Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler

Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani

IBM Research, Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan
ishizaki@trl.ibm.com

ABSTRACT

This paper describes the system overview of our Java Just-In-Time (JIT) compiler, which is the basis for the latest production version of IBM Java JIT compiler that supports a diversity of processor architectures including both 32-bit and 64-bit modes, CISC, RISC, and VLIW architectures. In particular, we focus on the design and evaluation of the cross-platform optimizations that are common across different architectures. We studied the effectiveness of each optimization by selectively disabling it in our JIT compiler on three different platforms: IA-32, IA-64, and PowerPC. Our detailed measurements allowed us to rank the optimizations in terms of the greatest performance improvements with the smallest compilation times. The identified set includes method inlining only for tiny methods, exception check eliminations using forward dataflow analysis and partial redundancy elimination, scalar replacement for instance and class fields using dataflow analysis, optimizations for type inclusion checks, and the elimination of merge points in the control flow graphs. These optimizations can achieve 90% of the peak performance for two industry-standard benchmark programs on these platforms with only 34% of the compilation time compared to the case for using all of the optimizations.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Processors – *runtime environment, compilers, optimization.*

General Terms: Algorithms, Measurement, Performance, Experimentation, Languages.

Keywords: Java, just-in-time compiler, optimization.

1. INTRODUCTION

The execution model of the Java language [1] is to execute Java bytecode on a virtual machine (VM) in order to run unchanged programs on all platforms. The VMs evolved through three generations. In the first generation, a VM consisted of only an interpreter. This was very portable, but very slow. In the second generation, a VM consisted of a lightweight Just-In-Time (JIT) compiler to compile all the methods in a given program. This achieved acceptable performance, but the JIT compiler could not perform more advanced optimizations, because they usually required an excessively long compilation time. In the third generation, a VM consists of an interpreter and a highly optimizing JIT compiler that compiles only the "hot" methods in a given program. This system can perform more time-consuming optimizations by compiling only 10% - 20% of all the methods [2, 3]. It can achieve high performance and avoid a long compilation times.

Considering the cost-effective development of JIT compilers for many different platforms as software products, it is ideal to share as many optimizations as possible in common across all of the platforms. At the same time, it is desirable to tune the performance for the target architecture to achieve the highest possible performance. Thus, we adopted the following design:

- Use a compact stack-based intermediate representation (IR) for method inlining to expand the scope of optimizations in the earlier phases. This IR is shared in common among all of the platforms.
- Use other two register-based IRs for advanced optimizations in the later phases. These two IRs are also shared in common on multiple platforms, but a compiler generates a different pattern of the IRs for the target architecture. This makes it possible to share optimizations across multiple platforms and at the same time to customize some sequences of the IRs for the target architecture.
- Provide a few architecture-specific optimizations only for register allocation, instruction scheduling, and code generation, all of which have strong dependencies on the target architecture.

This paper focuses on the design and the empirical evaluation of cross-platform optimizations on multiple platforms. For more details, we describe an overview of advanced optimizations on two register-based IRs, quadruples and directed acyclic graphs, which have been newly developed for our latest Java JIT compiler since the previously reported version [4]. We also describe optimization features that are unique to Java, including exception elimination optimizations [5] and direct devirtualization [6].

The major motivation for our evaluation here is to clarify how each optimization feature contributes to the overall performance improvement in terms of the cost as measured by its compilation time. Our goal is to identify which optimizations are most cost-

OOPSLA '03, October 26-30, 2003, Anaheim, California, USA.

effective across multiple platforms. That is, we would like to select those optimization features with which we can get the largest performance improvement at the expense of the smallest compilation time overhead. In summary, we will show that, with a small set of the selected optimization features, we can achieve 90% of the peak performance for SPECjvm98 [7] and a special version of SPECjbb2000 [8], with only 34% of the total compilation time in comparison to the case in which all the optimization features are enabled.

This paper makes the following contributions.

- **Design and implementation of a Java JIT compiler**: We describe the design and implementation of a production Java JIT compiler. In particular, we focus on the cross-platform optimization features that are common across different processor architectures. These components, listed below, are designed to be shared by using our common intermediate representations and to optimize them to generate different machine code for the target architecture.
  - Method inlining
  - Exception check optimizations
  - Scalar replacement for instance and class fields
  - Optimizations for type inclusion checks
  - Elimination of merge points in the control flow graphs
  - Optimizations on directed acyclic graphs such as loop versioning and code scheduling.
- **Detailed evaluation of the cross-optimization features and their effectiveness**: We empirically evaluate the performance improvements and the compilation time overhead of each optimization by selectively disabling it in our JIT compiler on IA-32, IA-64, and PowerPC. We show which optimizations improve the overall performance at the cost of small compilation times and which optimizations improve the performance at the expense of large compilation times. As a result, we identify a set of the effective optimizations, listed below, with which we can achieve 90% of the peak performance for SPECjvm98 and a special version of SPECjbb2000 at the expense of only 34% of the total compilation time in comparison to the case in which all of the optimizations are enabled.
  - Method inlining only for tiny methods
  - Exception check optimizations using forward dataflow analysis and partial redundancy elimination
  - Scalar replacement for instance and class fields using dataflow analysis
  - Optimizations for type inclusion checks
  - Elimination of merge points in the control flow graphs

The rest of the paper is organized as follows. Section 2 describes the overview of the JIT compiler. Section 3 gives the detailed description of the design and implementation of optimizations. Section 4 shows the results of the experiments. Section 0 describes related work, and Section 6 gives the conclusions.

