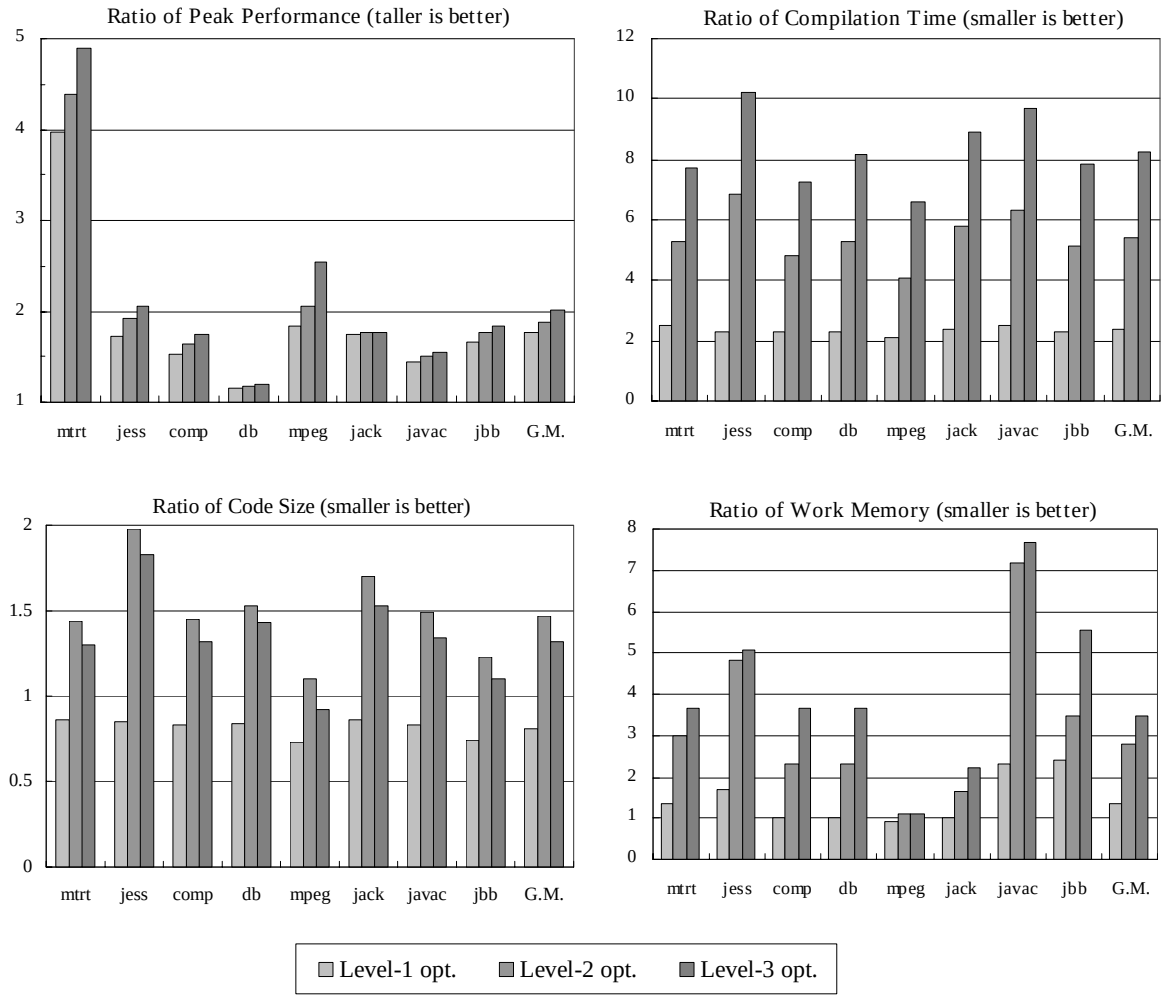


Figure 3.4: Comparisons of peak performance, compilation time, compiled code size, and peak compiler memory use between level-1 to level-3 optimizations. Level-0 compilation is used as a baseline in these graphs, in which we disable all the optimizations and generate code directly from the intermediate code. The benchmarks were run in the single execution mode without using MMI and profiling system.

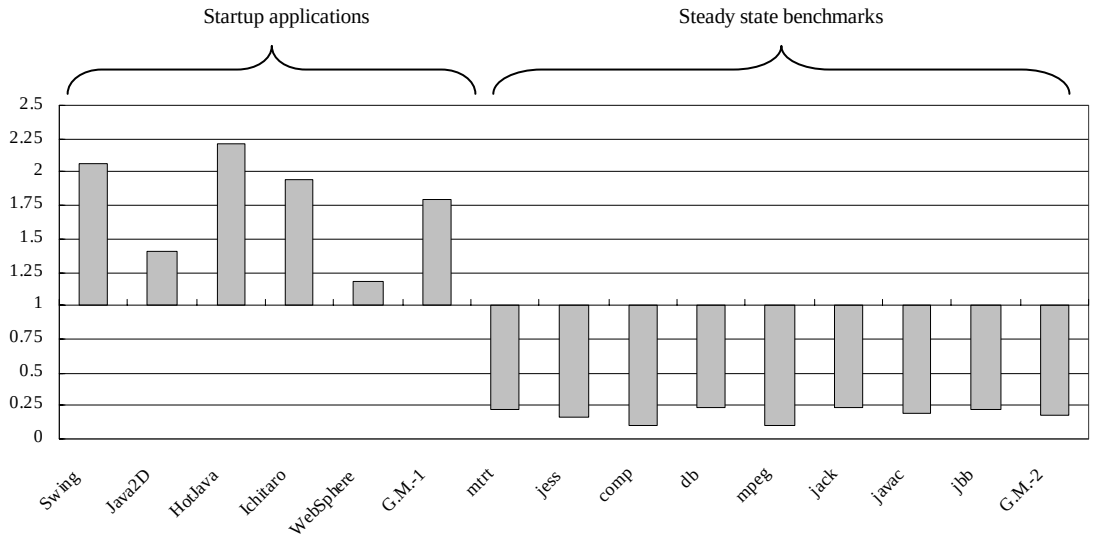


Figure 3.5: MMI performance for both startup applications and steady state benchmarks (SPECjvm98 and SPECjbb2000) over the level-0 execution. A geometric mean is computed separately for each category of applications.

timization, the classification of optimizations to each level, and the profiling and method promotion system, can affect the decisions. In our work, we conducted a number of experiments for evaluating the performance contribution and the compilation overhead for each optimization [67].

Figure 3.4 shows various data for compilation overhead such as compilation time, compiled code size, and peak compiler memory use for the three optimization levels. It also shows how much performance improvement comes from each of the optimizations. These measurements were done by running the SPECjvm98 and SPECjbb2000 benchmarks in the single execution mode for each level of optimization (without the MMI and the profiling system). The scores for *level-0 (L0) compilation* are used as the baseline of these graphs, where we disable all of the optimizations and just generate code directly from the intermediate code. This alternative level-0 execution mode is used for evaluating our system against the compile-only approach in Section 3.4.

Figure 3.5 shows MMI performance for both startup applications and steady state benchmarks (SPECjvm98 and SPECjbb2000) over the level-0 execution. The JIT compiler was not used throughout the program execution in this measurement. The graph shows that the performance of our MMI is nearly two times faster on average than that of the level-0 execution mode for startup applications. This means that our MMI is reasonably fast and performs comparable to the level-0 execution mode when the target methods are executed only a few times.

Overall, these results show a reasonable trade-off between the compilation costs and the benefits from the optimizations. At the lower optimization level, the performance benefit is large relative to the cost of the optimizations, which means the optimizations are

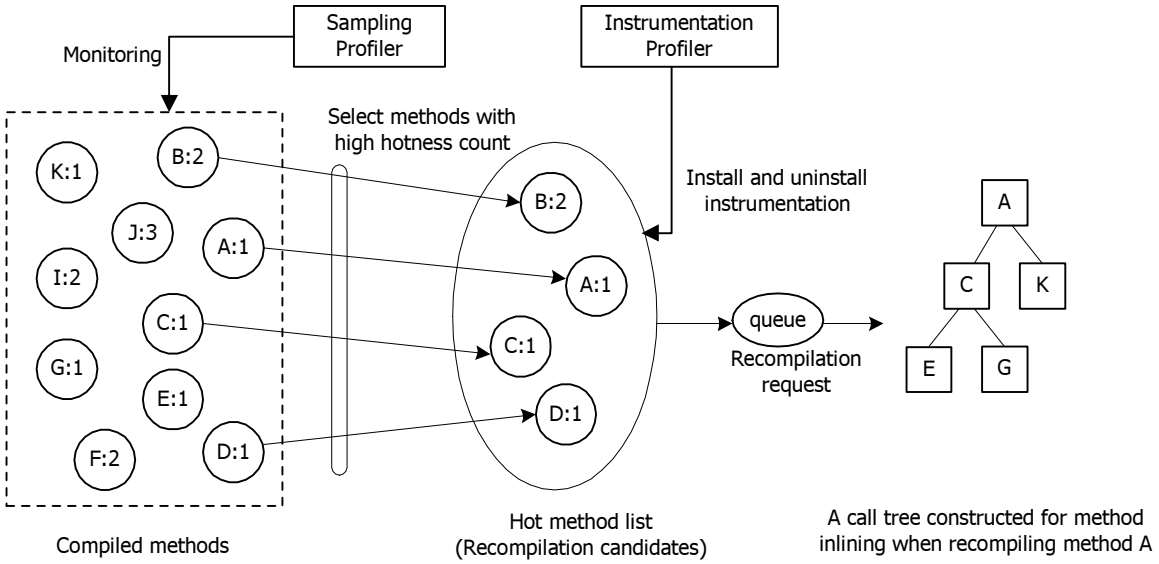


Figure 3.6: Two-stage profiling in our dynamic optimization framework. Each circle represents a compiled method, with a letter indicating the method name and a number indicating the optimization level. The instrumentation profiling is applied only for the compiled methods that are selected by the sampling profiler (methods A, B, C, and D). When the method A is recompiled, however, it may include code from non-instrumented methods (methods E, G, and K) as a result of method inlining.

applicable for a broader set of methods. However as we move to the higher optimization levels, the cost for applying each new set of optimizations is increasing compared to the corresponding return in performance gain. We can conclude from these results that assuming an effective profiling and method promotion system, the current 4-mode system (interpretation and three levels of optimization) can provide an adequate trade-off for recompilation decisions.

3.3.3 Profiling System

We combine three different profiling techniques in our system as shown in Figure 3.2, namely the counter-based MMI profiler and the sampling-based timer ticking profiler are used for hot methods detection, and the instrumentation-based profiler is used for value sampling.

From the mixed mode interpreter to the level-1 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 level-2 and level-3 recompilation from lower level compiled code is through the sampling profiler.

The reason we chose two different ways of method promotion comes from the consideration of the advantages and disadvantages of the two 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 the 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 the system misses the optimum point to trigger the first 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.[3] 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. Our implementation of the sampling profiler shows that the overhead introduced can be below the noise while retaining sufficient accuracy for identifying recompilation candidates by using an appropriate timer interval.

After methods are promoted to compiled code, the sampling profiler and the instrumenting profiler are used. The profiling work is split between these two profilers, one detecting the program's hot methods and the other collecting the detailed information on those methods detected by the first profiler.

This is the *two-stage profiling* design employing two different techniques for efficient online profiling,[4] as shown in Figure 3.6. With this design, we can benefit from several beneficial properties of online profiling.

First, the sampling profiler can be as lightweight as possible with virtually no performance penalty, which is a requirement for continuous monitoring operations throughout the entire program execution. Second, the instrumenting profiler also takes advantage of low overhead, because we can limit the duration of the profiling by uninstalling the instrumentation code after a preset number of samples are collected. We can also limit the targets of the instrumentation to only those selected methods we are interested in for value profiling. Without these mechanisms, the value profiling would be prohibitively expensive for use in dynamic execution environments. Third, we can expect reliable profiling results, because we can obtain representative and fine-grained profile information for real applications by delaying the instrumentation installation until the program hot regions are discovered.

On the other hand, when a selected method is recompiled, it is not necessarily the case that the profile information is available for all of the target code. This is because the inliner uses its own static heuristics to perform method inlining, and it may include a method

[3]There are some compilation techniques available to decrease the cost of counter updates in compiled code such as exploiting instruction-level parallelism or using loop unrolling.

[4]As stated in [14], this approach can be generalized to multiple stages of profiling schemes, where each subsequent profiler provides additional information on a smaller part of the application, but incurs additional overhead.

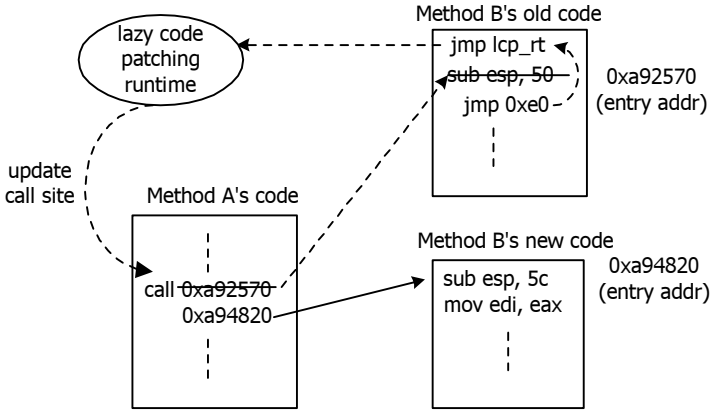


Figure 3.7: A lazy code patching mechanism for updating directly bound call sites to switch from the old version code to the new recompiled code. The dotted arrows show the flow of the first execution after the recompiled code is generated and installed, updating the target address of the corresponding call site. The code replacement is done atomically.

that has not been selected by the sampling profiler and thus has no instrumentation. Figure 3.6 includes a call tree example constructed for method inlining when recompiling the method A. The method C within the tree was instrumented and thus the profile information is available, but the inlined code from other methods in the tree does not have any profile information. Thus, the dynamic optimization system may not rely solely on the profile information for guidance, depending on the nature of the optimizations.

3.3.4 Recompilation and Code Management

Since the level-1 optimization inlines only tiny methods, 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 higher level optimizations. 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. This problem is addressed in the profile-directed method inlining described in Section 4.3.

After the recompilation is done for a method, it is registered by a runtime system called the *compiled code manager*, which controls and manages all of the compiled code modules by associating them with their corresponding method structures and with information such as the compiled code optimization level and specialization context. This means all of the future invocations of this method through indirect method lookup will be automatically redirected to the new version of the code, instead of to the existing compiled code. Static and nonvirtual call sites are exceptions, where direct binding call instructions to the old version code have already been generated.

A *lazy code patching* technique is used in order to change the target of the direct binding call. This is done by putting a jump instruction at the entry point of the old code to a runtime routine, which then modifies 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 to the new code will occur from the next call. Figure 3.7 shows a graphic example of this. In order to ensure thread safety, the code patching is performed with an atomic operation as follows. For replacing the old version entry code, we generate at compile time an unconditional jump instruction into the runtime routine for any versions of code that can be promoted to the next level. This kind of jump instruction is placed in the code header at a location several bytes ahead of the entry address. Thus we can put a short jump instruction (2 bytes long) atomically at the entry point to form the cascading jump into the runtime. For updating the call sites, we generate the directly bound call instructions in an address that fits in a single cache line in order to ensure that the call target (4 bytes long) can be replaced atomically. We add padding (nop) instructions if necessary to adjust the address for the call instruction generation.

For those threads currently executing old version code, the new optimized code will be used from the next invocation. We currently do not employ a mechanism for on-stack replacement [62], the technique of dynamically rewriting stack frames from one optimization version to another. Also the problem of cleaning up the old version code still remains in our system. The major difficulty with reclaiming old code is how to guarantee that no execution is currently in progress or will occur in the future using the old compiled code. The apparent solution would be to eagerly patch all the directly bound call sites, rather than to patch them lazily as in our current implementation, and then to traverse all of the stack frames to ensure no activation record exists for this old code. This traversal could be done at some appropriate time (like garbage collection time). Kistler and Franz [73] took this approach in implementing their continuous program optimization system.

3.4 Experimental Evaluation

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

3.4.1 Methodology

All the measurement results presented here were obtained on an IBM IntelliStation M Pro 6850 (Intel Xeon 2.8 GHz processor with 1,024 MB memory), running Windows XP SP1, and using the JVM of the IBM Developer Kit for Windows, Java Technology Edition, Version 1.3.1 prototype build. We conducted two sets of measurements for evaluating our

Table 3.1: List of applications used for start-up evaluation.

Program	Description
SwingSet	GUI component demo, version 1.1
Java2D	2D graphics library specified with -delay=0 -runs=1
IchitaroArk	Japanese word processor
HotJava	HotJava browser, version 1.1.5
WebSphere Studio	Java IDE platform, version 5.0

Table 3.2: List of benchmarks used for steady-state evaluation.

Program	Description
_227_mtrt	Multi-threaded image rendering
_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 JDK 1.02
SPECjbb2000	Transaction processing benchmarks, version 1.02

dynamic compilation system, startup runs and steady state runs. The benchmarks we chose for each category are shown in Table 3.1 and Table 3.2. For the startup performance evaluation, we selected a variety of real-world applications, ranging from a simple Internet browser to a complex Java IDE platform. For evaluating the steady state performance, we used two industry standard benchmark programs for Java, SPECjvm98-1.04 and SPECjbb2000-1.02 [94].

There are different requirements for the best performance in 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 time 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 time from the issuing of the command until the program terminated. For WebSphere Studio [64] and IchitaroArk [69], we used an empty workspace and measured the time to bring up the initial console window. For the steady state measurements, we took the best time from 10 repetitive autoruns for each test in SPECjvm98 running in test mode with the default large input size, with the initial and maximum heap size of 128 MB. Each distinct test

was run with a separate JVM unless otherwise stated. SPECjbb2000 was run in the fully compliant mode with 1 to 8 warehouses, with the initial and maximum heap size of 256 MB, and we reported the best scores among these runs.

The following parameters were used throughout the experiments.

- The threshold in the mixed mode interpreter to initiate first dynamic compilation (with level-1 optimization) was set to 500.
- The timer interval for the sampling profiler for detecting hot methods was 3 milliseconds. The controller examined the list of hot methods every 200 sampling ticks for recompilation decisions. The decay parameter was set to 0.3.
- No profile-directed optimization was employed in these measurements, and thus no instrumentation profiling code was generated or installed.
- The priority of the sampling profiler thread and the compilation thread was set above that of the application threads.

In the next two subsections, we compare performance and compilation overhead (measured in compilation time, peak compiler memory use, and the compiled code size) on several different configurations to see how well our system behaves for the execution of a variety of applications and benchmark programs compared to other configurations. To simulate a compile-only approach in our system, we provided the level-0 (L0) compilation as an alternative to the level-0 execution with MMI. In the L0 compilation, the intermediate representation (IR) is commonly used, but the native code is generated directly after constructing the IR without applying any optimizations. This L0 compilation is used as the baseline of the comparisons. We tried the following sets of compilation schemes.

1. No MMI configurations (compile-only approach)

– Level-1 optimization after level-0 compilation (L0-L1)
– Level-2 optimization after level-0 compilation (L0-L2)
– Level-3 optimization after level-0 compilation (L0-L3)
– Level-1 to 3 optimizations for adaptive recompilation after level-0 compilation (L0-all)

2. With MMI configurations

– Level-1 optimization with MMI (MMI-L1)
– Level-2 optimization with MMI (MMI-L2)
– Level-3 optimization with MMI (MMI-L3)
– Level-1 to 3 optimizations for adaptive recompilation with MMI (MMI-all)

For the no MMI configurations, the MMI was not used and all methods were L0 compiled at their first invocations. Then the hot methods identified by the sampling profiler were reoptimized with the corresponding optimization levels. The performance and compilation overhead of our L0 compilation relative to other optimization levels are shown in Figure 3.4, however, this may have different characteristics from the baseline compiler or the fast code generator in other systems using the compile-only strategy, especially because our L0 compilation is not by 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.

When we measured the compilation times, we instrumented the compiler with several hooks for each part of the optimization process to record the value of the processor's time stamp counter. Since the priority of the compilation thread is set higher than normal application threads, the difference between each value from the time stamp counter is guaranteed to provide the time spent performing the corresponding optimization work, and also the difference in the value between the beginning and the end of the compilation should be the total compilation time.

As we mentioned in Section 3.3.4, our current implementation does not reclaim the old version of code after the recompiled new version is made available. Thus, the compiled code size in the following results as an accumulated value from all of the optimization levels shows the total memory requirements for our system.

The peak compiler memory use is the maximum amount of memory allocated for compilation activity. Our memory management routine allocates and frees memory in 1 Mbyte blocks for compiling each method. We measured the maximum usage of the memory for all of the compilations.

3.4.2 Application Startup Time Performance

Figure 3.8 shows the comparisons of both performance and compilation overhead in the application startup phase. The top graph indicates that the performance of our dynamic optimization system, MMI-all, is almost comparable to that of the lightweight configuration of MMI-L1. On the other hand, the four configurations with the compile-only approach show poor performance for all of the programs apparently due to the bottom-line overhead of compiling all of the executed methods. Even L0-all, the recompilation system with the compile-only approach, does not perform better than level-0 compilation only (baseline), and has no advantages over the other compile-only configurations.

Part of the reason for this performance difference between the MMI and no-MMI configurations is that we used the sampling profiler to promote methods from level-0 compiled code to higher optimizations in the compile-only configurations, while our MMI used a counter for method invocation and loop iterations to trigger the compilations. In fact, the number of methods that are compiled at each optimization level between each

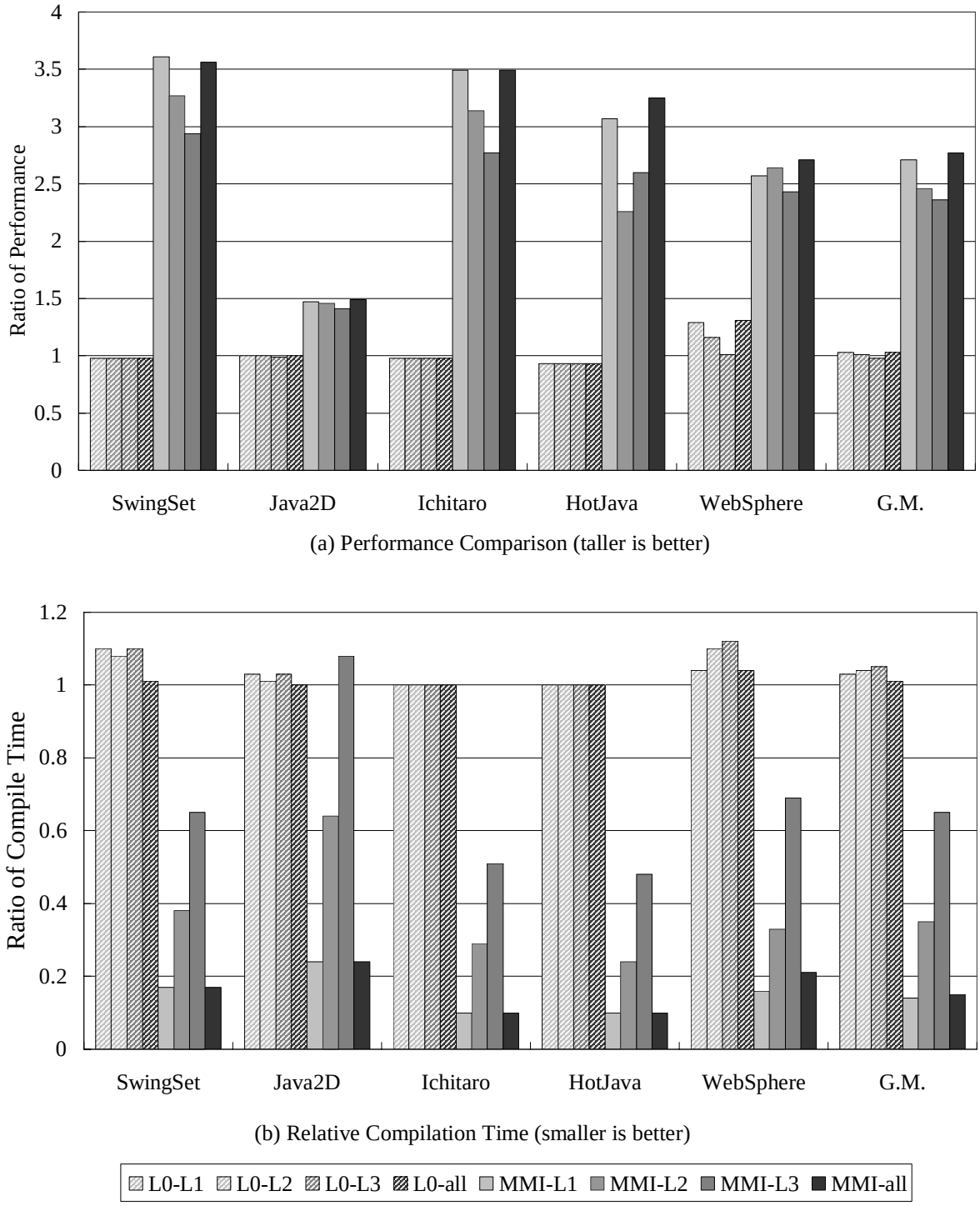


Figure 3.8: Startup comparison on both performance and compilation overhead (1 of 2). Each bar indicates the relative numbers to L0 compilation without MMI. The taller bar shows better performance in (a), while the shorter bar means smaller overhead in (b).

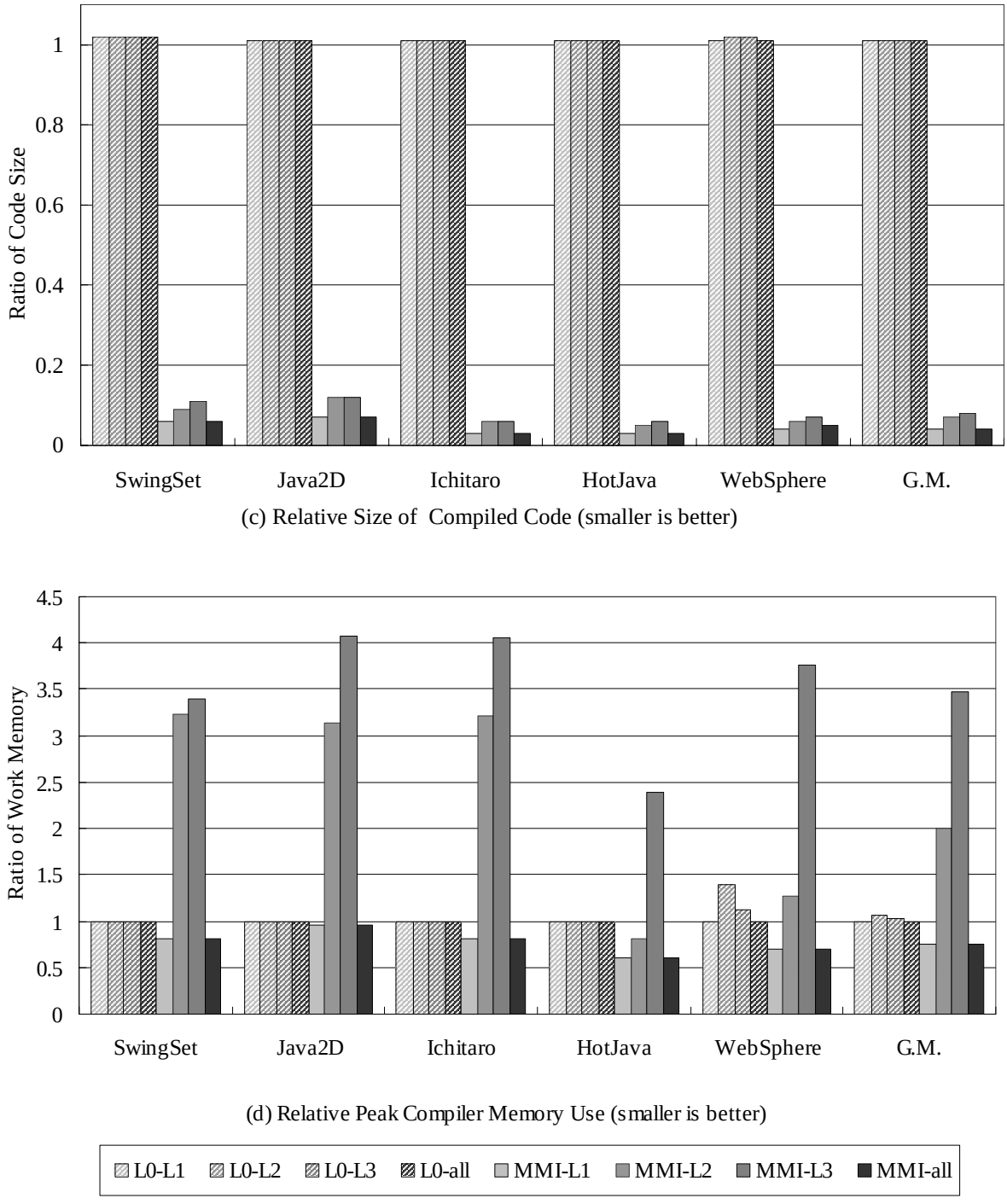


Figure 3.8: Startup comparison on both performance and compilation overhead (2 of 2). Each bar indicates the relative numbers to L0 compilation without MMI. The shorter bar means smaller overhead in (c) and (d).

Table 3.3: Percent of the compilation time to the execution time for start-up applications.

Program	L0-L1	L0-L2	L0-L3	L0-all	MMI-L1	MMI-L2	MMI-L3	MMI-all
SwingSet	17.9	17.6	17.9	16.5	20.3	36.4	51.0	19.4
Java2D	11.0	10.8	10.8	10.7	8.0	18.0	27.7	7.9
Ichitaro	16.6	16.6	16.6	16.6	12.1	29.7	42.0	12.3
HotJava	19.6	19.6	19.6	19.6	11.1	22.7	38.8	10.6
WebSphere	15.9	15.1	13.4	16.1	8.4	18.7	28.2	9.8

Table 3.4: Percent of the code size (upper row) and the peak compiler memory use (lower row) to the maximum live object size for start-up applications.

Program	L0-L1	L0-L2	L0-L3	L0-all	MMI-L1	MMI-L2	MMI-L3	MMI-all
SwingSet	28.5	28.5	28.5	28.5	1.6	2.6	3.0	1.6
	19.4	19.4	19.4	19.4	15.5	62.7	66.0	15.5
Java2D	48.0	48.0	48.0	48.0	3.4	5.8	6.0	3.4
	28.9	28.9	28.9	28.9	27.6	90.9	117.9	27.6
Ichitaro	65.9	65.9	65.9	65.9	1.8	3.7	4.2	1.9
	39.3	39.3	39.3	39.3	31.3	126.8	159.7	31.3
HotJava	30.8	30.8	30.8	30.8	0.9	1.5	1.8	0.9
	32.6	32.6	32.6	32.6	19.4	25.9	77.7	19.4
WebSphere	27.0	27.1	27.1	27.0	1.1	1.7	2.0	1.3
	8.3	11.6	9.3	8.3	5.8	10.5	31.4	5.8

pair of corresponding configurations with and without MMI (such as L0-L1 and MMI-L1) shows that the sampling profiler did not select as many methods as the counter based system in MMI in this application startup phase. This verifies our previous reasoning regarding the tradeoff for the two profiling systems in Section 3.3.3. However, apart from this problem, the MMI configurations still outperform the compile-only configurations, reflecting that the performance of our MMI-only configuration is nearly two times faster on average than that of the L0-only configuration, as stated in Section 3.3.2.

