Evolution of a Java just-in-time compiler for IA-32 platforms

T. Suganuma
T. Ogasawara
K. Kawachiya
M. Takeuchi
K. Ishizaki
A. Koseki
T. Inagaki
T. Yasue
M. Kawahito
T. Onodera
H. Komatsu
T. Nakatani

Java™ has gained widespread popularity in the industry, and an efficient Java virtual machine (JVM™) and just-in-time (JIT) compiler are crucial in providing high performance for Java applications. This paper describes the design and implementation of our JIT compiler for IA-32 platforms by focusing on the recent advances achieved in the past several years. We first present the dynamic optimization framework, which focuses the expensive optimization efforts only on performance-critical methods, thus helping to manage the total compilation overhead. We then describe the platform-independent features, which include the conversion from the stack-semantic Java bytecode into our register-based intermediate representation (IR) and a variety of aggressive optimizations applied to the IR. We also present some techniques specific to the IA-32 used to improve code quality, especially for the efficient use of the small number of registers on that platform. Using several industry-standard benchmark programs, the experimental results show that our approach offers high performance with low compilation overhead. Most of the techniques presented here are included in the IBM JIT compiler product, integrated into the IBM Development Kit for Microsoft Windows®, Java Technology Edition Version 1.4.0.

Introduction

Java** has gained widespread popularity, and an efficient Java virtual machine (JVM**) and just-in-time (JIT) compiler are crucial in providing high performance for Java applications. Since the compilation time overhead of a JIT compiler, in contrast to that of a conventional static compiler, is included in the program execution time, JIT compilers should be designed to manage the conflicting requirements between fast compilation speed and fast execution performance. That is, we would like the system to generate highly efficient code for good performance, but at the same time, the system should be lightweight enough to avoid any startup delays or intermittent execution pauses caused by the runtime overhead of the dynamic compilation.

We previously described the design of version 3.0 of our JIT compiler [1] that was integrated into the IBM Development Kit (DK) 1.1.7. It was designed to apply only relatively lightweight optimizations equally to all of the methods so that compilation overhead would be minimized but competitive overall performance could still be achieved at the time of product release. We used an intermediate representation (IR) called *extended bytecode* (EBC)—a compact and stack-based representation similar to the original Java bytecode. The conversion from the bytecode to the IR is relatively straightforward. Using the IR, we performed both classic optimizations, such as constant propagation, dead code elimination, and common subexpression elimination, and Java-specific optimizations, such as exception check elimination. Without imposing a large impact on compilation overhead, we applied method inlining only to small target methods to alleviate the performance problem caused by frequent calls of relatively small methods. The devirtualization of virtual method call sites was performed using the guard code for testing the target method.

0018-8646/04/$5.00 © 2004 IBM

In the previous version, we also introduced a mixed-mode interpreter (MMI) to allow efficient mixed execution between interpreted mode and compiled code execution mode. We begin the program execution using the MMI as the first execution mode. When the frequently executed methods or critical hot spots[1] of the program are identified using method invocation and loop iteration counters, we invoke the JIT compiler to obtain better performance for those selected methods. The MMI handles a large number of performance-insensitive methods, typically 80% or more of the executed methods in the program, and thus provides an efficient execution mode with no compilation overhead.

To achieve higher performance, we needed to apply more advanced optimization techniques, including aggressive method inlining, dataflow-based optimizations, loop optimizations, and more sophisticated register allocation. However, from the viewpoint of JIT compilers, these are all relatively expensive optimizations, and simply putting these optimizations in the existing JIT framework might have caused unacceptably high compilation overhead, typically large delays in application startup time. Using a two-level execution model with the MMI and a single-level, highly optimizing JIT compiler, the system would not be able to manage the balance between optimization effectiveness and compilation overhead because of the increasing gap of the tradeoff level between the two execution modes. It was therefore imperative to provide multiple, reasonable steps in the compilation levels with well-balanced tradeoffs between cost and expected performance, from which an adequate level of optimization could be selected that would correspond to the execution context.

In this paper, we describe the design and implementation of version 4.5 of our Java JIT compiler, specifically developed for IA-32 platforms, which has been integrated into the IBM DK 1.4.0. To manage the tradeoff, we did several things:

- In addition to the EBC, we introduced two register-based intermediate representations to perform more effective and advanced optimizations. We evaluated existing and new optimizations by considering the compilation costs (both space and time) and applied each of them on an appropriate IR.
- We constructed a dynamic optimization framework using a multilevel recompilation system. This enabled us to focus expensive optimization efforts on only performance-critical methods. We classified all optimizations into several different levels based on the cost and the benefit of each optimization. The profiling system continuously monitors the hot spots of a program and provides information for method promotion.
- We evaluated our approach using industry-standard benchmarks and obtained significant performance improvement for each optimization, while keeping the compilation overhead—in terms of compilation time, compiled code size, and compilation peak memory usage—at a low level.

Since this paper focuses on the evolution of our JIT compiler from the previous version 3.0 in terms of its internal architecture and a variety of added optimizations, we do not address profile-directed optimizations, which are more advanced techniques. The goal of this paper is to describe what we have done in the design of our JIT compiler to balance both the performance improvement and compilation overhead without using the profile-directed optimizations.

This paper is organized as follows. We first give an overview of our JIT compilation system, covering both the overall compiler structure with three different IRs and a dynamic optimization framework using the interpreter and multilevel optimizations. We then describe the platform-independent optimizations performed on each IR, followed by IA-32-specific optimizations, especially for the efficient use of the small number of registers. We next present the experimental results on both performance and compilation overhead. Finally, we summarize the related work and then conclude.

System architecture

This section provides an overview of our system. We present the overall structure of the JIT compiler by describing the three different IRs and list the major optimizations performed on each IR. We then describe the dynamic optimization framework, in which each optimization performed in the three IRs is mapped to an appropriate optimization level based on the compilation cost and the performance benefit.

Structure of the JIT compiler

Figure 1 shows the overall structure of our JIT compiler. We employ three different IRs to perform a variety of optimizations. At the beginning, the given bytecode sequence is converted to the first IR, called the *extended bytecode* (EBC), which was used in the previous version of our compiler as the sole IR. 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

[1] A hot spot is a method invocation within a loop or an indirect method invocation from a loop.

method, we insert `opc_syncenter` and `opc_syncexit` 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 the unguarded devirtualized code is invalidated due to the dynamic class loading.

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 result 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. We have two separate budgets for performing inlining, one for tiny methods and the other for all other types of methods. Inlining the tiny method is always considered beneficial without imposing any harmful effects on compilation time and code size growth. In the next section we give a more detailed description for our inlining policy. Other optimizations performed on EBC include switch statement optimization, typeflow analysis, and exception check elimination.

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 if the target class is resolved but not yet initialized for the relevant bytecode instructions. Another example is the virtual method invocation. The single bytecode instruction is separated into several operations: the operation for setting each argument, 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 commoning optimization 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** shows a simple example of the translation. The direct translation of stack operations produces many redundant copy operations, as shown in Figure 2(b). We apply copy propagation and dead code elimination immediately after

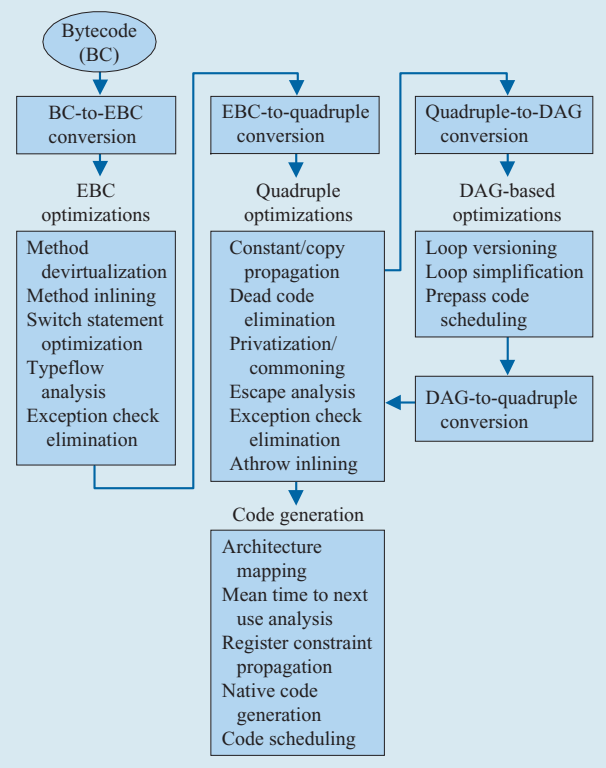

Figure 1

Structure of the JIT compiler. (DAG: directed acyclic graph. Athrow inlining: see the subsection on athrow inlining and Reference [2].)

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(c).

We apply a variety of dataflow-based optimizations on the quadruples. Some dataflow analysis, 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 in 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 (all described in the section on optimizations on quadruples).

The third IR is called a *directed acyclic graph* (DAG). (We ignore the back edges of the loop when we call the IR *acyclic*.) This is also a register-based representation and is in the form of a static single assignment (SSA) [3]. It consists of nodes corresponding to quadruple instructions and edges indicating both data dependencies

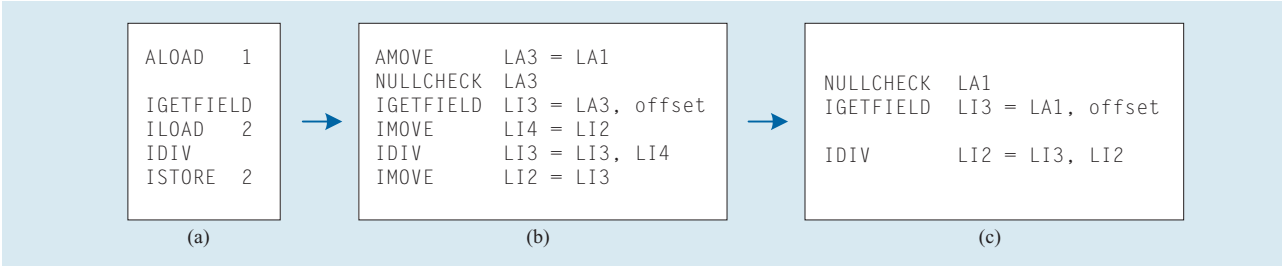

Figure 2

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

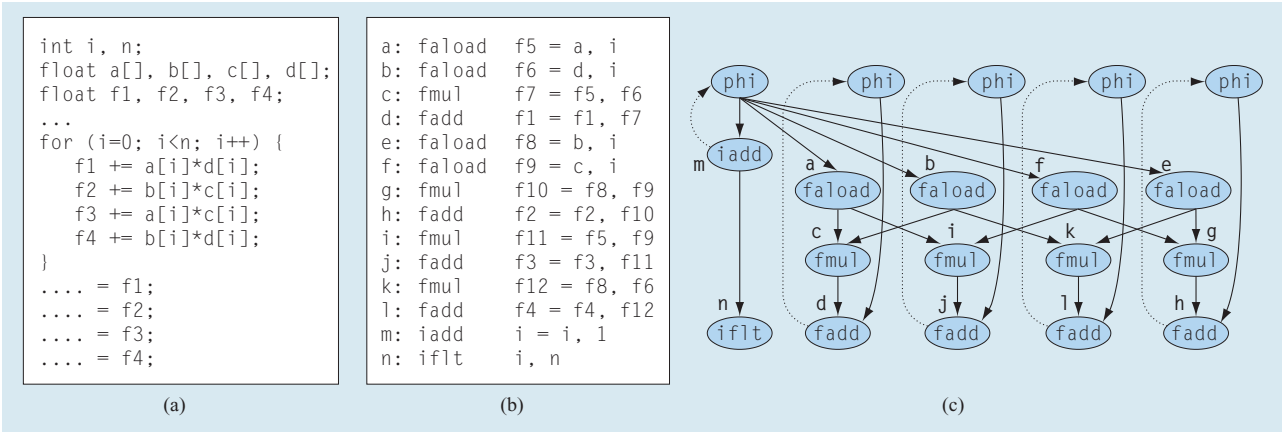

Figure 3

Example of constructing a DAG for a loop-containing method: (a) Original code; (b) quadruples for the loop body; (c) DAG for the loop body. For the sake of simplicity, this ignores all of the exception checks necessary for the array accesses within the loop.

and exception dependencies. **Figure 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 completely an optional pass, and we need to apply this set of these optimizations judiciously and for only 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 [4]. The BBs are placed using the branch frequencies collected during the MMI execution. Since the branch history information is limited to the first few executions in the MMI, we do not use code positioning guided solely by profile information [5]. 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 `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 optimizations are skipped. Method inlining can be applied either for tiny methods only or based on more aggressive

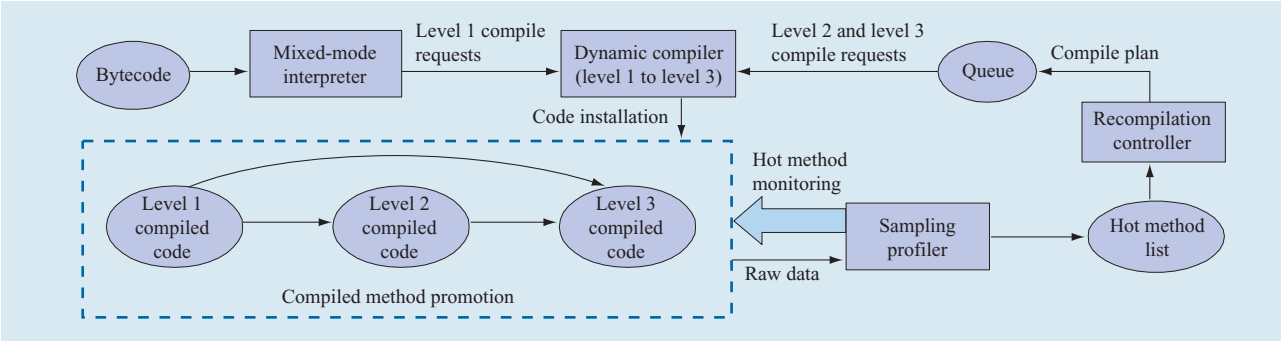

Figure 4

System architecture of our dynamic optimization framework.

static heuristics. The number of iterations on the dataflow analyses can be adjusted. The generation of a DAG representation and the optimizations on the DAG are optional. We exploit this capability in the design of the dynamic optimization framework described in the next section.