2. SYSTEM OVERVIEW

This section gives an overview of the IBM Java JIT compiler. First, we give the overview of the runtime environment and the structure of the JIT compiler in the IBM Developer Kit (DK).

2.1 Runtime Environment

The VM in the IBM DK is derived from an implementation of the Sun Classic VM. The IBM DK has many enhancements to improve the performance. Among these, we focus on three major enhancements.

The first enhancement is the object layout known as handleless objects. The VM in the IBM DK uses handleless object with direct pointers. The header and body of an object are allocated as one block. This has an advantage for high performance.

The second enhancement is the synchronization known as *Tasuki Lock* [9] derived from the *Thin Lock* [10] technique. The *Thin Lock* allows a synchronization operation to be performed with just a few machine instructions in the uncontended case. The Tasuki Lock further improves on robustness and performance over Thin Lock .

The third enhancement is the memory management system related to object allocation and garbage collection (GC). The system provides a global heap and thread-local heaps. While large objects are allocated from the global heap with appropriate synchronization, most objects are allocated from thread local heaps without synchronization, thus making object allocation very fast. The IBM DK basically uses a conservative, stop-the-world mark-sweep-compact GC [11], but it makes the mark and sweep phases parallel to reduce the pause time [12].

To allow efficient execution of a Java program, the VM in the IBM DK consists of a mixed-mode interpreter, which supports mixed execution of interpreted and compiled code [2], and a JIT compiler[1]. At the beginning, all methods are executed by the interpreter. An execution counter is provided for each method and initialized as zero. The counter is incremented at the method entry and at a loop backedge. When the counter exceeds a threshold value, the JIT compiler is invoked for the method.

The interpreter also records the runtime trace information for conditional branches and switch instructions. For any conditional branches encountered, the interpreter stores the information for the JIT compiler to predict the branch directions. For any switch instruction encountered, the interpreter stores the information for the JIT compiler to predict which case label is most frequently selected. Such trace information is used for the JIT compiler to recognize hot paths in a method.

2.2 Compiler Structure

Figure 1 shows the overall flow diagram of the JIT compiler, which uses three IRs. The first IR is called an *extended bytecode* (EBC) that holds the same stack-based semantics as the original

[1] The previous paper [2] described a dynamic optimization framework using a sampling-based profiler with an optimizing compiler at different levels of optimizations. Since the sampling-based profiler depends on the features provided by the Windows API, it was disabled for the study in this paper in order to evaluate the same set of optimizations on multiple platforms.

Java bytecode. It also maintains all of the type information for the destination of every instruction, unlike the original Java bytecodes, some of which lack explicit type information for the destinations. This requires only a small amount of memory, similar to the Java bytecode. The second IR is a *quadruple* (QUAD) that is a register-based representation. This is a tuple format with an opcode and zero or more operands, depending on each instruction. Its instructions explicitly represent potentially excepting instructions (PEIs) [13] to support the correct semantics of Java exceptions. The QUAD is the base IR designed to support various optimizations such as dataflow analysis. The third IR is a *directed acyclic graph* (DAG) that is also a register-based representation. This consists of nodes corresponding to QUADs and edges indicating both data dependencies and other dependencies such as exception dependencies [14]. These three IRs are grouped into basic blocks (BBs). BBs are not terminated by method calls or by PEIs like factored control flow graph [15].

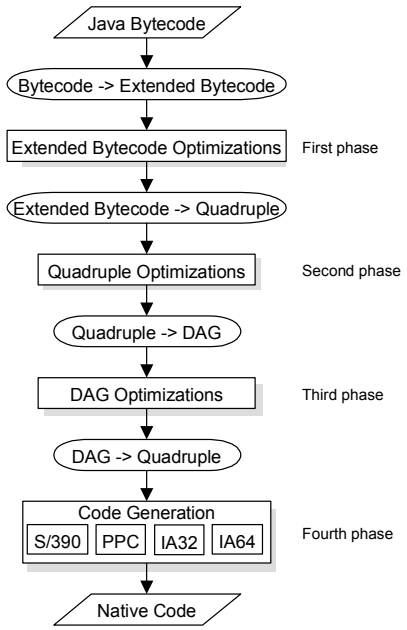


Figure 1. The overall structure of the JIT compiler

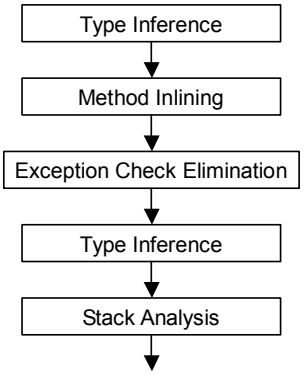


Figure 2. The sequence of optimizations on EBC

3. OPTIMIZATIONS

This section describes the optimizations on each IR, the EBCs, the QUADs, and the DAG. All of these optimizations are architecture-independent and common across all of the platforms. Finally, we describe the code generation, which is architecture-dependant.