The other three graphs in the figure show the significant advantages for all three compilation overhead metrics of the MMI configurations over the compile-only approaches, especially for MMI-L1 and MMI-all. The compilation time with MMI-L3 sometimes shows a larger overhead relative to the other MMI configurations. This is because a few methods that triggered compilation from the MMI caused large overheads due to the aggressive inlining and other optimizations applied, but those methods were not promoted in the corresponding compile-only configuration (L0-L3). This also explains the higher overhead of the peak compiler memory use with MMI-L2 and MMI-L3. MMI-all, however, avoids this anomaly by employing level-1 optimization at the time of the first compilation.

As for the generated code size, it is an order of magnitude larger with compile-only

configurations than with the MMI configurations. The code expansion factor from the bytecode size can actually be up to 10x without the MMI, while it is less than 1x for the MMI configurations.

Table 3.3 shows the ratios of the compilation time to the execution time for the start-up applications. The ratios for the no-MMI configurations are not very high (10% to 20%), but the compilation times are actually quite large because of the long execution times. For MMI configurations, the ratio consistently increases as we move from MMI-L1 to MMI-L3, and the execution time increases accordingly. MMI-all is almost at the same level as MMI-L1.

Table 3.4 shows the ratios of the compiled code size and the peak compiler memory use to the maximum size of the live object data in the Java heap. We measured the maximum live object size using the -verbosegc command line option. While the size was measured reflecting the granularity of the GC events and thus is not exact, the ratios shown in the table indicate approximately how much of the space was consumed for both compiled code and compiler work memory in terms of the live objects in the heap for each application. Please note that the baseline for comparison is fixed for each application across all configurations, unlike Table 3.3.

3.4.3 Steady State Performance

Figure 3.9 shows the comparisons of performance and compilation overhead in the steady state program runs. From the top graph of this figure, four configurations, L0-L3, L0-all, MMI-L3, and MMI-all, are the top performers in this category. These are all configurations involving level-3 optimization, and it shows the sampling profiler appropriately selected performance-critical methods for L0-L3, L0-all, and MMI-all configurations.

For the compilation overhead, the differences between the corresponding compile-only and MMI configurations (such as L0-all and MMI-all) are relatively small for both compilation time and peak memory use compared to the application startup phase. But the compile-only approach still shows a problem with the large size of the compiled code, although it is less dramatic than the comparison in the application startup phase. This means that the extra overhead of level-0 compilation for all methods is essentially negligible, for both compilation time and memory use, when compared to the cost of recompilation with higher optimization levels, especially level-2 and level-3. However, the code size has a cumulative effect and thus cannot be hidden by the cost with higher optimization levels.

From both the performance and compilation overhead in this category, the two recompilation configurations, L0-all and MMI-all, are the two best configurations and are almost comparable for both compilation time and peak compiler memory use. However L0-all still suffers from the large code size expansion. This is thought not to be specific to our system that only simulates the compile-only approach, but will be true even with an actual implementation fully optimized for this approach.

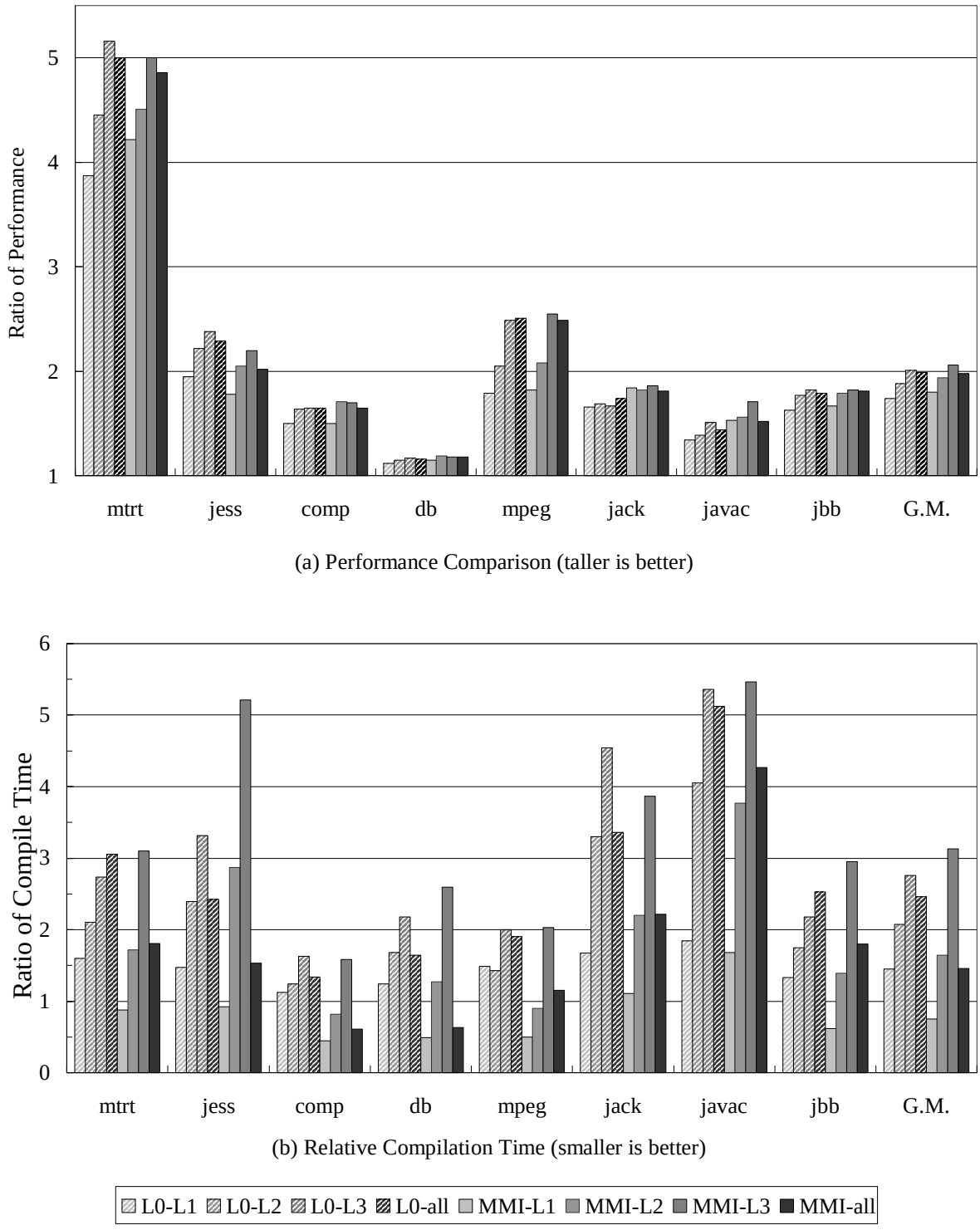
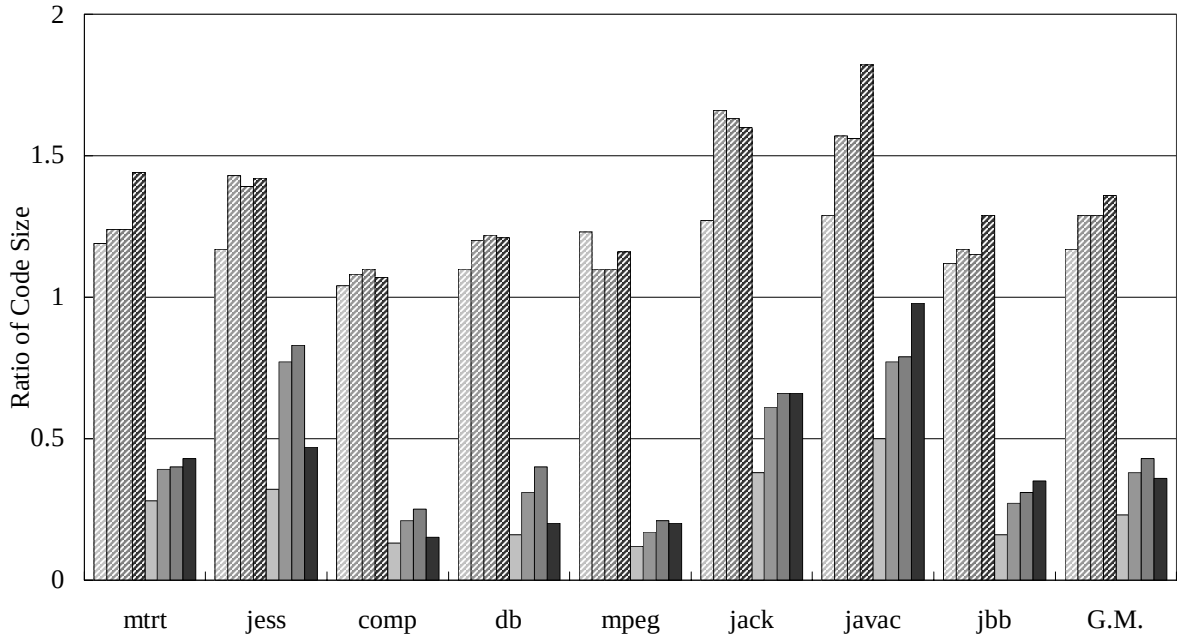
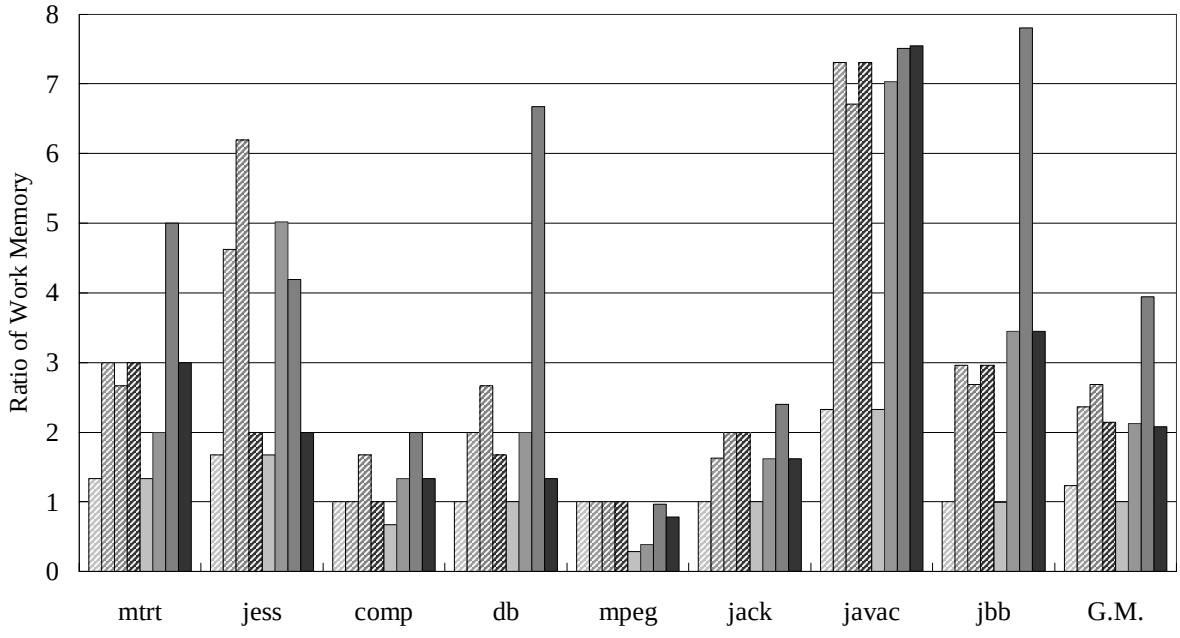


Figure 3.9: Steady state comparison on both performance and compilation overhead (1 of 2). Each bar indicates the relative numbers to L0 compilation without MMI. The taller bar shows better performance in (a), while the shorter bar means smaller overhead in (b).



(c) Relative Size of Compiled Code (smaller is better)



(d) Relative Peak Compiler Memory Use (smaller is better)

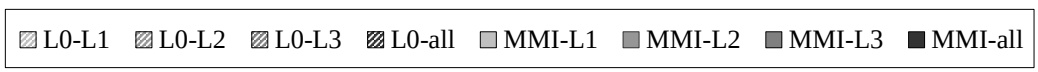


Figure 3.9: Steady state comparison on both performance and compilation overhead (2 of 2). Each bar indicates the relative numbers to L0 compilation without MMI. The shorter bar means smaller overhead in (c) and (d).

Table 3.5: Percent of the total compilation time over 10 repetitive runs to the best execution time when running each test of the SPECjvm98 benchmark in autorun mode. SPECjbb2000 is excluded, since it does not give us the execution time.

Program	L0-L1	L0-L2	L0-L3	L0-all	MMI-L1	MMI-L2	MMI-L3	MMI-all
_227_mtrt	38.3	58.1	87.6	94.7	23.1	48.2	96.2	54.5
_202_jess	24.5	45.6	67.5	47.5	13.9	50.2	97.9	26.5
_201_compress	6.3	7.6	10.0	8.3	2.5	5.2	10.0	3.8
_209_db	2.7	3.8	5.0	3.7	1.1	2.9	6.0	1.5
_222_mpegaudio	21.1	23.2	39.3	37.9	7.2	14.8	40.9	22.5
_228_jack	26.8	53.5	73.1	56.3	19.5	38.5	69.3	38.7
_213_javac	26.2	59.2	85.3	77.8	27.1	62.0	96.6	68.2

Table 3.5 shows how much of the compilation time has been spent before obtaining the best execution performance for the SPECjvm98 benchmark when running each test 10 repetitive times in autorun mode. SPECjbb2000 is excluded from the table because it does not give us the execution time. This table is different from the corresponding Table 3.3 in the previous section, since it does not show the compile time ratio actually occupied within the duration of a single execution. Instead, the table shows how much effort we needed over 10 repetitive runs to eventually obtain the best execution time. The ratio varies from one benchmark to another. For example, _201_compress and _209_db have very low ratios, meaning that the working set of these benchmarks is small, while _213_javac and _228_jack have flat profiles and thus higher ratios due to many methods being compiled. The high ratio of _227_mtrt and _202_jess is probably due to the very short execution times of these benchmarks. From this table, when comparing the corresponding configurations with and without MMI, the MMI configurations have smaller compilation ratios, except for MMI-L3.

Table 3.6 shows the ratios of the compiled code size and the peak compiler memory use to the maximum size of the live object data in the Java heap, similar to Table 3.4 in the previous section. The very high ratios of _202_jess, _222_mpegaudio, and _228_jack are caused by the very small sizes of the live object data (about 1 MB to 3 MB) throughout the programs' executions.

Overall, our dynamic optimization system, MMI-all, shows the best performance with the minimum compilation overhead among all configurations in both the application startup and steady-state phases.

3.4.4 Execution Mode Ratios

Figure 3.10 shows the percentage of each of the four execution modes, MMI and level-1 to level-3 compiled code, when running each benchmark with the MMI-all configuration. Each benchmark has two bars. The first bar is the percentage of the methods executed in

Table 3.6: Percent of the code size (upper row) and peak compiler memory use (lower row) to the maximum live object size for steady-state applications.

Program	L0-L1	L0-L2	L0-L3	L0-all	MMI-L1	MMI-L2	MMI-L3	MMI-all
_227_mtrt	5.2	5.4	5.4	6.3	1.2	1.7	1.8	1.9
	19.4	43.7	38.8	43.7	19.4	29.1	72.8	43.7
_202_jess	20.0	24.6	23.8	24.3	5.4	13.2	14.2	8.0
	83.6	231.6	310.3	100.3	83.6	251.7	210.2	100.3
_201_compress	5.6	5.9	6.0	5.8	0.7	1.1	1.4	0.8
	22.4	22.4	37.3	22.4	14.9	29.8	44.7	29.8
_209_db	3.9	4.3	4.3	4.3	0.6	1.1	1.4	0.7
	14.1	28.3	37.7	23.6	14.1	28.3	94.3	18.9
_222_mpegaudio	85.9	76.6	76.8	80.8	8.4	12.1	14.5	13.8
	510.3	510.3	510.3	510.3	148.4	198.0	494.9	396.0
_228_jack	46.8	61.1	60.0	59.1	14.0	22.4	24.4	24.4
	163.5	266.4	327.1	327.1	163.5	264.4	392.5	265.4
_213_javac	5.6	6.8	6.8	7.9	2.2	3.4	3.4	4.2
	19.0	59.3	54.5	59.3	19.0	57.1	61.0	61.3
jbb2000	2.5	2.6	2.5	2.8	0.4	0.6	0.7	0.8
	2.9	8.6	7.8	8.6	2.9	10.0	20.3	10.0

each execution mode, and the second bar is the percentage of the execution time in each of the four modes.

When counting the number of methods, we classified each method under the highest optimization level it reached. That is, if a method is executed in MMI first, and then promoted to level-1, then it is counted only in level-1 execution mode. For the execution time in each execution mode, we collected the data by using a profiling tool called real-time arcflow [4]. The tool instruments each method at entry and exit using a JVMPI interface to collect the time stamp, and then post-processes the collected data to provide the proportion of the execution time spent in each method. Again, we classified the execution time for each method according to the highest level the method reached. That is, if a method is eventually compiled with level-3 optimization, then we attributed all the time spent in that method to level-3 execution time. Native methods are not included for both of these data sets.

The number of methods compiled with level-1 optimization is, except for _228_jack and _213_javac, around 20% of the total number of methods, among which the number of recompiled methods with higher optimization levels is roughly 15% to 20%. Therefore we can achieve the best possible performance with the optimization capabilities available in our system by focusing on merely 3% to 4% of all methods. In comparison, the execution time spent in level-2 and level-3 execution mode is more than 80% of the total time, again except for _228_jack.

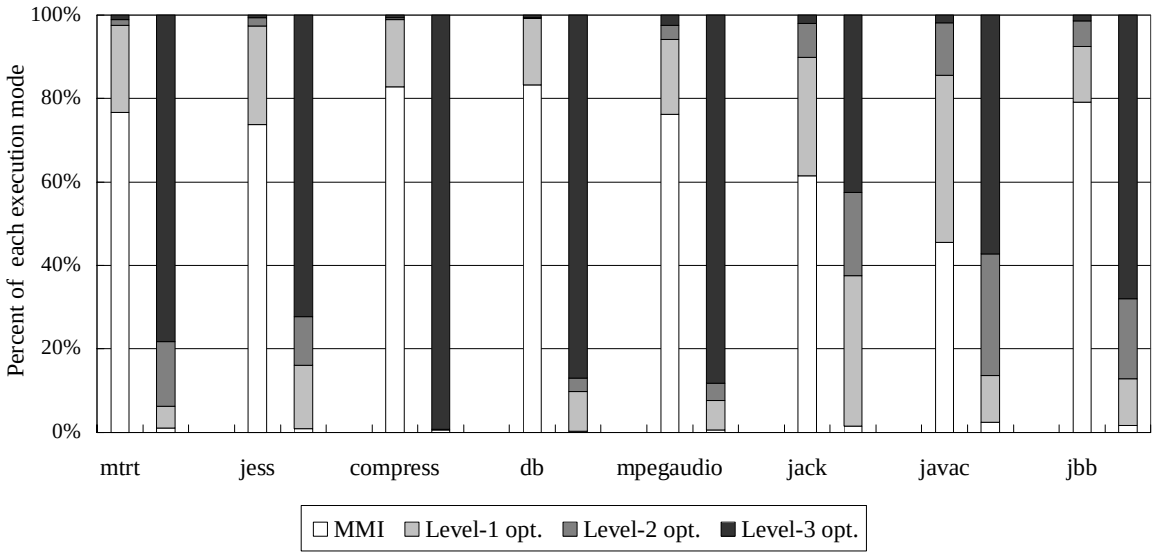


Figure 3.10: Statistics showing the percentage of each of the four execution modes for the SPECjvm98 and SPECjbb2000 benchmarks. The first bar for each benchmark is the number of methods in each execution mode, and the second bar is the execution time spent in each mode. Native methods are not included.

This shows the well established rule that programs typically spend most of their execution time in only a small portion of the code. In this case, it roughly follows the "80/20 rule" meaning that the proportion of the number of methods between MMI and compiled code is about 80 to 20, while that between level-1 and level-2/3 compiled code is again close to 80 to 20. In contrast, the proportion of the execution time is roughly 10 to 90 between the level-1 and the level-2/3 compiled code.

As described in [3], `_213_javac` has a flat profile, involving the execution of many equally important methods, and thus poses a challenge for any recompilation system. The same thing is true for `_228_jack`. This characteristic caused a relatively larger number of level-1 optimizations in our system, 30% to 40% of the total number of methods executed. However, the number of level-3 optimized methods is still limited to around 4%, similar to the other test cases, and those methods cover about 40% to 60% of the execution time. This shows that our recompilation decision process works quite well.

3.4.5 Compilation Activity

Figure 3.11 shows 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 in the autorun mode, ten times each with a single JVM. The horizontal axis of the graph is divided equally for each run of the tests. The bar chart (left vertical axis) indicates the number of compiled methods at each level of optimization, and the line graph (right vertical axis) indicates the changes of the execution time from the first to

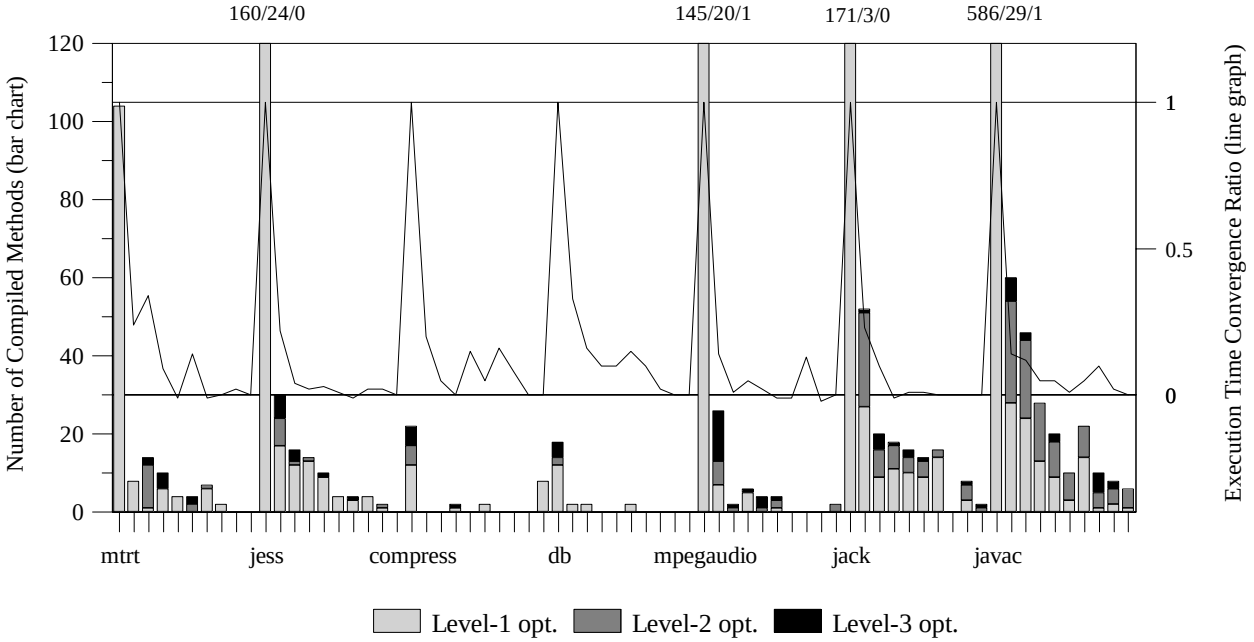


Figure 3.11: Change of compilation activity and the program execution as program shifts its execution phase. The bar chart (left Y-axis) shows the number of compiled methods for level-1 to level-3 optimizations within each run of the tests, and the line graph (right Y-axis) shows execution time ratio from 1st to 10th runs. For those bars extending out of the visible range, the number of compiled methods with level-1, level-2, and level-3 are shown at the top of each bar using the separator "/".

the tenth run normalized by the time differences. That is, 1 corresponds to the execution time in the first run and 0 corresponds to that in the tenth run, and the line graph shows how quickly the execution converges to the best running time for each test. Therefore this figure shows the relationship between the compilation activity and its impact on the performance improvement as the program dynamically changes its execution phases.

Overall, 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 level-1 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 level-2 or level-3 optimization. Note that there are some overlaps in the methods across these tests in the SPECjvm98. A method shared between several tests may be compiled with level-1 optimization first in an earlier test. Since the sampling profiler monitors all the code compiled so far, the method can be promoted to a higher level later if it is identified as hot.

The execution time is almost consistently improved after the first run for all of the tests by successful method promotions from the interpreted code to the level-1, and then to the level-2 and level-3 compiled code. In some of the tests, the cost of recompilation seems

to appear in the execution time. This is partly because we performed the measurement on a uniprocessor machine and cannot hide the background compilation costs completely. No significant overhead can be observed for most of the 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.

3.5 Summary

We have described the design and implementation of our dynamic optimization framework that consists of a mixed mode interpreter, a dynamic compiler with three optimization levels, a sampling profiler, a recompilation controller, and an instrumenting profiler. The experimental results show that the system can effectively work to initiate each level of optimization, and can achieve high performance and low compilation overhead in both program startup and steady state measurements in comparison to other configurations, including those with 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 transition between optimizations.

Based on the data shown in this chapter, MMI-all is our configuration of first choice, achieving the two goals described in Section 3.2: the best performance for varying phases of application programs, and system extensibility for adding profile-directed optimizations. However, using a simple system, such as MMI-L1 in our system, might be a reasonable approach depending on the target platform, where the simplicity and robustness of the system is a primary goal. That is, for a specific platform one may decide that it is more practical to simplify the system rather than introducing the implementation complexity and then facing the resulting code maintenance problem. Standing in this position, MMI-L1 may be a preferable system with an acceptable level of performance and sufficiently lightweight compilation overhead.

Chapter 4

Profile-Directed Optimizations

4.1 Introduction

From the results shown in the previous chapter, we can see the advantages of our dynamic optimization framework with the MMI and three optimization levels in terms of performance and compilation overhead in both the application startup and steady state phases. In addition, we can further explore the opportunity for more advanced optimizations by exploiting the dynamic profile information. In this chapter, we describe the two profile-directed optimizations, method inlining and code specialization, designed and implemented on top of the dynamic compilation system described in Chapter 3.

Section 4.2 describes how the profile data can be collected for extracting potential candidates to drive these optimizations. Section 4.3 and 4.4 then provide the details of the profile-directed method inlining and the code specialization, respectively. Section 4.5 shows the impact on the performance and the compilation overhead when applying these optimizations, both alone and in combination. Finally, Section 4.6 gives the summary of this chapter.

4.2 Dynamic Instrumenting Profiler

We employ a dynamic instrumentation mechanism to collect the runtime information, such as the distribution of call sites, parameter values, and other specified field values. The instrumenting profiler generates code, according to the plans specified by the recompilation controller, for each of the target methods or target variables to be monitored.

Figure 4.1 shows a graphic example when the instrumentation code is generated and installed in the target compiled code. The entry instruction of the target code, after it is copied into the instrumenting code region, is dynamically replaced with an unconditional branch instruction to direct control to the generated profiling code. At its first entry, the instrumentation code stores the current time stamp. It then examines the specified variables, records them in a table, and then jumps back to the next instruction after

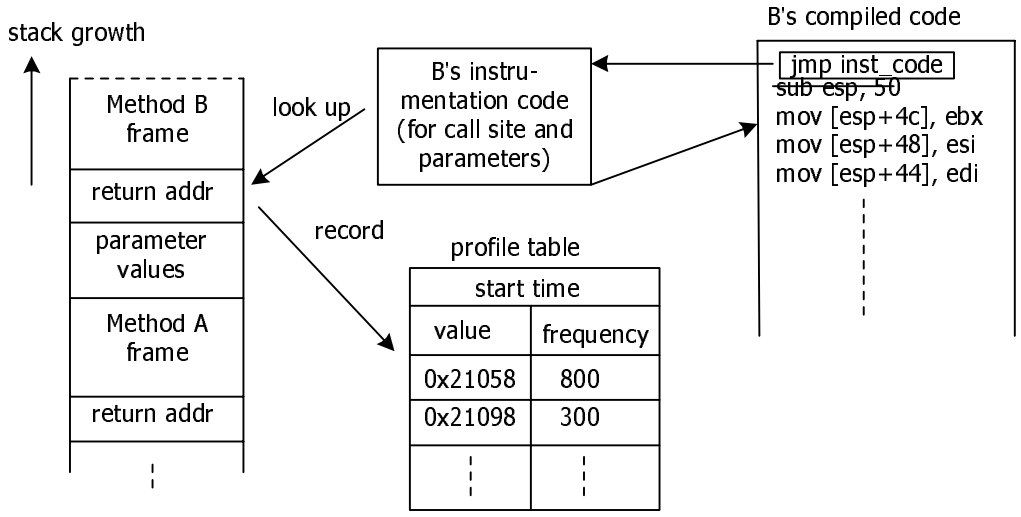


Figure 4.1: Dynamic instrumentation for collecting information on call site distribution and/or parameter and field values. The entry code is patched to direct control to the generated instrumentation code.

the entry point. After the predetermined number of samples has been collected, the code again checks the current time and records how long it took to collect those samples by subtracting the start time. The code then uninstalls itself from the target code by restoring the original code at the entry point.

The profiler can collect any or all of the following information depending on what optimizations are under consideration: the return address stored on the top of the current stack, the values of parameters on either designated registers or in stack locations, and the values of class variables or object fields from the parameters. When examining the caller's address, only the top-level history in the call stack is recorded because of the potential overhead. 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 tables with values or types of variables and their corresponding frequencies. The data table or the counter for storing information is allocated separately so it can be passed back to the controller. The maximum number of entries in the table is fixed.[1]

Thus two kinds of information can be collected for each method during an interval in the program's execution: the distribution of values and the frequency of invocations. The time recorded for sample collection indicates the call frequency for the method for that fixed number of instrumentation samples, and is useful information for method inlining decisions. This is because the hotness-count provided by the sampling profiler is strongly

[1]The instrumentation code is just a small amount of code, typically less than 100 instruction bytes, and the corresponding table space is in general less than another 100 bytes. This space is treated in the same way as compiled code so that it can be reclaimed upon class unloading.

dependent on the size of the target method (it is easier for larger methods to get more sampling ticks), so small but very frequently called methods may not receive the attention they deserve. In that sense, the fine-grained frequency information can be useful in identifying hot call edges critical for inlining, and in compensating for the rather coarse-grained hotness count of the sampling profiler.

The instrumentation is successively applied to a group of recompilation target methods provided by the sampling profiler at every fixed sampling interval. After installation, the recompilation controller periodically monitors the completion of the profile collections on all of these methods, and if completed, it proceeds to the analysis step described in the next two sections. This is implemented by setting a maximum wait count for the recompilation controller to give up the compilation thread execution. If necessary, it stops the profiling for methods that are invoked very infrequently and which will never reach the predetermined number of samples by directly manipulating the current profile count for those methods.