Dynamic optimization framework

Figure 4 depicts the architecture of our dynamic optimization framework[2] [6]. This is a multilevel compilation system with an MMI and three compilation levels (level 1 to level 3). All of the optimizations described in the section above on the structure of the compiler are classified into these three levels. Basically, the lightweight optimizations, such as those with linear order to the size of the target code, are applied in the earlier optimization levels, and those with higher costs in compilation time or greater code size expansion are delayed to the later optimization levels. The classification is based on the empirical measurements of both compilation cost and performance benefit of each optimization [7].

The three optimization levels in our JIT compiler are the following:

1. *Level 1 (L1) optimization* employs only a very limited set of optimizations for minimizing the compilation overhead. For the EBC, it performs devirtualization of the dynamically dispatched call sites based on class hierarchy analysis (CHA) [8]. For the resulting devirtualized call sites and static/nonvirtual call sites, it performs inlining only when the target method is tiny. Switch statement optimization is also performed. For quadruples, it enables dataflow optimizations only for very basic copy propagation and dead code elimination that eliminate the redundant operations resulting from the EBC-to-quadruple translation. No other dataflow-based or DAG-based optimizations are performed.
2. *Level 2 (L2) optimization* enhances level 1 by employing additional optimizations. For the EBC, method inlining is performed not only for tiny methods, but also for all types of methods based on static heuristics. Pre-existence analysis using the results of object typeflow analysis is performed on both the EBC and the quadruples to safely remove guard code and backup paths that resulted from devirtualization. For quadruples, it performs a set of dataflow optimizations iterated several times, but the maximum number of iterations is still limited to a low value. The other dataflow-based optimizations, such as synchronization optimization and athrow inlining, are also enabled at this level. DAG-based optimizations are not yet performed.
3. *Level 3 (L3) optimization* is augmented with all remaining optimizations available in our system. New optimizations enabled at this level include escape analysis (including stack object allocation, scalar replacement, and synchronization elimination), code scheduling, and DAG-based optimizations. It also increases the maximum iteration count for the dataflow-based optimizations on quadruples. There is no difference between L2 and L3 optimizations in terms of the scope and aggressiveness of inlining.

The compilation for L1 optimization is invoked from the MMI and is executed within the application thread, while the compilation for L2 and L3 is performed by a separate compilation thread in the background. The upgrade recompilation from L1-compiled code to higher-level optimized code is triggered on the basis of the hotness level of the compiled method as detected by a timer-based sampling profiler [9]. Depending on the relative hotness level, the method can be promoted from L1-compiled code either to L2 or directly to L3 optimized

[2] ©ACM, 2001. This subsection is from the work published in [6]. Used with permission.

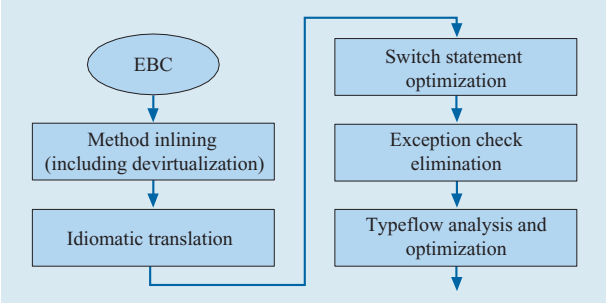

Figure 5

Sequence of optimizations on EBC.

code. This decision is made on the basis of different threshold values on the hotness level for each L2 and L3 method promotion.

The sampling profiler periodically monitors the program counters of application threads; it keeps track of methods in threads that are using the most central processing unit time by incrementing a *hotness count* associated with each method. The profiler keeps the current hot methods in a linked list, sorted by the hotness count, and then groups them together and gives them to the recompilation controller at every fixed interval for upgrade recompilation. The sampling profiler operates continuously during the entire period of program execution to effectively adapt to the behavioral changes of the program.

Platform-independent optimizations

This section provides a detailed description of the platform-independent optimization features performed on each of the three IRs.

Optimizations on EBC

Figure 5 shows the sequence of optimizations performed on EBC. 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 the compilation target. All of the optimizations on EBC following the method inlining are then performed to simplify the control flow or to eliminate redundant code so that we can reduce the target code size before converting to the second IR, quadruples.

Method inlining

Our dynamic compiler is able to inline any methods, regardless of the context of the call sites or the types of 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 runtime handling of exceptions and synchronization, we create a data structure indicating the inline context tree within the method and map each instruction or call site address that may constitute an exception with the corresponding node in the tree structure. Using this information, the runtime system can identify the associated try region identification or 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.

Many methods in Java, such as accessor methods, simply result in a small number of instructions of code when compiled; 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.[3]

Since invocation and frame allocation costs outweigh the execution costs of the method bodies for these methods, inlining them is considered completely beneficial without causing 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 in increasing compilation time, while it produces significant performance improvements over the no-inline case [10]. Thus, they are always inlined at all compilation levels. 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; then, to come up with a final tree, it looks at the total cost by checking each individual decision. 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.

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

- 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, 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 the inlining is tried for a deeper nest of call chains than for outside a loop.

Throughout the process, the inliner devirtualizes dynamically dispatched call sites using both the CHA and the typeflow analysis. It produces either guarded code (via class test or method test [11]) or unguarded code (via code patching on invalidation [12]), depending on whether the call site can be assumed to be monomorphic. When more than two target methods are found, it performs devirtualization only when the profile is available and the most beneficial target can be selected. A backup path is generated for both 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. At this point, the nonvirtual call sites in the fast path may or may not be inlined according to the static rules described above.

Idiomatic translation

This component recognizes the sequences of bytecode instructions or special method calls (primarily the *math.class* library calls), and replaces those bytecode instructions with simpler operators, mostly binary or unary operators, expressed in the EBC. For example, if the compiler finds a bytecode sequence that results from a `max` or `min` operation expressed with explicit control flow (`if-then-else`) or an arithmetic `if` operator, this is converted to a single EBC instruction with the corresponding operator. This code is generated using the `cmov` instruction for efficient execution on IA-32.

For the math. class library calls, such as the square root, ceil/floor functions, and some transcendental functions, we can exploit the IA-32 native instructions or specially prepared library calls, instead of invoking standard native methods using the Java Native Interface (JNI), if they are not specified as `strictfp` operations. Thus, we replace the sequence of bytecode instructions for these method invocations with single binary or unary operations. This significantly simplifies the control flow at compile time and also reduces the path length at execution time. In addition, we can eliminate both F2D and D2F conversions produced for single-precision operations before and after the original math library calls [13]. This is because the math libraries are provided with only double-precision versions, and thus the inefficient conversion operations are necessary when calling the library functions.

Switch statement optimization

The switch statement sometimes has a long list of case labels, and, in general, this makes the generated code inefficient because of the complicated control flow and many compare-and-branch instructions that have to be executed sequentially. In some cases, however, we can optimize the switch statement to simplify the control flow or to decrease the number of compare-and-branch instructions executed before finding the appropriate target statement. We consider two such optimizations here: transformation to a table lookup operation and the extraction of frequently executed case statements.

The transformation to a table lookup operation is made when the switch statement is used as an assignment of a different value to a single local variable according to the input parameter. **Figure 6** shows a simple example of this transformation. We first examine the structure of the switch statement to see whether it is applicable for this transformation by checking each case statement. The check includes the number of case labels and the density of case key values, as described below. If the conditions are satisfied, we introduce a compiler-generated table that holds the values as its elements corresponding to each assignment statement and convert the switch and the set of case statements into a simple table lookup operation.

As shown in the figure, the new code consists of the index variable normalization, the range checking of the index, the loading from the compiler-generated table, and the default value setting code for an index out-of-the-table range. If the default setting is not explicitly specified in the original switch statement, the conversion is in a slightly different form. That is, when the index is within the table range but turns out to be a value not specified in the case statement (through a dummy value stored in the table), we need to set the variable back to its original value before entering the switch statement.

Because of the base overhead of the index normalization and range checking, the transformation is not actually effective in a small number of case statements. Also, if the value density between the lowest and the highest keys in the case labels is not sufficiently high, the compiler-generated table may simply be a waste of memory. Thus, we check the total number of case

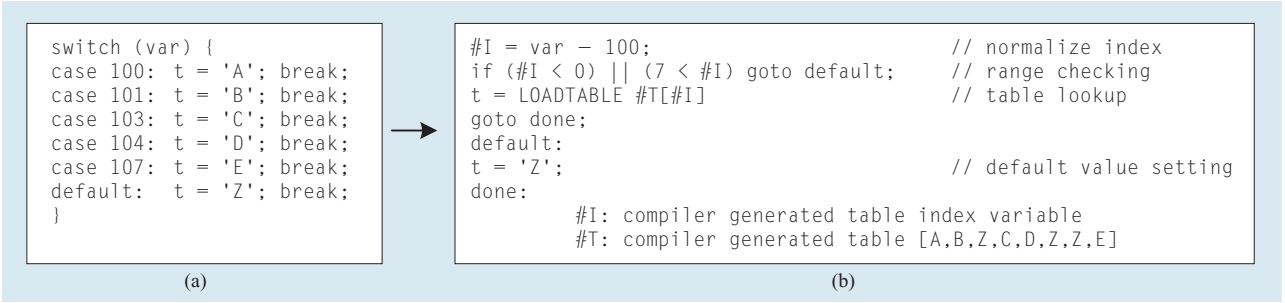

Figure 6

An example of the transformation of the switch statement into a simple operation of loading from a compiler-generated table: (a) Original switch statement; (b) pseudocode after transformation.

labels and the density of the values in the case labels before applying this transformation.

The second optimization is the extraction of frequently executed case statements. Using the execution frequencies for individual cases and default statements collected during the MMI execution, we extract some number of frequently executed cases and put explicit compare and branch instructions at the beginning of the switch statement. We compute the expected number of comparison instructions to be executed before reaching each body statement by assuming either linear search or binary search based on the number of case labels, as performed in the code generation. We then try to minimize this expected number by the case extraction. When the number of case statements is small and the frequency between the cases is clearly ordered, we replace the whole switch statement with that control flow; otherwise, the switch statement still remains after removing some number of extracted cases. For the profile collection, we limit the number of profiles both by the number of executions at the switch statement entry and by the number of executions at individual case statements in order to save the profile buffer space. We collect several different profiles for the default case, one for the input value above the highest key within the case labels, one for the value below the lowest key, and several others between two adjacent keys within the range. On the basis of these profiles, we apply the same transformation for the default case.

Exception check elimination

The redundant exception check elimination (both null check and array bound check) in the EBC level is based on simple forward dataflow. The dataflow analysis propagates both non-null and range expressions to determine where they must be checked and where they can be skipped. Since an exception is not explicitly represented in the EBC, all results of the exception check redundancy are marked as attributes in the relevant instructions. When the EBC is converted to quadruples, the explicit exception checking code is generated only for those without the attribute, and thus the size of the quadruples can be reduced.

Typeflow analysis

We perform typeflow analysis at various phases in the optimization process to infer the static type of each object reference. In the method inlining, we used the results of typeflow analysis for determining the types of receiver objects of virtual method invocations to increase the opportunities of devirtualization. At this phase, we use the results of typeflow analysis for three optimizations: elimination of backup paths of devirtualized code, elimination of redundant type checking code, and conversion of the native method call `System.arraycopy( )` to a special operator instruction.