3.1 Optimizations on EBC

This subsection describes the design and implementation of the optimizations based on the EBC, whose sequence is described in Figure 2.

First, flow-sensitive type inference [16, 17] computes a type for every object reference within the entire method to identify the possible set of classes of the receiver of each virtual method call. For each object reference, the compiler computes the dataflow information on its static types based on signatures, class instantiations such as new().

Second, method inlining, which replaces calls to methods by copies of their bodies, is performed to expand the scope of optimizations [18]. A program written in Java tends to have many small methods, such as accessor methods, which are called frequently. We call these small methods *tiny methods*. The compiler builds a possibly large call tree of inlined scopes with allowable sizes and depths, and then calculates the total cost by checking each decision. The compiler manages two separate budgets: one for tiny methods, and the other for non-tiny methods. By using separate budgets, the compiler attempts to inline as many tiny methods as possible. It also attempts to inline as many non-tiny methods as possible based on the following static heuristics until the predetermined budget is used up:

- If the total estimated size of the compiled code for both the caller and callee methods exceeds a threshold, stop inlining the method.
- If the estimated size of the compiled code for the callee method exceeds a threshold, stop inlining the method to avoid wasting the budget on a single method.
- If the call site is within a loop, perform inlining for a deeper level of the tree than that for a call site outside of a loop.
- If the total number of local variables and the stack height for both caller and callee methods exceed a threshold, stop inlining the method.

A static method call can be inlined in a straightforward manner by replacing the call to the method with a copy of its body. A dynamic method call may have several target methods, and thus devirtualization techniques must be used. When a dynamic method call is found during method inlining, class hierarchy analysis (CHA) [19] is performed to determine a set of possible targets of the dynamic method call by combining the static type of the object with the class hierarchy of the entire program. If it can be proved that the method call has only a single target, the compiler incorporates the target method without any guard code via *code patching* [6]. If the call has more than one target method, the compiler incorporates one of the target methods with guard code using a *method test* [20]. Any of these devirtualization techniques generates a *backup path* that is executed if the assumption fails during the execution of the program. After the devirtualization, both the static and dynamic method calls are inlined based on the same heuristics described above.

Third, the compiler eliminates redundant nullchecks and array bound checks. It eliminates redundant nullchecks along an execution path using forward dataflow analysis [5]. Then, it eliminates redundant array bound checks using forward dataflow analysis based on the extension of Gupta's algorithm [21] as described in [22]. The dataflow analysis propagates range expressions to determine where they must be checked. All the redundant checks for eliminating them in the later phase are marked as attributes of the EBC. This phase can greatly reduce the size of the QUAD, since the QUADs explicitly represent PEI.

Fourth, type inference is performed again for eliminating redundant backups and type inclusion checks (TICs) that determine whether two object references are related by a subtyping relationship. It proves that only a single class is reached at the receiver of the dynamic method call, the method call can be directly devirtualized without any backup path. The property of preexistence[2] [20] can then be used to directly devirtualize a dynamic method call without any backup path. To check for the preexistence property of a receiver, invariant argument analysis is performed using the result of type inference. If the receiver of a dynamic method call is directly reachable from an argument of the method and the method call has only a single target at compilation time, then the method call can be directly devirtualized without any backup path. When a method is overridden by dynamic call loading, the method will be recompiled at the next invocation. Therefore, any of these existing backup paths can also be eliminated. In addition, if all the members of the set of classes at a source operand of a TIC are a subtype of a class of the cast target at compilation time, then the compiler can eliminate that TIC. In Figure 2, there are two phases for type inference. The first type inference (before method inlining) increases an opportunity for direct devirtualization, while the second type inference (after exception check elimination) eliminates redundant TICs and backup paths in a wide compilation scope.

Finally, stack analysis is performed to identify the type of every stack operand and local variable using forward dataflow analysis. This type information is used during the translation from the EBC to the QUAD for mapping stack operands and local variables to symbolic registers.

3.2 Optimizations on QUAD

This subsection describes the design and implementation of optimizations based on the QUADs, whose sequence is described in Figure 3.

The QUAD representation is finer grained than the bytecode. For example in Figure 4, an *iaload* bytecode is divided into several QUADs: a QUAD (NULLCHECK) to check whether the given object reference is null, a QUAD (ARRAYLEN) to load the length of the array object, a QUAD (BOUNDCHECK) to check whether the given array index is legal, some QUADs to obtain the address of the specified array element, and a QUAD (IALOAD) to read the value from the address. The QUAD representation makes it easy to eliminate redundant exception checks and common sub-expressions.