4.3 Profile-Directed Method Inlining

Method inlining has significant impact on both costs and benefits of compilation as shown in Figure 3.3. In our dynamic optimization system, we have applied method inlining only for tiny methods in the level-1 optimization, while in the level-2 and level-3 optimizations we use static heuristics for aggressive inlining. However, we can exploit dynamic program execution behavior to improve the cost and benefit ratio of this expensive optimization. When enabling profile-directed inlining, we basically abandon any existing static heuristics, except for always inlining tiny methods, and totally rely on the online profile information for driving method inlining in the level-2 and level-3 optimizations.

For collecting call site distribution, we basically apply the instrumentation to all hot methods that are detected by the sampling profiler and which are candidates for promotion to the next level of optimization. Since the instrumentation code in this case is to find the most beneficial candidates among the call sites where the current target method can be inlined, we need to ensure that the current target is appropriate for inlining, or otherwise adding the instrumentation is just overhead. Thus those methods that are apparently not inlinable into other methods (due to excessively large size, for example) are excluded as targets for instrumentation. Methods already compiled with the highest level of optimization are also excluded, since they are methods already considered for inlining but not inlined in the previous round of compilation. All these decisions are made through the instrumentation planning by the recompilation controller.

The controller examines the recompilation candidates grouped together by the sampling profiler to identify among them those hot call paths appropriate for inlining. Upon completion of collecting the call site information, the decision on requesting method inlining proceeds with the following steps:

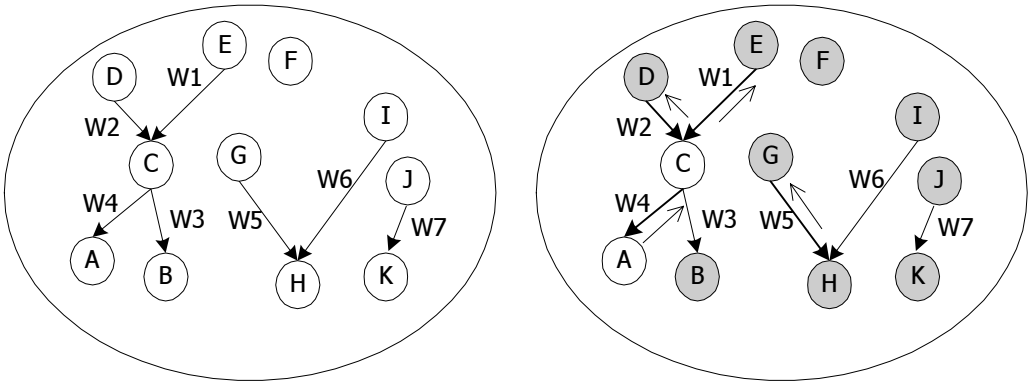


Figure 4.2: (a) An example of the partial call graph, where nodes indicate recompilation candidate methods (A to K), and edges show the call connections from caller to callee, each associated with a strength factor (W1 to W7). Each node is assumed to have a hotness count, although it is not shown above. (b) The graph example of the inlining steps. The hot call paths are identified (the bold edges) and three inlining requests are made (A to C to D, A to C to E, and H to G). The hotness counts of these methods are then adjusted along the arrows shown beside the hot call paths, resulting in the zero hotness counts for A and C. Finally the recompilation requests are made for the shadowed methods, excluding the methods with low hotness counts.

1. Partial call graph construction: The controller first constructs a partial call graph[2] by examining the call site distribution profile for each of the instrumented methods. Nodes represent recompilation candidate methods, each of which has a hotness count, and edges show the call connections from caller to callee, each associated with a strength factor computed from the call frequency and the distribution ratio, as shown in Figure 4.2(a). The distribution ratio is based on the count given to the corresponding call site address in the profile.[3]

2. Exact call path identification: The controller then identifies the exact call paths appropriate for inlining. It begins with a method having no outgoing edge, and successively selects call edges whose strength factor is above a threshold by traversing the connections up to the top of the call chain.

3. Method inlining request: The controller issues inlining requests for the hot call paths identified above.[4] These requests are stored in a persistent database so that

[2]The call graph is partial in the sense that it is constructed from the profile data and therefore it includes only call edges that were found in a certain interval of the program's execution.

[3]Since tiny methods are always inlined in the compiled code and thus the call site found in the profile may actually be within an inlined tiny method, the runtime structure called the inlined method frame is consulted to get the exact call path from the call edge. This structure was originally created to support the runtime system for exception handling and security managers.

[4]Since the raw profile data indicates a set of compiled code addresses where the invocation was made

inlining decisions can be preserved for future recompilation to incorporate the previous inlining requests. This is to avoid any performance perturbation resulting from oscillating between two compiled versions of a method, each with a different set of inlining decisions [59, 10].

4. Hotness count adjustment: The controller then corrects the hotness count for each method by assuming that inlining was performed as requested in the previous step. A portion of the hotness count of the callee (including values from its children, if any) is distributed to each of its callers according to the distribution ratio for each incoming edge. When dividing the hotness counts from children into their multiple callers, the distribution ratio cannot be known precisely, because the profile information is not available for the multiple-level histories of the call paths. Therefore, we use a *constant ratios assumption* [92] to approximate the actual ratio. That is, we assume the ratio is the same as the one recorded in the profile.

5. Recompilation request: The controller again traverses the list of recompilation candidates to issue the final recompilation requests. If a method in the list is recommended for inlining into another method, and there is no other call site for this method found in the profile information, then the method is removed from consideration for recompilation. Likewise, those methods whose hotness counts are below a threshold value as a result of the hotness count adjustments are discarded. A method already compiled with the highest optimization level will be recompiled again only when a new inlining request has been made and its estimated impact is above a threshold value.

Figure 4.2(b) shows a graphic example of the Steps 2 to 5 above. We assume here that W1, W2, W4, and W5 are the only strength factors above the threshold, and thus the three hot call paths along the bold edges (D-C-A, E-C-A, and G-H) are identified. We also assume the distribution ratio of the incoming edges along the hot paths is dominant for the methods A and C but not for method H, which results in the different recompilation decisions for these methods.

Since we consider the inlining possibilities among hot methods appearing in the group of hot methods, and since this group comes from sampling during an interval in the program execution, the resulting inlining requests are expected to contribute for a performance boost. In Step 2, however, there may be a call path where a caller is not included in the current group of hot methods, and it is possible to request inlining for such a case as well. This can be considered to be more aggressive in the profile-directed inlining. However, this aggressive inlining policy did not produce additional benefit in our experience [99].

to the current target method, it does not specify the exact call sites in the original form of the method bodies. The code address is therefore converted to the offset of the corresponding bytecode instructions using a table generated at compile time.

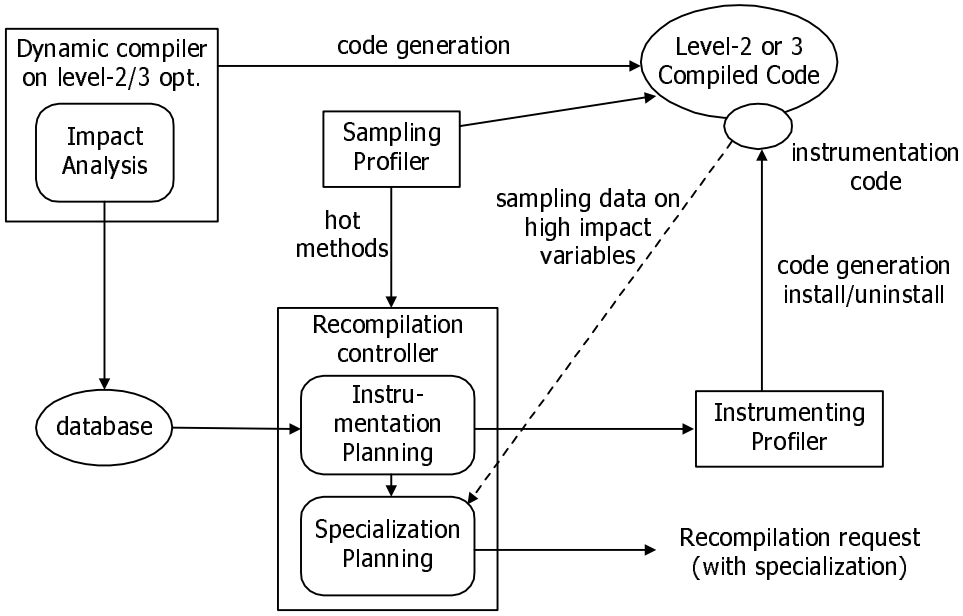


Figure 4.3: Flow of code specialization decisions. The recompilation controller directs the instrumenting profiler to dynamically install instrumentation code by specifying specific variables based on the impact analysis results. It then utilizes the profile information to decide how those methods should be specialized by taking advantage of the skewed behavior of the application.

4.4 Dynamic Code Specialization

Code specialization is enabled only on the highest optimization level based on the profile data collected in the previous version of the compiled code. Figure 4.3 shows the flow of control regarding how the decision on code specialization will be made. It consists of two parts, impact analysis by the dynamic compiler and specialization decisions by the recompilation controller. These are described in detail in the following sections. The code specialization is applied for an entire method, not for a smaller region in the method such as a loop.

4.4.1 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 benefits of code specialization, is used to make a prediction as to how much better code we would be able to generate if we knew a specific value or type for some variables. The impact analysis is performed during the level-2 or level-3 compilation and the results of the analysis are stored in a persistent database, so that the controller can make use of it for the next round of recompilation plans.

The specialization targets can be either method parameters or non-volatile global variables,[5] such as class 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 within basic block n.[6]
$Out(n) = In(n)$ - $Kill(n)$
$In(n) = \bigcap_{m \in Pred(n)} Out(m)$ (for $n \neq$ entry basic block)

This means that after the computation $In(n)$ is the set of global variables referenced within the method and guaranteed not to be updated along any paths reachable from the top of the method to the entry point of the basic block n. Each global variable reference within the basic block can then be checked as to whether it can be included in the specialization target from the *Kill* set for each instruction. That is, we can compute all the global variables that are safe for *privatization* at the entry point for each method. The set of specialization targets, both the global variables obtained as above and the argument variables, is then fed to the impact analysis. The pseudo-code of the impact analysis is shown in Figure 4.4.

For each specialization target V, the specialization point can be expressed as a triple (V, L, S), where L denotes a local variable directly or indirectly defined from V, and S denotes the statement in the method in which the variable L is defined. The algorithm traverses the dataflow through a def-use chain for each use of the variable L and its derived variables, tracking any possible impact on each operation for V. The impact value of the specialization type T on operation Op appearing in the pseudo-code can be expressed as:

$Total\_saved\_cost(Op, T) = Unit\_saved\_cost(Op, T)$ * $f(loop\ nest\ level)$
$Impact\_value(Op, V, T) = Total\_saved\_cost(Op, T)$ / $SST(V, T)$

In these equations, $Unit\_saved\_cost$ represents the cost savings attributed to a single operation Op with the specialization type T. If the operation is located within a loop, then the cost saving is scaled with a fixed value that represents the loop iterations to reflect the greater benefit that can be expected.

The baseline of the impact is the execution cost of the *specialization safety test* (SST), which is the guard code generated at the entry point of the specialized method. This can vary from a simple compare and jump instruction to a set of multiple instructions

[5] 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 [51], so we cannot treat it as a specialization target.

[6] Any method invocations, synchronization instructions (`monitorenter` and `monitorexit`), and exception checking operations for the variable are treated as barrier instructions. We assume the optimization of exception check elimination [71] is performed prior to the impact analysis.

```
 1:  Each specialization point is expressed as a triple (V, L, S), where
 2:      V: a parameter or a global variable that can be a target for
 3:         specialization
 4:      L: local variable that is directly or indirectly defined from V
 5:      S: statement where L is defined
 6:
 7:  Impact_Analysis () {
 8:      for each specialization target V {
 9:          Weight(V, *) = 0;
10:          /* estimate the impact for a given specialization target */
11:          Estimate (Weight, V, V, method entry point);
12:      }
13:  }
14:
15:  Estimate (Weight, V, L, S) {
16:      for each operation Op which uses L and is reachable from S {
17:          if (Op can be simplified or eliminated by a specialization type T) {
18:              Weight(V, T) += Impact_value(Op, V, T);
19:              if (Op is converted to a constant assignment) {
20:                  /* recursively call itself for the derived variable */
21:                  Estimate (Weight, V, LHS variable of Op, location of Op);
22:              }
23:          }
24:      }
25:  }
```

Figure 4.4: Pseudo-code for the impact analysis. The algorithm tracks any possible impact of operation for each use of the specialization targets and its derived variables. The impact value is defined based on the type of operations simplified and the execution cost of the guard code to be generated, as described in the text.

depending on the variable V and type T. The impact value is then computed by weighing the expected benefit ($Total\_saved\_cost$) from the operation Op with the specialization type T against the expected cost SST.

The cost savings can be quite different for each of the operations. For example, the elimination of `checkcast` or `instanceof` operations can have a large effect,[7] while it would be much smaller when getting a constant operand in a binary operation. In particular, `instanceof` has a large impact because it is often used as a conditional part of an if-statement and the elimination of this operation can lead to the elimination of the entire *then* or *else* part of the if-statement, resulting in straight-line code.

The final result of the impact analysis for the estimated specialization benefit is the *specialization candidate set SCS*, each member of which is a *specialization candidate SC*;

[7]Our implementation of the subtype checking operation is based on a 1- or 2-element software cache (positive cache for `checkcast` and both positive and negative caches for `instanceof`). This still requires 6 to 9 instructions in the fast execution pass. The *display* implementation in HotSpot [39] is more efficient and requires only 3 instructions for most cases.

$$SCS = \{(V, T) \mid Weight(V, T) > minimum\ threshold\}$$

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 or a switch statement.
- An exact object type, which allows removal of some unnecessary type checking operations and leads to more opportunities for devirtualization and inlining by improving the precision of the class hierarchy analysis.
- 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 sequencing for some instructions (e.g. `invokevirtualobject`, `checkcast`).
- Equality of two parameter objects, allowing method bodies to be significantly simplified.
- The thread locality of objects[8] which allows the removal of unnecessary synchronizations.

4.4.2 Specialization Decision

When a hot method has been identified in the level-2 or level-3 compiled code, the controller checks the results from the impact analysis stored in the code manager database. If it finds a candidate *SC* for the method that looks promising as a justification for performing specialization, then the controller dynamically installs the instrumentation code into the target native code, using the mechanism described in Section 4.2, to decide whether it is indeed worth specializing with the specified type. Currently a minimum threshold is used to allow all candidates to be selected as instrumentation installation targets.

Upon the completion of the value profiling, the controller then makes a final decision regarding whether or not it is profitable to specialize with respect to a specialization candidate SC. The metric we use for this decision can be expressed as follows:

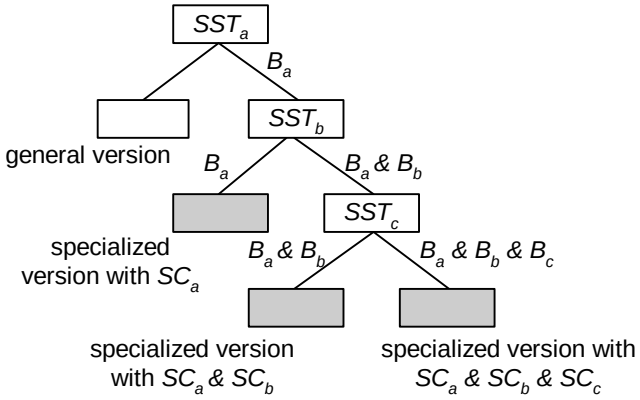
$$f\ (Weight,\ Sample\ Ratio,\ Code\ Size,\ Hotness\ Count)$$

This function indicates that the impact analysis result, the ratio of bias in the corresponding profile data, the size of the recompilation 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

[8] We use the escape analysis to identify thread-local objects.

Specialization Candidate	Sample Bias Ratio	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) Three specialization candidates



(b) Construction of decision tree with three specialization canidates

Figure 4.5: An example of specialization decision on multiple candidates. (a) Three specialization candidates available. (b) Construction of decision tree on generating three specialized versions.

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 ($\in SCS$ for $i = a, b, c$) available for a method as in Figure 4.5(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.

All of the nodes that are not selected for a particular specialization are contracted to a single node. For example, the left node from the root represents a general version of the code (that is, the level-2 or unspecialized level-3 compiled code). Thus, the number of leaf nodes in the tree is exactly the number of specialization candidates plus one. 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.

Two strategies can be considered for selection: benefit ordered and sample ratio ordered. In the benefit-ordered strategy, a specialization candidate having a larger value of B_i is selected as a higher level of node, reflecting a greater expected benefit when the condition holds true. In the sample-ratio-ordered, the value of R_i is regarded as a more important factor with expectation of a higher rate of executing the specialized versions. In Figure 4.5(b), assuming that SC_i is in the order of a, b, c by either criteria, so the decision-tree is constructed with the number of specialized versions limited to three.

The transformations with code specialization proceed with "ordinary" optimizations by using the set of the specialization variables and with their corresponding values. Those optimizations include constant folding, null-check and array-bound-check elimination, type check elimination or simplification, and virtual method inlining. When there is an SST failure upon entry into the specialized code block, the control is directed to the general version of the code, which is the level-2 or unspecialized level-3 code. This can occur as a result of changes in the execution phase of the program, so that the specialized methods are 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 a 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 avoid excessive code growth.

4.4.3 Specialization Example

Figure 4.6 shows an example of the code specialization. For the example program (a), the three instance field variables ($this.offset$, $this.count$, and $this.value$) can be identified as specialization targets, since there are no barrier instructions within the method for these global variables to be privatized at the method entry point. Thus, the impact analysis is performed for the three global variables and two parameter variables as specialization targets.

Suppose the result of the impact analysis is as shown in the table of Figure 4.6(b). This shows that, for example, if we know the value of the parameter $fromIndex$ as a constant, we would be able to optimize the program with a factor of 2.0 compared to the cost of the necessary guard code. Two of the specialization targets, however, have no impact on any type of specialization. The specialization candidate set SCS in this method is {($fromIndex$, $constant$), ($this.count$, $constant$), ($this.offset$, $constant$)}.

If this method is identified as hot and is being recompiled with level-3 optimization, the recompilation controller checks the result of the impact analysis, and performs the value profiling for these three specialization candidate SCs that have possible optimization benefits. Assuming that profile results are well biased for these variables as shown in the table, the controller may decide that it is profitable to specialize with these SCs based on the expected benefit and the hotness count of this method.

The specialization is thus organized to apply these SCs in the order of the expected benefit. Figure 4.6(c) shows a specialized version when we apply all of these SCs. We assume here that we construct two versions of the specialized code. The failure of the first two SST guard tests results in jumping to the general version, but when the third test fails, another version of specialized code is called. Given the <variable,value> pair of the specialization, the optimizer performs constant propagation and loop simplification, an existing set of optimizations, to produce the specialized code.

```
public int indexOf (int ch, int fromIndex) {
   int max = this.offset + this.count;
   char v[] = this.value;

   if (fromIndex < 0) {
      fromIndex = 0;
   } else if (fromIndex >= this.count) {
      return -1;
   }
   for (int i = this.offset + fromIndex; i < max; i++) {
      if (v[i] == ch) {
         return i - this.offset;
      }
   }
   return -1;
}
```

(a) Example program

```
public int indexOf_specializedA (int ch, int fromIndex) {
   /* specialization safety test (SST) */
   if (this.offset != 0) goto general version;
   if (this.count != 4) goto general version;
   if (fromIndex != 0) goto indexOf_specializedB;
   /* specialized version with this.offset = 0,
      this.count = 4, and fromIndex = 0 */
   char v[] = this.value;

   if (v[0] == ch) return 0;
   if (v[1] == ch) return 1;
   if (v[2] == ch) return 2;
   if (v[3] == ch) return 3;
   return -1;
}
```

(c) A specialized version of the example program.

Impact analysis result			Value profiling result (value: sample ratio)	Expected benefit *B*
Specialization target *V*	Specialization type *T*	*Weight(V,T)*		
fromIndex	*constant*	2.0	0: 100%	2.0
ch	*any type*	0.0		
this.count	*constant*	5.0	4: 70%, 10: 8%, 12: 7%, ...	3.5
this.offset	*constant*	6.0	0: 95%, 5: 1%, 3: 0.5%, ...	5.7
this.value	*any type*	0.0		

(b) Result of the impact analysis and value profiling

Figure 4.6: An example of code specialization. (a) shows an example program, and (b) shows the result of the impact analysis and value profiling. A specialized version of the program (c) has the SST guard code for specialization with this.offset = 0, this.count = 4, and fromIndex = 0.

4.5 Experimental Evaluation

This section shows the impact on both performance and compilation overhead when applying these profile-directed optimizations, both separately and in combination.

4.5.1 Methodologies

All of the measurement conditions are the same as those described in Section 3.4.1. Other conditions specific to the measurements here are as follows.

- For the instrumentation-based profiling for hot methods, a maximum of 10,000 values were collected for each of the target parameters, global variables, or return addresses. The maximum number of data variations recorded was 8.

- The number of code duplications allowed for specialization was set to one at a time, regardless of the target method code size.[9] The decision-tree construction was based on the benefit-ordered strategy.

- The maximum number of level-3 compilations for the same method was set to three. This is for both profile-directed inlining and code specialization with different profile information.

Four variations of profile-directed optimizations are compared as described below, two with the above optimizations applied separately, and the others with the optimizations applied in combination. The baseline of the comparison is with the MMI-all configuration described in Section 3.4.1.

- inline: This enable only the profile-directed method inlining on the baseline. None of the static heuristics for inlining are used except for tiny methods, which are always inlined. No profile data for specialization is collected, nor is any impact analysis performed.

- specialization: This enable only the code specialization on the baseline. No profile data for inlining is collected. Method inlining is performed using the default static heuristics as in the baseline.

- inline+spec: This enable both profile-directed inlining and the code specialization. The profile information for both inlining and specialization is collected at the same time for the given methods.

- inline+spec+edo: In addition to the above inline+spec configuration, *exception-directed optimization* (EDO) [85] is enabled. This is another example of a profile-directed optimization technique implemented in our framework to optimize frequently raised exception paths. A recompilation request from the EDO profiler is processed at the same optimization level with special treatment for method inlining for the specified hot exception paths.

When measuring the compilation overhead, we measured only level-2 and level-3 compilation statistics, since level-1 is the same for all four cases. The compiled code size includes additional code that was dynamically generated for instrumentation.

4.5.2 Profile-Directed Inlining

Figure 4.7 shows the comparisons of both performance and compilation overhead in the steady state benchmarks, and we can make the following observations.

[9] We have tried allowing multiple versions of specialized code allowed for each method, but found it not very effective [97].

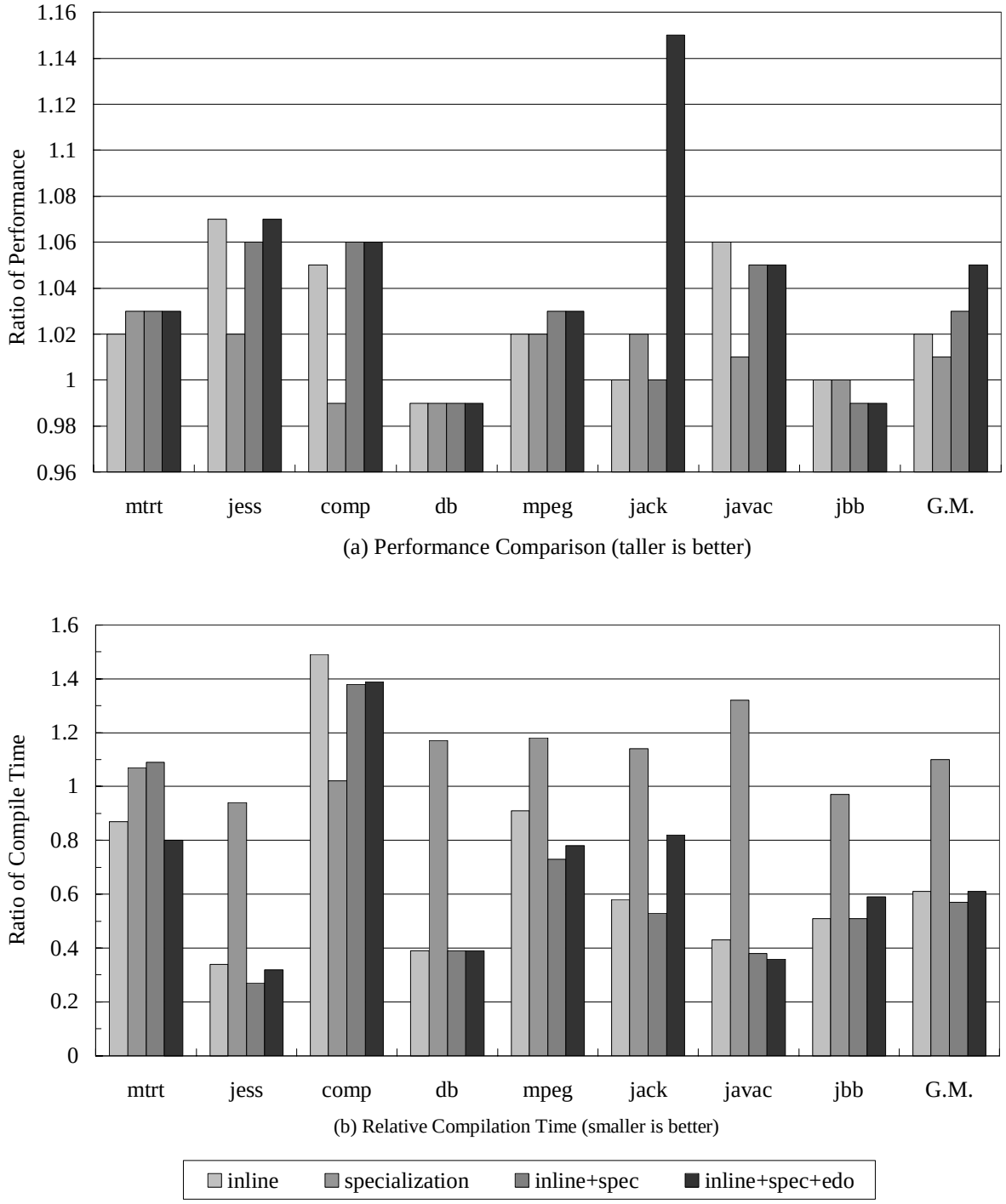


Figure 4.7: Steady state comparison on both performance and compilation overhead with four variations of profile-directed optimizations (1 of 2). Each bar indicates the relative numbers to the MMI-all configuration. The taller bar shows better performance in (a), while the shorter bar means smaller overhead in (b).

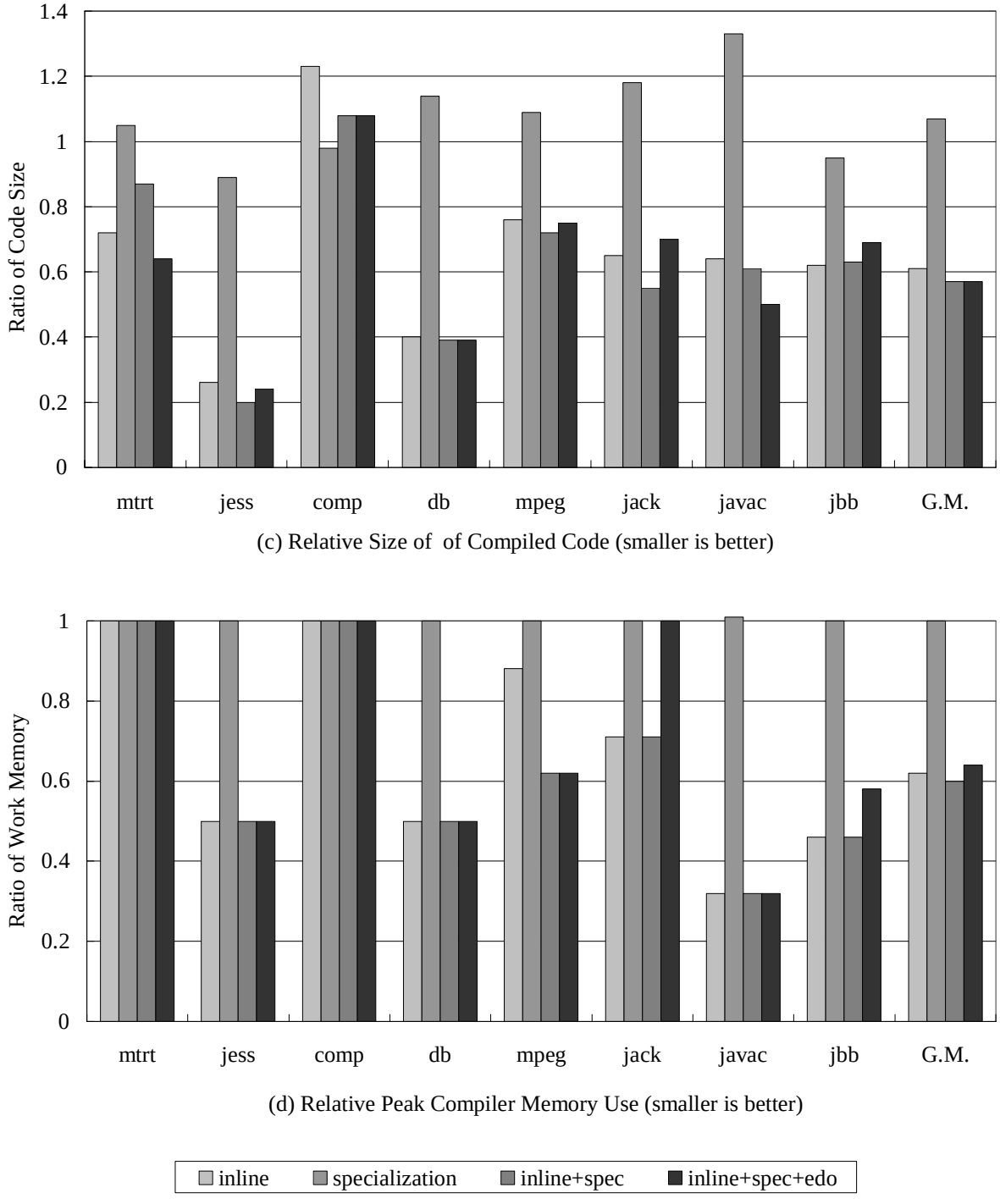


Figure 4.7: Steady state comparison on both performance and compilation overhead with four variations of profile-directed optimizations (2 of 2). Each bar indicates the relative numbers to the MMI-all configuration. The shorter bar means smaller overhead in (c) and (d).

The top graph in the figure shows that the profile-based inlining performs about as well as or better than the static-heuristics-based inlining for most of the benchmarks. The use of profiling seems particularly effective for _202_jess, _201_compress, and _213_javac.