We exploit the "preexistence" [11] to safely eliminate devirtualized-call-site backup paths without requiring a complex on-stack replacement technique. The preexistence is a property of virtual call receiver objects in which the receiver is preallocated before the calling method is invoked. If the receiver for a devirtualized call site has this property and the devirtualization is made for the target method of a single implementation, we can eliminate the backup path without causing incorrect behavior because we can guarantee that the single implementation assumption is true on entry of the calling method. Even if another implementation of the target method is loaded because of the dynamic class loading later, we can modify the affected method to be recompiled at its entry before making the loaded class available. To prove preexistence, we perform an additional argument analysis to check whether the receiver object of each devirtualized call site is directly reachable from an

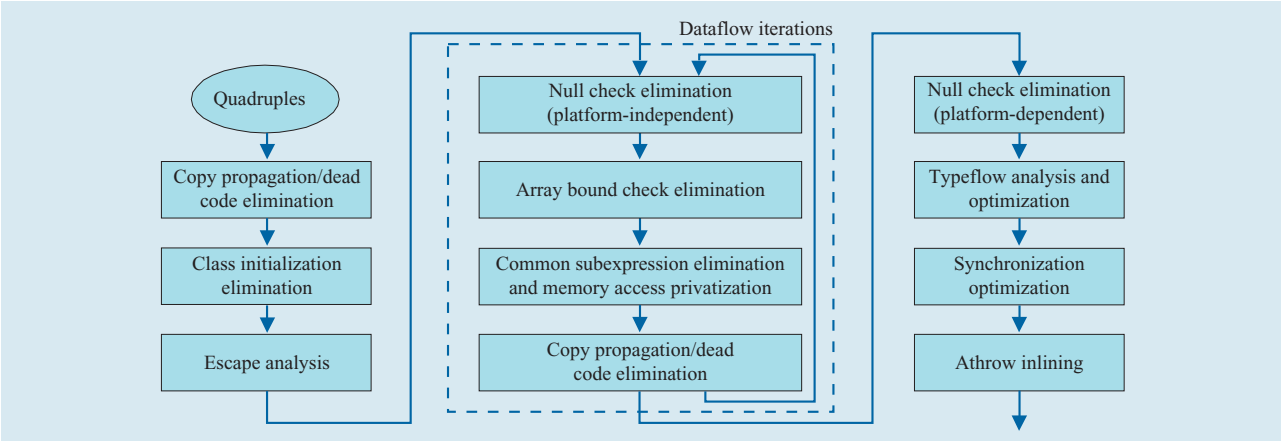

Figure 7

Sequence of performance optimizations on quadruples.

argument of the calling method. This optimization results in eliminating the merge points from virtual call sites in the control flow and thus can improve the precision of dataflow analyses in the later phases.

We also perform the elimination of redundant type checking code using the typeflow analysis. The type checking code examines whether two object references are related by a subclass relationship and is required for `checkcast`, `instanceof`, and `aastore` bytecode instructions. By propagating the type information along the forward dataflow from several seed instructions—such as object allocation, argument specification, and instance checking code—we can identify and eliminate redundant type checking operations in some cases. For example, the following statement is quite common in practice, and the second instance checking for the `checkcast` instruction is eliminated by propagating the type information already checked:

```
If (anObj instanceof aClass) {
       aClass ao = (aClass) anObj;
}
```

Another optimization is for the special method call, `System.arraycopy`( ). This is one of the most frequently called native methods and takes five arguments: the source array and its start index, the destination array and its start index, and the number of elements to be copied. If we know from the results of the typeflow analysis that an array store exception cannot be raised for the given pair of arrays, we convert the method call to a single EBC operator. The typeflow analysis also provides the information that the source and destination arrays are not aliased on the basis of the corresponding seed instruction. Additional attributes for this operator, such as the length of the array, the start index, and the number of copy elements, may be added by the optimizations of constant propagation and copy propagation performed in the quadruple optimization phase. At the code generation phase, depending on the information available for the operator, we inline the code by using the special instruction `movs` with the repeat prefix for IA-32. In the best case, the inlined code is quite simple because we know that the range of the copy operation is guaranteed to have no overlap and the array bound checking code is not necessary.

Optimizations on quadruples

We perform a variety of dataflow-based optimizations on quadruples. **Figure 7** shows the sequence of optimizations performed on quadruples. In the following, we describe each optimization in detail.

Copy propagation and dead code elimination

These optimizations are designated to eliminate the redundant register-to-register copy operations in the code directly translated from the stack-based EBC, as described above in the section on the structure of the JIT compiler. The optimizations are applied globally across basic blocks.

Class initialization elimination

Since we separate the resolution of a class from the execution of its class initializer, we can try to resolve a class at compile time without violating the language specification. For a resolved but not yet initialized class access, we explicitly generate a quadruple operator for class initialization, because we have to execute the class initializer (i.e., ⟨`clinit`⟩) once at every first access for each class. The class initialization is a method invocation

and thus has to be treated as a barrier in the following exception check elimination and memory access privatization phases. We attempt to eliminate the redundant class initializations at this stage using simple forward dataflow analysis.

Escape analysis

Escape analysis examines each dynamically allocated object and deduces the bounds on the lifetimes of those objects to perform optimizations. If all uses of the allocated object are within the current method, we can allocate the object on the stack to improve both the allocation and reclamation costs of memory management. If the object is proved to be nonescaping from the current thread, we can eliminate any synchronization operations on this object.

Our escape analysis is based on the algorithm described in [14]. This is a compositional analysis designed to analyze each method independently and to produce summary information that can be used at every call site that may invoke the method. Without the summary information, the analysis treats all arguments as escaping at the given call sites. Hence, the analysis result can be more precise and complete, since more of the invoked methods are analyzed. However, if backup paths for devirtualized call sites have been eliminated by the pre-existence analysis, we suppress generation of the summary information. This is because the escape analysis will be based on the optimistic assumption without considering those potentially escaping paths eliminated by the pre-existence analysis. In that case, we may incorrectly conclude in some of its caller methods that the object passed as a parameter is nonescaping by using the summary information. When dynamic class loading invalidates the pre-existence assumption, we will then have to recompile not only the current method, but all caller methods that have used the summary information as well.

The stack allocation of an object basically holds the space of the whole object, including the object header, within the stack frame. Since it may increase memory consumption by extending the lifetime of objects until the method returns and the stack is rolled back, we perform the stack allocation only when the allocation site is not within a loop and the size of the allocated objects (both individually and collectively) is within a preset threshold value.

When the only uses of an object are to read and write fields of the object, we can perform scalar replacement for those field variables instead of allocating the entire object. For example, if an object is passed as an argument to other methods or is used in type checking instructions, it is not eligible for scalar replacement. For this transformation, we introduce a new local variable for each of the object fields and simply replace the allocation instruction with a set of initialization instructions for those new locals. Each reference to the object is also replaced with a simple copy instruction from or into the new local variable. The computational redundancies with this transformation can be eliminated in the subsequent phase of copy propagation and dead code elimination.

Dataflow iterations

We perform exception check elimination, common subexpression elimination, and memory access privatization iteratively using the partial redundancy elimination (PRE) technique [15, 16]. Within each iteration, the null check elimination first moves null checks backward and eliminates redundant null checks within a loop. This optimization increases the opportunities for the following array bound checking elimination, since the bound checking code is now allowed to move in a wider range in the program beyond the original place of the null checking code. After the exceptions are eliminated from a loop, we apply common subexpression elimination and privatize memory (instance, class, and array variable) access operations. Again, these optimizations should be effective here because many of the barriers for code motion were eliminated in the previous steps. The common subexpression elimination and the memory access privatization can, in turn, expose new opportunities for the exception elimination optimization, so we iterate these dataflow optimizations several times.

Figure 8 shows an example in which these optimizations in the dataflow iteration can affect each other successively to produce better code. The original program (a) is converted to the quadruple representation (b). With a simple approach of eliminating redundant exceptions using forward dataflow, the first null check for the array a in (b) cannot be lifted out of the loop, since the exception has not been checked in the code reaching to the entry of the loop. Our algorithm uses PRE and moves this null check out of the loop, as shown in (c). This creates an opportunity in (d) to move the bound checking code out of the inner loop, which in turn enables privatization to be applied for the loop invariant array access in the first dimension, as shown in (e). This again creates another opportunity (f) for elevating the null checking code out of the inner loop.

We assign different values for the maximum number of iterations of these dataflow optimizations depending on the platform (i.e., the number of available registers), optimization level, and whether or not the method contains a loop. For IA-32, we chose four iterations as the maximum case when using the highest (L3) optimization and for a method containing a loop. We adjust this number for the lower-level optimizations and for a method without a loop. In any case, the specified number is a limit

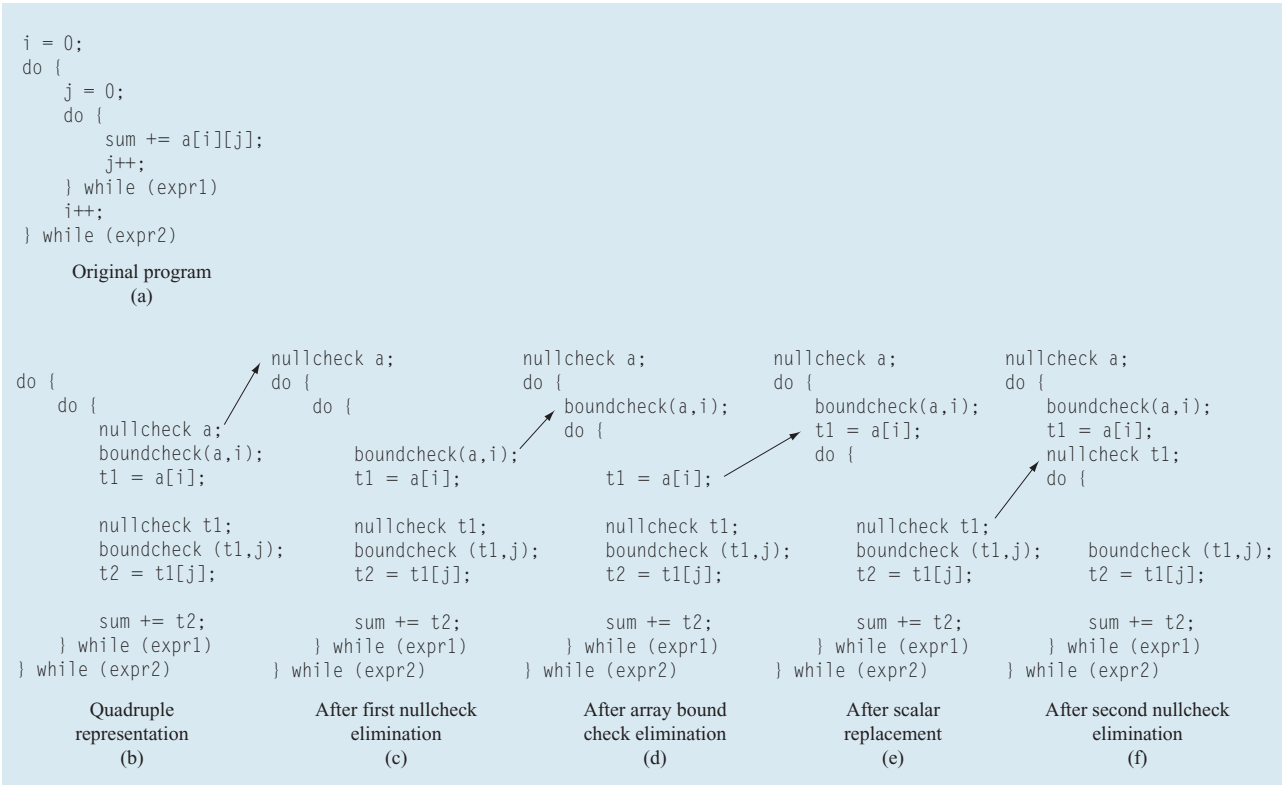

Figure 8

An example of successive optimizations with dataflow iterations. Note that for simplicity the code for initializing and incrementing loop indices in the original program is not shown in (b) to (f) . This is a minor revision of the figure published in [15]. ©ACM, 2000; used with permission.

for the iterations. We also stop the iterations if an iteration produces no new transformations.

Null check elimination (platform-dependent)

This is the platform-dependent part of our null check elimination algorithm [15]. On IA-32 Windows and Linux**, we can utilize the hardware trap mechanism for accessing the zero page. We call the null check *implicit* when we can rely on the hardware trap without generating actual checking code, and *explicit* otherwise. We need to use an implicit null check wherever possible to minimize the overhead of null exception checks.

Our algorithm treats all null checks as explicit at first. We then perform forward dataflow analysis to compute the latest point in the region at which each null check can be moved forward. We then insert either an explicit or implicit null check at that point, depending on whether the next instruction following the insertion point will raise a hardware trap for the object access. Finally, we perform backward dataflow analysis to eliminate redundant explicit null checks. This optimization substitutes explicit null checks wherever possible by finding and using the memory accessing instructions that raise the hardware traps.

Typeflow analysis

The typeflow analysis and optimizations performed on quadruples are basically the same as those performed on the EBC. That is, the results of the analysis are used for the elimination of devirtualized code backup paths, the elimination of the redundant type checking code, and the conversion of the calls to the native method `System.arraycopy( )` into the internal operator. We perform the typeflow analysis and the same set of optimizations here again to identify the type of variables more precisely than for the EBC. This is because other optimizations—in particular, copy propagation and memory access privatization—allow better dataflow propagation and thus expose more opportunities for the three optimizations using the result of the typeflow analysis.

Synchronization optimization

Since we explicitly express the synchronization enter and exit operations (`syncenter` and `syncexit`) in our IR with its target object, it is easy to identify the opportunities for eliminating redundant synchronization operations. There are two scenarios in which we can

perform the optimization. One is a nested synchronization, that is, a `syncenter` operation on the same object twice followed by two `syncexit` operations, where we can remove the inner pair of the synchronization. The other is two separate synchronization regions, such as a pair of `syncenter` and `syncexit` operations on one object followed by another pair of `syncenter` and `syncexit` operations on the same object. There may be some other code between the two regions. In this case, we have an opportunity to reduce the number of synchronization operations by extending the scope of synchronization and creating a single larger synchronization region.