[2] If the receiver for a dynamic method call has been allocated before the invocation of its calling method, then that method cannot be overridden during the execution of the caller.

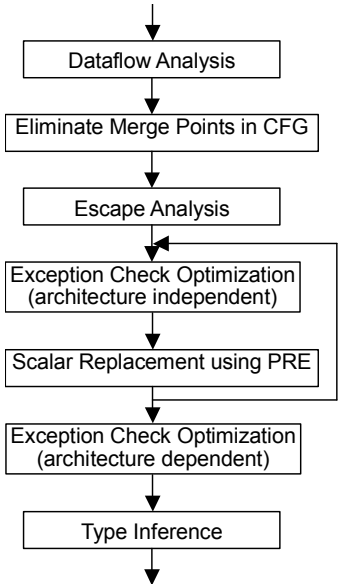


Figure 3. The sequence of optimizations on QUADs

Most QUADs have a one-to-one correspondence to a native instruction, except for a few complex QUADs that correspond to object allocations and unresolved class references. The QUAD is designed for different processor architectures. A set of QUADs to obtain the address of an array element is different for each platform. For example, it needs only a single QUAD for the architecture supporting a scaled-index-addressing mode with displacement, such as mov eax, [ecx+ebx*4+8] on the IA-32 architecture. On the other hand, it needs multiple QUADs that generate an effective address for an array element followed by a load instruction from the address on the PowerPC, the IA-64, and the S/390 architectures. Figure 4 shows an example of translating from a sequence of EBCs to QUADs for these different architectures. Since many of the same QUADs are used in common, we can share many optimization phases across the different platforms while generating efficient code for each architecture.

First, after translating the EBCs to the QUADs, dataflow optimizations, such as copy propagation, constant propagation, and dead code elimination [23], are performed to remove redundant QUADs caused by the static semantics.

Second, merge points are eliminated by method splitting [24], which is a kind of tail duplication. Direct devirtualization by code patching [6] creates a diamond control flow including the devirtualized method on one side and a backup path including the original dynamic method call on the other side. The merge point in a control flow graph (CFG) may limit the JIT compiler from performing dataflow optimizations, as discussed in [25]. Since naïve splitting causes an exponential code explosion, we use a *frequency-directed splitting*, in which the BBs in frequently executed paths are duplicated to expose an opportunity for optimizations, while those in rarely executed paths, such as backup paths, are not duplicated. Figure 5 shows an example to describe the difference between naïve splitting and our frequency-directed splitting for eliminating the merge points in the CFG. The figure shows that our splitting can reduce the number of BBs without introducing any merge point in the frequently executed path (a->b->d->e->g).

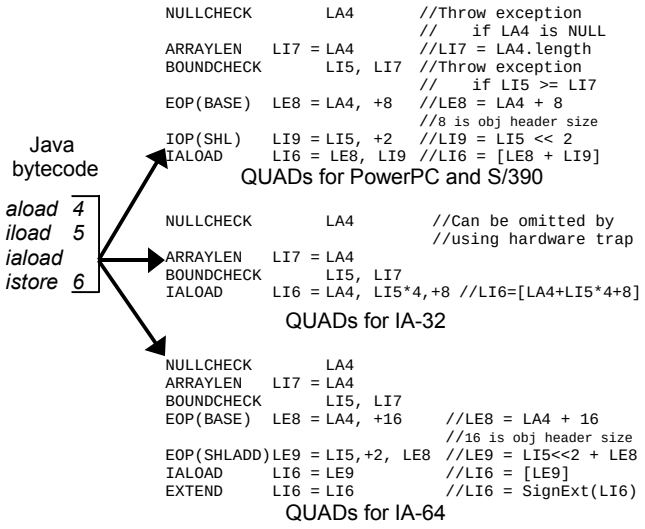


Figure 4. An example of the translation from bytecode to QUADs

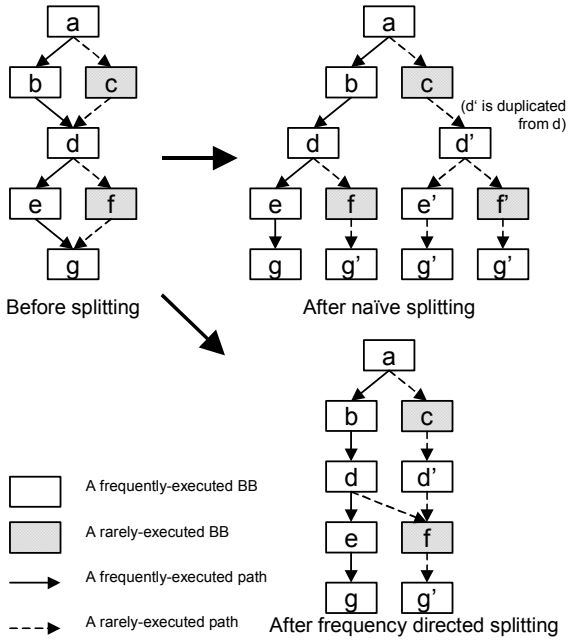


Figure 5. An example of eliminating merge points in the CFG

Third, compositional escape analysis [26] is performed based on the point-to escape graphs, which characterize how local variables, instance, and class fields refer to objects. If an object does not escape from its allocating thread, the compiler eliminates all the synchronization operations on the object. If it is found that an object does not escape from the current method, the JIT compiler allocates it in the method's local stack instead of the heap. To reduce the compilation time overhead, the JIT compiler produces summary information for every call site that may call the same method. If a callee method has not yet been analyzed at a call site, the JIT compiler proceeds with a pessimistic assumption that all the arguments are escaping at the call site.