The comparisons of the actual inlining results between profile-based and static-heuristics-based inlining policies show that there were several critical call paths that were not inlined with the static heuristics, but which were inlined with profile-based decisions, and this led to performance differences. One probable reason for these differences is that the chance to inline performance-critical call paths is lower for the static heuristics because the limited inlining budget is wasted by greedily inlining methods based on static assumptions.

The performance is slightly degraded with the profile-based inlining for _209_db. Investigation shows that the current profile-based inlining is missing some opportunities for inlining performance-sensitive call sites in this benchmark. This is because the sampling profiler overlooked as recompilation candidates target methods that should have been inlined. Our profile-directed inlining is only possible when the target method is selected as hot for dynamic instrumentation. Although tiny methods are always inlined, there are still some chances that other methods with very small bodies will be missed for selection as recompilation candidates due to the sampling nature of the profiler, and in that case it is unable to consider the inlining possibilities for those methods. This problem may be resolved by expanding the tiny method criteria, but this would go against the goal of our basically relying only on profile information with minimal heuristics. We need to come up with an effective way to deal with this situation.

As for the compilation overhead in the other three graphs, the profile-based inlining shows significant advantages over the existing static-heuristics-based inlining for most of the benchmarks. In the best case, it reduces the overhead by nearly 70%, and on average the reduction is about 40% in all three metrics of the compilation overhead. This is apparently due to limiting the inlining requests only to those call sites deemed beneficial for performance improvements. The compiled code size is greatly reduced for some benchmarks, even taking the instrumentation code into account.

There is an exception to these general observations, where _201_compress shows degradation with profile-based inlining. This benchmark has a very few hot methods, resulting in a relatively low level of compilation overhead regardless of the inlining policy. In the profile-based inlining case, two inlining requests were actually made for the same method, Compressor/compress, but with different timings because of the group of hot methods supplied by the sampling profiler. Thus the method was compiled twice with level-3 optimization, first with one inlining request and then with an additional request identified later. Because of the low level of compilation overhead, this additional level-3 compilation was enough to cause a seemingly big difference in these ratios.

This problem is considered inherent to an online profiling and feedback system, since the information is partial, not complete, at any point in time, and we do not know the best time to drive recompilation with the limited available information. When additional information that can contribute to the performance becomes available later, we then have

Table 4.1: Statistics of compiled and instrumented methods for each optimization level. The number of inlining requests is shown for each depth of call paths that was requested for inlining.

Program	L1		L2		L3	inline requests			
	comp'd	inst'd	comp'd	inst'd	comp'd	d=1	d=2	d=3	d>4
_227_mtrt	212	16	12	3	7	13	2	1	-
_202_jess	310	27	18	11	9	21	9	3	-
_201_compress	130	5	1	-	4	3	-	-	-
_209_db	129	5	1	1	3	6	-	-	-
_222_mpegaudio	221	23	23	11	16	8	3	2	3
_228_jack	387	58	43	17	15	19	4	1	1
_213_javac	838	117	128	25	18	20	3	1	-
SPECjbb2000	583	130	133	33	27	20	6	1	1

to decide whether or not it is better to trigger additional recompilations. Currently we drive recompilation by considering the estimated performance benefit of the additional inlining request based on the relative hotness counts of the caller and the callee, but we also limit the maximum number of level-3 compilations for the same method in order to avoid a code explosion. We need to explore better ways to manage this situation by, for example, introducing additional metrics for the impact of inlining on a particular call path. We would need to have a code reclamation function to remove this constraint, as in the continuous optimization framework described in [73].

Table 4.1 shows the compilation statistics when only profile-directed inlining is enabled. The numbers are for compiled and instrumented methods for each of the optimization levels. No instrumentation is applied to L3 compiled code, as mentioned in Section 4.3. The count is cumulative, so if a method is L1 compiled and then promoted to L2, then it is counted in both categories. The last four columns show the total number of inlining requests for each depth of call paths. The number is per call site, so if two inline requests are issued for a method, inlining it into two different call sites, then both are counted as separate inline requests.

4.5.3 Code Specialization

With code specialization only a modest performance improvement, from 2% to 3%, can be observed for four benchmarks, _227_mtrt, _202_jess, _222_mpegaudio, and _228_jack, while the others do not show any significant difference. Two of them, _201_compress and _209_db, have spiky profiles and only a few methods are heavily executed. Our impact analysis could not find any good candidates for specialization among these hot methods. In contrast, _213_javac and SPECjbb have many equally important methods, and specializing only a few of them does not seem to provide any additional speedup.

The code size growth observed was approximately 7% to 30%. The increased code

Table 4.2: Statistics of the number of methods and variables for specialization candidates and those actually get specialized.

Program	# of candidates		# of specialized		total L3
	methods(L2/L3)	variables(L2/L3)	methods	variables	compiled
_227_mtrt	5 / 1	8 / 2	4	6	9
_202_jess	5 / 0	5 / 0	4	4	14
_201_compress	2 / 0	6 / 0	2	3	5
_209_db	1 / 0	1 / 0	1	1	5
_222_mpegaudio	18 / 2	32 / 7	14	26	23
_228_jack	12 / 1	20 / 1	8	11	21
_213_javac	15 / 0	30 / 0	8	16	22
SPECjbb2000	23 / 1	42 / 2	12	15	38

size for _209_db, _228_jack, and _213_javac seems to be excessively high relative to the resulting performance gain. The hit ratio of specialized version code was 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 the data for parameters or global variables is relatively small within a single benchmark.

Table 4.2 shows the statistics for the numbers of methods and variables for the code specialization. The second and third columns are the specialization candidates shown in the number of methods and variables for each L2 or L3 compiled code. The value profiling is performed for these variables. The next two columns are the code specialization actually applied among these candidates when they are recompiled with L3 optimization. The total number of L3 compiled methods is in the last column. The table shows that except for _201_compress and _209_db a fair number of methods and variables are specialized.

These results collectively show that the code specialization seems to have only a modest impact on performance, at least for these benchmarks and with our implementation strategy. This is especially true when considering the moderately complicated technique that involves impact analysis, value profiling, and the specialization decision process. The results seem to be a contrast to what has been reported with specialization in other languages. For example, the system with the *alto* optimizer for C programs [83] shows an average of 4.6% (14.1% maximum) performance benefit over 8 integer SPEC-95 benchmarks, and the scheme in the Vortex compiler for Cecil [45] provides 20% to 100% improvement over a CHA-based optimization system on four selected benchmark programs.

We can think of several reasons for the differences between these systems and ours in the effectiveness of the code specialization. First, there may be problems in our implementation strategy for the specialization. For example, the target of specialization is the whole method, rather than a smaller region in the method, and this may make the opportunities for specialization smaller, because only those variables that can be guarded at the method entry are specialization candidates. If we move to specialization for a part

of a 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. Also we currently limit the maximum number of level-3 recompilations to three. This is related to the problem of specialization on the whole method, but the lack of a code reclamation capability is another reason for this restriction. Second, the two cited systems are both static compilers and the profile data is collected offline in a separate run. Thus call graphs for the most profitable sets of specializations can be constructed. Our system considers specialization individually within a single method. Third, Cecil is, like Self, a pure object-oriented language, so all data are objects and message passing is the only way to manipulate objects. This makes the overhead for dynamic method dispatch very high within the system, and the specialization that converts dynamic calls to static calls can be much more effective than in statically typed languages such as Java.

Overall, we need to investigate more deeply for a broader set of applications and benchmarks regarding whether or not the code specialization for Java can be effective in a dynamic optimization system. We should allow more aggressive code specialization after implementing both the specialization on any region of a program and a code GC mechanism.

4.5.4 Combination of Profile-Directed Optimizations

When enabling both the profile-directed inlining and the code specialization, the results are even better for both performance and compilation overhead. The compilation overhead is still held at a very low level, around 60% of the baseline for all three metrics, because of the profile-directed inlining. For the performance, we can see the same level of improvement that came from either of the two profile-directed optimizations, and in some benchmarks such as `_201_compress` and `_222_mpegaudio`, we see better results than from applying the optimizations separately.

EDO is especially effective for `_228_jack`. This benchmark is known to frequently raise exceptions during the program execution, and EDO seems to optimize the code quite well by detecting some hot exception paths and by inlining the exception throwing methods into their corresponding catching methods. This produces another version of the code with additional inlining and can cause significant compilation overhead. The figure shows, however, that the overhead is at a reasonable level, and by applying this technique with profile-directed inlining, the total overhead can still be kept below the level of the baseline configuration.

4.6 Summary

This chapter described the design and implementation of the profile-directed method inlining and the dynamic code specialization. The profile-directed method inlining is performed in level-2 and level-3 optimizations, while the code specialization is applied

only at the highest optimization level. These techniques both exploit the dynamically generated instrumentation mechanism for collecting runtime information, such as call site distribution, parameter values, and global variable values.

The code specialization employs impact analysis to estimate the benefit of specialization and to decide for which variables the runtime information needs to be collected. The results show that profile-directed inlining has the potential for significantly reducing the compilation overhead, for improving performance, or both, compared to the static inlining heuristics. In contrast, the code specialization only produced modest performance improvements for a few benchmark programs. This limited performance impact overall may be due to our current design for code specialization, but needs to be investigated further, especially to clarify the difference between Java and other languages and systems that benefited more greatly from specialization.

In a dynamic compilation environment, we have to be very careful about performing any optimizations that can have significant impact on compilation overhead. We demonstrated that the online profiling can provide useful information to guide inlining and specialization opportunities only for those call sites or variables that can be expected to produce performance improvements.

Chapter 5

Region-Based Compilation

5.1 Introduction

Method inlining and dataflow analysis are two major components for effective program transformations at the higher optimization levels. However, methods often contain rarely or never executed paths even in the selected hot methods, as shown in previous studies [21, 114], and this can cause some adverse effects that reduce the effectiveness of these optimizations. For example, method inlining can be restricted due to the excessive code size caused by the rarely executed code in a target method. This is because dynamic compilers usually manage the inlining process with a fixed budget using metrics such as the estimated code size and the number of local variables in order to avoid an excessive compile-time overhead and explosion in the code size. Some methods may include a large amount of rarely executed code at the beginning, and this may prevent them from being inlined at the corresponding call sites. Others can grow large from the cumulative effects of sections of rare code after several stages of inlining, and this may prevent other hot methods from being inlined. Also, dataflow analysis is often hindered by kill points existing in those rarely executed paths, whose control flow may merge back to non-rare paths, and this can prevent the propagation of accurate dataflow information on non-rare paths.

The problem here lies in the fact that we implicitly assume methods are the units for compilation. Even if we perform inline expansion, we either inline or do not inline the entire body of a target method, regardless of the structure and dynamic behavior of the target method. Method boundaries have been a convenient way to partition the process of compilation, but methods are not necessarily the most suitable units for optimizations. If we can eliminate from the compilation target those portions of the code that are rarely or never executed, we can focus the optimization efforts only on non-rare paths. This would make the optimization process both faster and more effective.

In this chapter, we describe the design and implementation of a region-based compilation (RBC) technique in our dynamic optimization framework. In this framework, we

no longer treat methods as the units of compilation, as in traditional method-based or function-based compilation (FBC). Instead, we select only those portions that are identified as non-rare paths. The term *region* refers to a new compilation unit, which results from collecting code from several methods of the original program but excludes all rarely executed portions of these methods. Regions are inherently inter-procedural, rather than intra-procedural, and thus the region selection needs to be interwoven with the method inlining process. We also describe two RBC-related optimizations, partial escape analysis and partial dead code elimination, which take advantage of the selected regions.

The notion of region-based compilation was first proposed by Hank et al. [56], as a generalization of the profile-based trace selection approach [79]. They showed some experimental evidence of the potential impact by allowing the compiler to repartition the program into desirable compilation units. An improved region formation algorithm was then proposed by combining region selection and the inlining process [111, 112]. The goal of this prior work was to expose as many scheduling and other optimization opportunities as possible to an ILP static compiler without creating an excessively large amount of code due to the aggressive inlining. In a dynamic compilation environment, however, this technique is especially useful for several reasons. First, dynamic compilers can take advantage of runtime profile information from currently executing code and use this information for the region selection process. Second, they are very sensitive to the compilation overhead, and this technique may significantly reduce the total compilation time and code size. Third, they can avoid generating code for unselected regions until the code is actually executed at runtime.

The key components for the RBC approach are region selection, partial inlining, and the region exit handling. For region selection, we employ both static heuristics and dynamic profiles to identify seed blocks of rare code and then propagate them to form rare code sections. The region selection process and method inlining can affect each other, in the sense that method inlining exposes new targets for region selection, and the region selection process in turn conserves the inlining budget, allowing more methods to be inlined. Thus the inlining process should be performed for parts of a method, not for the entire body of the method. When the program attempts to exit from a region boundary at runtime, we trigger recompilation and rely on *on-stack replacement* (OSR) [59], which is a technique to dynamically replace a stack frame in one form with another form, in order to continue the execution from the corresponding entry point in the recompiled code.

In the rest of this chapter, Section 5.2 first describes the existing RBC strategies from three important perspectives relative to this new technique, and discusses the issues and challenges for performing RBC in dynamic compilers. Section 5.3 then describes the design and implementation strategy of our region-based compilation technique, including intra-method region identification, partial inlining, and other optimization techniques that are aware of region information. The region exit handling with the on-stack replacement mechanism is also described for safely executing the exit path from the selected region. Section 5.4 presents the experimental results on both performance and compi-

lation overhead based on the actual implementation in our production-level Java JIT compiler. Finally, Section 5.5 gives the summary of this chapter.

5.2 RBC Strategies and Issues

There have been several previous efforts involving region-based compilation and the related techniques that addressed the problem of rarely executed paths affecting optimization opportunities. These systems range from static compilers and dynamic compilers to dynamic binary optimizers. We can examine these systems from the three key aspects of the technique – how the regions or the units of optimization are extracted, how the region formation and method inlining interact, and how those unselected regions and the region boundary points are handled. This section overviews the previous approaches for each of these three aspects and discusses the issues and challenges when performing effective RBC in the dynamic compilers.

5.2.1 Region Formation

The effectiveness of the region-based compilation largely depends on how the system performs the region formation by extracting units of optimization that are more representative of the dynamic behavior of the program than the original units partitioned by method boundaries. Figure 5.1 shows an example of region formation by three systems, a static compiler, a dynamic binary optimizer, and a dynamic compiler. The strategies used by these systems are described below.

One strategy for region formation is the offline-profile-based approach. This approach was used in a region-based compilation framework in [56, 55] for an ILP static compiler in order to expose many interprocedural scheduling and optimization opportunities. The region formation algorithm in this framework uses the following four steps, based on execution frequency information being available for all blocks: 1) selection of the seed basic block with the highest execution frequency, 2) region expansion to the successors of the seed block based on the execution frequency, 3) region expansion to the predecessors of the seed block (analogous to successors), and 4) region expansion to the successors of all of the blocks in the seed path. The expansion of the region is halted by several factors, such as the region size, the minimum acceptable execution frequency of the given block relative to the seed block, and the presence of optimization hazards [57].

The last step of the above algorithm expands the region along multiple control flow paths from a single trace of the control of the seed path. Thus the regions formed with this algorithm contain a wider range of program fragments than simply selecting a single trace at a time. The use of offline and complete profile information in this algorithm provides compilation units that reflect the bias in the actual behavior of the program. With this approach, the whole program can be partitioned into non-overlapping regions.

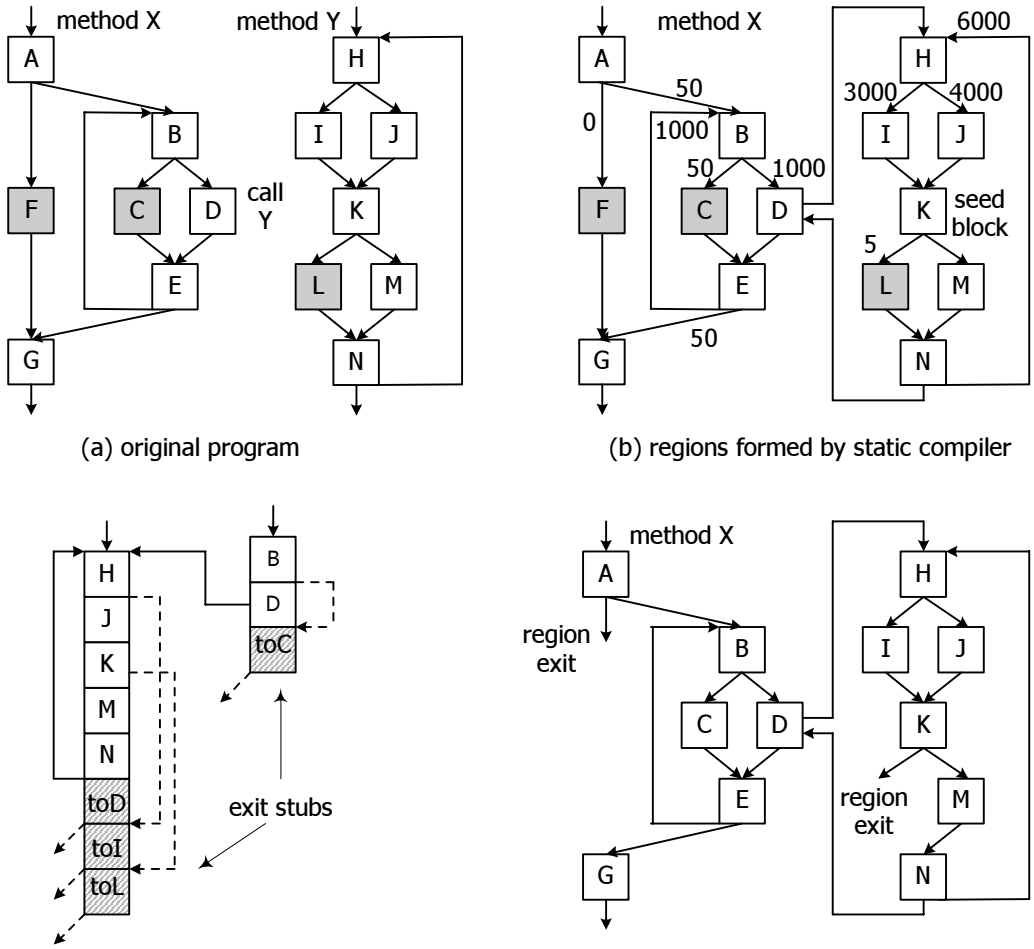


Figure 5.1: An example of regions formed by a static compiler, dynamic optimizer, and dynamic compiler. The shaded blocks in (a), C, F, and L, are assumed to be rarely executed. A static compiler may obtain the offline collected profiles as shown in (b), and select a region excluding the shaded blocks, assuming that method Y can be inlined into method X. A dynamic binary optimizer may generate two traces as shown in (c), one for the inner loop and the other for the outer loop. A dynamic compiler may remove the two rarely executed blocks from the compilation target as shown in (d), again assuming that method Y can be inlined into method X.

In Figure 5.1(b), the seed block K is selected first based on the profile information available (we assume method Y is inlined into method X). From the seed block, the region is then expanded to both successors and predecessors, resulting in the seed path A-B-D-H-J-K-M-N-D-E-G. The block I is then added to the region as a frequent successor of the seed path. The remaining shaded blocks C, F, and L are infrequent successors and thus form separate single block regions.

Another strategy is the online-trace-based approach that is typically used in dynamic binary optimization systems such as Dynamo [19], DynamoRIO [22], and Mojo [33]. These systems take a native instruction stream as input, which comes from a statically optimized binary, and reoptimize the code at runtime. When they identify a program hot trace as an optimization target, they extract the frequently executed paths (traces) to form a single-entry multiple-exit contiguous sequence as a unit for optimization. The online nature of these systems, however, requires that the trace be identified quickly, or the overhead of the system outweighs the benefit of the reoptimizations. For example, Dynamo [19] employs a technique called NET (Next Executing Tail) [48] to minimize the total profiling overhead. A counter is associated with each selected start-of-trace point, such as the target address of a backward branch. If the counter exceeds a preset threshold value, the next executing trace is assumed to be hot and is recorded as a target for performing dynamic optimizations.

Figure 5.1(c) shows two separate traces selected with this approach. The first trace starts from the inner loop entry and ends when it hits the backward branch. Note that either block I or J can be included in one trace, even if both are executed frequently. The second trace starts from the outer loop entry and ends when it hits the entry of the already generated trace. The exit point of the second trace is linked to the entry of the first trace.

In dynamic compilers, we cannot use either of these approaches for the following reasons. First the offline-profile-based approach assumes that a complete profile that covers all of the code within the target method is always available. It also implicitly assumes that the profile incurs no runtime cost because it is collected during a separate profile run. Neither of these assumptions is true with dynamic compilers. The profile needs to be collected at runtime from the currently executing program. Also, no profile information may be available for some parts of the target code, as described in the previous section.

Second, the online-trace-based approach may cause relatively frequent region exits at runtime, especially when the selected trace is hot but not a dominant one. That is, if there are several different paths that are frequently executed equally, as in the case of blocks I and J in the example, there will be no single dominant trace we can effectively pick up for dynamic optimization. The frequent region exits are likely to lead to performance degradation rather than improvement in the case of dynamic compilers.

This trace-based approach can certainly be effective for a system that dynamically reoptimizes code on top of already (statically) optimized generic binaries, since the system

Table 5.1: A summary of the region definition and region formation in three systems.

System	Region definition, formation, and handling of non-selected code
Static compiler	Region is a program fragment (not a trace) that is identified as hot. Region formation is solely based on offline-collected profile. Unselected blocks are separately compiled (cold) regions.
Dynamic binary optimizer	Region is a single trace of control that is identified as hot. Region formation is solely based on online profiling. Unselected code is in an existing binary (and thus does not matter).
Dynamic compiler	Region is a program fragment after removing rarely executed code. Region formation is based on heuristics combined with online profile. Unselected blocks are not handled until being executed at runtime.

simply creates a specialized, reoptimized version for the fragment of the code, and the original code exists when the control actually exits from the reoptimized code. This is quite a different compilation model from our dynamic compilation environment. The cost of handling the region exits can be significantly higher when the control escapes from the regions, since we need to create the backup compiled code on the fly or execute with the interpreter.

For the above reasons, some dynamic compilers use the strategy of removing rarely executed code blocks based on some heuristics combined with available profile information, and perform a very limited form of region-based compilation. For example, both SELF-91 [29] and SELF-93 [59] systems employ a strong type prediction system and remove the code blocks for messages sent to receiver classes that are predicted to be uncommon. Similarly the HotSpot server [87] mainly focuses on the code of uncommon virtual method targets and references to uninitialized classes, and avoids code generation for those code blocks. The Jikes RVM [50] uses online profile information, in addition to class hierarchy analysis, to identify and remove the rarely executed code blocks. This strategy generally creates a more conservative region than the other two strategies described above.

In Figure 5.1(d), two blocks, F and L, that are never or rarely executed, are removed from the compilation target (we again assume method Y can be inlined into method X). The block C is not removed, since it is executed relatively infrequently but still quite a few times. The compilation for the removed (unselected) blocks is deferred until they are actually to be executed at runtime.

Table 5.1 summarizes the differences in the definition of a region and the strategy of the region formation for these three systems.

5.2.2 Method Inlining

The interprocedural regions are the keys for region-based compilation to effectively expose optimization opportunities that are missed due to procedure boundaries. Thus it

is important to see how the region formation and the method inlining processes interact with each other.

Under the original framework of [56], the inlining process and the region formation process were separate processes. The target program was first flattened with as much inlining as possible, and the region formation was performed for the resulting code after the inlining process. Since one of their motivations for the region-based compilation was to create compilation units that are roughly similar in size, in contrast to the method-based compilation units, this approach worked well by putting an upper bound on the allowable region size.

However, this approach has the following problem. The amount of a priori inlining performed may place an upper bound on the quality of the regions that can be extracted and thus on the effectiveness of optimizations on those regions. In other words, if we want to explore more effective regions, the inlining needs to be performed in an aggressive or sometimes overly aggressive way to expose a larger scope of code for partitioning.

Way et al. [111] addressed this problem by extending the region formation algorithm to make region-sensitive inlining decisions and to identify interprocedural regions. In the improved algorithm, the inlining is not performed a priori, but it is interwoven with the region formation process. When a call site is encountered as a most frequent successor or predecessor of the seed block, the algorithm first determines whether the call site is inlineable, and if so, it then continues forming regions within the target callee code. Therefore, the inlining is driven by the demands made at call sites as a region is formed, and the interprocedural regions are identified by having the region formation process cross method boundaries. Because the inlining is performed for the target callee as part of the region formation, this leads to the effect of partial inlining, making it easier and more natural to achieve such partial inlining.

Procedure Boundary Elimination (PBE) [107, 106, 105] addressed the same problems, but in a different way. PBE first unifies the whole program into a single compilation unit, which they call the procedure unification process, and then repartitions the unified program into compilation units suitable for optimizations. In the repartitioning phase, PBE uses the same algorithm as the one originally used by Hank et al. [56]. The unification is different from inlining in that it simply replaces all call and return instructions with normal branch instructions, not with the procedure bodies, taking care of the parameter passing and local variable renaming. PBE may effectively solve the problem of placing an upper bound on the amount of inlining and the quality of the regions that can be obtained, but this technique pays little concern to the compilation overhead caused by handling the entire program through the procedure unification.

In a dynamic compilation environment, all of the existing systems perform method inlining first, followed by region selection (removing rare code) as in the framework of [56], and thus have the same problems described above. We cannot afford to perform aggressive inlining for extracting better regions when there is a risk of causing excessive compilation overhead and code size growth. It is better to perform partial inlining by

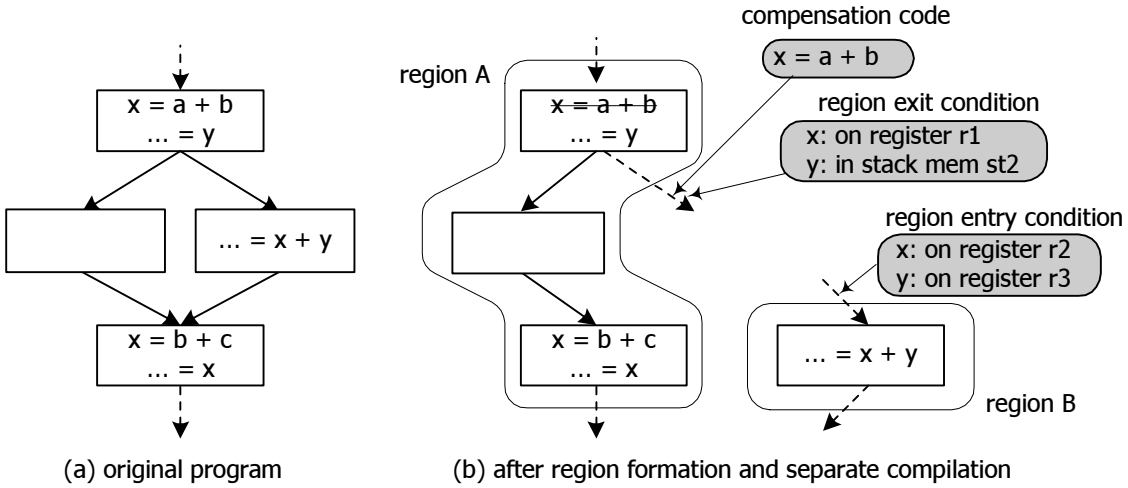


Figure 5.2: An example of reconciliation and compensation code required between two regions.

integrating the method inlining and region formation processes.

With the online-trace-based approach [19], dynamic execution traces are by definition contiguous instruction sequences that can extend across static program boundaries, and thus can be part of a method or span multiple methods. This results in an effect similar to partial inlining. In fact, one of the important optimization opportunities exploited in these native-to-native binary optimizers is straightening the code by eliminating procedure calls, returns, indirect branches, and so on. Note that the effect of inlining here is achieved by extracting a runtime trace that crosses method boundaries existing within the already compiled application binary. In contrast, partial inlining in static and dynamic compilers is performed by repeated interaction between inlining a single method and applying region selection to the method being inlined.

5.2.3 Region Exit Handling

As we saw when reviewing the region formation strategies in Section 5.2.1, a region (or a trace) is an arbitrary fragment in the program selected as an appropriate compilation unit. Differing from the method-based compilation units, where the procedure calling convention defines clear interfaces at the boundaries, there may be any number of variables that are live across each region's entry and exit points. Since each region is separately compiled, regions need to be *reconciled* at the boundaries by considering register allocation and memory dependencies for those live variables.

In addition, some optimizations applied to the region may transform the code to remove redundancies or to replace instructions with more efficient operations. This requires, however, that *compensation code* be generated at the region boundaries to compensate for any effects of the optimization on outside regions and to ensure correctness when control actually exits the region.

Figure 5.2 shows an example of both reconciliation and compensation code required for the region exit handling. The partial redundancy for the computation of x in the original program (a) becomes fully redundant when two regions are formed and the region A is compiled separately in (b). The redundancy elimination requires compensation code to be generated along the region-exiting path, since the variable x is live at the boundary and needs to be made available to the outside regions. Also, because of the separate compilation for each region, agreement on which registers or memory locations the live variables exist in is necessary between adjacent regions, another reason that reconciliation code is required.

In a static compilation environment, such as the framework of [56], the compiler determines the order in which the regions are compiled, organizing them from important, potentially more frequently executed regions to less important regions. Both compensation code and reconciliation code are pushed into unprocessed regions. The region exit conditions of register and stack location mappings for live local variables are propagated from the current region into all subsequent regions, and this information is incorporated in the compilation of the next region. The reconciliation code is generated to relocate the live variables at the entry of the region. Thus, the code for a region boundary is generated in the less frequently executed region.

In the dynamic binary optimizers, each exit of a newly formed region (trace) is linked to an entry point of an already existing optimized trace. Most of the optimizations performed involve redundancy removal operations (such as redundant branch elimination, redundant load removal, and redundant assignment elimination). The compensation code is generated at the bottom of the trace to incur the overhead only when control exits from the trace (see the exit stub in Figure 5.1(c)). The optimizer preserves the original register and stack location mappings, and therefore reconciliation code is not necessary. Thus efficient execution crossing multiple traces can be performed with simple branch instructions.

In dynamic compilers, we need both compensation code and reconciliation at region boundaries. Unlike static compilers, however, we cannot push the code into unprocessed regions, since they are not handled until they are actually being executed at runtime. The compensation code is thus generated in specially generated blocks provided at region exit paths. The reconciliation is performed by on-stack replacement (OSR) using the mapping information for local variables provided at each exit point. There are three options for handling region exits:

1. Simply fall back to the interpreter. This is the option taken by the HotSpot server compiler [87] when class loading invalidates inlining or other optimization assumptions. This relies on the underlying recompilation system to promote the "decompiled" interpreted method once again, if required.

2. Drive recompilation with deoptimization. This is the policy used by the SELF-93 system [59] and Jikes RVM [50]. The unoptimized code is generated at the

first transition from the region, and used only for handling the transitions from the optimized code, not for future invocations of the method. If the original optimization assumptions turn out to be wrong and the rare cases happen frequently, then the system reoptimizes the method to reflect the new reality, replacing the original code for future invocations, by using the underlying recompilation system. Thus this option can produce two additional versions of the code, the unoptimized code to handle transitions and the reoptimized code to handle new invocations.