This second optimization requires somewhat careful consideration to guarantee the correct program behavior. Suppose there is an object field access in the code between the two synchronization regions. If we perform this optimization, the code of the field access can be moved within the new wide synchronization region and may be reordered with other memory operations on different objects that were placed within either of the two original synchronization regions. If the program assumes the order of computation forced by the two original synchronization regions, this optimization can lead to an unintended behavior. Thus, we perform this optimization only if there is no quad instruction causing any side effect, such as an object write instruction and a method invocation between the two synchronization regions.

Another possibility for optimization is to lift the synchronization operation out of the loop. This situation occurs when a synchronized method is inlined within a loop. This optimization is efficient and probably legal in Java, but the resulting behavior is not what most programmers would expect. If the loop is long-running, other threads trying to access that method will be starved. However, most programmers writing such a loop would expect the lock to be released and reacquired each time through the loop to allow any other thread to have a chance to execute. Thus, we currently do not perform this optimization on synchronization.

Athrow inlining

This is a part of exception-directed optimization (EDO) [2], which is a technique to identify and optimize frequently raised exception paths. It first collects exception path profiles. An exception path is a stack trace from the method that throws an exception through a method that catches the exception. It is expressed with the sequence of a pair of the method and the program counter of the call sites. Whenever the runtime library traverses the stack to find the method that catches the given exception, it registers the exception path and increments the count. After identifying a particular exception path that is executed frequently enough, EDO drives recompilation for the exception-catching method with a request to inline all of the methods along the call sequence corresponding to the exception path down to the exception-throwing method.

After exception paths are inlined, the compiler examines each athrow instruction that can be eliminated by linking the source and destination of the exception paths within the same compilation scope. This consists of three steps. For the first step, we perform exception class resolution, in which we identify the class of every possible exception object that can be thrown at each athrow instruction using the typeflow analysis. In the simplest case, the exception object is explicitly created immediately before the athrow, and thus the class can easily be identified. In a more challenging case, the exception object is created and saved in a class field and is used repeatedly by loading the class variable before the athrow. Even in this case, we can determine the set of possible classes using the current class hierarchy structure.

For the second step, we perform the exception-handling analysis within the current method. When we identify the exact class of the exception object, we can find the corresponding catch block. When we determine the multiple exception classes under a class hierarchy in the previous step, we find the corresponding catch block for each of the possible exception classes.

For the third step, we perform the conversion of the athrow instruction to either an unconditional or a conditional branch instruction, depending on whether or not the exact class of the exception object has been identified. If necessary, to create a separate entry point for the branch instruction, the entry basic block for the target catch block is divided into two: one for loading the exception object and the other for the body of the catch block. The athrow instruction itself may remain in a branch of a conditional statement if we cannot find the catch block for all possible exception classes in the current method. After the conversion, the exception object may no longer be accessible if there is no reference within the catch block. In this case, the explicit object allocation before raising the exception will be eliminated in a subsequent phase of dead code elimination.

Optimizations on DAGs

The sequence of optimizations performed on DAGs is shown in Figure 1.

Loop optimizations

We perform two kinds of loop optimizations: loop versioning and loop simplification. Loop versioning [1, 17] is a technique to eliminate the exception-checking code within a loop by creating two versions of the loop: an optimized loop and an unoptimized loop. Guard code is provided at the loop entry for checking the possibility of raising an exception within the loop, and—depending on

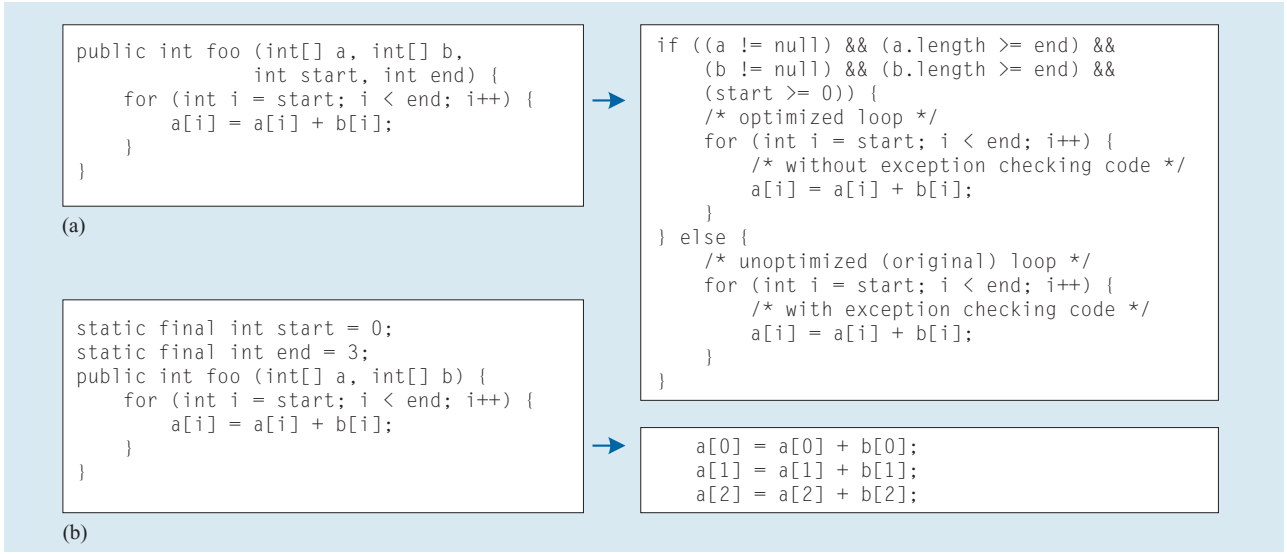

Figure 9

An example of (a) loop versioning and (b) loop simplification.

the result of the entry check—either the optimized loop or the unoptimized loop is selected at runtime. The guard code is provided for both null pointer checks and the array bound checks for the entire range of the loop index. The exception-checking code is eliminated in the optimized loop, while the unoptimized loop retains the exception-checking code. **Figure 9(a)** shows an example of this transformation. We can expect the optimized loop to be executed for most of the time, since exceptions are rarely thrown in practice.

For nested loops, we apply loop versioning to both outer and inner loops by creating only two versions of the loop: the optimized version for both outer and inner loops and the unoptimized version for both loops. The unoptimized version is executed upon any failure of the entry tests for both the outer loop and the inner loop. Otherwise, the optimized loop is executed.

In the optimized loop, this optimization not only eliminates the runtime overhead of the exception checking code, but also eliminates barriers for code motion and thus creates more optimization opportunities in the later phases. On the other hand, code growth is the major drawback of this transformation. In order to avoid an unacceptable level of code growth and to limit the compilation overhead, we check the number of basic blocks and instructions within the loop body, and proceed only if it is within a predetermined threshold.

Loop simplification is another optimization we perform in this phase. As shown in **Figure 9(b)**, this transformation unfolds the loop completely if we know the start and end of the loop index at compile time. Again, code growth is the main limiting factor of this transformation, and we check the size of the loop body before its application.

Prepass code scheduling

Finally, we perform prepass code scheduling within each basic block using a simple list-scheduling algorithm [18]. We integrated register minimization into list scheduling in order to avoid the pass-ordering problem between the register allocation and scheduling. The prepass scheduler traverses the DAG in its topological order. It selects the node that has the highest scheduling priority among ready nodes and appends the selected node to the end of the sequence of scheduled nodes. During scheduling, the scheduler also maintains a set of currently used registers.

Our algorithm uses two different scheduling policies: maximization of instruction-level parallelism (ILP) and minimization of the number of used registers. When there are plenty of available registers on the target architecture, the scheduler prefers a node on the critical path of the DAG to maximize the parallelism. When the number of available registers falls below a certain threshold, the scheduler then invokes the second list-scheduling routine to minimize the register usage; that is, it prefers the node that will release the largest number of registers. In the final phase of the scheduling, we reduce the number of local variables by assigning the same variable when the lifetime of the two variables does not overlap.

Figure 10 shows quadruples after we assigned pseudoregisters in the example shown in Figure 3. We denote an integer register as R*n* and a floating-point

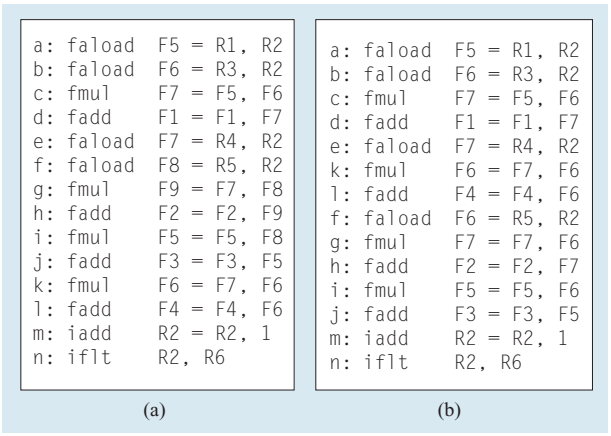

Figure 10

An example of prepass code scheduling. (a) Instruction ordering with maximum instruction-level parallelism. (b) Instruction ordering with minimal register usage.

register as F*n*. Figure 10(a) shows the result with the original instruction ordering. This ordering maximizes ILP and requires nine floating-point registers. Figure 10(b) shows the result when we applied prepass scheduling with the minimal register usage policy. After the node e is scheduled, the ready nodes are node f and node k. While node f requires another floating-point register to hold the loaded value, node k does not increase the number of used registers. Thus, the scheduler selects node k after node e. In this case, the resulting instruction ordering requires only seven floating-point registers.

Platform-specific optimizations

This section describes IA-32-specific optimizations implemented in the code generation phase. The sequence of optimizations is shown in Figure 1. First the architecture mapping takes the platform-independent IR (quadruples) and converts it to a form suitable for the target architecture. For IA-32, we have two phases. One is to convert it from a three-operand format to a two-operand format as required by the architecture, except for the case in which we can exploit the lea instruction for three-operand addition and subtraction operations. We insert an additional copy instruction for each statement to be converted. The other phase is to specify where we can use memory operand instructions in the code generation. For each memory load instruction, we examine three conditions:

1. Whether the corresponding use for the loaded value is single.
2. Whether the instruction using the value allows the memory operand format.
3. Whether the source operands for the load instruction are all live at the corresponding use.

If these conditions are met, we mark both the definition and the use of the variable with an attribute indicating that the load instruction can be skipped by using the memory operand instruction at the corresponding use statement. A typical example is an array bound check, which usually consists of loading the array size followed by comparing it with a given index. If the size of the array is not used after the bound check, both the definition and the use of the array size variable are marked as a memory operand.

After the architecture mapping, we perform a set of optimizations for efficient usage of the registers. Since we do on-the-fly local register allocation during the code generation phase for IA-32,[4] it is important to provide the register manager with a global view regarding the use of registers in order to avoid inefficient register shuffling or spilling code. The next sections show how the decision for spill candidate is made by the register manager, describe how the register preference is used for allocating registers, and describe our technique in the code generation phase for increasing the number of available registers.

Mean-time-to-next-use analysis

When no register is available, the register manager has to decide which register should be spilled out into memory. With the stack-based IR (EBC), as in the previous version of the compiler, we first searched for the appropriate one from the local variable registers and then from the stack variable registers, on the basis of the assumption that stack variables are generally short-lived and thus more likely to have a shorter interval to the next reference. With the register-based IR (quadruples), the register manager has to know how soon each of the local variables is accessed next from the current instruction in order to make the right decision for spill-out registers. We call this information *the mean time to next use*.

We compute the mean time value as the number of quadruple instructions between each definition and reference of the local variables by using the backward dataflow analysis. In the local analysis, the local propagator initializes a counter whenever it encounters a reference to a new local variable. It increments all of the active counters as it scans each instruction in the quadruples. When it reaches a local variable reference with an active counter, it updates the mean time value of the operand with the counter value. The counter is cleared

[4] We use different register allocation schemes depending on the target platform: a register manager for IA-32, a linear scan allocator [19] for PPC, and a graph-coloring allocator [20, 21] for IA-64. This is based on the consideration of both the number of available registers and the restrictions on register usage for each platform.

at its definition point. In the global analysis, the results from local analysis are combined by selecting the smallest mean time value. In the case of loop back edges or edges with biased branch frequencies, the local result on a particular single basic block is selected. We iterate the local and global dataflow analysis several times with a certain limit on the mean time value.

In the code generation phase, the register manager uses the mean time value by updating its internal register table. It copies the value from the operand for the variable defined or referenced in the current IR instruction, or otherwise decrements the value within the table for the variables not currently referenced. Thus, the register manager keeps the mean time value up to date in the register table throughout the code generation and uses the information for making spill decisions. In addition, when a local variable reference is in its last use, the mean time value in the operand is zero, and the register manager can invalidate the corresponding register immediately after its code generation and put it in the list of available registers.