In addition, for those objects that are not escaping, if the object header is not used within the method, the compiler performs scalar replacement for all the fields in the object to replace global variables with scalar temporaries to allocate them to registers.

Fourth, exception check optimizations (architecture independent) and scalar replacements are performed iteratively, as described in [5]. The exception check optimization for nullchecks uses partial redundancy elimination (PRE) [27] to remove exception checks out of loops. The exception check optimization for array bound checks uses forward and backward dataflow analyses based on the extended Gupta algorithm [22].

Then, scalar replacement for instance and class variables follows with PRE, which eliminates redundant computations of common subexpressions and moves invariant accesses out of loops. In addition, it moves redundant computations in the frequently executed path aggressively before conditional branches [22]. Since each of these optimizations can expose new opportunities for the other, the compiler iterates this phase several times.

Next, after completing the iterations, exception check optimizations (architecture dependent) are performed. First, forward dataflow analysis is made to minimize exception checks by utilizing a hardware trap mechanism. If a hardware trap for accessing the zero address (page) is available for the target architecture, explicit nullchecks are converted into implicit nullchecks. Then, backward dataflow analysis is performed to eliminate redundant exception checks. In our current implementations, this phase is enabled both on Windows/IA-32 and Linux/IA-32 platforms.

Finally, type inference is performed. Flow-sensitive type inference determines a set of classes reachable at each object reference within the entire method. This is basically the same process as the one performed on the EBC, as described in Section 3.1, but it is more effective since many merge points in the CFG were eliminated prior to this phase. The result is used for eliminating backup paths and TICs.

A TIC determines whether two types are related by a given subtyping relationship. Instructions requiring TICs such as *instanceof*, *checkcast*, and *aastore* bytecodes are executed frequently in Java, and thus it is important to reduce their runtime overhead. If the set of classes at a source operand of a TIC is known to be a subtype of a class of the cast target at compilation time, the compiler can eliminate that TIC.

In addition, to improve the runtime performance of each TIC, we generate a simple block of inlined code [28] to test the most-frequently occurring case, as shown in Figure 6. Line 1 checks whether the referenced object (from) is NULL. Line 2 checks whether the referenced object is an array. If the JIT compiler knows that an array object never reaches the referenced object, this statement can be removed. Line 3 checks whether the actual class of the referenced object is identical to that of the destination operand (Type). Line 4 (Line 5) checks whether the class cached in the referenced object by the last successful comparison (the last failed comparison) is identical to that of the destination operand. Each of these inlined tests can be done in only two or three machine instructions. If all of these tests fail, the C runtime routine expensive_testC() is called for traversing the class hierarchy. This implementation is based on our finding that most of the test cases are handled by the inlined tests. Though it is not shown in this paper, we observed that our performance is almost

comparable with that of Cohen's algorithm [29] using SPECjvm98 and SPECjbb2000 [8].

```
Java program
   Type to = (Type)from;

Generated code
1: if (from == NULL) {to = NULL;}
2: elif (is_array_object(from)) {if expensive_testC(...) ...}
3: elif (from.type == Type) {to = from;}
4: elif (from.type.lastsucc == Type) {to = from;}
5: elif (from.type.lastfail == Type) {throw exception}
6: elif (expensive_testC(...)){to=from;from.type.lastsucc=Type;}
7: else {from.type.lastfail = Type; throw exception}
```

Figure 6. Code of a Type Inclusion Check

A throw elimination that is a part of Exception-Directed Optimization [30] is also performed using the results of the type inference. If the object of a class thrown by an *athrow* bytecode is caught by a surrounding exception handler, the JIT compiler can replace the *athrow* with a direct branch to the corresponding catch block. This can eliminate the overhead of throwing an exception and searching for a corresponding handler at runtime when an *athrow* bytecode is executed.

3.3 Optimizations on DAG

This subsection describes the design and implementation of optimizations based on the DAGs, whose sequence is described in Figure 7.

The DAG consists of nodes and directed edges. Each node corresponds to a QUAD, and it has one or more source operands, a destination operand, and a special operand to describe if there is any side effect. Each directed edge represents a data dependence, or other dependence such as synchronization dependence or exception dependence [14], between two nodes.