3. Drive recompilation with the same optimization level as that used for the original code, but without region optimization. In this recompilation, inlining is performed for the same call sites and for the same target methods as applied when compiling the original region optimized version. Thus the recompiled version has all of the entry points corresponding to region boundaries in the original version, and it is used for both current transitions and future method invocations.

The mapping information we need to keep for restoring the state at the region boundary should be basically the same for all three of these options, but the methods used to reconstruct the new stack frame are different. In particular, the options (1) and (2) need to create a set of stack frames according to the inlined context at the transition point, while the option (3) simply creates a single new stack frame for the recompiled method.

There are both advantages and disadvantages for each of these three options, and which option we should choose depends on the meaning of "rare code". If we are very conservative, saying that only extremely rare cases are rare,[1] then the option (1) is probably the best choice, since the current version can still serve for future invocations and it is only necessary to handle transitions that may happen very infrequently. However, this means the compiler may miss some additional optimization opportunities for frequent cases due to being too conservative. On the other hand, if we use an overly aggressive strategy for optimizing away rare code, these cases will actually tend to occur frequently at runtime, and for the sake of overall performance it may be better to replace the compiled code as quickly as possible to avoid too many expensive OSR operations.

Ideally it might be better to choose from these options depending on the predicted execution frequency for each region exit point. For example, we could use branch prediction from the dynamic profile information.

5.3 Our Approach

This section provides a detailed description of our region-based compilation technique [100, 101]. Section 5.3.1 describes our design goals and the strategies when we modify

[1] We call this conservative since it removes less of the code from the compilation target. However, this may lead to less effectively optimized code on the frequently executed paths. This can happen because of inlining and dataflow analysis problems related to relatively infrequent code which is not "extremely rare".

our dynamic compiler to perform the RBC optimizations. Section 5.3.2 describes the intra-method region selection process using both static heuristics and dynamic profiles. Section 5.3.3 shows how the region identification and method inlining are interrelated to obtain more desirable targets as compilation units. Section 5.3.4 describes our implementation of OSR as support for handling region exit points, and Section 5.3.5 gives some useful optimizations we implemented to exploit the characteristics of the target code in the selected regions.

5.3.1 Design Goals and Strategy

Our region-based compilation approach assumes a multi-level dynamic optimization environment, with a lightweight mechanism for detecting and promoting hot methods to higher levels of optimizations, as described in Section 3.2. Our RBC approach exploits this framework to understand the dynamic profiles of the program and to provide better optimization opportunities for the hottest methods. Our goals for the region-based compilation approach are:

- Improve the application performance. Our primary goal is to improve the performance in the application's steady state by effectively exposing optimization opportunities through region selection. Method inlining and dataflow analyses are two major areas where we can expect better optimizations.

- Reduce the total compilation overhead. The secondary goal is to reduce the total compilation overhead (both time and space) by narrowing the compilation target through effective region selection. The overhead here includes the costs for the recompilations that can occur for handling region exits at runtime.

In order to achieve these goals, we use region-based compilation only in level-2 and level-3 optimizations, using the rare code profile information collected from instrumenting level-1 compiled code. Level-2 and level-3 optimizations are applied only on selected hot methods, but they are a major part of the compilation overhead and have a large impact on performance. Since level-1 optimization is designed to be very lightweight with limited inlining applied, as described in Section 3.2, little benefit could be expected from region selection at this level.

The following are the decisions used in our RBC implementation for the three key design issues described in Section 5.2:

- We perform region selection by identifying never or rarely executed portions that we can remove from the compilation target, as done in other dynamic compilers and as discussed in Section 5.2.1. We use both static heuristics and dynamic profiles to identify and eliminate rare sections of code.

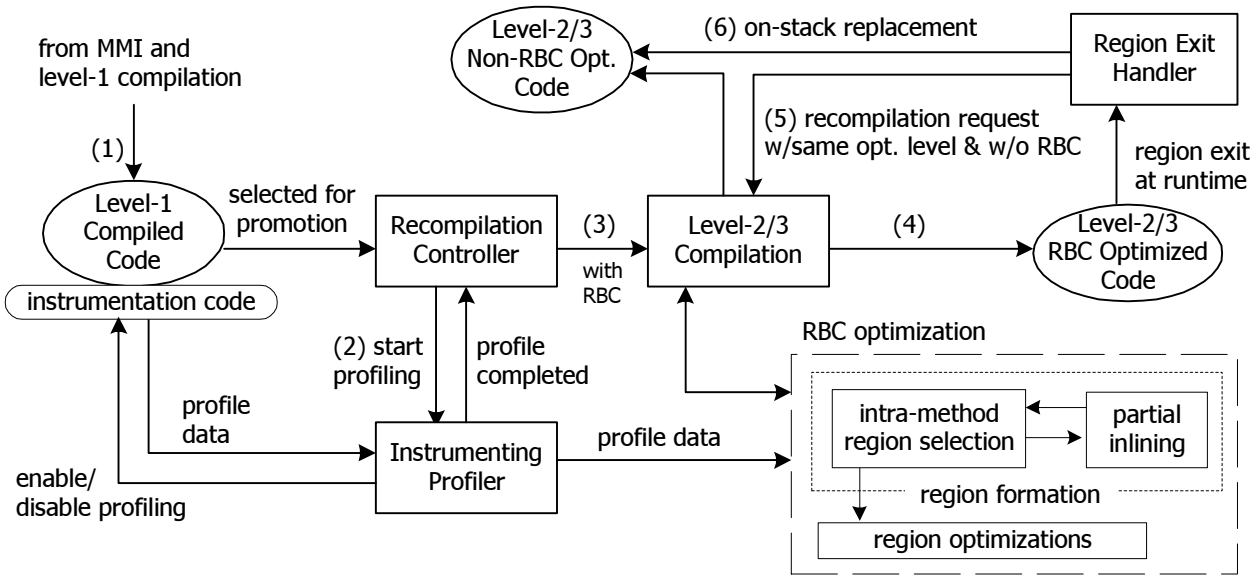


Figure 5.3: The high level view of our configuration with RBC-based optimizations. Both static heuristics and dynamic information provided by the instrumenting profiler are used in the region formation process. The numbers from (1) to (6) shown in the figure correspond to each step described in the text.

- We allow method inlining and region selection to interact with each other to direct the process of interprocedural region formation. We apply region selection for each target method first within the inlining process, and then perform partial inlining, only for the selected parts of the target rather than for the entire body of the method. When making a decision on inlining the selected regions, we use the same set of criteria and inlining budget as used in our system without RBC.

- For the region exit handling, we use the strategy of recompiling with the same optimization level as that used for the target code, the option (3) in the list of Section 5.2.3. This is because we apply RBC only at the higher optimization levels (level-2 and level-3). Those methods where region exits occur are already known to be hot and performance critical, and thus it could cause significant degradation of the overall performance to deoptimize or decompile those methods. We assume there is no significant difference in the status of class resolution and class hierarchy for the recompilation to have the same inlining scope for generating all of the corresponding entry points.

Figure 5.3 shows a high-level view of the configuration, indicating how a method is RBC optimized and then how the region exit handling is processed for that method. The details are described here:

1. Execution begins with the MMI. The MMI selects a hot method based on the invocation frequency to generate level-1 compiled code. The RBC optimization is

not applied during the level-1 compilations. The instrumentation code for profiling is generated in the code, but left disabled.

2. When a level-1 compiled method is selected for higher optimizations (level-2 or level-3), the recompilation controller directs the instrumenting profiler to enable the instrumentation code for this method. The profiler disables the profiling when enough samples have been collected.

3. The controller then drives recompilation for this method at either level-2 or level-3, depending on the hotness level. The RBC optimization is applied at this time using static heuristics and the profile data provided by the instrumenting profiler.

4. The level-2 or level-3 RBC-optimized code is generated. If the code is level-2, it may be promoted to level-3 by the sampling profiler. In that case, the RBC optimization is applied again using the same profile data (not shown in Figure 5.3). As long as the execution stays within the selected region, the next two steps do not occur.

5. When the control exits from the selected region of the RBC-optimized code, the region exit handler forces recompilation to generate new code for the target method. The code is optimized at the same level as the level of the RBC-optimized code, but RBC optimization is not used in the recompilation. The new code includes all of the entry points for transitions. This step is done only for the first region exit from the RBC-optimized code block.

6. The region exit handler then uses on-stack replacement to seamlessly continue the execution from the corresponding entry point in the new code.

5.3.2 Intra-Method Region Selection

Our intra-method region selection algorithm is shown in Figure 5.4. We assume each method is represented as a control flow graph (CFG) at this point with a single entry block and a single exit block. In the algorithm, *Gen* represents a 2-bit flag for the rare or non-rare for the seed basic blocks, and *In* and *Out* indicate the flag at the entry and exit of each basic block, respectively.

The algorithm begins by marking the *Gen* flag to seed basic blocks by employing both heuristics and dynamic profile results. We currently use the following heuristics.

- A backup block generated by compiler versioning optimization (such as devirtualization of method invocation) is rare.
- A block that ends with an exception-throwing instruction (`OPC_ATHROW`) is rare.
- An exception handler block is rare.
- A block containing unresolved or uninitialized class references is rare.
- A block that ends with a normal return instruction is non-rare.

```
 1:  procedure RegionSelection (M)
 2:  Input:  target_method M
 3:  Output: target method M' after removing rare blocks
 4:
 5:  Rare, NonRare: a bit representation, 10 and 01, respectively
 6:  In, Out, Gen: a 2-bit flag for each basic block and initialized with zero
 7:
 8:  /* initialization phase (set flag for seed blocks) */
 9:  for each basic block bb in any order {
10:      /* mark Gen flag based on heuristics */
11:      if (matched to static heuristics) {
12:          Gen(bb) = Rare or Gen(bb) = NonRare;
13:      }
14:      /* mark Gen flag based on profile information (overriding heuristics) */
15:      if (profile_info_available(bb)) {
16:          if (profile_count(bb) == 0) {
17:              Gen(bb) = Rare;
18:          } else if (profile_count(bb) > threshold) {
19:              Gen(bb) = NonRare;
20:          }
21:      }
22:  }
23:
24:  /* iteration phase (grow rare and non-rare regions) */
25:  do {
26:      changed = false;
27:      for each basic block bb in reverse post order {
28:          /* compute Out flag from all successors' In flag */
29:          Out(bb) = Union of In(succ(bb)) for all successors of bb;
30:          if (Out(bb) & NonRare) {
31:              /* unmark Rare since at least one successor is marked NonRare */
32:              Out(bb) = NonRare;
33:          }
34:          /* select either Gen or Out flag and then update In flag */
35:          temp = (Gen(bb) == 0) ? Out(bb) : Gen(bb);
36:          if (temp != In(bb)) {
37:              In(bb) = temp;
38:              /* need to iterate further */
39:              changed = true;
40:          }
41:       }
42:  } while (!changed)
43:
```

Figure 5.4: Algorithm for intra-method region selection (1 of 2). It first selects seed blocks based on both heuristics and profile results, and then propagates the flags along the backward data flow.

```
44:  /* final phase (remove basic blocks marked rare) */
45:  perform live variable analysis over the current method M;
46:  for each basic block bb in depth first order {
47:      for each successor succ-bb of bb {
48:          if (In(bb) == NonRare && In(succ-bb) == Rare) {
49:              /* found a boundary from a non-rare block to a rare block */
50:              new-bb = create a new basic block as RE-BB;
51:              new-inst = create a special instruction in new-bb;
52:              operand(new-inst) = set all live variables at succ-bb entry;
53:              for each predecessor pred-bb of succ-bb {
54:                  remove an outgoing edge to succ-bb;
55:                  create an outgoing edge to new-bb;
56:              }
57:          }
58:      }
59:  }
```

Figure 5.4: Algorithm for intra-method region selection (2 of 2). In the final phase, it detects a transition point from non-rare to rare, and replaces the rare block with a new basic block containing a special instruction.

The first heuristic above is the same as trusting the compiler's assessments of which blocks of the versioned code are executed most frequently. The second and third heuristics are based on the general observation that exception operations are extremely rare in Java programs. The fourth heuristic for unresolved and uninitialized class references is from the fact that the block containing such references cannot have been executed through the MMI and level-1 compiled code. Finally, we treat all return instructions as the places where the control normally exits from methods.

If dynamic profile information is available and it shows that a block was never executed, then we mark that block as rare. If the profile count value is above a predetermined threshold, the block is marked as non-rare. The dynamic profile information is given priority when there are conflicts with the static heuristics.

In the iteration phase, we propagate this information along the backward dataflow until it converges for all of the basic blocks. The *Out* flag of a basic block is marked non-rare if any of the successor's *In* flags is marked non-rare. If no successor has an *In* flag marked non-rare and any one of the successors has a rare *In* flag, then the *Out* flag of the basic block is marked rare. If there is a conflict for rare or non-rare between the *Gen* flag and the *Out* flag of a basic block, the *Gen* flag is selected to propagate further. Thus a region of rare code can grow backwards until it encounters a non-rare path, or a statically identified rare region can be blocked from growing by a profile-based non-rare block along its path. When converged, the rare regions should have reached the points where the branches are expected to be rarely taken from the non-rare paths. The *In* flag indicates the result of rare or non-rare for each basic block.

In the final phase, live variable analysis is performed first for the given method. This

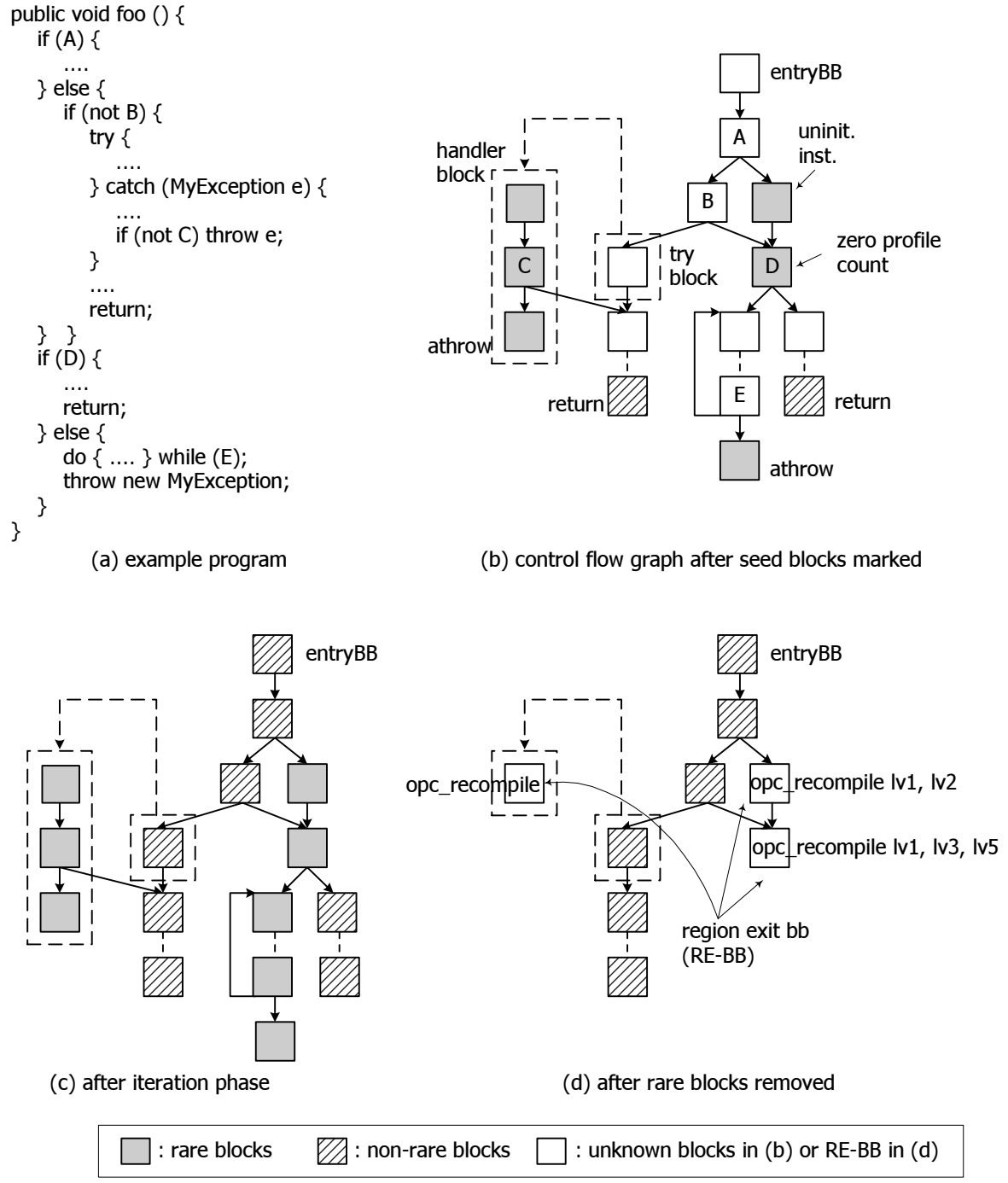


Figure 5.5: An example of intra-method region selection, showing (a) a Java pseudocode program, (b) the control flow graph with markings on some seed blocks, (c) the graph after completing the iteration phase of the algorithm, resulting in all blocks marked either rare or non-rare, and (d) the graph after removing the rare blocks and providing the region-exit basic blocks (RE-BBs) containing all of the live variable information.

is to find the set of live variables at the entry of each basic block before removing the rare blocks from the control flow graph. We then traverse the basic blocks to determine the transition points from a non-rare block to a rare block, and generate a new *region exit basic block* (RE-BB) for each boundary. This new RE-BB contains a single special instruction, called `OPC_RECOMPILE`, which holds all of the live variables for the rare block as its operands. When executed, this instruction calls a runtime routine that triggers recompilation if required and then performs OSR. Finally, we remove all the existing control flow edges going to the rare block, and create the corresponding edges from the same source blocks into the new RE-BB. This means that all of the rare blocks that originally existed following the boundary point are no longer reachable from the top of the method and thus can be eliminated in the succeeding control flow cleanup phase.

Figure 5.5 shows an example of the intra-method region selection. For the original program (a), the control flow graph is shown in (b) with markings on some seed blocks based on both heuristics and profile results. The heuristics used are for a block ending with an exception-throwing instruction, blocks within the exception handler, a block containing an uninitialized class reference instruction, and for blocks ending with a normal return instruction. These rare or non-rare flags in the seed blocks are then propagated along the backward dataflow path, resulting in the final state shown in (c). Finally, (d) shows the pruned control flow graph after the rare regions are replaced by the RE-BBs at each boundary point.

5.3.3 Partial Inlining

Partial inlining begins by performing region selection for the given target method as shown in Figure 5.6. The inliner then builds a call tree[2] for the possible inlined scopes from this target method based on the allowable call tree depths and callee method sizes. Thus, the initial call tree indicates a larger scope of inlining than that which can actually be performed within the inlining budget.

The actual inlining pass iterates over each call edge of the call tree and checks each individual decision for inlining against the total cost. Specifically, the inliner tries to greedily incorporate as many methods as possible using static heuristics until the pre-determined budget is used up. Tiny methods are always inlined without qualification [99]. Otherwise the target method is first processed by region selection, and then it is determined whether or not the method is inlineable based on this reduced code size. If inlineable, the inlining is performed only for the non-rare part of the code, and the current cost is updated with the reduced size of the method.

An important part of this process is to update the live variable information in each special instruction provided in the RE-BB for the method being inlined. There are two things that need to be done. One is to rename those live variables by reflecting the

[2]Since we apply method inlining differently depending on the call context (such as for a call site within a loop or outside of a loop), this structure needs to indicate call-site-specific information.

```
 1:  procedure PartialInline (R, B)
 2:  Input: target_method R, inlining_budget B
 3:  Output: compilation unit (selected region) for R after inlining
 4:
 5:  /* setup for the target method */
 6:  R' = RegionSelection (R);
 7:  construct a (possibly large) call_tree T rooted from R';
 8:  set current_cost C = 0;
 9:
10:  /* actual inlining pass */
11:  do {
12:      select an unprocessed call edge E in call_tree T;
13:      M = callee method of E;
14:      if (is_tiny(M)) {
15:          /* tiny methods are always inlined */
16:          perform inlining M into R';
17:      } else if (C < B) {
18:          /* decision is made after region selection */
19:          M' = RegionSelection (M);
20:          if (inlinable(M')) {
21:              perform inlining M' into R';
22:              update live information for each RE-BB in M';
23:              C += cost(M');
24:          }
25:      }
26:      if (inline performed) {
27:          update call_tree T
28:      }
29:  } while (an unprocessed call edge exists in T)
```

Figure 5.6: Algorithm for interaction of intra-method region selection and inlining process, leading to partial inlining.

mapping into the caller's context, just like other local variable conversions when inlined. The other is to add the live variables at the call site to reflect the complete set of live variables at each region exit point. This operation is necessary to automatically hide the effect of later optimizations such as copy propagation, constant propagation, and any other program transformations.

When the inlining has been performed on a method during an iteration, the call tree is then updated in order to encompass the new candidates exposed within the inlined code. This is because the devirtualization of the dynamically dispatched call sites is performed after the caller method has actually been inlined.

The direct advantages of the partial inlining in our framework are twofold:

- Since we first remove the rarely executed paths before trying to find the next inlining candidate, we never inline methods at call sites within rare portions of code, since they are no longer included in the current scope. This avoids performing inlining at

performance-insensitive call sites and can conserve the inlining budget.

- Methods being inlined are first processed through intra-method region selection before actual inlining. Inlining is considered and carried out against the reduced target code after removing the rare portions of the code, and this contributes to conserving the inlining budget.

Assuming the given set of criteria for method inlining is reasonable and thus fixed, we can then use the saved budget from the above steps and try to inline other methods in the call tree, which is expected to be more effective and can contribute to further improving performance. Other indirect benefits due to partial inlining include that 1) instruction cache locality can be improved since non-rare parts of the code tend to be better packed into the same compilation unit, and 2) later optimizations in the compilation process can be more effective with increased optimization scope.

5.3.4 Region Exit Handling

As described in Section 5.3.2, we provide an `OPC_RECOMPILE` instruction at each region exit point. This is a first-class operator in our compiler's intermediate representation, as in the previous implementation of OSR [50, 114]. We keep all live variables at that program point in the given bytecode sequence as operands of this special operator for both local and stack variables. This is done at a very early stage in the compilation, so that any optimizations in later phases can rely on the use of those variables across each region exit boundary for each valid transformation. At the final stage of the compilation, we create a map at each region exit point indicating the final locations of those variables within the stack frame. This is similar to the region exit condition shown in Figure 5.2. The map includes other information for performing frame conversion, such as frame size and the callee-saved registers.

Figure 5.7 illustrates how a transition from RBC-optimized code to recompiled code is handled, using the same example method shown in Figure 5.5. The numbers in the figure indicate the sequential steps of handling the region exit process. When the `OPC_RECOMPILE` instruction is executed at one of the region exit points within the method, the region exit handler is called (Step 1). If this region exit event is the first one for the method, the handler drives recompilation for this method to create the code that contains the whole body of the method (Step 2). As mentioned in Section 5.3.1, we perform the recompilation with the same optimization level as that used for compiling the current RBC-optimized code, but at this time do not apply the RBC optimization to avoid recursive recompilation. This recompiled version prepares all of the entry points within the method for possible future transitions, not just the entry point for the current transition, as shown in the figure.[3]

[3]This means the compilation target must include the complete body of the inlined methods that the original code included. This may result in an excessively large compilation target in a pathological case,

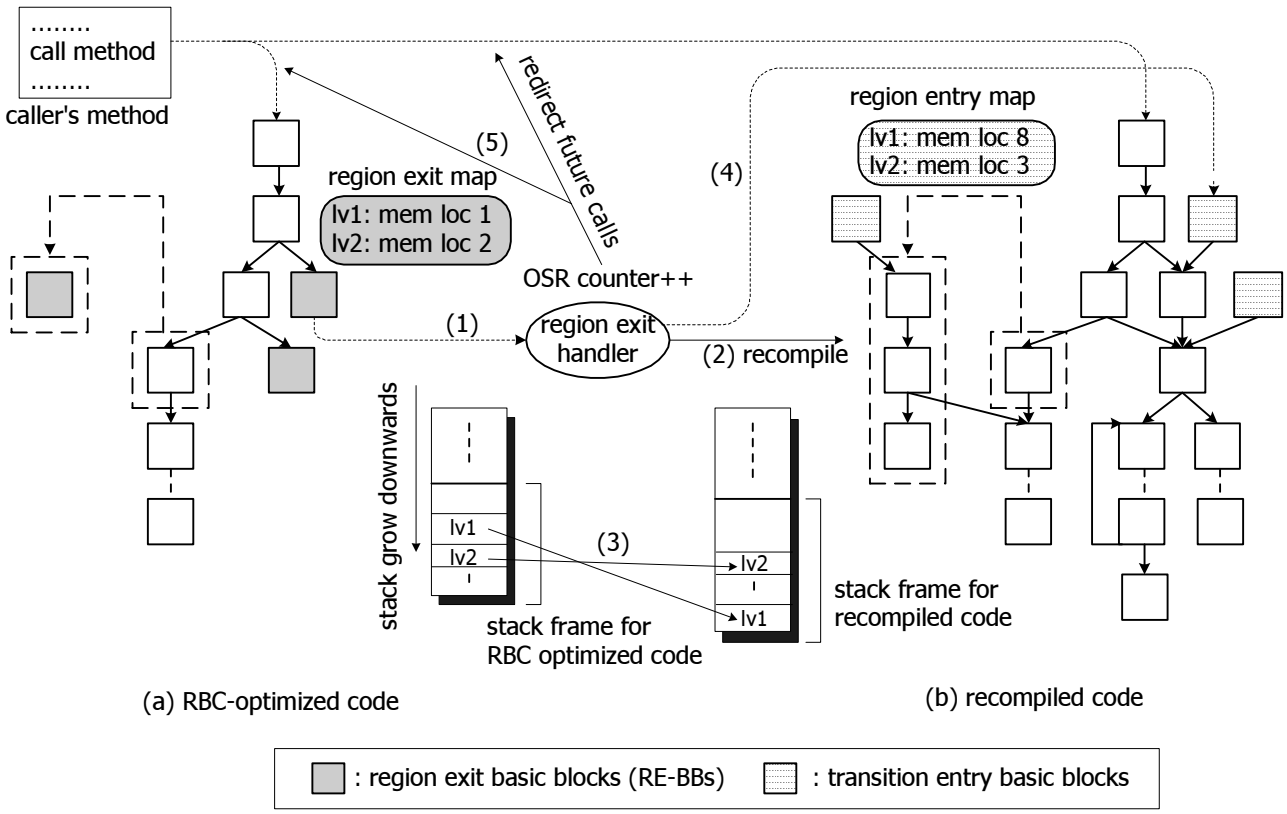


Figure 5.7: An example of transitions from RBC-optimized code to recompiled code. The numbers in parentheses indicate the steps (described in detail in the text) to be followed when a region exit event occurs.

Next, the region exit handler performs OSR to dynamically reconstruct the frame using the maps of both the source and the destination points (Step 3). All registers holding live variables are first spilled out into the stack frame. The handler then sets up new callee save registers, reorders the live local and stack variables, and adjusts the size of the frame, all based on the information contained in the maps. One convenient optimization here is that when the source and destination stack frames turn out to be the same shape (i.e. the corresponding map information is the same) in the first OSR event for each region exit point, then we can patch the instruction at the region exit point with an unconditional jump instruction directly into the corresponding entry point, so that we can skip the expensive OSR operation from the next time. After performing the OSR, the region exit handler transfers the control to the instructions at the region entry point

because we may have aggressively inlined methods using partial inlining for producing RBC-optimized code. We have not actually encountered this problem, but we can work around this situation in several ways (though we have not implemented them), such as 1) including only the region exit point for the current transition, 2) not applying inlining to all methods but dividing them into several units for recompilation, or 3) reverting back to the interpreter.

(Step 4).

The final step (Step 5) is to determine whether or not we should switch from the RBC-optimized version to the new recompiled version for the future invocations of this method. Since our region selection is inevitably speculative (based on static heuristics and incomplete dynamic profiles), it is possible that the assumptions for RBC optimization at compile time fail and control frequently exits from certain region boundary points. However, if the frequency is small enough, we want the original RBC version to still serve for future invocations since it is likely to be better optimized. Thus, we provide a counter for each RBC version to count the number of actual OSR events that occurred from the given method. Initially the recompiled version is used for the region exit transitions only, not for future invocations. If the OSR counter exceeds a certain threshold, we conclude that our speculative region selection did not work well for this method, and redirect the control of all future invocations into the recompiled version.

5.3.5 Region Optimizations

All of the analyses and optimizations that follow the region formation phase can proceed normally. Since there are no longer any merge points from the rare code to the non-rare code, the dataflow optimizations should be able to work more effectively for program transformations. The operands in the special instruction `OPC_RECOMPILE` provided in the RE-BB are all live variables at each region exit point, and they work as anchors to preserve the necessary variables through any optimizations. These operands may be renamed to other variables or replaced by constants during the optimization phases such as constant propagation and copy propagation. The RE-BB serves as a placeholder for any optimizations that require generating compensation code for an exiting path.

Thus most of the existing dataflow optimizations can proceed effectively and safely without any special treatment for the selected region, but there are some special optimizations that can take advantage of the region by making them aware of the region exit points. Examples include our implementations of partial dead code elimination and partial escape analysis, as described in [114].

Partial dead code elimination

We have to keep all live variables (both local and stack) in the bytecode-level code at each RE-BB to be able to restore the state correctly when a region exit occurs at runtime. This means the live range of some of those variables becomes larger than in the FBC approach. For example, in the example program shown in Figure 5.8(a), the second reference to variables a and b for defining y will be eliminated by applying common subexpression elimination in the FBC, and thus the live range for a and b will be terminated at the statement in the first block. In the RBC shown in Figure 5.8(b), however, these variables have to be passed on region exit because they are used in the bytecode-level code. This live range problem in the RBC will be expressed in the final code as increased register

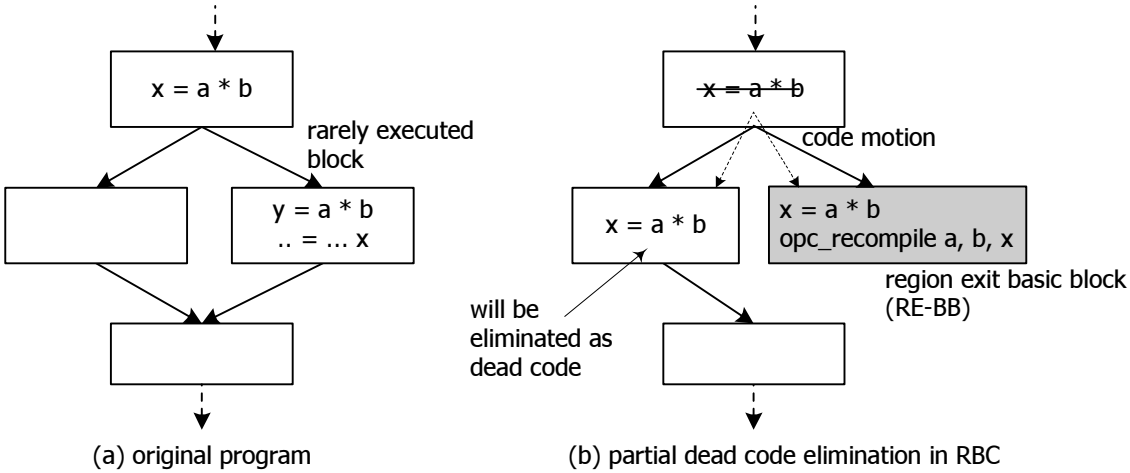


Figure 5.8: An example of partial dead code elimination. The computations, whose defined variables are included in the live set from the RE-BBs but not in the set from the other non-rare path, are moved and copied into both the RE-BB and the non-rare path. The copy in the non-rare path is then eliminated in the following dead code elimination phase.

pressure, extra instructions to spill into memory in the non-rare paths, and larger frame sizes.