Register constraint propagation

There are constraints for the use of registers for some IA-32 instructions; that is, a particular operand must reside in a designated register. For example, the shift instructions require `ecx` to be the second operand. The integer division instructions are another example. Also, only four of the eight general-purpose registers can hold 8-bit values. We have to satisfy these requirements during code generation by moving values between registers or between registers and memory, if necessary. Traditionally, the register-coalescing technique [22] in the graph-coloring register allocation addressed this problem by merging nodes representing local variables and physical registers in order to eliminate redundant moves. The register constraint problem was also discussed in the linear scan register allocation, where register-preallocated live ranges [19] were used. We use the register constraint propagation to help the register manager assign the optimal registers [23].

Since the register manager works from the top to the bottom of the program, registers are assigned to local variables at their definition and released at their last use. To track the register constraint information for the register manager, we first add the register constraint seed information to the source operands of the given quadruples and then propagate the information backward for the corresponding definitions. We currently use the following as the seeds of the register usage constraints:

- *Calling convention:* We follow the calling convention to use `eax`, `edx`, and `ecx` for the first three integer parameters (long parameters are treated as two integer parameters) for method invocations. The return of integer-type values is through `eax` and the return of long-type values is through `eax` and `edx`.
- *IA-32 architectural requirements:* The integer division, integer multiplication, and shift instructions require specific registers to be used for some operands. The compare-and-swap instruction, which we use for generating code for the synchronization operation, requires that its operands reside in specific registers. Instructions dealing with 8-bit values for byte-type operations (e.g., `baload`) allow use of only four of the eight general-purpose registers.
- *Preference for variables referenced across method calls:* For local variables whose live ranges cross a method call or a C-runtime library call, it is preferable to assign nonvolatile registers to reduce the cost of saving and restoring registers over the call. In contrast to the above two requirements, which we have to satisfy in the code generation, this is just a preference.

We use a bit vector representation with the number of available registers to express the register constraint information. The constraint expression can be either positive or negative; that is, we can specify that a local variable be assigned with a certain specific register, or *not* be assigned with specific registers. This negative specification is important to avoid a situation in which the specified register is occupied by another variable whose live range overlaps the live range of the target variable.

Second, we propagate the seed information within each basic block. The local propagator manipulates the array of the register constraint bit vector for local variables and a temporary called the currently occupied register bits (CORB) as it scans the code from the bottom to the top. If it encounters a local variable reference with a positive register constraint, it updates the array for the local variable with the bits indicating the specified registers and then clears the bits for all of the other local variables. The CORB status also reflects the corresponding register bits. Likewise, if it encounters a local variable reference with a negative constraint, it updates the array for the local variable with the inverse bits of the CORB. If it reaches a local variable definition, it clears all of the bits in the array for the local variable and the corresponding bits in the CORB. **Figure 11** shows an example of the local constraint propagation. The register constraints, both positive and negative, are expressed in the bit vectors as shown in the figure. Each bar represents the live range of a local variable within a basic block, and two of them have seed information at the reference points. After the local propagation, each live range has the propagated register constraint information at its definition point.

Finally, the local information is propagated across the entire method using backward dataflow. The aggregation

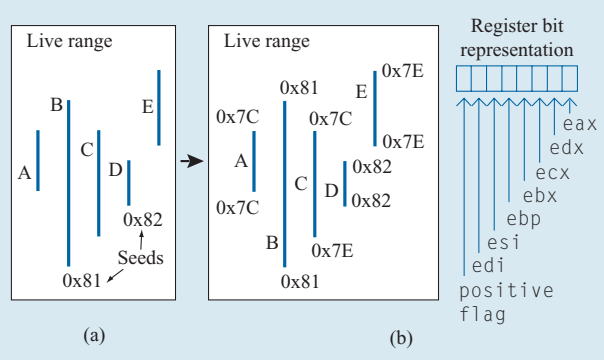

Figure 11

An example of local constraint propagation (a) before and (b) after local propagation. Each bar represents the live range of a local variable associated with a register constraint value.

function in the dataflow equation is a bitwise operation between the constraints of successors for each local variable, as follows:

- If the constraints of the successors are all positive, perform a bitwise `OR` operation.
- If the constraints of the successors are all negative, perform a bitwise `AND` operation.
- If the constraints of successors are mixed positive and negative, perform a bitwise `AND` operation and set the positive flag.

We iterate the local and global dataflow analysis several times. In our current implementation, we iterate as many times as the maximum depth of the loop nest existing in the method.

Code generation

Since the number of registers is limited on IA-32, we must use these scarce resources efficiently to achieve high performance. We use three techniques: reclamation of the Frame Pointer register, fast accesses to the thread local storage without using a dedicated register, and management of two different sets of floating-point registers.

Frame Pointer register

We reclaimed the Frame Pointer register and freed up `ebp` to be used as a general-purpose register. All of the local variables in memory are accessed through the stack pointer. The method prologue and epilogue code become simpler and more efficient just by increasing or decreasing the stack pointer with the current frame size. If there is no stack memory operation within the method, we can even skip creating the stack frame, saving the two instructions for sliding the stack pointer. We reorder local variables that require spill code within the stack frame on the basis of the access count, so frequently accessed variables are located within the 1-byte offset from the stack pointer.

When a hardware trap or a software exception is raised at runtime, the exception handler and the stack walker must know the frame base address in order to unwind the chain of stack frames. We create a map for the runtime to indicate the current stack offset from the base for each instruction. Throughout code generation, we maintain the current offset value of the stack pointer, and at the end of the compilation we create the mapping table for every instruction in the method that potentially raises an exception and where the stack offset value is changed. Each entry in the map includes a pair of the stack offset value and the corresponding instruction address. Since our frame size is basically fixed and varies only when the arguments are pushed for method calls or C-runtime calls, the map can be kept reasonably small. With an appropriate compression technique, the map size is kept within 10% of the generated code size for most methods.

The MMI- and JIT-generated code share the same execution stack in our system. Since the MMI uses `ebp` as a frame pointer, the exception handler first identifies which type of frame, MMI or JIT, is currently on the stack, and then extracts the frame base address accordingly.

Thread local storage pointer

Instead of having a dedicated register as a pointer to the JVM/JIT thread local storage, we use the FS segment register (one of the four segment registers for pointing to data segments in IA-32 architectures). The segment register is usually reserved for use by the operating system for its own purposes and cannot be used arbitrarily by applications. Microsoft Windows** maintains a thread information block (TIB) structure for every thread by the FS segment register. The first entry in this structure is a pointer that contains the head of the exception-handler chain. We create an exception-handler record when starting the execution for each thread and register with the system by setting the pointer to the offset 0 into the segment pointed to by the FS register. Our JIT compiler then uses one entry in this structure as the pointer to our own thread local storage. Thus, we can obtain the thread local storage address through one indirection of memory access.

On Linux, the FS segment register is not reserved for use for any particular purpose, and thus any application can use the segment register for improving its own performance, which means that using the segment register in our JIT compiler can cause an unexpected conflict with other applications. Instead, we use the topmost entry of

the stack space for the pointer to the thread local storage. The entry can be accessed by masking the stack pointer so that we can still reach the thread local storage through one indirection, as in the Windows implementation.

Floating-point registers

We use streaming single-instruction multiple-data (SIMD) extensions (SSE and SSE2) when they are supported by the underlying machine and the operating system. We can generate more efficient code using these new instructions, first because the associated XMM registers are designed to be flat, in contrast to the existing x87 floating-point stack registers; second, because the instructions are divided between single-precision and double-precision operations, we do not have to do extra precision control operations. On the other hand, we have other kinds of challenges for register allocation and code generation, as follows:

- SSE/SSE2 instructions do not support all of the existing floating-point functions (such as the remainder operation and transcendental functions) using the x87 registers, and we have to move values from one register set to another for those operations.
- In some operations, such as the conversion from a floating-point number to an integer number, SSE/SSE2 instructions are much faster than x87 instructions. We want to generate the faster code, even if we pay extra for memory operations that transfer values between registers.
- When there are more than eight live floating-point variables, it is better to use both register sets rather than sticking to one register set and generating memory operations for the extra variables. Thus, we have to deal with two kinds of instructions and registers for efficient floating-point code generation.

We follow lazy migration policy for transferring values between the two sets of registers. When the source operands are in only one of the sets of registers, we use the instructions available for those registers, except for those operations having constraints or special preferences, as mentioned above. When the source operands are mixed between the two register sets, we basically move values to XMM registers, if available, and generate SSE/SSE2 instructions.

It is possible to apply the register constraint back propagation, as we did for integer registers. We can assign seed registers for those operations with the register constraints and preferences, and if we choose to pass floating-point parameters via XMM registers, we can include parameter-setting operations as well. Currently we do not do this, but the constraint propagation would provide useful information for avoiding inefficient register move operations, since we cannot directly transfer values between the two sets of registers, but have to use expensive memory operations.

Experimental results

This section presents some experimental results. We first outline our experimental methodology and then present and discuss our results for performance improvement and compilation overhead of individual optimizations and for the effectiveness of our dynamic optimization framework. The last two sections below present the history of performance improvement with each release of our JIT compiler.

Benchmarking methodology

All of the performance results presented in this section were obtained on an IBM IntelliStation* M Pro 6850 (Intel Xeon** 2.8-GHz uniprocessor with 1024-MB memory), running Windows XP SP1. We used the JVM of the IBM DK for Windows, Java Technology Edition Version 1.4.0 prototype build, except for the results reported in the section on the evolution of performance improvement, where we used each version of the JVM/JIT combination. We used SPECjvm**98 and SPECjbb**2000 [24] as common benchmarks for evaluating our system in all of the experiments. SPECjvm98 was run in the test mode with the default large input size, and in the autorun mode with ten executions for each test. Each distinct test was run with a separate JVM with the initial and maximum heap size of 128 MB. SPECjbb2000 was run in the fully compliant mode with one to eight warehouses, with the initial and maximum heap size of 256 MB, and we reported the best scores among these runs. To evaluate our dynamic optimization framework in more detail, we also present results for larger benchmarks—Java Grande [25], XSLTMark [26], and SPECjAppServer**2002 [24]—in the section on evaluation of the dynamic optimization framework. The system configuration and parameter settings for measuring these benchmarks are described in that section.

When measuring the compilation overhead, we use a separate thread for compiling all methods. The compiler is instrumented with several hooks for each part of the optimization process to record the processor timestamp counter value. The priority of the compile thread is set higher than the priority of the normal application threads. We measure the time difference between the beginning and the end of the compilation from the timestamp counter.

Evaluation of individual optimizations

We first evaluate how much benefit each optimization brings to the system at what compilation cost. We disable

Table 1 Variations for evaluating cost and benefit of individual optimizations. These correspond to the bars in Figure 12 from left to right.

Abbreviation	*Description*
No-AggrInline	This disables all of the static heuristics for aggressive inlining. Tiny methods are always inlined.
No-IdiomSwitch	This disables both the idiomatic translation and the switch optimizations.
No-ExceptionCheck	This disables the exception check (null check and array bound check) elimination at the EBC level.
No-Escape	This disables escape analysis. No scalar replacement or stack object allocation is performed.
No-DataflowIter	This disables the iteration of dataflow analyses performed at the quadruple level. It also disables the platform-dependent nullcheck elimination.
No-Typeflow	This disables the typeflow analysis performed at both the EBC and quadruple levels. The optimizations using this analysis result, backup path elimination, redundant type checking elimination, and arraycopy JNI call conversions, are not performed.
No-DAGOpt	This disables the DAG-based optimizations (loop optimizations and prepass code scheduling). The IR conversions from quadruples to DAGs and then back to quadruples are not performed.
No-RegOpt	This disables optimizations performed in the code generation phase, that is, mean-time-to-next-use analysis, register constraint propagation, the reclamation of the frame pointer register, and the use of XMM registers.

each optimization individually and examine how much impact it has on both performance and compilation overhead. We use a single optimization level for this measurement, instead of using the three-optimization-level recompilation system, to get a clear picture of the cost and benefit of each optimization before classifying them into the three optimization levels. We evaluated the eight variations shown in **Table 1**. The baseline of the comparison is the configuration with all optimizations enabled within the system.

Figure 12 shows changes in both performance and compilation overhead when each optimization is disabled. We use three metrics for compilation overhead: compilation time, compiled code size, and compilation time peak work memory usage. A smaller bar indicates a larger impact from that optimization for the system. The peak memory usage is the maximum amount of memory allocated for compiling methods. Our memory management routine allocates and frees memory in 1-MB blocks.

There are several observations for these measurement results. First, the static-heuristics-based inlining (leftmost bar) contributes an average of 10% performance improvement but causes a large compilation overhead. Without the aggressive inlining, the overhead is halved or reduced to less than half for all three metrics. In particular, the compilation time in *jess*, *db*, the code size in *jess*, and the work memory usage in *db*, *javac*, and *SPECjbb* are all reduced by more than 60%. The performance impact for *jack* is larger than that for the others because EDO cannot effectively optimize frequently raised exception paths without performing aggressive inlining.

Second, the DAG-based optimization (second bar from the right) shows the next largest compilation overhead. This occupies an average of 20 to 30% of the total compilation overhead, depending on the metric. In particular, this creates large overheads for *db* and *SPECjbb* in both compilation time and work memory usage. Furthermore, the performance contribution of this optimization is limited to only a few programs, such as *mpegaudio*. We have to apply this expensive optimization very carefully.