We begin by translating the QUADs to the static single assignment (SSA) form [31], and generate a minimal-representation SSA [32] by inserting phi-functions. Second, we apply loop versioning [4] to hoist array bound checks outside of a loop in an optimized loop along with the original loop. The code for exception checks is added only at the entry to the optimized loop to examine the whole range of the index at once for the entire loop. As a result, all the array bound checks on the first dimension of the array are eliminated. Figure 8 shows such an example. In addition, we apply loop versioning to make sure that the array accesses by load or store instructions are not aliased to any other accesses. This will make the following scalar replacement more effective. Figure 9 shows such an example, in which the original loop has two accesses to the array a[i], where a[i] may be aliased to b[i]. In the optimized loop, the second access to a[i] can be replaced with a reuse of the first access to a[i] after ensuring a[i] is not aliased to b[i].

Third, for each striding array access to an induction loop index, we generate an appropriate update-type memory access instruction available on the target architecture. For example, we use an stwu or lwzu instruction in the update form on the PowerPC, whereas we use an ld4 or st4 instruction in the immediate base update form on the IA-64. For each countdown loop, we use a loop count register supported by the target architecture.

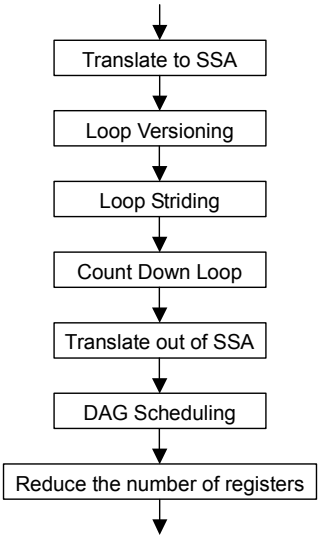


Figure 7. The sequence of optimizations on DAG

```
if ((array != NULL) && (0 <= start) && (end <= array.length)) {
    /*  optimized loop
        eliminate all array bound exception checks for array[] */
    for (i = start; i < end; i++) {
        array[i] = array[i] + 1;
    }
} else {
    /*  original loop
        original loop with array bound exception checks */
}
```

Figure 8. An example of loop versioning for eliminating exceptions [4]

```
Java program
for (i = 0; i < n-1; i++) {
  x = a[i];
  b[i] = y*a[i+1]; // a may be aliased to b
  z = a[i];
}

Generated code
if (a != b) {
  for (i = 0; i < n-1; i++) {
   x = a[i];
   b[i] = y*a[i+1];// a are not aliased to b
   z = x;          // replace an array access
                   // with a scalar variable
  }
} else {
  for (i = 0; i < n; i++) {
   x = a[i];
   b[i] = y*a[i+1];// a are aliased to b
   z = a[i];
  }
}
```

Figure 9. An example of loop versioning for scalar replacement

After we translate the SSA form of the QUADs back to the non-SSA form [33], we perform pre-pass code scheduling for each basic block using a list scheduling algorithm. The scheduling policy of our algorithm is adaptive. When the available registers are scarce, the algorithm attempts to minimize their usage. When they are not, it attempts to maximize the instruction-level parallelism. Finally, we create a pre-allocation of the registers in order to reduce the register usage by allocating the same register number to those variables whose lifetimes do not interfere with each other.

3.4 Code Generation

This phase consists of three parts, as shown in Figure 10. First, an architecture mapping is made. Then, register allocation is performed using physical registers. Finally, code emission is made with post-pass code scheduling to generate the native code for the target platform.

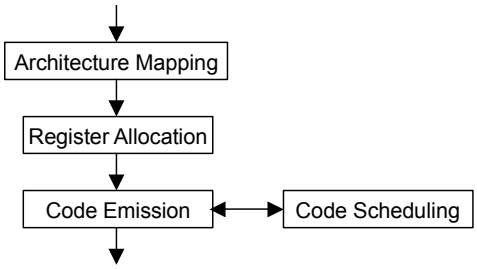


Figure 10. The sequence of code generation

First, the architecture mapping translates a sequence of QUADs to the one suited for the target architecture. For the IA-32, the JIT compiler can specify at most two operands for each machine instruction, and it can use a memory location as an operand. For the IA-64 architecture, it translates a conditional branch to a compare instruction to set a result in a predicate register and a branch instruction guarded by the predicate register. In addition, the JIT compiler performs *if-conversion* [34] to translate an acyclic region of BBs to a *hyperblock* [35], a single branch-free block with a single entry and multiple exits, in order to improve the performance by eliminating branch instructions. Currently, the JIT compiler support only simple if-then-else blocks, which are transformed to simple hammocks.

Java specifies the frequently used "int" type as a signed 32-bit data type [36]. If such Java programs are executed on a 64-bit architecture, 32-bit values must be sign-extended to 64-bit values for many integer instructions. This extension operation will cause serious performance degradation, and thus we implemented an efficient algorithm for eliminating sign extension effectively [37]. While this optimization is beneficial in general for the effective address computation of array accesses, it is also useful for the IA-64 architecture that does not support load instructions with sign extension.

To improve the instruction cache locality, the compiler also reorders BBs in the CFG by moving the rarely executed regions to the bottom and placing the frequently executed regions as close as possible.