Partial dead code elimination [74] can partly alleviate this negative effect of the RBC by eliminating the computations on non-rare paths. Our implementation of partial dead code elimination is a simple code motion followed by dead code elimination. We maintain two sets of live variables, one from the RE-BBs and the other from the non-rare paths. Using a standard code motion algorithm, we move the computations whose defined variables are included in the set from the RE-BBs but not included in the other set. The computations are copied once into both the appropriate RE-BB and the non-rare path in the other branch direction, but the copy in the non-rare path can then be eliminated in the following dead code elimination phase. In the example of Figure 5.8(b), the computations with the variables a and b no longer exist along the non-rare paths, differing from the FBC case, where they must be retained.

Partial escape analysis

Escape analysis as applied to stack object allocation, scalar replacement, and synchronization elimination is very effective for improving performance. However, quite often this optimization suffers from the fact that objects escape from only rarely executed code, especially from the backup path of a devirtualized method call, as shown in Figure 5.9(a). Thus its effectiveness with the FBC approach has been limited in practice.

In the region-based compilation, we can analyze more objects as non-escaping by focusing only on the non-rare paths in the target code. To do this, the escape analysis simply ignores region exit points. Any optimization based on this analysis is legal as long

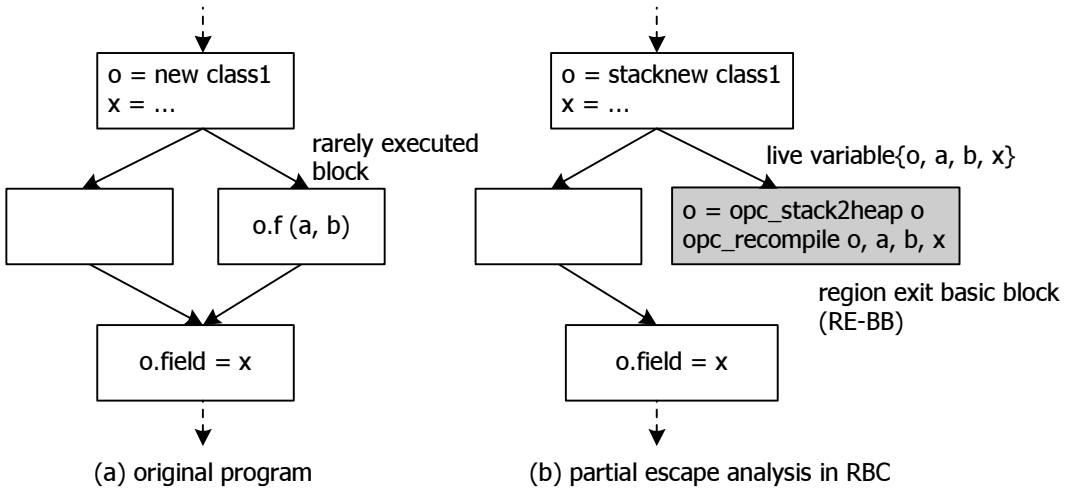


Figure 5.9: An example of partial escape analysis. In the original program (a), the object o is escaping and cannot be optimized. In (b), we ignore the RE-BB for the analysis and the object allocation is optimized (stack allocated). Compensation code is generated to allocate the object in heap space on the region-exiting path.

as the execution stays within the selected region at runtime. Thus, the object allocation in the example program can be optimized (stack allocated) along the non-rare paths as shown in Figure 5.9(b).

We need to generate compensation code for this optimization on region-exiting paths. In the final stage of the analysis, we check each of the objects identified as stack allocateable or as replaced by scalar variables to see whether the object is one of the live variables at each region exit point (as listed in the operands of the `OPC_RECOMPILE` instruction). If the object is live, we insert a special `OPC_STACK2HEAP` instruction in the RE-BB immediately before the `OPC_RECOMPILE` as shown in Figure 5.9(b). When executed, this special instruction calls a runtime module that does the following:

- allocates the object on the heap and does its initialization,
- copies the object content from stack to heap or copies the scalar-replaced variables to heap,
- updates the location of the object reference, if the object is referred to among the other objects, and
- synchronizes the allocated object, if that operation has been eliminated in the non-rare paths but is necessary at the exit point.

Our escape analysis is based on the algorithm described in [117]. This is a compositional analysis designed to analyze each method independently and to produce a parameterized analysis summary result that can be used at all of the call sites that may invoke the method. Without the summary result, the analysis has to treat all the arguments

Table 5.2: Additional benchmark used for RBC evaluation: Java Grande Section 3

Program	Description
Euler	Computational fluid dynamics
Molydyn (md)	Molecular dynamics simulation
Montecarlo (mc)	A financial simulation using Monte Carlo method
Raytracer (ray)	3D ray tracer
Search	Alpha-beta pruned search

as escaping at the given call sites. Hence the analysis result can be more precise and complete as more of the invoked methods are analyzed.

However, if an argument of object type is included in the list of live variables at any region exit point within the method, we suppress the generation of the summary results. This is because the escape analysis is based on the optimistic assumption of ignoring rare regions. We may create an optimistic summary result in which an object given as an argument is non-escaping within the method, although the object can actually escape from a region exit point. This poses a problem, since the analysis of its caller methods using such optimistic summary results is also optimistic. When the execution exits from a region boundary in a method, we would have to recompile not only the current method but all of the caller methods that used the optimistic summary results directly or indirectly.

5.4 Experimental Evaluation

This section presents some experimental results showing the effectiveness of the RBC in our dynamic compilation system.

5.4.1 Methodology

All of the measurement conditions are basically the same as those described in Section 3.4.1. We used the same applications and benchmarks shown in Table 3.1 and Table 3.2, but in order to evaluate for a broader set of Java programs, we added Java Grande Section 3 from [68] as described in Table 5.2. We ran each test of the Java Grande benchmark separately with the "Size B" problem (large data set), and with the initial and maximum heap sizes of 512 MB. Unlike SPECjvm98, this benchmark includes the JIT compilation time in the execution time.

The number of samples to be collected in the instrumentation-based profiler was set to 10,000. The threshold for the number of OSRs to redirect the future method invocations to recompiled code was set to 10. This threshold was determined to be at most about 1% of the method invocation count.

As in previous chapters, we compared both performance and compilation overhead

(measured in compilation time, compiled code size, and compile time peak memory use) of our RBC approach against the conventional FBC approach. We used the dynamic optimization framework described in Chapter 3, but for the Java Grande and the start-up applications we ran them with the configuration of MMI and level-3 compilation, instead of the multi-level compilation scheme, in order to apply RBC to more of the methods in the program execution. That is, the upgrade recompilation was not enabled, and thus the profile information on level-1 compiled code is not available. We applied RBC using only the static heuristics for these benchmarks.

In our current implementation of partial inlining, we perform region selection for an inlining target method and estimate the cost for the reduced target code, but temporarily inline the entire body of the method. The rare portion of the code identified in the region selection is removed immediately after this inlining process. This should not greatly affect the compilation time or the code size, but the compile-time peak work memory usage will be larger than it should be with an optimized implementation.

5.4.2 Statistics

Table 5.3 and Table 5.4 show the statistics when running the benchmarks with RBC. The second to fourth rows are execution and compilation statistics, showing the total number of methods executed for each test, the total combined number for methods compiled at level-2 or level-3, and the total number of RBC-optimized methods, respectively.[4] These numbers include methods from the required library routines as well as from the benchmark or the application itself. The level-2 and level-3 compiled methods in the third row are the targets for RBC optimizations, out of which the fourth row shows the actual number of methods where rare regions were identified and where the RBC optimizations were performed.[5] Both the third and fourth rows include the percentage of the level-2/3-compiled methods over the total number of methods executed, and the percentages of the RBC-optimized methods compared to the number of level-2/3-methods, respectively. Each compilation unit forms a single region in our RBC approach, and thus the number of RBC-optimized methods corresponds to the number of regions formed.

The next five rows show the breakdown of the region exit points classified with each rare type, based on the profile results or heuristics, for the RBC-optimized methods. In other words, these rows show how the rare regions were identified and removed in the region selection process for the methods shown in the fourth row. An RBC-optimized method generally has several region exit points, and these numbers give a cumulative picture of how the methods were RBC optimized. The last three rows show the number of recompiled methods due to region exits (with the percentages of the RBC-optimized

[4]We count only the methods directly driven to compilation, and do not include the methods inlined into the caller methods.

[5]The numbers in the fourth row represent subsets of the numbers in the third row. All of the third row methods were selected by the sampling profiler, but only some of those methods (counted in the fourth row) were found to have rare regions for RBC optimization.

Table 5.3: Statistics of the region-based compilation for SPEC benchmark runs. The top three rows are execution and compilation statistics, the middle five rows show how rare regions are identified in the RBC-optimized methods, and the bottom three show the runtime behavior for recompilations and OSRs.

Benchmarks	mtrt	jess	comp	db	mpeg	jack	javac	jbb
Total executed methods	455	734	325	321	495	561	1085	2818
L-2/3 compiled methods	33	41	7	7	56	81	157	166
(% to total methods)	7.3	5.6	2.2	2.2	11.3	14.4	14.5	5.9
RBC optimized methods	28	22	1	6	19	58	101	123
(% to L-2/3 methods)	84.8	53.7	14.3	85.7	33.9	71.6	64.3	74.1
Profile-based rare path	3	12	1	7	2	7	14	35
Devirtualized backup path	483	36	0	19	16	136	644	1244
Exception throwing path	19	22	0	7	33	114	169	171
Exception handler block	2	3	0	6	0	69	54	96
Uninitialized code path	0	0	0	0	0	1	0	4
Recompiled methods	0	0	0	0	1	9	4	1
(% to RBC opt. methods)	0.0	0.0	0.0	0.0	5.3	15.5	4.0	0.8
Region exited points	0	0	0	0	1	10	4	1
(OSR skipped points)	(-)	(-)	(-)	(-)	(0)	(4)	(1)	(0)
Number of OSR events	0	0	0	0	6	52	34	10

Table 5.4: Statistics of the region-based compilation for Java Grande and other benchmark runs. These benchmarks were run with the configuration of MMI and level-3 compilation, and thus the profile information for rare region identification was not available.

Benchmarks	euler	md	mc	ray	search	taro	j2d	swing	was
Total executed methods	616	726	745	670	636	7398	7365	7298	17204
L-2/3 compiled methods	49	30	105	39	28	515	1157	870	1451
(% to total methods)	8.0	4.1	14.1	5.8	4.4	7.0	15.7	11.9	8.4
RBC optimized methods	23	12	46	12	11	231	390	335	610
(% to L-2/3 methods)	46.9	40.0	43.8	30.8	39.3	44.9	33.7	38.5	42.0
Profile-based rare path	0	0	0	0	0	0	0	0	0
Devirtualized backup path	10	2	52	2	2	457	836	706	2083
Exception throwing path	48	23	98	26	23	522	507	496	725
Exception handler block	3	2	10	2	2	258	241	231	324
Uninitialized code path	0	0	0	0	0	0	2	0	34
Recompiled methods	1	0	0	0	0	12	9	13	24
(% to RBC opt. methods)	4.3	0.0	0.0	0.0	0.0	5.2	2.3	3.9	3.9
Region exited points	1	0	0	0	0	12	9	12	24
(OSR skipped points)	(0)	(-)	(-)	(-)	(-)	(3)	(4)	(6)	(10)
Number of OSR events	11	0	0	0	0	63	59	73	164

methods), the number of region exit points where control actually escaped at runtime (including the number of points OSR skipped due to the stack frame layout being the same), and the total number of OSR events that occurred, respectively.

Except for _201_compress, quite a few methods were optimized with RBC. The number was roughly in the range of 40% to 80% of the level-2 and level-3 compiled methods, showing that many benchmarks and applications do contain rare regions even in hot methods and we can use some optimizations for these methods. A backup path for a devirtualized method invocation is the most common kind of rare region identified, followed by an exception-throwing path. The majority of rare regions were identified on the basis of the static heuristics, and the numbers of profile-identified rare paths were relatively small. This is because profile results are not always available for RBC optimization target methods as described in Section 3.3.3, and because we remain conservative when working from the profile results, considering the fact that our profiles are based on samples collected for short intervals of program execution.

The numbers of recompiled methods and OSRs is relatively large in _228_jack, _213_javac, and the applications shown in the last four columns in Table 5.4, in comparison to the other benchmarks. The _228_jack and _213_javac benchmarks are known to frequently raise exceptions during program execution, and control exited from certain exception-throwing paths we had presumed to be rare and which we had removed from the compilation target. For the startup applications, many of the speculative region selections turned out to be invalid because of the dynamic class loading activity during the startup runs. This suggests that the RBC optimization should be performed after the program enters a steady state with a stable working set. However, the number of recompiled methods for these benchmarks was still kept within a reasonable level of the percentage of the total number of the RBC-optimized methods, up to 15% for _228_jack and around 2% to 5% for some others. The number of OSRs is well constrained, owing to the mechanism of dynamic OSR counting and the control of future invocations for the recompiled methods based on those counts. The optimization of skipping the OSR operation was applied to about one-third to a half of the region exit points where the control actually escaped. As described in Section 5.3.4, this optimization can be performed when the source and destination stack frames turn out to be the same shape. The region exits through the direct jump after this optimization was performed are not included in the number of OSR events. The other benchmarks show none or very small numbers of recompilations and OSRs.

5.4.3 Performance

Figure 5.10(a) shows the performance improvements with RBC over the conventional FBC approach. We took the best time from 10 repetitive autoruns for each test in SPECjvm98, and the best throughput from a series of successive executions from 1 to 8 warehouses for SPECjbb2000. We took the self-reported score for the Java Grande benchmarks, and

measured the time as described in the previous section for the rest of the benchmarks. The figure shows that RBC performs significantly better than FBC for some benchmarks, with a 24% improvement for `_227_mtrt`, and 4% to 8% improvements for some others.

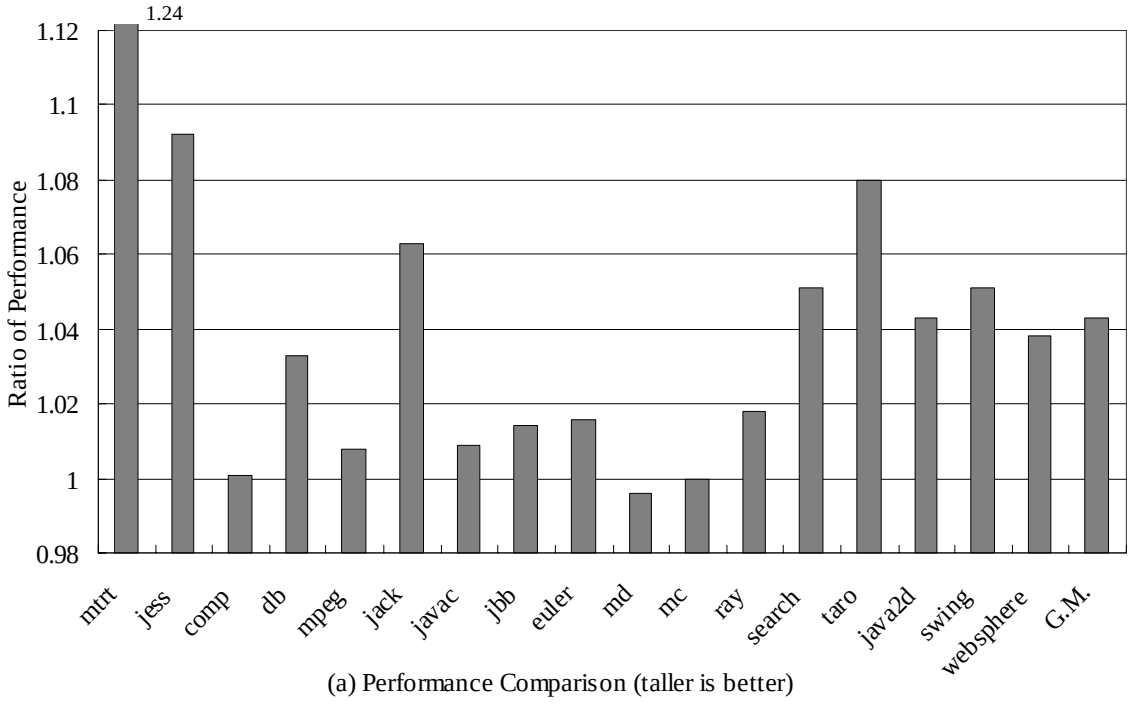
The majority of the performance gain for `_227_mtrt` comes from the elimination of backup paths for devirtualized method calls and their exploitation by the partial escape analysis. `Mtrt` has many virtual method invocations, most of which are devirtualized and the target methods are inlined. These methods are never overridden and retaining the backup paths for dynamic class loading does nothing but prevent the escape analysis from working well. However, simply removing the backup paths as a part of region selection does not solve this problem. Without partial escape analysis, the performance actually degrades, because the escape analysis now has to treat region exit points as globally escaping points for all live variables, not only for the variables passed as arguments in the original virtual invocation call sites. As a result, more objects will be analyzed as escaping, and the number of captured objects that are allocated on stack or replaced with scalar variables will be decreased.

The partial inlining contributes to a significant improvement for some other SPEC benchmarks, especially for `_202_jess` and `_228_jack`. For example, the large performance improvement in `_228_jack` results from the additional inlining performed in the partial inlining process which then allows the escape analysis to recognize some frequently allocated objects as captured and makes those objects stack allocated in one of the core methods of the benchmark. This is a good example of the indirect effect of partial inlining.

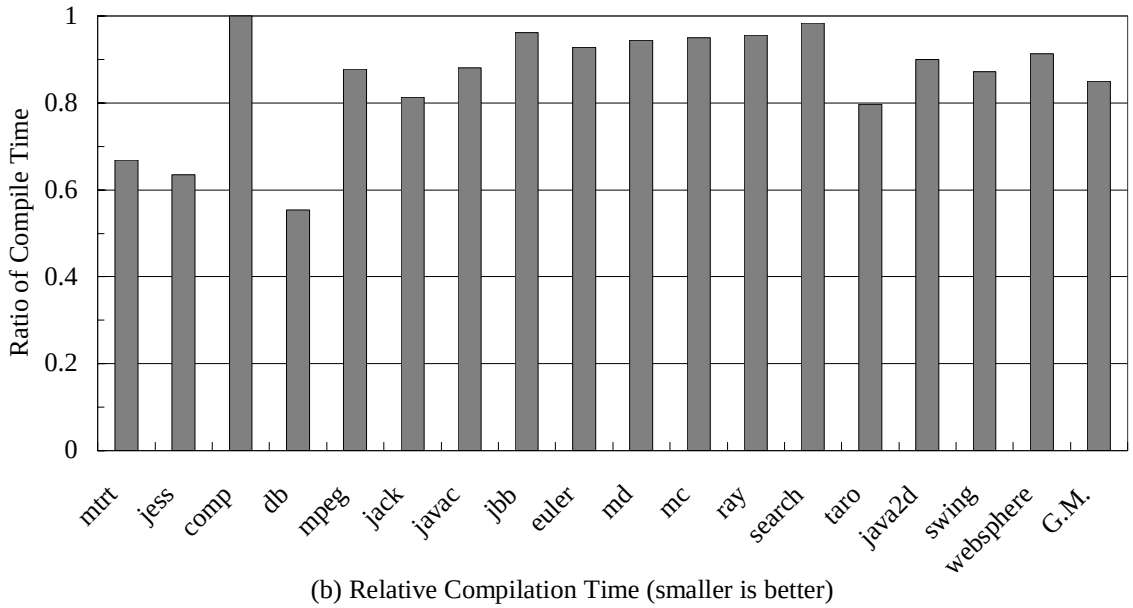
For the Java Grande benchmarks, the performance improvement with the RBC approach is not very significant, except for `Search`. These benchmarks were not identified as having sufficiently large rare regions by the region selection process, as shown in Section 5.4.2. This makes the partial inlining ineffective for producing performance differences.

The performance of the application startup consistently improves in the 4% to 8% range. In contrast to the Java Grande benchmarks, we found a considerable number of rare regions were identified in these applications to use the RBC optimizations. On the other hand, we also had relatively higher numbers of recompilations and OSR events due to region exit, as shown in Section 5.4.2. Since these events are during the application startup, it is expected that we will have a higher failure rate of speculative optimizations than in the steady state. Nevertheless, we observed consistent performance improvements.

To assess the time to reach steady state performance, we rely on the underlying recompilation system to identify the performance-critical hot methods and to promote them to higher optimization levels, where we can then apply RBC optimization. Thus the time to reach steady state performance is basically the same with and without RBC optimization, except that some RBC-optimized methods are recompiled upon region exit.

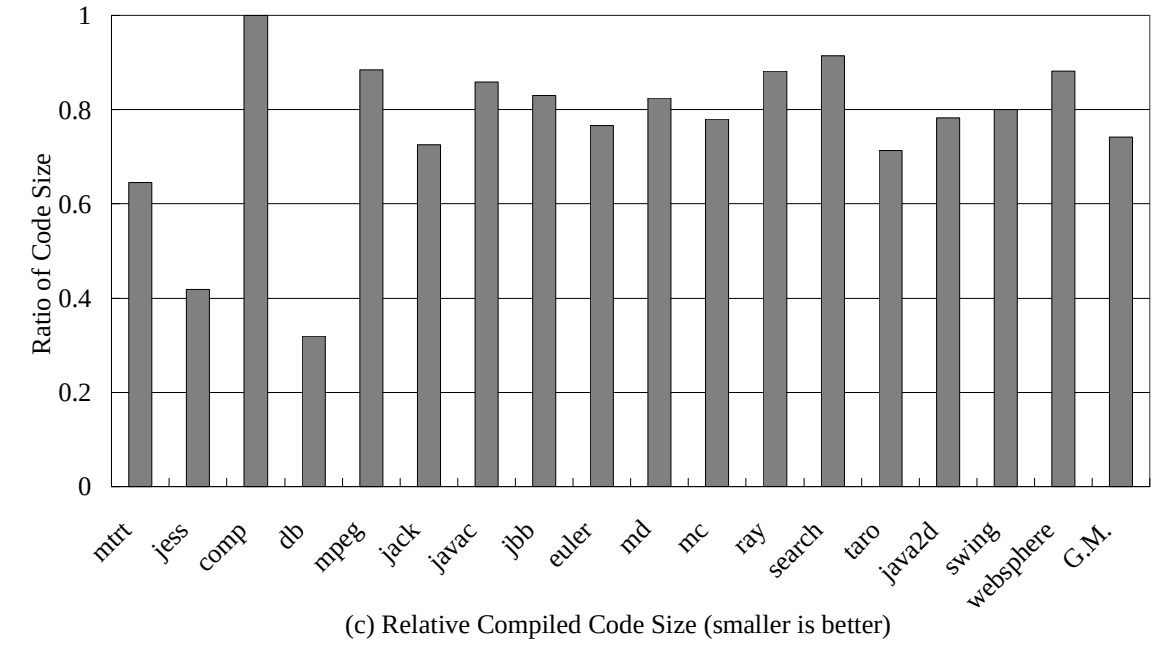


(a) Performance Comparison (taller is better)

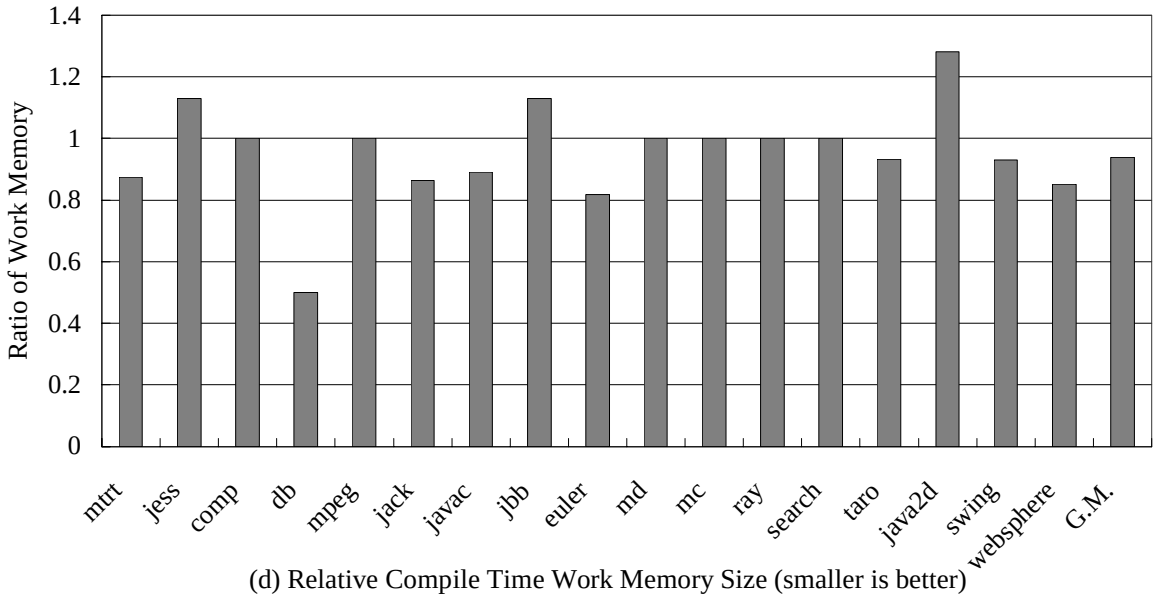


(b) Relative Compilation Time (smaller is better)

Figure 5.10: Comparison on performance improvement and compilation overhead (1 of 2). Each bar indicates the relative numbers of RBC to FBC approach. The taller bar shows better performance in (a), while the shorter bar means smaller overhead in (b).



(c) Relative Compiled Code Size (smaller is better)



(d) Relative Compile Time Work Memory Size (smaller is better)

Figure 5.10: Comparison on performance improvement and compilation overhead (2 of 2). Each bar indicates the relative numbers of RBC to FBC approach. The shorter bar means smaller overhead in (c) and (d).

5.4.4 Compilation Overhead

Figure 5.10, parts (b) to (d) shows the ratios of compilation overhead for RBC over FBC using three metrics: the compilation time, the compiled code size, and the compilation time peak work memory usage. Smaller bars mean better scores in these figures. We measured only level-2 and level-3 overhead, since level-1 is shared between the RBC and FBC configurations. All these figures for RBC include additional overhead that results from the recompilations due to region exits occurring at runtime. The peak memory usage is the maximum amount of memory allocated for compiling methods, including the space for constructing call trees for method inlining. Since our memory management routine allocates and frees memory in 1 Mbyte blocks, this large granularity masks the minor differences in memory usage and causes the results of the (d) figure to form clusters.

The reduction in compilation time is up to 40% for `db`, and on average slightly below 20%. Overall, the SPEC benchmarks show relatively better reduction ratios than the other benchmarks. All of the Java Grande benchmarks show very small reductions (less than 10%) due to the small sizes of the rare regions identified in the region selection process. The reductions for the startup applications are between 10% and 20%.

The reduction of compiled code size is relatively larger but shows tendencies similar to those of the compilation time. In particular, the reduction exceeds 60% for `_202_jess` and `_209_db`, and is between 20% and 40% for many other benchmarks. Again, the SPEC benchmarks show relatively better reduction ratios than the other benchmarks. Besides the compiled code space shown in the figure, RBC actually requires another runtime memory area, the map for each region exit point, which is additional overhead specific to RBC. The size of the map depends on the number of live local and stack variables for each region exit point, but the map typically requires around 50 to 70 bytes per exit point. Even if we take this map space into consideration, the total size is still well under the FBC code size for most of the benchmarks.

The reduction in compilation time work memory size is less dramatic, compared to the other metrics. Although we can observe significant reductions for some benchmarks (50% for `db`, and around 20% for `Euler` and some others), the overall reduction is less than 10% and there is even an increase for a few benchmarks. This problem of relatively higher overhead is caused by our current implementation of partial inlining, as described in Section 3.4.1. That is, we temporarily inline the entire body of the target methods before removing the rare portions of the code. With an optimized implementation of partial inlining, this problem will disappear and the reduction ratio of the work memory usage overhead should be almost similar to the ratios for the two other overhead metrics.

Overall, the RBC approach shows significant advantages over the conventional FBC approach for all three metrics of the overhead in most of the benchmarks. The reductions are between 10% and 30% on average, depending on the benchmark and the metric. This significant reduction is not surprising, since the rare regions are identified and removed from the target code, and all of the optimizations and code generation are done for this smaller amount of target code. The cost of recompilations due to runtime region

Table 5.5: Percent of the compilation time to the execution time, and percent of the code size and the peak work memory size to the maximum live object size.

Benchmarks	Compilation time		Compiled code size		Work memory size	
	FBC	RBC	FBC	RBC	FBC	RBC
_227_mtrt	41.3	33.9	0.9	0.6	38.8	33.9
_202_jess	16.7	11.6	3.2	1.3	100.3	113.8
_201_compress	1.1	1.1	0.1	0.1	29.8	29.8
_209_db	0.3	0.2	0.2	0.1	18.9	9.4
_222_mpegaudio	12.1	10.7	4.9	4.3	395.9	395.9
_228_jack	24.6	21.1	12.4	8.9	265.4	228.9
_213_javac	46.2	41.0	2.3	1.9	62.9	55.9
jbb2000	-	-	0.4	0.3	10.0	11.3
euler	1.4	1.3	0.3	0.2	24.9	20.4
moldyn	0.1	0.1	2.8	2.3	249.1	249.1
montecarlo	0.8	0.7	0.1	0.1	0.6	0.6
raytracer	0.2	0.2	1.5	1.2	119.7	119.7
search	0.7	0.7	0.5	0.4	140.3	140.3
ichitaroark	43.3	37.4	10.3	7.3	159.7	148.9
java2d	25.8	24.2	11.9	9.3	112.6	145.7
swingset	47.7	43.9	6.8	5.4	79.3	73.8
websphere	28.9	27.4	3.8	3.3	31.4	26.7

exits for some RBC-optimized methods is effectively offset by the benefits of the reduced compilation overhead with the RBC approach.

Table 5.5 shows the ratios of the compilation time to the execution time, and the ratios of the compiled code size and the compile time work memory size to the maximum live object size in the Java heap for each benchmark. For the ratios of the compilation time for SPECjvm98 benchmarks, the numbers show how much of the compilation time has been spent in level-2 and level-3 over 10 repetitive runs (in autorun mode) for obtaining the best execution performance. SPECjbb2000 is excluded because it does not give us the execution time. For the other benchmarks, the numbers show the compilation time ratio actually occupied within the duration of a single execution.