Third, the largest performance contributor is the dataflow iteration (fourth bar from the right). It improves the performance by an average of 20% and up to 50% for *mpegaudio*. The kernel of this benchmark is a heavily iterated, doubly nested loop that accesses several two-dimensional arrays. The null check elimination, array bound check elimination, common subexpression elimination, and memory access privatization are all effective for optimizing this kernel loop. The compilation time overhead with this optimization is around 10 to 20%, except for *mpegaudio*, for which the compilation time is significantly increased with the dataflow iterations turned off. This anomaly is due to DAG-based optimization, by which compilation time is increased more than fourfold. This is because the implementation of DAG-based common subexpression elimination is not very efficient, but it is invoked when the quadruple-based common subexpression elimination within the dataflow iterations is disabled.

Fourth, the typeflow analysis (third bar from the right) contributes to both performance improvement and reduction of compilation overhead. In particular, it has a large impact on *mtrt*, nearly 10% for performance and 20 to 40% for

Figure 12

Evaluation of individual optimizations for both performance impact in (a) and compilation overhead in (b) to (d). Each bar indicates the relative number against the baseline, the configuration with all optimizations enabled. SPECjvm**98 consists of seven benchmarks: *mtrt*, *jess*, *compress*, *db*, *mpegaudio*, *jack*, *and javac*. G.M. = geometric mean.

compilation overhead, primarily by enabling the elimination of devirtualized code backup paths. This optimization consistently contributes to many programs other than *mtrt* without causing any extra compilation cost.

The two EBC-level optimizations, idiom/switch optimizations and the exception check elimination (second and third bars from the left), cause little increase in compilation overhead, but they improve performance in some benchmarks. In particular, idiom/switch optimization contributes to performance improvements in *mtrt* and *javac*. The register usage optimization in the code generation phase (rightmost bar) also contributes to a significant performance improvement, more than 20% for *mtrt* and *mpegaudio*, while it causes little additional compilation overhead. The escape analysis improves performance only for *mtrt* by about 10%, but causes around 10% of compilation time overhead for all benchmarks.

Evaluation of dynamic optimization framework

In this section, we evaluate our dynamic optimization framework. In the measurements, the timer interval for the sampling profiler for detecting hot methods was set to five milliseconds, and the list of hot methods was examined every 200 sampling ticks.

Figure 13 compares both performance and compilation overhead with our dynamic optimization framework (MMI-L1/L2/L3) against those in three single-level compilation configurations: MMI to L1 only, MMI to L2 only, and MMI to L3 only. The performance with MMI-L1/L2/L3 is slightly less than that of the best-performing configuration, MMI-L3. The difference is up to 4% and, on average, 2%. For the compilation overhead, MMI-L1/L2/L3 shows a significant advantage over both MMI-L2 and MMI-L3 configurations. In the best case, the overhead is almost equal to the baseline (MMI-L1). In the worst case, with *SPECjbb*, it is around 3.5 times the baseline for the compilation time. On average, the overhead is less than twice the baseline in all three metrics, and is around 30 to 70% of that with MMI-L3 configuration.

Figure 14 shows the corresponding results for the larger benchmarks, Java Grande section 3, XSLTMark Version 2.1.0, and SPECjAppServer2002. Java Grande section 3 consists of five benchmarks, *Euler*, *MolDyn*, *MonteCarlo*, *RayTracer*, and *Search*. We ran each of these benchmarks separately with the "Size B" problem (large dataset) and with the initial and maximum heap sizes of 512 MB. Unlike SPECjvm98, this benchmark includes the JIT compilation time in the best execution time. XSLTMark consists of 40 test cases designed to assess a variety of functional areas of extensible stylesheet language transformation (XSLT) processors. We used Xalan–Java Version 2.5.2 [27] as the XSLT processor, and measured the performance using the aggregated result of all of the test cases. SPECjAppServer2002 models the representative business process used at a Fortune 500 company, and its workload emulates heavyweight manufacturing, supply chain management, and order/inventory systems. We configured a three-tier system (**Figure 15**) by providing a separate machine for a database tier (using DB2* 8.1.4) and an application server tier (using WebSphere* Application Server Version 5.0.2). We used version 1.3.1 of the IBM DK, not version 1.4.0, for this benchmark, because of the product restrictions of this version of WebSphere, but we used the same code base for the JIT compiler as that used in the other benchmarks. We ran the application server with the initial and maximum heap sizes of 1200 MB and compared the self-reported score (transactions per second) for each of the four configurations. The ramp-up time, steady-state time, and ramp-down time were set to 300, 300, and 150 seconds, respectively.

Figure 14 shows the same overall characteristics as does Figure 13, but there are some interesting differences. First, MMI-L1/L2/L3 is the best performer for four benchmarks, and among them MMI-L1 is the second best for *MolDyn* and *SPECjAppServer*. Although we have not investigated the reasons in detail, we believe that the aggressive method inlining performed in L2 and L3 may cause code locality to be decreased. In fact, by disabling the aggressive method inlining in MMI-L2 and MMI-L3 configurations, performance is restored to a level higher than that with MMI-L1 for both of these benchmarks. Second, MMI-L1/L2/L3 shows a compilation overhead similar to or larger than that of MMI-L3 in all three metrics for *Euler* and *Search*. For these benchmarks, the recompilation system correctly picked up performance-critical methods to promote to higher optimizations, but their code size happens to be quite large after inlining. In some cases, those methods were compiled through all optimization levels, L1, L2, and then L3, and thus the overhead of repetitive compilation for those selected methods exceeded the overhead caused by compiling all methods with L3 optimization under MMI-L3 configuration. For other benchmarks, MMI-L1/L2/L3 works effectively to reduce the total compilation overhead.

Overall, our dynamic optimization framework performs best when both performance and compilation overhead are considered. These are the results even without enabling any profile-directed optimizations. The MMI-L1/L2/L3 is the only configuration that can enable those additional optimizations; thus, it has the potential to outperform other configurations even more without causing any significant increase in compilation overhead.

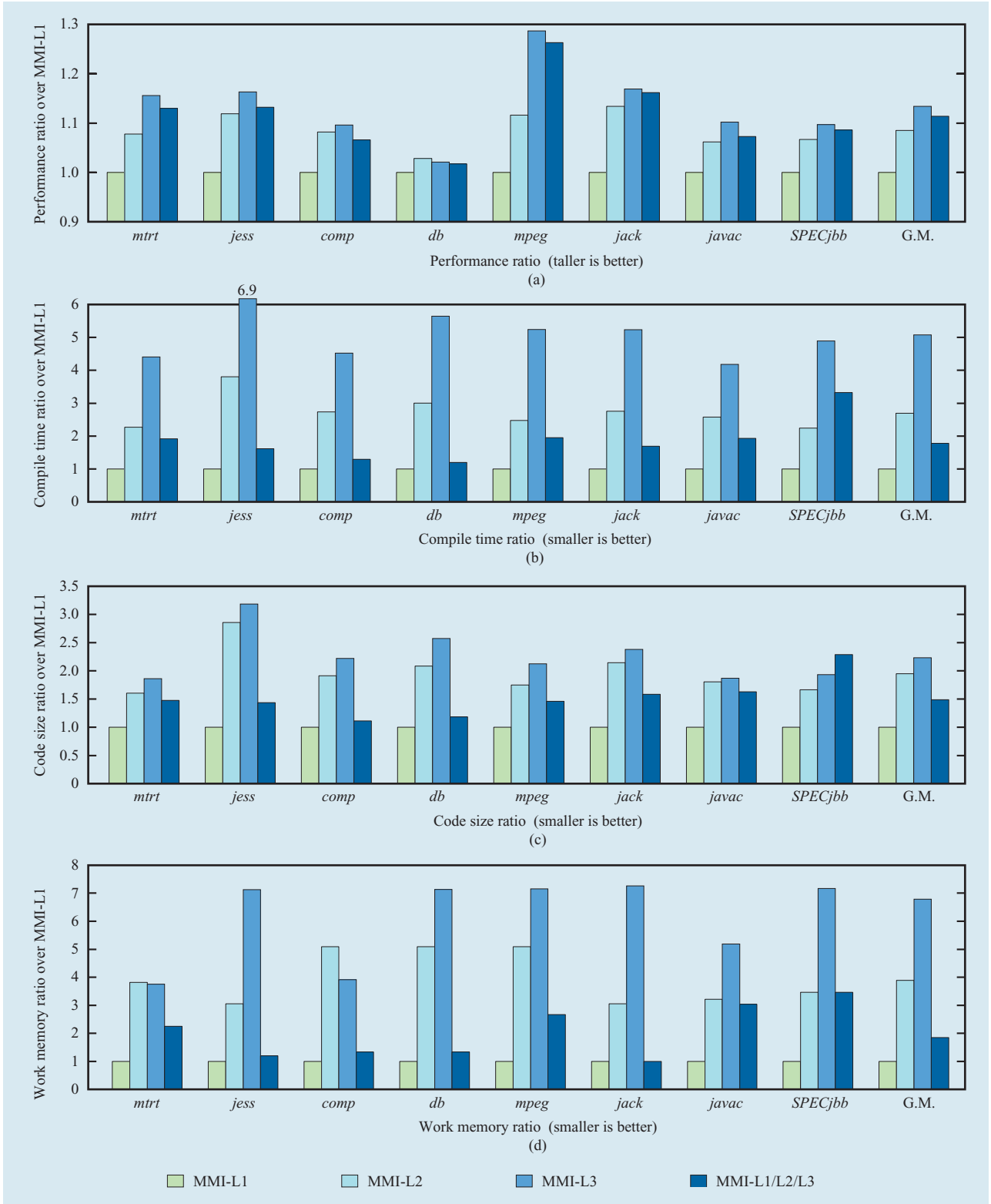

Figure 13

Evaluation of dynamic optimization framework for both performance (a) and compilation overhead (b) to (d). The baseline of the comparison is MMI-L1.

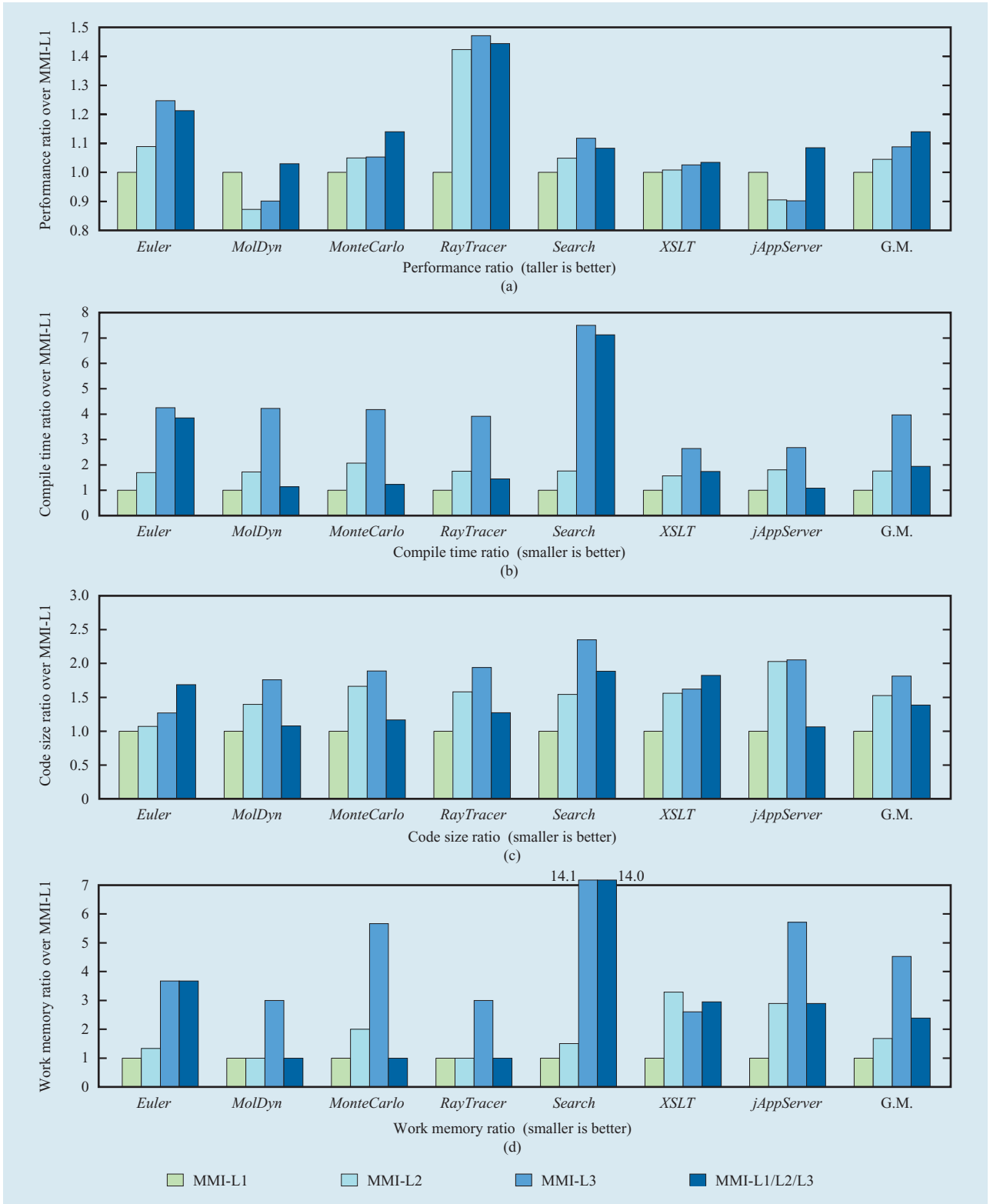

Figure 14

Evaluation of dynamic optimization framework for Java Grande, XSLTMark, and SPECjAppServer2002 benchmarks. The baseline of the comparison is MMI-L1.