Second, we perform register allocation using different algorithms depending on the number of registers available for the target architecture. For example, we use a special allocator for the IA-32, which has only eight general-purpose registers (GPRs). It begins by allocating registers to the frequently accessed operands. If there are still some registers available, it allocates those registers to the short-lived operands. We use various heuristics to cope with the non-orthogonal usage of registers. We attempt to use MMX, SSE, and SSE2 instructions where possible. For the PowerPC, which has 32 GPRs and 32 floating-point registers (FPRs), we use a linear scan register allocator [38] to minimize the overhead of the compilation time. Note here that it would be ideal to use a single register allocator for all of the platforms, and a register allocator based on the preference-directed coloring algorithm [39] is one of the candidates for that goal.

Finally, we perform code emission to generate the machine instructions for the target architecture in cooperation with post-pass code scheduling. It is fairly straightforward to generate efficient machine instructions since in general each QUAD has a corresponding machine instruction. There are a few exceptions, such as the QUAD for object allocation and that for a reference to an unresolved class, either of which needs to call a runtime routine for a special handling.

The code emission is performed in conjunction with post-pass code scheduling within each BB. The code scheduler puts a ready instruction in the earliest available slot in a first-fit manner [4]. Since the IA-64 architecture can issue several instructions simultaneously, forming a bundle of instructions that can be executed in parallel is important to extract the instruction-level parallelism from a given program. The critical part of the bundle formation is the set of heuristics that determine the order of instructions in a bundle to satisfy various constrains. Based on the heuristics, the code scheduler swaps instructions in a bundle to maximize the utilization of the non-uniform execution units.

4. EXPERIMENT RESULTS

This section describes the results of several experiments showing the effectiveness in performance and reduced compilation time of the optimizations in the JIT compiler. We outline the experimental methodology for the benchmarks, and then discuss the experimental results.

4.1 Benchmark Methodology

All the results presented in this section were obtained using the VM of the IBM Developer Kit, Java Technology Edition, Version 1.4.0. The threshold in the interpreter to initiate the JIT compiler was set to 1,000 on all platforms.

We used SPECjvm98 [7] and pseudojbb [40] (denoted as pjbb in the graphs), which is a fixed-work version of SPECjbb2000 [8]. For SPECjvm98, the measurements were performed from five executions of the autorun sequence in the test mode (not in the SPEC-compliant mode) with the count of 100. For pjbb, a fixed number of transactions is executed to compare the execution time and compilation time.

We conducted experiments on three platforms. The IA-32 platform is an IBM IntelliStation (Pentium 4 Xeon 2.8 GHz dual-processor with 1 GB memory), running Windows 2000. The IA-64 platform is an IBM IntelliStation (Itanium 800 MHz dual-processor with 2 GB memory), running Windows .NET server. The PowerPC platform (denoted as PPC in the graphs) is an IBM eServer pSeries 630 (POWER4 1.0 GHz 4-way processor with 2 GB memory), running AIX 5L Version 5.1.

4.2 Experimental Results

This subsection presents experimental results to show how various sets of optimizations affect the execution time and the compilation time for each benchmark program. Here, execution time means the best execution time in all the sequence for each benchmark program, and compilation time means the compilation time in all the sequence for each benchmark program. In this subsection, all the execution times are normalized relative to the result with all optimizations enabled (denoted as Base in the graphs). In the graphs for the relative execution time, the taller bars show higher performance. All the compilation times are also normalized

relative to the result with all optimizations enabled (denoted as Base in the graphs). For those results, the shorter bars show that less time is used.

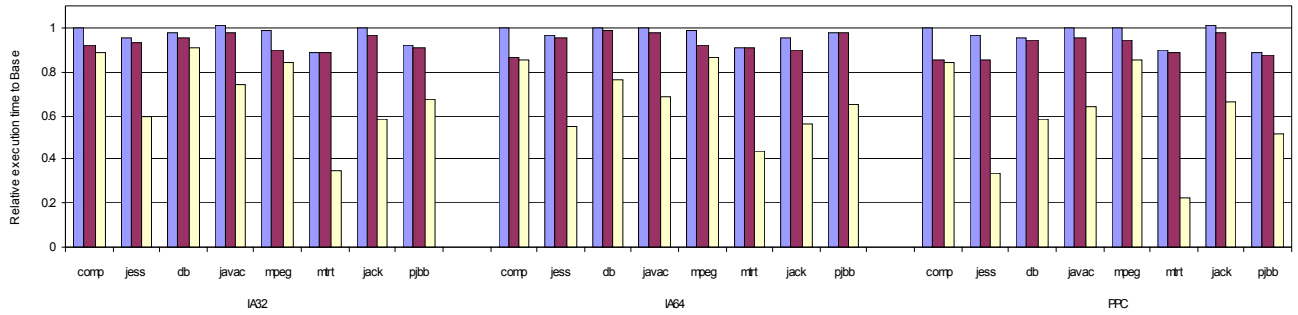
We categorize the optimizations described in Section 3 as follows: method inlining, exception check optimizations, scalar replacement, optimizations for TICs, elimination of merge points in the CFG, and optimizations on the DAG. For the rest of this subsection, we evaluate each of these optimizations by selectively disabling it on the three platforms.