The ratio varies from one benchmark to another. For example, _201_compress, _209_db, and the JavaGrande benchmarks have very low ratios, meaning that the working set of these benchmarks is very small. On the other hand, _213_javac and _228_jack have flat profiles and thus higher ratios due to many methods being compiled at level-2 and level-3. The startup applications spend almost a quarter to a half of their execution time in compiling methods with level-3. The high ratio of _227_mtrt is due to the very short execution time. When comparing the FBC and RBC configurations, RBC consistently has smaller compilation ratios over FBC, despite the significantly improved execution

times for some benchmarks.

For the ratios of the compiled code size and the compile-time work memory size, we used the maximum live object size in the heap as an indicator of runtime program behavior for memory consumption. We measured the live object size after each round of garbage collection by using the -verbosegc command line option. While the size was measured reflecting the granularity of the GC events and thus is not exact, the ratios shown in the table indicate approximately how much of the space was consumed for both compiled code and compiler work memory in terms of the live objects in the heap for each benchmark. Again, the ratio varies widely between benchmarks due to the size of the live object data. For example, `moldyn`, `_222_mpegaudio`, and `_228_jack` are very small (about 0.8 MB to 1.6 MB), while `montecarlo` and `jbb2000` need large amounts of memory for their live objects (about 515 MB and 90 MB, respectively) throughout those programs' execution. From this table, the reduction of the compiled code size and the compile-time work memory use with RBC approach is up to 3% and 10% of the maximum of the live objects allocated in the heap, respectively.

5.4.5 Discussion

Overall, this study shows the advantages of the RBC approach in both performance and compilation overhead over the traditional FBC approach. It shows the potential for significantly reducing the compilation overhead, measured in time, work memory, and code size, and for improving performance. In a dynamic compilation environment, we have to be very careful in performing any optimizations that have significant impact on compilation overhead, so RBC is a promising strategy for dynamic compilers.

We did not use *exception directed optimization* (EDO) [85, 86] in our current implementation. This is a technique to monitor frequently raised exception paths and to optimize them by inlining and converting exception throwing instructions to simple jump instructions into their corresponding handlers. This is complementary to our RBC approach, since EDO effectively eliminates frequently excepting instructions from the current control flow, before those instructions are treated as rare in our static heuristics for region selection. By applying EDO, we could ensure that the remaining exception-throwing instructions are truly in rarely executed paths. As shown in Section 5.4.2, many of the recompilation and OSR events occurring in our current implementation are due to region exits from exception paths, so this optimization is expected to decrease the probability of region exits without reducing the effectiveness of the RBC approach.

As described in Section 5.2.3, it would be useful to support several options for OSR and employ them depending on the characteristics of each region exit point regarding how the rare paths were eliminated. For example, the current strategy of recompilation with the same optimization level works fine for the exit point of a devirtualized call site backup path, because once the control escapes from one of those exit points due to the loading of a dynamic class, it will most likely escape from this exit point in subsequent

executions. On the other hand, when a region exit occurs from an exception-throwing path, it may be sufficient to fall back to the interpreter, since exception-throwing paths (especially after EDO) need not be optimized, considering the inherently high overhead of runtime exception handling. This mechanism of selective region exit strategies may allow more aggressive rare path elimination than we currently use.

We can still use the dynamic counts of the OSRs to identify from which region exit points the control is frequently escaping, and thereby drive recompilation, rather than waiting for the promotion to be performed by the underlying recompilation system. However, we need to check whether or not the use of absolute counts is appropriate here. Any sufficiently long-running program will eventually cause the counter to hit the threshold, assuming the event frequency is non-zero, and we may loose the benefits of RBC optimization. This will be a problem in production use. We can explore counter decay and other techniques to address such problems.

We can explore further opportunities for identifying rarely executed code to increase the effectiveness of the RBC approach. For example, loop versioning [96, 81] is an optimization technique for hoisting the array-bound exception-checking code for an individual array access outside a loop by providing two copies of the loop: the unoptimized loop, where exception checking code is retained as in the original loop, and the optimized loop, where all array exception-checking code is eliminated. Guard code is provided to examine the whole range of the index against the bound of the arrays accessed within the loop, and depending on the result of this test, either the optimized or unoptimized loop is selected at runtime. This is an effective optimization, but entails a significant code size increase. It is expected that the guard code will succeed in most of the cases and thus the unoptimized loop will rarely or never be executed. We could have a significant code size reduction if we can integrate this opportunity into the RBC strategy. We could even use on-the-fly generation of an unoptimized loop when the test in the entry guard code fails.

Method splitting, also called procedure splitting [88], is a technique that can complement the RBC strategy. This is to place relatively infrequent code away from common code, typically in a separate page, in order to improve instruction cache locality. Our region selection process does not identify these relatively infrequent but still executed portions of the code, since over-aggressive region selection will lead to too many recompilations and can degrade performance. In other word, the selected region still contains some relatively infrequent code. We could increase the code locality even more by integrating the method splitting technique into our framework.

5.5 Summary

In this chapter, we have described the design and implementation of a region-based compilation technique in our Java JIT compiler. We presented our design decisions for region formation, partial inlining, and region exit handling, and described the algorithm in detail for the intra-method region selection and its integration in the inlining process. We

implemented this RBC framework in our dynamic optimization framework described in Chapter 3, and evaluated the technique using several industry standard benchmarks. The experimental results show the potential to achieve better performance and improved compilation overhead in comparison to the traditional FBC approach.

Chapter 6

Related Work

This chapter describes some previous work in the area of dynamic optimizations. There are several research topics related to the work presented in this dissertation, including dynamic and adaptive optimization frameworks, low-overhead profiling techniques, profile-directed method inlining, dynamic code specialization, and compilation techniques exploiting rarely executed regions. To help clarify the differences between previous work and this dissertation, this chapter is organized as follows.

Section 6.1 describes several dynamic optimization systems currently available for Java, all of which use some form of automatic profile-driven adaptive optimizations. These systems are the work most closely related to this dissertation. We describe several important points that distinguish our work from these earlier approaches. Section 6.2 describes several dynamic optimization systems in other programming languages. In particular, the SELF implementation was a pioneering work in the area of dynamic optimization, introducing many important concepts such as customization, type feedback, adaptive optimization, deoptimization, and splitting. These techniques are still used, with various improvements, in many of the current state-of-the-art Java virtual machines. Previous work in this category also includes dynamic binary optimizers, where online profile information is used to improve the quality of the code produced by the optimizing compiler.

Section 6.3 describes previous work that used offline profile information, collected in a training run, to drive optimization. These systems focus on some of the problems addressed in this dissertation (using profile information to improve code quality), but they do not address the issues surrounding online profiling and dynamic optimization. The section also includes some previous work that studied a single topic or issue related to this dissertation.

6.1 Dynamic Optimization System for Java

This section describes several major dynamic compilation systems currently available for Java, either as product JVMs or as research JVMs, all with some form of automatic,

profile-driven adaptive optimization. We evaluate the characteristics of these systems for the following three criteria:

- Whether the system uses a compile-only approach or uses an interpreter to allow for a mixed execution environment with interpreted and compiled code,
- How the system monitors the application program to promote methods from a lower optimization level to a higher level, and
- What profile information the system collects online to be exploited by the higher optimization levels, and how it does so.

These features characterize dynamic compilation systems in general. That is, how the system addresses these three aspects can greatly affect some important features of the dynamic compilation systems, such as the total system performance during both the application startup time and the steady state, the underlying bottom-line overhead for program monitoring, the flexibility and adaptability of the system against changes in the applications' dynamic behaviors, and the extensibility of the system for further reoptimization.

6.1.1 Open Runtime Platform

The Open Runtime Platform (ORP) [37, 36] is a well known research virtual machine released as open source. It uses a compile-only approach, and implements dynamic optimization with two execution modes by providing two different compilers: a fast code generator [2] and an optimizing compiler [38, 1]. While the fast code generator produces code directly from bytecode with only limited and lightweight optimizations, the dynamic compiler uses an IL to apply aggressive optimizations, such as method inlining, global dataflow-based optimizations, and loop transformations. The dynamic compiler also uses the profile information collected in the first execution mode to guide optimization decisions such as the inlining policy, where to apply expensive optimizations, and the code layout in the final code emission.

As a way of triggering recompilation, the system provides two mechanisms using a counter associated with each method. The first mechanism inserts code to test the values of the counters. As soon as the counters reach a threshold, the code jumps immediately to the recompilation routine to optimize the method used by the executing thread. The second mechanism uses a separate thread to scan the counters, looking for recompilation candidates in order to optimize them in parallel with the executing application. In both of these mechanisms, however, the system inserts counter-updating instructions in the first level compiled code for both method entry points and for loop-backward branches.

The disadvantage of this type of profiling and method promotion system is the performance overhead. Although the instrumentation code is not generated in the second-level

compiled code, it can incur a continuous bottom-line performance penalty. The target code has to be recompiled if we want to remove the overhead of these instructions. This problem may not be apparent for applications with spiky profiles, where there are only a few very hot methods, and compiling those methods with the highest optimization level is sufficient to obtain the best possible performance. However, flat profile applications have a large number of methods that are almost equally important, and thus a significant portion of those methods can remain unoptimized. It is therefore possible that the performance will be noticeably degraded due to the instrumentation code left in that unoptimized code. In fact, in our prototype implementation of this type of profiling system, we measured an approximately 15% performance penalty for the SPECjvm98 `javac` test.

Also, the fact that the recompiled code is not instrumented implies further reoptimization is not possible in this system. The system currently does not have dynamic profile collection or any optimization feature using online profile information, and the system is considered not extensible for adding profile-based optimizations either, due to the lack of any continuous program monitoring capability. In contrast, our system uses a low-overhead profiling system for continuous sampling throughout the entire program execution, and thus allows for further reoptimizations, such as code specialization.

6.1.2 Jikes RVM

The Jikes RVM (previously called Jalapeño) [24, 5] is another research JVM implemented in Java itself, and uses a compile-only approach. The system is implemented with a multi-level recompilation framework using a baseline compiler and an optimizing compiler with three optimization levels. The recompilation is driven based on the estimated cost and benefit of compiling methods at each optimization level. The system shows good performance improvements in both startup and steady state regimes compared to other non-adaptive configurations or adaptive but single level recompilation configurations [10].

Their overall system architecture is quite similar to ours, but the major differences lie in its compile-only strategy and in how the profiling system works. The compile-only strategy can incur a significant overhead for the system. Although their baseline compiler was designed separately from the optimizing compiler to minimize the compilation overhead, and thus the additional compilation time may be negligible, the system can still result in a large memory footprint. The integration of an interpreter into the system may cause some extra engineering and implementation complexity, for example, in distinctively handling both interpreter frames and compiled code frames in the stack traversal, but this additional effort would certainly be rewarded. As shown in the measurements in Section 3.4.4, a large number of the dynamically executed methods are not frequently called nor computation intensive in many applications. A mixed mode interpreter can execute those infrequently called methods without imposing any cost in compile time or code size growth. This allows the recompilation system to be more flexible and aggressive in its reoptimization policy decisions.

Like our system, the Jikes RVM combines two separate profiling techniques. The system first uses a timer-based sampling mechanism to find hot methods, and then it employs a heavier instrumentation mechanism for the hot methods to collect more detailed information. The sampling profiler is based on polling at yield points (method prologues and loop back edges), taking advantage of the system's existing thread-switching mechanism for quasi-preemption. This sampling profiler continuously works to detect a program's hot methods during the course of the entire program execution. This continuous profiling throughout the application's lifetime is desirable for flexible and adaptive responses to the changes in an application's dynamic behavior.

The Jikes RVM was released under an open source license in 2003 [6], and it has been the most popular and widely used platform in the Java virtual machine research community. Many advanced features, ranging from low overhead instrumentation to a variety of adaptive optimization techniques to sophisticated garbage collection algorithms, have been implemented and studied using the platform. The rest of this subsection reviews some of the noteworthy examples.

Arnold and Ryder implemented a framework called *instrumentation sampling* [15, 9] to reduce the instrumentation overhead in an online system. This technique introduces a second version of the code, called checking code, to reduce the frequency of executing the instrumented code. This allows a variety of profiling techniques to be integrated in the framework. Since this is a sampling-based instrumentation technique, the profiling overhead can be kept very small, in comparison to a bursty profiling system like ours. The main concern, however, 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 code for value sampling at the method entry points, and thus it is more space efficient.

Arnold et al. [13] present a comparative study of static and profile-based heuristics for inlining with several limits on code expansion. In considering three inlining heuristics, based on a static call graph, a call graph with node weights, and a dynamic call graph with edge weights, they regard the selection of inlining candidates as a knapsack problem, and employ a greedy heuristic based on the benefit/cost ratio as a meta-algorithm for approximating the NP-hard problem. Their experiment, done with offline-based profiling and an ahead-of-time compilation framework, shows that a substantial (sometimes more than 50%) performance improvement can be obtained with the heuristics based on the dynamic call graph with edge weights over the static call graph, even with modest limits on code size expansion. This work became the basis for the online profile-directed method inlining.

The Jikes RVM employs two separate profiling techniques, a sampling profiler and an instrumenting profiler, in order to keep the overall profiling overhead low, but stack traversals are required by the sampling profiler for the online profile-directed method inlining [10]. This means the system periodically takes a statistical sample of its caller methods at the method prologue yield points by traversing the stack frame, and it maintains an

approximation of the dynamic call graph. A possible problem in this design is that there will be a continuous bottom-line overhead for the extra work of traversing the stack at the time of every sampling, and that profile information can be rather coarse-grained for use in reoptimization. In contrast, the lightweight sampling profiler in our system focuses only on detecting the hot methods, and relies on a more intrusive instrumentation profiler for collecting any information that is necessary for guiding optimizations.

Arnold et al. [14] continued to further improve the effectiveness of the profile-directed inlining by collecting dynamic execution frequencies of the control flow edges between basic blocks, using the *instrumentation sampling* framework, and letting this information be used for fine-tuning the inlining decisions. This call-edge frequency information was also applied to other online-feedback-directed optimizations, such as splitting, code positioning, and loop unrolling. By combining these four optimizations using profile information, they showed performance improvements ranging from 0.9% (`javac`) to 16.9% (`mtrt`), with a geometric mean of 4.3% for the SPECjvm98 benchmark.

Hazelwood and Grove [58] studied the use of adaptive, context-sensitive profile information for improving online method inlining. Their dynamic call graph is periodically maintained based on multi-level call stack sampling. For a level of context sensitivity, they attempted several adaptive schemes to find the ideal degree for each call site, since too much context sensitivity can degrade performance. They showed that the context sensitivity can make possible a significant number of reductions in both compile time and compiled code size, while keeping the performance impact very close to zero.

Fink and Qian [50] described a new, relatively compiler-independent mechanism for implementing OSR, and applied this technique to integrate the *deferred compilation* strategy in the Jikes RVM adaptive optimization system. Since they did not at that time implement optimizations that could take advantage of the deferred compilation, the performance improvement was small, but the compilation time and code size showed modest improvements.

6.1.3 HotSpot Server

The HotSpot server [87] is a JVM product implementing an adaptive optimization system, and uses both an interpreter and an optimizing compiler to support a mixed execution environment, as in our system. The interpreter employs separate counters for method invocations and loop back edges, and drives the compilation immediately when a preset threshold value is reached and the method is presumed to be hot. The transition from the interpreter to compilation can occur even in the middle of a method using on-stack replacement. HotSpot seems to monitor program hot spots 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 (including how many optimization levels it has) is not available in the published literature.

The system collects profiles of receiver type distribution online in the interpreted execution mode, and this information, together with class hierarchy analysis (CHA) [46], is used when optimizing the code for virtual and interface calls. Although collecting the profile information in interpreted mode has low overhead given the already poor performance of the interpretation, the duration of the profile collection is limited to only the early periods of a method's execution and thus the system is vulnerable to mispredictions. If the behavior of the early execution of the method is not representative of the long run, the optimization based on the rather inaccurate profile information can be either ineffective or even harmful to the performance.

The HotSpot employed a technique called *uncommon trap*, which was first implemented in SELF-93 system as described in Section 6.2.1, to avoid code generation for uncommon cases. The system always falls back to the interpreter at a safe point after converting the stack frame when an uncommon path is actually taken. The HotSpot treats both the backup code of devirtualized calls and references to uninitialized classes as uncommon paths, as clearly described in the literature. It is also said to employ other heuristics similar to ours for determining uncommon cases, such as blocks ending in exception throws or in creation of exception objects, special cases for checkcast operations, and use of profile information to override the static heuristics. The HotSpot is the first and currently only production JVM that supports an uncommon trap technique and OSR.

6.1.4 JRockit

The JRockit JVM [93] is another JVM product designed for server applications. It takes a compile-only approach, and relies upon a fast JIT compiler for compiling methods quickly at their first invocation, as opposed to interpretive bytecode execution. It employs a sampling-based profiling system to monitor the program's hot spots. If the system determines that a compiled method is causing a performance bottleneck, it reoptimizes the method with a secondary compilation with full optimizations. The details of the compiler structure and contents of the optimizations are not available. But they reported on the inferiority of the intrusive profiling mechanism of method invocation counters generated in the unoptimized code in comparison to the sampling-based profiler to detect hot methods.

6.1.5 Joeq

Joeq [115, 116] is a research platform for virtual machine and compiler technology, released as open source software. It is implemented entirely in Java, but was designed to support multiple languages, C, C++, and even x86 binary code, as well as Java bytecode. It includes an interpreter, an optimizing compiler, and a profiling system that drives dynamic recompilation. The optimizing compiler employs a common unified internal representation, and all analyses and optimizations are performed uniformly across all the different

type of input code. As in our system, Joeq provides two different profilers, the timer-based sampling profiler [113] and the instrumenting profiler. The profile information is used by various compiler optimizations to improve their effectiveness, such as inlining frequently invoked call sites.

Whaley [114] described a technique for performing partial method compilation using basic-block-level offline profile information. This system eliminates rarely or never executed code portions and applies optimizations only on frequent cases. When a branch is taken to execute those eliminated paths at runtime, the system falls back to an interpreter. The technique allows most optimizations to completely ignore rare paths and fully optimize the common cases. This system also assumes falling back to an interpreter when the rare path is taken.

The technique is implemented on Joeq virtual machine, but the Joeq Java compiler does not perform significant optimizations and thus the effectiveness is limited. Thus he estimated the effectiveness of the technique by collecting basic-block-level profiles offline and then using this information to refactor the affected classes with the Bytecode Engineering Library [7]. The interpreter transition points at rare block entry are replaced with method calls to synthetic methods that contain all of the code separated from the transition point, so that the compilation of those rarely executed blocks can be avoided. Thus the result is an ideal case, since the compiler need not retain any information to restore the interpreter state.

6.2 Dynamic Optimization Systems in Other Languages

6.2.1 SELF

SELF [30] is a dynamically typed object-oriented language originally designed in 1986 as a refinement and simplification of the Smalltalk-80 language. In order to implement the pure object-oriented language efficiently, many novel techniques were created during the course of the project.

SELF-91

Chambers [27] collected a variety of techniques developed in the early stages of the project into what was called the SELF-91 system. It included several important optimization techniques, but the ones most relevant to this dissertation are the customizations, deferred compilations, and inlining trials.

Customization [28] 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 the self object. This can be regarded as a restricted form of specialization. This technique allowed the compiler to customize each version to the specific receiver type, such as static binding

of method invocations or even inlining, leading to an order of magnitude performance improvement.

The selective specialization technique [44, 45] then corrected the problem of both overspecialization and underspecialization in the customization by specializing only heavily used methods for their most beneficial argument classes, and by specializing those methods based on their arguments, rather than focusing on 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 specializations. 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.

Chambers and Ungar [29] also described a technique called *deferred compilation* for uncommon branches where a skewed execution frequency distribution seems likely. They use type information to defer compilation for messages sent to receiver classes that are presumed to be rare. When the rare path is actually executed, the compiler generates code for the *uncommon branch extension*, which is a continuation of the original compiled code from the point of failure to the end of the method. The extension is unoptimized to avoid recursive uncommon branches, and reuses the stack frame created for the original common case version. It was demonstrated that the technique increases compilation speed significantly, by nearly an order of magnitude, but there were both performance and code size problems when the compiler's uncommon code predictions were wrong.

Dean and Chambers [42, 43] describe the *inlining trial* method. This technique uses the first compilation as an experimental version and records inlining decisions and the resulting effects of the optimizations in a database. The compiler can then take advantage of the recorded information for future inlining decisions by searching the database for that information about the receiver and arguments. They do not exploit runtime profile information in their system, though the expected execution frequency of the call site is used for the inlining decisions. Central to this technique is type group analysis, which determines how much of the argument type information available at the call site was actually used during optimizations of the inlined code. This has a similar effect to our impact analysis, but the impact analysis is considered more general in the sense that it can handle not only method parameters but also global variables such as object instance fields.

SELF-93

The SELF-93 system [59, 62] enhanced the SELF-91 implementation in many important ways, including a type-feedback system, profile-directed adaptive recompilation, and dynamic deoptimizations with an on-stack replacement (OSR) technique.

SELF-93 pioneered the online-profile-directed adaptive recompilation systems. The goals of this system were to avoid long compilation 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 to reflect the invocation frequencies and to avoid eventually reaching the threshold for recompilation for relatively unimportant methods.

The system also collects call-site-specific profile information for receiver class distributions (type feedback) in unoptimized runs, and then when the method is recompiled, makes use of this information to optimize dynamically-dispatched calls by predicting likely receiver types and inlining calls for these types. It was demonstrated that the performance of many programs written in SELF can be substantially improved with this technique.

Grove et al. [54] complement this result by studying the various characteristics of the profiles of receiver class distributions collected offline, such as the degree of bias, the effectiveness of deeper granularity, and the stability across input and programs. They reported that the compiler could effectively use deeper granularity of the profile context to predict more precisely the target of dynamically dispatched procedure calls.

The problems in deferred compilation described in the previous section were fixed by treating the occurrence of uncommon cases as another form of runtime feedback and replacing overly specialized code with less specialized code, instead of just extending the specialized code with an unoptimized extension code. This approach was made possible by introducing both an OSR technique and an adaptive recompilation system. OSR allowed the dynamic deoptimization [61] of the target code and the replacement of the stack frame containing the uncommon trap with several unoptimized frames. When it was found that this unoptimized compiled code was executed frequently, the recompilation system could optimize the method again based on the feedback from the unoptimized code. Both SELF-91 and SELF-93 systems focused only on uncommon branches for virtual method calls for the deferred compilation.

6.2.2 Dynamic Binary Optimizers

There are several binary translation systems for profile-based native-to-native reoptimization, such as Dynamo [19], its descendent DynamoRIO [22], HCO [40], and Mojo [33]. These systems identify frequently executed paths (traces) and optimize them by exploiting code layout and other runtime optimization opportunities.

For example, Dynamo is a fully transparent dynamic optimization system, requires no user intervention, and takes an already compiled native instruction stream as input and reoptimizes it at runtime. The use of the interpreter is to identify the hot paths for reoptimization rather than to reduce the total compilation cost as in our system. They employ a technique called NET (next executing tail) [48] to identify hot paths as quickly as possible. If a counter provided at each selected start-of-trace point, typically the target

address of a backward branch, exceeds a preset threshold value, then the trace at the next execution time is selected as a target of dynamic optimizations.

Since the trace is a single-entry multiple-exit contiguous sequence, and can extend across static program boundaries, the trace has arbitrary sub-method and cross-method granularity as the unit of optimization, similar to the effect of our partial inlining in the region-based compilation. However, they operate only on a single trace at a time, and optimizations can be less effective when the selected trace is hot but not a dominant one. This may not be a problem for these systems, since the statically optimized generic code is already there and a specialized reoptimized version is being generated. In our dynamic compilation environment, we have to be more conservative not to frequently exit from selected regions, and thus employ more general regions to contain arbitrary numbers of hot traces by removing blocks of rarely executed code.

6.2.3 Oberon

Kistler and Franz [72, 73] describe a continuous optimization system that constantly monitors the system's state and re-performs optimizations in the background during idle time. The profiling components collect a variety of runtime information, such as basic-block-level edge counts, individual path counts, the execution times of individual procedures, as well as the execution frequency for each procedure. They found that recompiling even fully optimized code in response to the changes in profiling information could give rise to real performance improvements. A code reclamation mechanism is mandatory in this system, since it can produce an unlimited number of optimized versions for each procedure due to the continuous optimizations.

6.2.4 Staged Compilation

There has been much work in the area of dynamic code generation and specialization, most of which require either source language extensions, such as the tcc system [91], or programmer annotations such as Tempo [80] and DyC [16, 52]. These systems perform staged specialization. That is, a static compiler first performs the majority of the optimization work and prepares for a dynamic compilation process by generating templates, and a dynamic compiler then instantiates the templates using the values available at runtime. They rely on programmer intervention to clarify what to specialize and what variables to specialize for.

Calpa [82] automated this process of generating annotations for the DyC dynamic compiler. It evaluates the information that was collected offline regarding basic block execution frequencies and a value profile based on its own cost/benefit model, and determines runtime constants for specialization and dynamic compilation strategies. The staged compilation framework has been further refined in their subsequent system [89].

CoCo [34] described a continuous compilation framework that combines both static

optimization at compile-time and dynamic optimization at runtime. At compile time, CoCo predicts the impact of applying an optimization without actually applying it, using three types of models: code application models, optimization models, and resource models. Using the prediction framework, CoCo determines the optimization plan regarding what optimizations to apply and in what order to apply them in a given code context. At runtime, CoCo applies the dynamic code transformation in response to changes in program behavior as guided by the dynamic optimization plans developed at compile time.

6.2.5 Others

The notion of mixed execution of interpreted and compiled code was considered as part of a continuous compiler or smart JIT approach in [90], and a study of three-mode execution using an interpreter, a fast non-optimizing compiler, and a fully optimizing compiler was reported in [3]. In both of these papers, it was proven that there are performance advantages from using an interpreter in the system to balance the compilation cost and the code quality, but the problems of the compilation memory footprint and the generated code size were not discussed.

Bruening and Duesterwald [21] explored the issues in finding optimal compilation unit shapes for an embedded Java JIT compiler. They demonstrated that always using method boundaries is a poor choice for compilation. They did not implement a working JIT compiler for evaluation, and only provided estimates of the code size reductions when using trace-based and loop-based strategies for compilation units. They found that a majority of the code in methods is rarely or never executed, and concluded that code size could be reduced drastically with only a negligible change in performance.

Ephemeral instrumentation [104] 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 a pointer to a general subroutine that updates a frequency counter for the corresponding edge. The collected data is then used offline for a static compiler. Our profiling system, on the other hand, is not limited to the branch target, but is applicable to any point in the program by generating the corresponding code for value sampling. Also the instrumentation system is integrated into the fully automated dynamic optimization system.

Krintz and Calder [76] describe an annotation framework for reducing compilation overhead for Java programs. One of the proposed annotations is method inlining on the basis of analysis of profile information collected offline, which allows substantial reduction of startup time compilation overhead.

Another profile-driven dynamic recompilation system is described in [23] for Scheme. They use edge-count profile information for basic block reordering in the recompiled code resulting in improved branch prediction and cache locality.

6.3 Static Compilation Systems

6.3.1 IMPACT

The IMPACT is an experimental framework developed at the University of Illinois at Urbana-Champaign to investigate architectural and compilation techniques to support ILP processors. Some of the work performed in this framework is relevant to this dissertation.

Chang et al. [31] described profile-guided procedure inlining in an optimizing C compiler. They first construct a weighted call graph using the profile information collected offline on the numbers of invocations of each function and the relative hotness counts of each call edge. A greedy algorithm is then applied bottom-up in the call graph to maximize the number of reductions of dynamic function calls while keeping the code size expansion within a fixed bound. Their result shows a significant performance improvement with a relatively low code expansion ratio. However they don't report on how much of the effectiveness is contributed by the use of profile information. Dynamically dispatched calls (through function pointers in C programs) were not inlined in this study.

Hank et al. [56] described the problems of the conventional function-based compilation strategy and demonstrated the potential of the region-based compilation technique by presenting the results of several experimentals, including results on the static code size savings. The proposed region formation was designed to perform a normal (possibly an aggressive) inlining pass first, followed by a partitioning phase that created new regions based on heuristics that used offline profile results.

Hank [55] further investigated the application of classical optimizations to region-based compilation units, and the issues involved in separate register allocations by the compilation units. Two additional steps, encapsulation and reintegration, were required to make regions look like ordinary functions for optimizations and then to reintegrate them into the containing function.

Trace scheduling [79] is a technique that predicts the outcome of conditional branches and then optimizes the code assuming the predictions are correct. It can suffer from the complexity involved in the compensation code generation. Superblock scheduling [63] simplifies the complexity by using tail duplication to create superblocks, single-entry multiple-exit regions. Both of these techniques were originally designed to extend the scope of ILP scheduling beyond basic block boundaries to encompass larger units. Other classic optimizations were also extended to exploit superblocks [32].

6.3.2 VELOCITY

The VELOCITY is a research compiler infrastructure developed at Princeton University primarily to address a new approach to compiler organization. The compiler organization work performed in this framework includes two techniques relevant to this dissertation: procedure boundary elimination and optimization space exploration.

Procedure Boundary Elimination (PBE) [107, 106, 105] is a framework for unrestricted whole-program optimization, and can be considered as an extreme case of the region-based compilation, targeting the entire body of the program. It first unifies the whole program into a single compilation unit, which is then repartitioned into units better suited to optimization than the original procedures. PBE uses the same algorithm proposed by Hank et al. [56] for the region formation from the unified program.

Optimization Space Exploration (OSE) [109, 108, 105] is an iterative compilation method, trying multiple optimization paths for searching which optimizations to apply and in what order. They first construct a decision tree, with the most important optimizations in the root node, and search the space by predicting the relative performance benefit with each optimization configuration based on a simplified machine model and on profile data. They observed 5% to 10% speedups in average compared to -O2 compilation for the Itanium, while the compilation time almost doubled even with compile-time search space pruning.

6.3.3 Others

Ball and Larus [20] proposed a heuristic approach for static branch prediction based on the data types and the types of comparisons used in the branches and the code in the target basic blocks. Hank et al. [57] also studied a similar approach using hazard avoidance and branch heuristics, and showed that the performance of static-analysis-based superblock formation and optimizations is comparable to profile-based methods for many benchmarks. We also used program-based static heuristics for the region selection, and propagated the rare and non-rare information using backward data flow to identify rarely or never executed regions.

Scheifler [92] shows that inlining optimization can be reduced to the well-known knapsack problem. He uses a greedy algorithm to minimize the estimated number of function calls subject to a size constraint. This relies on runtime statistics about the program to calculate the expected overhead of each invocation. The constant ratio assumption is used to avoid the cost of a multi-level history. Kaser and Ramakrishman [70] propose a probabilistic model to estimate the effect of using profile data, with a one-level history based on the constant ratio assumption. When evaluating inlinable calls remaining after optimization, they report good results with their technique compared to other compilers.