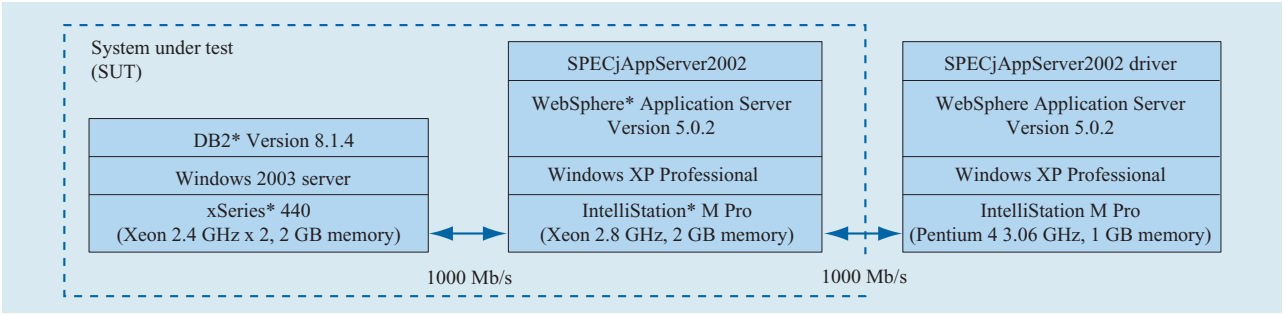

Figure 15

Experimental environment for running the SPECjAppServer2002 benchmark.

Evolution of performance improvement

Figure 16 shows the history of the performance improvement with each release of our JIT compiler after version 3.0. We have integrated the techniques described in this paper over several versions of the JIT compiler and have steadily improved the performance with every new release. With the latest version of the JVM and JIT compiler, we achieved on average more than a 30% performance improvement. In particular, the performance of *SPECjbb* nearly doubles. There have been significant contributions from our efforts in enhancing JVM, especially in the area of synchronization, object allocation, and class libraries, as well as from those in enhancing the JIT compiler for this achievement. Roughly speaking, half of the performance improvement can be considered as due to the JIT compiler, because the optimizations in JIT3.0 correspond approximately to the current L1 optimizations, and thus the performance of JIT4.5 is estimated to be about 15% better than JIT3.0, based on Figure 14(a), without the effect of JVM improvement. This is due to our rough estimate based on the average performance improvements, and the actual percent of the contribution differs greatly for each benchmark.

Figure 16 also shows an interesting trend that our overall performance improvement gains less from the improvement in the JIT compiler. In fact, the performance gain by JIT4.5 is much smaller than that achieved by each of the previous two releases. This is because we integrated most of the techniques described here in JIT3.5 and JIT4.0, including the two register-based IRs and a variety of optimizations on those IRs, and JIT4.5 added only a few remaining items. We currently believe that most of the optimization techniques that do not use profile information and are suitable for dynamic compilers have already been included in our JIT compiler, though there may be some additional opportunities we can find by investigating a wider range of applications or with the introduction of new architectural features. In the future, we need to explore some profile-directed optimizations to improve the performance further, as we have attempted for prototyping [6, 10, 28].

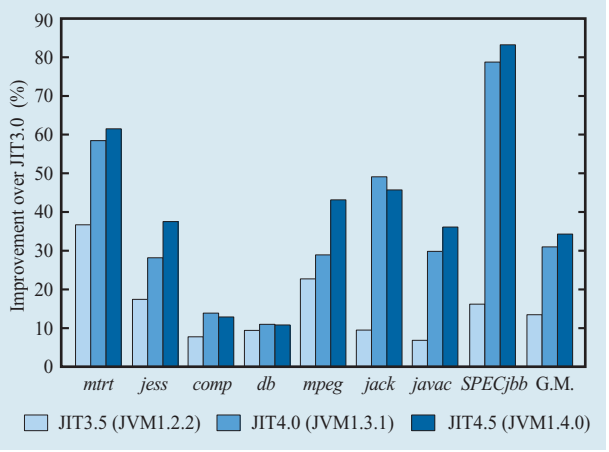

Figure 16

Performance improvements for each release of the JIT compiler. The baseline of comparison is JIT3.0 [1].

Related work

Besides the IBM DK, there are four other major Java virtual machines publicly available today. Sun Microsystems** HotSpot and BEA Systems** JRockit** are the two major production systems, whereas the IBM Jikes* Research Virtual Machine (RVM) and Intel Open Runtime Platform are the two major research virtual machines. All of these systems employ dynamic recompilation frameworks by using an interpreter and an optimizing compiler or by using two different compilers in order to focus the expensive optimization efforts only on the hot methods of a program.

The HotSpot server [29] is a JVM product that uses both an interpreter and an optimizing compiler to allow a mixed execution environment, as in our system. It uses an SSA-based IR for optimizations. The optimizer performs all of the classic optimizations, including common subexpression elimination, dead code elimination, constant propagation, and loop invariant hoisting. In addition, it employs more Java-specific optimizations, such as null check and range check elimination. It also collects profiles of receiver type distribution online in the interpreted execution mode. The profile information, together with CHA, is used when optimizing the code for virtual and interface calls.

The JRockit [30] 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. If the system determines that a compiled method is causing a performance bottleneck, it reoptimizes the method with a secondary compilation with full optimizations. While the details of the compiler structure and contents of the optimizations are not available, it seems to use an SSA-based IR for performing most of the optimizations.

The Jikes RVM [31], formerly called the *Jalapeño JVM*, is the most popular and widely used platform in the Java virtual machine research community and is released under an open source license. The system is implemented with Java itself and supports a multilevel recompilation framework using two different compilers: a baseline compiler and an optimizing compiler. The optimizing compiler has three IRs, all register-based, and performs a variety of optimizations on each IR. The bytecode is first translated to the high-level IR (HIR). The optimizations performed on the HIR are method inlining, some simple local optimizations within an extended BB, and flow-insensitive optimizations. The HIR is then translated to the low-level IR (LIR) by expanding some complex operators, where intraprocedural flow-sensitive optimizations based on the SSA form are performed by exploiting new opportunities exposed in the lower-level representation. The instruction selection then translates the platform-independent LIR into machine-specific IR (MIR) using a bottom-up rewriting system (BURS). The MIR has a one-to-one mapping between IR operators and the target instruction set architecture. Using MIR, the system performs live analysis and linear scan register allocation and then generates the native code. The optimizations on both HIR and LIR are divided into three levels in the optimizing compiler. The adaptive optimization system drives recompilation based on the estimated cost and benefit of compiling methods at each optimization level [32].

The Open Runtime Platform (ORP) [33] is another well-known RVM released as open source. It uses a compile-only approach and implements the dynamic optimization with two execution modes by providing two different compilers: a fast code generator [34] and an optimizing compiler [35, 36]. While the fast code generator produces code directly from bytecode with only limited and lightweight optimizations, the dynamic compiler uses an IR 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 determining the inlining policy, deciding where to apply expensive optimizations, and deciding upon the code layout in the final code emission.

The SELF system [37] was a pioneering work for dynamic optimizations, some of which are the basis of the techniques used today in many sophisticated JVMs and JITs. Some examples include code splitting [38], receiver class profiling and polymorphic inline cache [39], adaptive optimization system using online profile information [40], and dynamic deoptimization and on-stack replacement [41]. In particular, the adaptive recompilation system is now considered a standard feature for dynamic optimization systems to reconcile the high performance and low compilation overhead and is adopted in our system and in all of the above Java virtual machines. This also allows the system to exploit various kinds of profile information in the higher optimization levels to better guide optimizations using the dynamic behavior of a program.

Devirtualization of dynamically dispatched method calls has been extensively studied in static compilation environments. CHA [8] determines a set of possible targets of a dynamic method call by using the statically declared type of a receiver object and the class hierarchy of the whole program. If a target method is proved to be a single implementation, it can be inlined or the call site can be replaced with a fast direct call. Rapid type analysis [42] and variable type analysis [43] were proposed to be more precise than CHA by tightening the static type constraints on the receiver using object instantiation information. When a call site is not monomorphic but turns out to be almost monomorphic from receiver class profile information, a class test is provided to guard the devirtualized code [44]. All of these techniques were developed to statically analyze dynamic method calls under a closed-world assumption.

In the dynamic environment, dynamic class loading may cause the result of the compile-time analysis to be incorrect; thus, we need to provide guard code for all devirtualized call sites (unless it is provably monomorphic) and/or an invalidation mechanism to ensure safety in the occurrence of class loading. On-stack replacement [41]

is a technique to effectively invalidate currently executed unsafe code and to transfer the control to another version of the same method. HotSpot [29] uses this technique to eliminate backup paths for devirtualized call sites. The technique can be implemented with or without guard code and eliminates the merge points from virtual call sites in the control flow, improving the effectiveness of dataflow-based optimizations. Preexistence analysis [11] is a simple technique for invalidating unsafe code without requiring on-stack replacement, but it is effective for only a small portion of devirtualized call sites. Since methods are guaranteed to be safe until the next invocation, the guard code is not required with this technique. Our approach to code patching [12] can completely remove the overhead of executing guard code from any of the monomorphic call sites, but the diamond-shaped control flow with backup virtual dispatch still remains. Thin guards [45] reduce both the cost of guard code and the penalty of optimization restrictions by identifying regions of code for which speculative optimization can be performed and by mapping those multiple optimistic assumptions to a single guard.

Conclusions

We have described the design and implementation of version 4.5 of our Java JIT compiler, specifically for IA-32 platforms. Since the previous report [1], we have introduced new register-based IRs (quadruples and DAGs) and implemented more advanced optimizations to further improve the performance. At the same time, these optimizations were expected to cause high compilation overhead, and we needed to apply these optimizations more selectively and carefully than in the previous system. We designed a dynamic optimization framework with the interpreter and the system with three levels of recompilation to keep the total compilation overhead within an acceptable level.

We evaluated our approach using industry-standard benchmarks. The results—measured in compilation time, compiled code size, and compilation peak work memory usage—showed both the effectiveness of the optimizations on performance and the low compilation overhead with our dynamic optimization framework.

As we add more advanced and sophisticated optimizations in our JIT compiler, we have increasing numbers of heuristics, such as method inlining decisions, loop versioning decisions, the number of iterations for the dataflow, and triggers for recompilation, for controlling these optimizations. This is more or less inevitable, especially for dynamic compilers, since we need to balance two conflicting requirements: generating good-quality code and minimizing compilation overhead. The tuning of these heuristics is a difficult problem. We currently determine these various parameters from the empirical results by using a variety of industry-standard benchmarks, but we will soon need to include more customer applications to determine them.

In the future, we will further study the practical and effective ways of performing dynamic compilation. Instead of a predetermined optimization classification as currently implemented, it is desirable to dynamically select a set of suitable optimizations according to the characteristics of the target method, so that we can apply only those optimizations known to be effective for that method. In the dynamic compilation environment, we should avoid applying those optimizations that do not contribute to performance improvements but rather result in a waste of compilation resources. A possible way of doing this is to estimate the impact of each costly optimization on the basis of both the profile information and the structure analysis of the given method. The notion of *program metrics* [46] is one such attempt to achieve better optimization decisions.

The profile-directed optimizations are another potential route for future JIT evolution. Although we did not address any of those techniques in this paper, we have attempted some of them [6, 10, 28] in our framework, and see a potential for significant improvements. On the other hand, we believe that the introduction of the profile-directed optimizations may pose additional challenges from the reliability, availability, and serviceability (RAS) point of view, especially in product quality and code maintenance. When a problem arises around the profile-directed optimizations, it is in general quite difficult to reproduce the problem to trace back to the root cause. Future JIT compilers should address this problem while exploiting more of the profile information for advanced optimizations.

Acknowledgments

We thank the IBM Toronto Laboratory for continuing the development and services of the IBM Java JIT compiler. We also thank the IBM Java performance team in Austin, Texas, particularly Bill Alexander, Mike Collins, and Weiming Gu, for their helpful discussion and analysis for identifying opportunities to improve overall performance. We are also grateful to the former members of our group, Hiroshi Dohji, Masashi Doi, Shusaku Matsuse, Hiroyuki Momose, Arvin Shepherd, Janice Shepherd, Kunio Tabata, Akihiko Togami, and John Whaley, for their respective contributions to improve the JIT compiler. Finally, we thank the anonymous reviewers for their valuable comments and suggestions to improve this paper.

countries, or both.