Ayers et al. [18] describe the design and implementation of the inlining and cloning in the HP-UX optimizing compiler, and show the performance of the SPECint92 and 95 benchmarks can be substantially improved with their techniques. They use profile information collected offline to prioritize the inlining or cloning candidates of the call sites to be considered. The use of profile information is reported to be quite effective, but it is an intra-procedural profile of the basic block level execution frequency, not information on call edges for guiding inlining for a particular call path.

Autrey and Wolfe [17] proposed an analysis to identify *glacial variables* to find good

candidates for specialization. Their analysis is static, and the execution frequency is estimated by the loop nesting level, without using dynamic profile information. Calder et al. [26] showed the potential for performance improvement that the value-profile-based specialization can yield using several hand-transformed examples based on offline-collected value profiles. In contrast, we described a fully automated design and implementation of online dynamic code specialization that has been integrated into our production-level Java JIT compiler.

Muth et al. [83] described a fully automatic value-profile-based specialization system. They employ a cost-benefit analysis that is used both to reduce the overhead caused by value profiling (which they call *goal-directed value profiling* [110]) and also to identify the code to be specialized. This is very similar to our impact analysis, which provides specialization candidates to which we apply value profiling. However, their targets for profiling, and hence for specialization, are limited to those variables currently in registers.

Way et al. [111] improved the region formation algorithm of Hank [55] to make it scalable by combining region selection and the inlining process, and reduced the compilation time memory requirements considerably. They also evaluated region-based partial inlining that was performed through partial cloning, and observed small performance improvements [112]. This work was also performed for improving ILP static compilers, as in Hank's system.

Chapter 7

Conclusions

7.1 Summary

In this dissertation, we have described the design of dynamic optimization techniques for Java that exploit various kinds of runtime profile information. We implemented these techniques in an IBM Java Just-in-Time compiler product, and performed a detailed evaluation for each of the techniques using industry standard benchmark programs and several applications.

We first described in Chapter 3 how we can construct an efficient dynamic optimization framework for Java, and discussed several key design points. We also presented extensive experimental results, demonstrating the effectiveness for achieving high performance and low compilation overhead in both program startup and steady state measurements. Our contributions can be summarized as follows:

- We presented a system architecture for a simple, but efficient and high-performance, dynamic optimization framework in a production-level JIT compiler. We showed that use of a mixed mode interpreter is important in the framework to build an effective recompilation system, by taking advantage of the zero compilation cost and setting appropriate tradeoff levels for each transition between optimizations.

- We presented a program profiling mechanism, combining multiple different techniques depending on profiler characteristics and target code compilation levels. That is, a counter-based profiler, counting method invocation frequencies and loop iteration for detecting hot interpreted methods, a sampling-based profiler for detecting hot compiled methods, and an instrumenting profiler, installed and uninstalled dynamically for collecting more detailed runtime information for selected methods.

On top of this dynamic optimization framework, we designed and implemented several dynamic optimizations. In Chapter 4, we described the profile-directed method inlining and the dynamic code specialization. Both of these optimizations exploit the dynamically

generated instrumentation mechanism for collecting runtime information, such as call site distribution, parameter values, and global variable values. Our contributions here can be summarized as follows:

- We presented a profile-directed method inlining technique based on runtime call site distribution and frequency information collected with a dynamic instrumentation mechanism. We eliminated all static inlining heuristics except that we always inline tiny methods, and showed the significant advantages for both improving performance and reducing compilation overhead.
- We presented a code specialization technique that employs impact analysis to estimate the benefit of specialization and runtime value sampling for specialization decisions. This is a fully automated design with no programmer intervention required. We showed that the proposed technique can produce modest performance improvement for a set of benchmark programs.

Another dynamic optimization technique proposed in this dissertation is region-based compilation, as described in Chapter 5. We discussed the issues and challenges for performing effective region-based compilation in a dynamic compilation environment, in contrast to other environments, static compilers and dynamic binary optimizers, and proposed a technique that is the most suitable for Java JIT compiler. Our contributions in this area can be summarized as follows:

- We presented a design for region formation, partial inlining, and region exit handling that is effective for dynamic compilers, and described the algorithm in detail for the intra-method region selection and its integration in the inlining process.
- Together with two region-aware optimizations, partial escape analysis and partial dead code elimination that exploit region exit points, implemented in the region-based compilation framework, we showed that our proposed technique can contribute to significant advantages for both improving performance and reducing compilation overhead.

7.2 Future Work

There is a long history of research in the area of dynamic and adaptive optimizations, as summarized well in a recent survey [12]. We believe that dynamic optimization techniques will likely become more important as programs become more complex and rely more heavily on object-oriented language features, such as inheritance and polymorphism. Static analysis and optimization will become more difficult as program complexity increases, creating more opportunities for dynamic optimizations to identify and exploit runtime execution behaviors.

The following describes some of our subjective observations and future directions regarding the design and implementation of the system described in this dissertation.

7.2.1 Dynamic Optimization Frameworks

The ultimate goal of dynamic optimizations would be to construct a continuous program optimization (CPO) framework, where code is continually and automatically reoptimized in the background, capturing new optimization opportunities based on runtime profile values. When new code is generated, reflecting new program behavior, the control should be transferred from the currently executing code to the corresponding position within the newly generated code (hot swap). The background mechanism should then start finding yet more opportunities in the execution patterns for the next cycle of reoptimizations.

Our dynamic optimization system is still far from such a CPO framework. Although our profiling system operates continuously during the entire period of program execution, we have no mechanisms such as hot code swapping and code reclamation, which are mandatory for building a generalized CPO framework. In a CPO framework, we need to capture optimization opportunities in the currently executing code with minimal overhead, and we believe the concepts of impact analysis and two-stage profiling described in this dissertation can play a key role. This is because our concepts try to exploit knowledge of the program's structure to determine the best profiling strategies and to make a cost-benefit model for estimating the value of reoptimization opportunities.

Another interesting area for an efficient CPO framework would be to identify and exploit phase shifts in executing programs. CPO is based on an assumption that there is usually no single *typical* usage scenario in real applications, which could then be optimized with a single optimization strategy, but instead there are several *distinctive* usage scenarios calling for different strategies. A phase-shift detector could determine in a timely manner when an executing program is in a stable phase of program execution or when it is in a transition between phases. If this information were available at a reasonable cost, we could rely on the technologies in the CPO framework to determine when we should reconsider optimization decisions. The research on online phase detection mechanism is just emerging [84], and this will be a new research frontier.

7.2.2 Profile-Directed Optimizations

We need to further refine the system to improve the cost and benefit of the profile-directed optimizations. For example, we have considered so far only the relative strengths and distributions of the call edges when driving profile-directed inlining. However, the significant impact of method inlining is not only through the direct effect of eliminating call overhead, but also due to indirect effects of specializing an inlined method body into the calling context, with better utilization of dataflow information at the call sites. Since our instrumentation mechanism collects parameter values as well as return addresses, it

should be possible to estimate whether inlining a method will be beneficial in terms of the effect on these optimizations as well. We can extend our impact analysis in this direction. We will try to exploit this information in the future for pursuing better inlining strategies.

In the long term, however, the essential problem of the dynamic optimization systems is to decide whether or not the optimization effort can be offset by the performance benefit for a given program. Most, if not all, of the dynamic optimization systems currently available, including ours, predetermine the set of optimizations provided for each optimization level. Basically those optimizations considered to be lightweight, such as those with a linear relationship to the size of the target code, are applied at the earlier optimization levels, and those with higher costs in compilation time or greater code size expansion are delayed to the later optimization levels. However, the classification of optimizations into several different levels has been either intuitive work or just based on the measurements of compilation cost and performance benefit on a typical execution scenario analyzed offline. Consequently, the optimizations applied are not necessarily effective for the actual target methods compiled with the given optimization levels. That is, some optimizations may not contribute to any useful transformations for performance improvements, but can result in a waste of compilation resources.

The problem here is that we equally apply the same set of optimizations for those methods selected to compile at that optimization level, regardless of the type and characteristics of each method. Ideally it would be desirable to dynamically assemble a set of suitable optimizations depending on the characteristics of the target methods so that we can apply only those optimizations known to be effective for each method. It would be better for the total cost and benefit management if we could not only selectively apply optimizations on performance-critical methods, but also selectively assemble or customize a set of optimizations depending on the characteristics of the target methods so that we can apply only those optimizations known to be effective for the given methods.

For example, inlining decisions are now profile-based, and thus the compilation boundaries are dynamically determined. The next step will be to estimate the costs and benefits we can expect from optimizations when the target method is inlined, before performing actual inlining. We can employ a technique similar to impact analysis to estimate the benefit, because a majority of the inlining benefits come from specializing an inlined method body into the calling context with better utilization of dataflow information at the call sites. The cost estimation will be based on the structure of a given method, using indications such as loop structures, characteristics of field and array accesses, and other features, as suggested in [11]. *Program metrics* [72] are using a similar notion to try to estimate the potential benefit of program optimizations. *Online performance auditing* [77] is an approach to evaluate the effectiveness of optimizations online to automatically identify and correct performance anomalies at runtime.

7.2.3 Region-Based Compilation

While we obtained a significant improvement in both performance and compilation overhead reduction for a selected set of benchmarks and applications, we need to validate the technique for broader set of real applications, especially large-scale server applications. Large-scale applications are often constructed by piecing together separate components and libraries, producing programs where the code that performs the actual work is buried within and spread across many layers of abstraction. These applications generally have very flat profile when analyzed with method-level profiling [53], but they often show a significant bias in their execution profile when we look at them with basic-block-level profiling. Thus we believe that region-based compilation could be effective for these seemingly flat profile applications.

In our current implementation, we used a single method for handling rarely executed code regions. We drove recompilation with the same optimization level and performed on-stack replacement (OSR). This allowed a quick prototyping of the RBC technique, but poses some boundary case problems, such as different class resolution status or the hierarchy when performing recompilation. It also forces us to be conservative for region selection, because the cost of region exit was relatively high. We need to address these problems.

The approach we suggest here is, as briefly discussed in Section 5.4.5, to employ multiple methods for handling rare regions, including code splitting, method outlining, and rare-path cutting (both driving recompilation and backing to the interpreter through OSR), and then to select the best strategy among them based on a cost-benefit analysis. Note that OSR backing to the interpreter does not cause the above-mentioned boundary case problems we have in our current system. This approach will allow us to be more aggressive in the region selection process, and should result in forming better compilation or optimization units.

Rare-path cutting is an important option for handling rare regions because we can eliminate merge points while avoinding the code size and compilation overhead problems. In particular, rare-path cutting will be the best strategy if the expected execution frequency of a rare region is indeed extremely rare. Retaining rare-path cutting as one of the viable options allows us to form better compilation or optimization units. However, we need other options for handling relatively infrequent, but not extremely rare paths. Method splitting and method outlining will give us options for selective region exit strategies. Given a rare-region boundary point, it is important to select the best strategy from these options using a single cost-benefit analysis. The factors of the analysis will be as follows:

- Increased optimization opportunities (and the resulting code quality)
- Impact on compilation overhead
- Expected frequency of region exit and the costs of handling the region exits
- Expected execution speed when region exit occurs

We may also be able to restructure even non-rare regions in order to make optimizations work effectively. When we apply transformations, such as code splitting, method outlining, and peeling, they will be based on a cost-benefit analysis for each transformation. For example, given an object's allocation, if a basic block approach forces escape analysis to make a decision not to stack-allocate the object, it would probably be worthwhile to restructure the non-rare region to exclude the basic block.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Dong-Yuan Chen, Anwar Ghuloum, Vijay Menon, Brian Murphy, Mauricio Serrano, and Tatiana Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7(1):19–31, February 2003.

[2] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, Effective Code Generation in a Just-in-Time Java Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 280–290, New York, June 1998. ACM Press.

[3] Ole Agesen and David Detlefs. Mixed-mode Bytecode Execution. Technical report, SMLI TR-2000-87, Sun Microsystems, June 2000.

[4] William P. Alexander, Robert F. Berry, Frank E. Levine, and Robert J. Urquhart. A Unifying Approach to Performance Analysis in the Java Environment. *IBM Systems Journal*, 39(1):118–134, January 2000.

[5] Bowen Alpen, C.R. Attanasio, John Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn-Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, January 2000.

[6] Bowen Alpern, Steve Augart, Steve M. Blackburn, Maria Butrico, Antony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark Mergen, J. Eliot B. Moss, Ton Ngo, and Vivek Sarkar. The Jikes RVM Project: Building an Open Source Research Community. *IBM Systems Journal*, 44(2):399–417, May 2005.

[7] Apache Jakarta foundation. Byte Code Engineering Library (BCEL), 2006. http://jakarta.apache.org/bcel/index.html.

[8] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, Reading, Mass., 2000.

[9] Matthew Arnold. *Online Profiling and Feedback-Directed Optimization of Java.* PhD thesis, Rutgers University, New Jersey, October 2002.

[10] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimizations in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*, pages 47–65, New York, October 2000. ACM Press.

[11] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter 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)*, New York, December 2000. ACM Press.

[12] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449–466, February 2005.

[13] Matthew Arnold, Stephen J. Fink, Vivek Sarkar, and Peter F. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, pages 52–64, New York, January 2000. ACM Press.

[14] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online Feedback-Directed Optimization of Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, pages 111–129, New York, November 2002. ACM Press.

[15] Matthew Arnold and Barbara G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 168–179, New York, June 2001. ACM Press.

[16] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, pages 149–158, New York, May 1996. ACM Press.

[17] Tito Autrey and Michael Wolfe. Initial Results for Glacial Variable Analysis. *International Journal of Parallel Program*, 26(1):43–64, February 1998.

[18] Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, pages 134–145, New York, June 1997. ACM Press.

[19] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 1–12, New York, June 2000. ACM Press.

[20] Thomas Ball and James R. Larus. Branch Prediction For Free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, pages 300–313, New York, June 1993. ACM Press.

[21] Derek Bruening and Evelyn Duesterwald. Exploring Optimal Compilation Unit Shapes for an Embedded Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, New York, December 2000. ACM Press.

[22] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*, pages 265–275, Los Alamitos, March 2003. IEEE Computer Society.

[23] Robert G. Burger and R. Kent Dybvig. An infrastructure for Profile-Driven Dynamic Recompilation. In *Proceedings of the International Conference on Computer Languages*, pages 240–249, Los Alamitos, May 1998. IEEE Computer Society.

[24] Michael G. Burke, Jong-Deok Choi, Stephen J. Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM SIGPLAN Java Grande Conference*, pages 129–141, New York, June 1999. ACM Press.

[25] Brad Calder, Peter Feller, and Alan Eustace. Value Profiling. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, pages 259–269, Los Alamitos, December 1997. IEEE Computer Society.

[26] Brad Calder, Peter Feller, and Alan Eustace. Value Profiling and Optimization. *Journal of Instruction-Level Parallelism*, 1(1-6), March 1999.

[27] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Technical Report CS-TR-92-1420, Stanford, March 1992.

[28] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'89)*, pages 146–160, New York, July 1989. ACM Press.

[29] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems,*

Languages and Applications (OOPSLA'91), pages 1–15, New York, October 1991. ACM Press.

[30] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'89)*, pages 49–70, New York, October 1989. ACM Press.

[31] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-Guided Automatic Inline Expansion for C Programs. *Software Practice and Experience*, 22(5):349–369, May 1992.

[32] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software Practice and Experience*, 21(12):1301–1321, December 1991.

[33] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, New York, December 2000. ACM Press.

[34] Bruce R. Childers, Jack W. Davidson, and Mary Lou Soffa. Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation. In *Workshop on Next Generation Systems, held during the International Symposium on Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, April 2003. IEEE Computer Society.

[35] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs. In *Proceedings of the ACM Workshop on Program Analysis for Sotware Tools and Engineering*, New York, September 1999. ACM Press.

[36] Michal Cierniak, Marsha Eng, Neal Glew, Brian T. Lewis, and James M. Stichnoth. The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1):5–18, February 2003.

[37] Michal Cierniak, Brian T. Lewis, and James M. Stichnoth. Open Runtime Platform: Flexibility with Performance using Interfaces. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 156–164, New York, November 2002. ACM Press.

[38] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 13–26, New York, June 2000. ACM Press.

[39] Cliff Click and John Rose. Fast Subtype Checking in the HotSpot JVM. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 96–103, New York, November 2002. ACM Press.

[40] Robert Cohn and P. Geoffrey Lowney. Hot Cold Optimization of Large Windows/NT Applications. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pages 80–89, Los Alamitos, December 1996. IEEE Computer Society.

[41] Ron Cytron, J. Ferrante, B. K. Rosen, Mark N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[42] Jeffrey Dean and Craig Chambers. Training Compilers for Better Inlining Decisions. Technical report, 93-05-05, Department of Computer Science and Engineering, University of Washington, May 1993.

[43] Jeffrey Dean and Craig Chambers. Towards Better Inlining Decisions Using Inlining Trials. In *Proceedings of the ACM SIGPLAN Conference on LISP and Functional Programming*, pages 273–282, New York, June 1994. ACM Press.

[44] Jeffrey Dean, Craig Chambers, and David Grove. Identifying Profitable Specialization in Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–96, New York, June 1994. ACM Press.

[45] Jeffrey Dean, Craig Chambers, and David Grove. Selective Specialization for Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 93–102, New York, June 1995. ACM Press.

[46] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 77–101, Berlin, August 1995. LNCS 952, Springer-Verlag.

[47] David Detlefs and Ole Agesen. Inlining of Virtual Methods. In *13th European Conference on Object-Oriented Programming (ECOOP'99)*, pages 258–278, Berlin, June 1999. LNCS 1628, Springer-Verlag.

[48] Evelyn Duesterwald and Vasanth Bala. Software Profiling for Hot Path Prediction: Less is More. In *Proceedings of the 9th ACM Conference on Architectural Support on Programming Languages and Operating Systems*, pages 202–211, New York, November 2000. ACM Press.

[49] Eclipse Consortium. Eclipse Open Platform, 2002. Available at http://www.eclipse.org/.

[50] Stephen J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*, pages 241–252, Los Alamitos, March 2003. IEEE Computer Society.

[51] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Reading, Mass., 2005.

[52] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An Evaluation of Staged Run-Time Optimizations in DyC. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 293–304, New York, June 1999. ACM Press.

[53] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan. Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04)*, pages 151–162, Berkeley, May 2004. USENIX Association.

[54] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95)*, pages 108–123, New York, October 1995. ACM Press.

[55] Richard E. Hank. *Region-Based Compilation*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, 1996.

[56] Richard E. Hank, Wen-mei W. Hwu, and B. Ramakrishna Rau. Region-Based Compilation: An Introduction and Motivation. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pages 158–168, Los Alamitos, November 1995. IEEE Computer Society.

[57] Richard E. Hank, Scott A. Mahlke, Roger A. Bringmann, John C. Gyllenhaal, and Wen-mei W. Hwu. Superblock Formation Using Static Program Analysis. In *Proceedings of the 26th International Symposium on Microarchitecture (MICRO-26)*, pages 247–255, Los Alamitos, December 1993. IEEE Computer Society.

[58] Kim Hazelwood and David Grove. Adaptive Online Context-Sensitive Inlining. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*, pages 253–264, Los Alamitos, March 2003. IEEE Computer Society.

[59] Urs Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, Technical Report CS-TR-94-1520, Stanford, August 1994.

[60] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *5th European Conference on*

Object-Oriented Programming (ECOOP'91), pages 21–38, Berlin, July 1991. LNCS 512, Springer-Verlag.

[61] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, pages 32–43, New York, July 1992. ACM Press.

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

[63] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1-2):229–248, May 1993.

[64] IBM Corporation. WebSphere Studio Application Developer, 2002. Information available at http://www.ibm.com/software/ad/adstudio/.

[65] IBM Corporation. IBM Workplace Client Technology: Delivering the Rich Client Experience, 2004. White paper available at http://www.lotus.com/products/product5.nsf/wdocs/7231aae345edcabf85256e890071b1ad.

[66] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio 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 and Applications (OOPSLA'00)*, pages 294–310, New York, October 2000. ACM Press.

[67] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03)*, pages 187–204, New York, October 2003. ACM Press.

[68] Java Grande Forum. Java Grande Benchmark, 2000. Available at http://www.epcc.ed.ac.uk/javagrande/sequential.html.

[69] Just System Corporation. IchitaroArk for Java, Japanese word processor system, 1998. Available at http://www.justsystem.com/ark/index.html.

[70] Owen Kaser and C.R. Ramakrishman. Evaluating Inlining Techniques. *Computer Languages*, 24(2):55–72, July 1998.

[71] Motohiro Kawahito, Hideaki Komatsu, and Toshio 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 (ASPLOS-IX)*, pages 118–127, New York, November 2000. ACM Press.

[72] Thomas Kistler and Michael Franz. Continuous Program Optimization: Design and Evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[73] Thomas Kistler and Michael Franz. Continuous Program Optimization: A Case Study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[74] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, pages 147–158, New York, June 1994. ACM Press.

[75] Andreas Krall. Efficient JavaVM Just-in-Time Compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 205–212, Los Alamitos, October 1998. IEEE Computer Society.

[76] Chandra Krintz and Brad Calder. Using Annotations to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 157–167, New York, June 2001. ACM Press.

[77] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online Performance Auditing: Using Hot Optimizations Without Getting Burned. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 239–251, New York, June 2006. ACM Press.

[78] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, pages 36–44, New York, October 1998. ACM Press.

[79] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[80] Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient Incremental Run-Time Specialization for Free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 281–292, New York, June 1999. ACM Press.

[81] Vitaly V. Mikheev, Stanislav A. Fedoseev, Vladimir V. Sukharev, and Nikita V. Lipsky. Effective Enhancement of Loop Versioning in Java. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 293–306, Berlin, April 2002. LNCS 2304, Springer-Verlag.

[82] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In *Proceedings of the 33rd International Symposium on Microarchitecture (MICRO-33)*, pages 1–12, Los Alamitos, December 2000. IEEE Computer Society.

[83] Robert Muth, Scott A. Watterson, and Saumya K. Debray. Code Specialization based on Value Profiles. In *7th International Static Analysis Symposium (SAS 2000)*, pages 340–359, Berlin, June 2000. Springer-Verlag (LNCS vol. 1824).

[84] Priya Nagpurkar, Michael Hind, Chandra Krintz, Peter F. Sweeney, and V.T. Rajan. Online Phase Detection Algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*, pages 111–123, Los Alamitos, March 2006. IEEE Computer Society.

[85] Takeshi Ogasawara, Hideaki Komatsu, and Toshio 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 and Applications (OOPSLA'01)*, pages 83–95, New York, October 2001. ACM Press.

[86] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. EDO: Exception-Directed Optimization in Java. *ACM Transactions on Programming Languages and Systems*, 28(1):70–105, January 2006.

[87] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot Server Compiler. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 1–12, Berkley, April 2001. USENIX Association.

[88] Karl Pettis and Robert .C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*, pages 16–27, New York, June 1990. ACM Press.

[89] Matthai Philipose, Craig Chambers, and Susan J. Eggers. Towards Automatic Construction of Staged Compilers. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 113–125, New York, January 2002. ACM Press.

[90] Michael P. Plezbert and Ron K. Cytron. Does "Just in Time" = "Better Late than Never"? In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 120–131, New York, January 1997. ACM Press.

[91] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A System for Fast, Flexible, and High-Level Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, pages 109–121, New York, June 1997. ACM Press.

[92] Robert W. Scheifler. An Analysis of Inline Substitution for a Structured Programming Language. *Communications of the ACM*, 20(9):647–654, September 1977.

[93] K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm. Impact of JIT/JVM Optimizations on Java Application Performance. In *Proceedings of the 7th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-7)*, Los Alamitos, February 2003. IEEE Computer Society.

[94] Standard Performance Evaluation Corporation. SPECjvm98 and SPECjbb2000 benchmarks, 2000. Available at http://www.spec.org/osg.

[95] Toshio Suganuma, Takeshi Ogasawara, Kiyokuni Kawachiya, Mikio Takeuchi, Kazuaki Ishizaki, Akira Koseki, Tatsushi Inagaki, Toshiaki Yasue, Motohiro Kawahito, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Evolution of a Java Just-in-Time Compiler for IA-32 Platforms. *IBM Journal of Research and Development*, 48(5/6):767–795, September 2004.

[96] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, January 2000.

[97] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, pages 180–194, New York, October 2001. ACM Press.

[98] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Design and Evaluation of Dynamic Optimizations for a Java Just-In-Time Compiler. *ACM Transactions on Programming Languages and Systems*, 27(4):732–785, July 2005.

[99] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. An Empirical Study of Method Inlining for a Java Just-In-Time Compiler. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM'02)*, pages 91–104, Berkeley, August 2002. USENIX Association.

[100] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A Region-Based Compilation Technique for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 312–323, New York, June 2003. ACM Press.

[101] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A Region-Based Compilation Technique for Dynamic Compilers. *ACM Transactions on Programming Languages and Systems*, 28(1):134–174, January 2006.

[102] Sun Microsystems, Inc. Java Technology and Web Services. Available at http://java.sun.com/webservices.

[103] Sun Microsystems, Inc. Using Java Reflection. Available at http://java.sun.com/developer/technicalArticles/ALT/Reflection/.

[104] Omri Traub, Stuart Schechter, and Michael D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. Technical report, Division of Applied Science and Engineering, Harvard University, June 2000.

[105] Spyridon Triantafyllis. *Eliminating Scope and Selection Restrictions in Compiler Optimization*. PhD thesis, Princeton University, Princeton, September 2006.

[106] Spyridon Triantafyllis, Matthew J. Bridges, Easwaran Raman, Guilherme Ottoni, and David I. August. A Framework for Unrestricted Whole-Program Optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 61–71, New York, June 2006. ACM Press.

[107] Spyridon Triantafyllis, Manish Vachharajani, and David I. August. Procedure Boundary Elimination for EPIC Compilers. In *Proceedings of the Second Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology*, Los Alamitos, November 2002. IEEE Computer Society.

[108] Spyridon Triantafyllis, Manish Vachharajani, and David I. August. Compiler Optimization-Space Exploration. *Journal of Instruction-Level Parallelism*, 7(1-25), January 2005.

[109] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler Optimization-Space Exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*, pages 265–275, Los Alamitos, March 2003. IEEE Computer Society.

[110] Scott Watterson and Saumya Debray. Goal-Directed Value Profiling. In *10th International Conference on Compiler Construction*, pages 319–333, Berlin, April 2001. Springer-Verlag (LNCS vol. 2027).

[111] Tom Way, Ben Breech, and Lori L. Pollock. Region Formation Analysis with Demand-driven Inlining for Region-based Optimization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 24–36, Los Alamitos, October 2000. IEEE Computer Society.

[112] Tom Way and Lori L. Pollock. Evaluation of a Region-based Partial Inlining Algorithm for an ILP Optimizing Compiler. In *Proceedings of Conference on Parallel and Distributed Processing Techniques and Applications*, pages 552–556, Los Alamitos, June 2002. IEEE Computer Society.

[113] John Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of the ACM SIGPLAN Java Grande Conference*, pages 78–87, New York, June 2000. ACM Press.

[114] John Whaley. Partial Method Compilation using Dynamic Profile Information. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, pages 166–179, New York, October 2001. ACM Press.

[115] John Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. In *Proceedings of the ACM SIGPLAN Workshop on Interpreters, Virtual Machines, and Emulators (IVME)*, pages 58–66, New York, June 2003. ACM Press.

[116] John Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. *Science of Computer Programming Journal*, 57(3):339–356, September 2005.

[117] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 187–206, New York, November 1999. ACM Press.

[118] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik R. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 1–12, Los Alamitos, October 1999. IEEE Computer Society.

List of Publications

Refereed Papers

A1. ◯ Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. "A Region-Based Compilation Technique for Dynamic Compilers," *ACM Transaction of Programming Languages and Systems (TOPLAS)*, Vol. 28, No. 1, pages 134–174, January 2006.

A2. ◯ Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. "Design and Evaluation of Dynamic Optimizations for a Java Just-In-Time Compiler," *ACM Transaction of Programming Languages and Systems (TOPLAS)*, Vol. 27, No. 4, pages 732–785, July 2005.

A3. ◯ Toshio Suganuma, Takeshi Ogasawara, Kiyokuni Kawachiya, Mikio Takeuchi, Kazuaki Ishizaki, Akira Koseki, Tatsushi Inagaki, Toshiaki Yasue, Motohiro Kawahito, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. "Evolution of a Java Just-in-Time Compiler for IA-32 Platforms," *IBM Journal of Research and Development*, Vol. 48, No. 5/6, pages 767–795, September 2004.

A4. Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. "Structural Path Profiling: An Efficient Online Path Profiling Framework for Just-In-Time Compilers," *Journal of Instruction Level Parallelism*, Vol. 6, pages 1–28, April 2004.

A5. Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, Toshio Nakatani. "Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler," In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03)*, pages 187–204, October 2003.

A6. 安江 俊明, 菅沼 俊夫, 小松 秀昭, 中谷 登志男. "動的コンパイラにおける実行時経路情報の構造的収集手法の提案", 情報処理学会論文誌:プログラミング, Vol. 44, No. SIG15, 2003.

A7. Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. "An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers," In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, pages 148–158, September 2003.

A8. ◯ Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. "A Region-Based Compilation Technique for a Java Just-In-Time Compiler," In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 312–323, June 2003.

A9. ◯ Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. "An Empirical Study of Method Inlining for a Java Just-In-Time Compiler," In *Proceedings of the USENIX 2nd Java Virtual Machine Research and Technology Symposium (JVM'02)*, pages 91–104, August 2002.
∗ Received Symposium Best Paper Award.

A10. ◯ Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. "A Dynamic Optimization Framework for a Java Just-In-Time Compiler," In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, pages 180–194, October 2001.

A11. Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. "Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler," *Concurrency: Practice and Experience*, Vol. 12, No. 6, pages 457–475, May 2000.

A12. ◯ Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. "Overview of the IBM Java Just-in-Time Compiler," *IBM Systems Journal*, Vol. 39, No. 1, pages 175–193, January 2000.

A13. Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. "Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler," In *Proceedings of the ACM SIGPLAN Java Grande Conference*, pages 119–128, June 1999.

Unrefereed Papers

B1. 石崎 一明, 川人 基弘, 今野 和浩, 安江 俊明, 竹内 幹雄, 小笠原 武史, 菅沼 俊夫, 小野寺 民也, 小松 秀昭. "Java Just-In-Time コンパイラにおける最適化とその評価", 電子情報通信学会技術研究報告, CPSY-99-64, pages 17–24, 1999.

Others

C1. 水野謙, 菅沼 俊夫, 石崎 一明, 古関 聡, 上田 陽平, 小松 秀昭. "並列トランザクショナルアプリケーションのためのプログラミングフレームワーク", 情報処理学会論文誌:プログラミング, 掲載決定.

C2. 小松 秀昭, 菅沼 俊夫, 小笠原 武史, 石崎 一明, 郷田 修. "SP2 のための HPF コンパイラにおける最適化技術", 情報処理, Vol. 38, No. 2, pages 100–104, 1997.

C3. Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. "Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines," In *Proceedings of the 10th International Conference on Supercomputing (ICS'96)*, pages 18–25, May 1996.

C4. 郷田 修, 大澤 暁, 小松 秀昭, 菅沼 俊夫, 小笠原 武史, 石崎 一明, 中谷 登志男. "HPF コンパイラの実装と評価", 情報処理学会研究報告, HPC-57-20, pages 115–120, 1995.