References

1. T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler," *IBM Syst. J.* **39,** No. 1, 175–193 (2000); see *http://www.research.ibm.com/journal/sj/391/suganuma.pdf*.
2. T. Ogasawara, H. Komatsu, and T. Nakatani, "A Study of Exception Handling and Its Dynamic Optimization in Java," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2001, pp. 83–95.
3. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Programming Languages & Syst.* **13,** No. 4, 451–490 (October 1991).
4. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs," *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*, September 1999, pp. 21–31.
5. K. Pettis and R. C. Hansen, "Profile-Guided Code Positioning," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.
6. T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "A Dynamic Optimization Framework for a Java Just-in-Time Compiler," *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2001, pp. 180–194.
7. K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani, "Effectiveness of Cross-Platform Optimizations for a Java Just-in-Time Compiler," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003, pp. 187–204.
8. J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," *Proceedings of the 9th European Conference on Object-Oriented Programming*, August 1995; published in *Lecture Notes in Computer Science* **952,** 77–101 (1995).
9. J. Whaley, "A Portable Sampling-Based Profiler for Java Virtual Machines," *Proceedings of the ACM SIGPLAN Java Grande Conference*, June 2000, pp. 78–87.
10. T. Suganuma, T. Yasue, and T. Nakatani, "An Empirical Study of Method Inlining for a Java Just-in-Time Compiler," *Proceedings of the USENIX 2nd Java Virtual Machine Research and Technology Symposium*, August 2002, pp. 91–104.
11. D. Detlefs and O. Agesen, "Inlining of Virtual Methods," *Proceedings of the 13th European Conference on Object-Oriented Programming*, June 1999; published in *Lecture Notes in Computer Science* **1628,** 258–277 (1999).
12. K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A Study of Devirtualization Techniques for a Java Just-in-Time Compiler," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2000, pp. 294–310.
13. T. Ogasawara, H. Komatsu, and T. Nakatani, "Optimizing Precision Overhead for x86 Processors," *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium*, August 2002, pp. 41–50.
14. J. Whaley and M. Rinard, "Compositional Pointer and Escape Analysis for Java Programs," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, November 1999, pp. 187–206.
15. M. Kawahito, H. Komatsu, and T. Nakatani, "Effective Null Pointer Check Elimination Utilizing Hardware Trap," *Proceedings of the 8th International Conference on Architectural Support for Programming Language and Operating Systems*, November 2000, pp. 139–149.
16. M. Kawahito, H. Komatsu, and T. Nakatani, "Eliminating Exception Checks and Partial Redundancies for Java Just-in-Time Compilers," *Research Report RT-0350*, April 2000; IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan.
17. V. V. Mikheev, S. A. Fedoseev, V. V. Sukharev, and N. V. Lipsky, "Effective Enhancement of Loop Versioning in Java," *Proceedings of the 11th International Conference on Compiler Construction*, published in *Lecture Notes in Computer Science* **2304,** 293–306 (April 2002).
18. J. R. Goodman and W. C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," *Proceedings of the International Conference on Supercomputing*, July 1988, pp. 442–452.
19. M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Trans. Programming Languages & Syst.* **21,** No. 5, 895–913 (September 1999).
20. P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to Graph Coloring Register Allocation," *ACM Trans. Programming Languages & Syst.* **16,** No. 3, 428–455 (May 1994).
21. A. Koseki, H. Komatsu, and T. Nakatani, "Preference-Directed Graph Coloring," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002, pp. 33–44.
22. L. George and A. W. Appel, "Iterated Register Coalescing," *ACM Trans. Programming Languages & Syst.* **18,** No. 3, 300–324 (May 1996).
23. A. Koseki and H. Komatsu, "Register Constraint Back Propagation Working in Collaboration with a Register Manager," *Research Report RT-0551*, October 2003; IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan.
24. SPECjvm98, SPECjbb2000, and SPECjAppServer2002 Benchmarks available at *http://www.spec.org/osg/*; Standard Performance Evaluation Corporation (SPEC), 6585 Merchant Place, Suite 100, Warrenton, VA 20187.
25. Java Grande Forum Sequential Benchmarks available at *http://www.epcc.ed.ac.uk/javagrande/sequential.html*; The Java Grande Forum; see *http://www.javagrande.org/*.
26. XSLT Performance Benchmark available at *http://www.datapower.com/xml_community/xsltmark.html*; DataPower Technology Inc., One Alewife Center, 4th Floor, Cambridge, MA 02140.
27. Apache XML Project Xalan–Java Version 2.5.2 available at *http://xml.apache.org/xalan-j/index.html*; The Apache Software Foundation; see *http://xml.apache.org/*.
28. T. Suganuma, T. Yasue, and T. Nakatani, "A Region-Based Compilation Technique for a Java Just-in-Time Compiler," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003, pp. 312–323.
29. M. Paleczny, C. Vick, and C. Click, "The Java HotSpot Server Compiler," *Proceedings of the USENIX Symposium on Java Virtual Machine Research and Technology*, April 2001, pp. 1–12.
30. K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm, "Impact of JIT/JVM Optimizations on Java Application Performance," *Proceedings of the 7th Annual Workshop on Interaction Between Compilers and Computer Architecture*, February 2003, pp. 5–13.
31. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F.

Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño Virtual Machine," *IBM Syst. J.* **39,** No. 1, 211–238 (2000).
32. M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive Optimizations in the Jalapeño JVM," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2002, pp. 47–65.
33. M. Cierniak, B. T. Lewis, and J. M. Stichnoth, "Open Runtime Platform: Flexibility with Performance Using Interfaces," *Proceedings of the Joint ACM Java Grande–ISCOPE 2002 Conference*, November 2002, pp. 156–164.
34. A. R. Adl-Tabatabai, M. Cierniak, G. Y. Lueh, V. M. Parakh, and J. M. Stichnoth, "Fast, Effective Code Generation in a Just-in-Time Java Compiler," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998, pp. 280–290.
35. M. Cierniak, G. Y. Lueh, and J. M. Stichnoth, "Practicing JUDO: Java Under Dynamic Optimizations," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000, pp. 13–26.
36. A. R. Adl-Tabatabai, J. Bharadwaj, D. Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Spheisman, "The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments," *Intel Technol. J.* **7,** No. 1, 19–31 (February 2003).
37. C. Chambers, "The Design and Implementation of SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages," Ph.D. Thesis, Stanford University, CS-TR-92-1420, March 1992.
38. C. Chambers and D. Ungar, "Making Pure Object-Oriented Languages Practical," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 1991, pp. 1–15.
39. U. Hölzle, C. Chambers, and D. Ungar, "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches," *Proceedings of the 5th European Conference on Object-Oriented Programming*, July 1991; published in *Lecture Notes in Computer Science* **512,** 21–38 (1991).
40. U. Hölzle and D. Ungar, "Reconciling Responsiveness with Performance in Pure Object-Oriented Languages," *ACM Trans. Programming Languages & Syst.* **18,** No. 4, 355–400 (July 1996).
41. U. Hölzle, C. Chambers, and D. Ungar, "Debugging Optimized Code with Dynamic Deoptimization," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1992, pp. 32–43.
42. D. F. Bacon and P. F. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1996, pp. 324–341.
43. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical Virtual Method Call Resolution for Java," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2000, pp. 264–280.
44. G. Aigner and U. Hölzle, "Eliminating Virtual Function Calls in C++ Programs," *Proceedings of the 10th European Conference on Object-Oriented Programming*, July 1996; published in *Lecture Notes in Computer Science* **1098,** 142–166 (1996).
45. M. Arnold and B. G. Ryder, "Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading," *Proceedings of the 16th European Conference on Object-Oriented Programming*, June 2002; published in *Lecture Notes in Computer Science* **2374,** 498–524 (2002).
46. T. Kistler and M. Franz, "Continuous Program Optimization: Design and Evaluation," *IEEE Trans. Computers* **50,** No. 6, 549–566 (June 2001).

Received October 15, 2003; accepted for publication December 17, 2003; Internet publication September 17, 2004

Toshio Suganuma *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (suganuma@jp.ibm.com).* Mr. Suganuma is a Senior Advisory Researcher. He received B.E. and M.E. degrees in applied mathematics and physics from Kyoto University in 1980 and 1982, respectively, and received an M.S. degree in computer science from Harvard University in 1992. Since joining IBM in 1992, he has worked on the high-performance Fortran and Java JIT compiler projects. Mr. Suganuma's primary research interest is in the area of dynamic compiler optimizations.

Takeshi Ogasawara *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (takeshi@jp.ibm.com).* Mr. Ogasawara works in the areas of optimizing compilers. He joined IBM in 1991 at the Tokyo Research Laboratory after receiving B.S. and M.S. degrees in computer science from the University of Tokyo. He designed and implemented Exception Directed Optimization for the IBM Java JIT compiler and contributed to the runtime support of method inlining. Mr. Ogasawara's research interests include optimizations by compilers and their runtime systems.

Kiyokuni Kawachiya *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (kawatiya@jp.ibm.com).* Mr. Kawachiya is a Senior Advisory Researcher. He received B.S. and M.S. degrees in information science from the University of Tokyo in 1985 and 1987, respectively. Since joining IBM in 1987, he has been working on operating systems, multimedia systems, and mobile information devices. He joined the Java JIT compiler project in 1997, contributing primarily to its Linux version, IA-64 runtime, and synchronization optimization. Mr. Kawachiya's research interests include systems software and computer architecture.

Mikio Takeuchi *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (mtake@jp.ibm.com).* Mr. Takeuchi is an Advisory Researcher. He joined IBM in 1993 after receiving B.S. and M.S. degrees in mathematical engineering and information physics from the University of Tokyo. Until March 2003 he was a member of the Java JIT compiler project, where he worked on a lightweight register manager for the Intel IA-32 architecture and the compiler back end for the Intel IA-64 architecture. Mr. Takeuchi's research interests include compilers, runtimes, and tools for dynamic programming languages.

Kazuaki Ishizaki *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (kiszk@acm.org).* Dr. Ishizaki received B.S., M.S., and Ph.D. degrees in computer science from Waseda University in 1990, 1992, and 2002, respectively. He joined IBM in the Research Division in 1992, and has worked on the high-performance Fortran compiler. Dr. Ishizaki is currently involved with the Java JIT compiler; his research interests include optimizing compilers and processor architectures.

Akira Koseki *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (akoseki@jp.ibm.com).* Dr. Koseki received M.S. and Ph.D. degrees in computer engineering from Waseda University in 1994 and 1998, respectively. He joined the IBM Tokyo Research Laboratory in 1998. Dr. Koseki has been working primarily on code generation and register allocation.

Tatsushi Inagaki *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (e29253@jp.ibm.com).* Mr. Inagaki received B.S. and M.S. degrees from the University of Tokyo in 1993 and 1995, respectively. Since joining the IBM Tokyo Research Laboratory in 1998, he has been working on DAG-based optimizations in the Java JIT compiler project. His research interests include compiler optimizations and parallel processing.

Toshiaki Yasue *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (yasue@jp.ibm.com).* Mr. Yasue received B.S. and M.S. degrees from the School of Science and Engineering, Waseda University, in 1989 and 1991, respectively. Since joining IBM Japan in 1995, he has been working on the Java JIT compiler in the Network Computing Platform Group. Mr. Yasue's primary research interests include compiler optimization and parallel processing.

Motohiro Kawahito *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (jl25131@jp.ibm.com).* Dr. Kawahito received B.S. and Ph.D. degrees in computer science from Waseda University in 1991 and 2004, respectively. He joined IBM in 1991 as a member of the AIX Systems Group at the IBM Yamato Laboratory. He joined the Tokyo Research Laboratory in 1998 and is currently an Advisory Researcher. Dr. Kawahito worked on partial redundancy elimination, exception check elimination, constant propagation, dead store elimination, and class variable privatization for the Java JIT compiler project.

Tamiya Onodera *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (tonodera@jp.ibm.com).* Dr. Onodera is a Senior Technical Staff Member. He received B.S., M.S., and Ph.D. degrees in computer science from the University of Tokyo in 1983, 1985, and 1988, respectively. He joined IBM in 1988 and initially worked on the design and implementation of a new object-oriented language called COB. For the past seven years, he has been working on IBM Java virtual machines and JIT compilers. Dr. Onodera's primary research interest is efficient implementations of object-oriented programming languages.

Hideaki Komatsu *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (komatsu@jp.ibm.com).* Dr. Komatsu is a Senior Technical Staff Member. He received

B.S. and M.S. degrees in electrical engineering from Waseda University in 1983 and 1985, respectively, and a Ph.D. degree in computer science from Waseda University in 1998. Since joining the IBM Tokyo Research Laboratory in 1985, he has carried out research activities in the areas of optimizing compilers, Prolog compilers, dataflow compilers, the Fortran 90 compiler, the high-performance Fortran compiler, and the Java JIT compiler. Dr. Komatsu's research interests include compiler optimization techniques (register allocation, code scheduling, and loop optimizations) for instruction-level parallel architectures and loop optimizations for massively parallel computers.

Toshio Nakatani *IBM Research Division, IBM Tokyo Research Laboratory, 1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (nakatani@jp.ibm.com).* Dr. Nakatani received a B.S. degree in mathematics from Waseda University in 1975, and M.S.E., M.A., and Ph.D. degrees in computer science from Princeton University in 1985, 1985, and 1987, respectively. He joined the IBM Tokyo Research Laboratory in 1987 and is currently an IBM Distinguished Engineer and manager of the Network Computing Platform Group. Dr. Nakatani's research interests include computer architectures, optimizing compilers, and algorithms for parallel computer systems.