4.2.1 Method Inlining

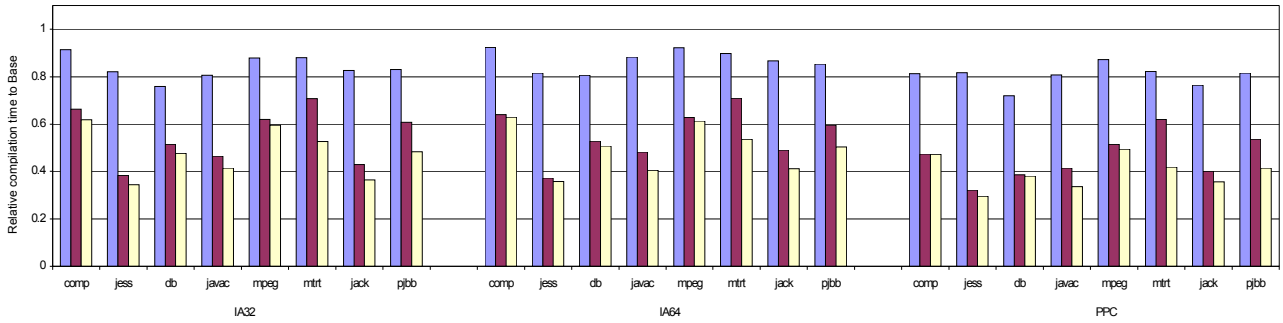
As we described in Section 3.1, three kinds of method inlining are performed: dynamic method inlining, static method inlining, and tiny method inlining both for static and dynamic methods. Figure 11 shows the relative execution times and compilation times from selectively disabling method inlining as in the table.

The results show that the case where all of the method inlinings are disabled degrades the execution time significantly. The degradations vary from 11% to 78%. Method inlining of tiny methods both for static and dynamic method calls is a simple heuristic with great effectiveness, resulting in a maximum execution time[3] degradation of only 15% from the peak execution time. Static method inlining is effective for compress and mpegaudio, which are loop-centric programs. Dynamic method inlining is effective for mtrt, which has a hot method OctNode.Intersect including many dynamic method calls.

When no method inlining is performed, the compilation time is reduced by an average of 55%. Method inlining for tiny methods increases the compilation time by an average of 6%. Static method inlining drastically increases the compilation time by up to 50% (with an average of 34%). In particular, it is remarkably expensive for jess, javac, and jack. Dynamic method inlining also increases the compilation time by up to 38% (with an average of 16%). On the other hand, these two method inlining techniques improve the execution time by up to 15% (with an average of 8%). The balance between the benefit of the execution time and its cost needs to be considered carefully.



a) Relative execution times (Taller bars are better).



b) Relative compilation times (Shorter bars are better).

The color of the bar	Dynamic method inlining	Static method inlining	Tiny method inlining both for static and dynamic methods
Base	ON	ON	ON
	OFF	ON	ON
	OFF	OFF	ON
	OFF	OFF	OFF

Figure 11. Measurements on method inlining.

[3] In this paper, 'improvement of the time' shows the simple difference between two values of relative time.

4.2.2 Exception Check Optimizations

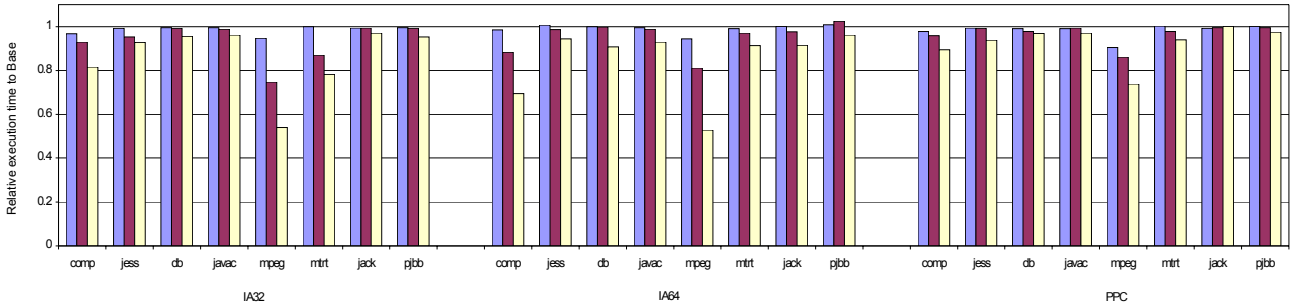
As we described in Sections 3.1, 3.2, and 3.3, three kinds of exception check optimizations are performed: exception check eliminations using loop versioning, exception check eliminations using PRE, and exception check eliminations using forward dataflow analysis. Figure 12 shows the relative execution times and compilation times by selectively disabling exception check eliminations as in the table.

Eliminating exception checks using forward dataflow analysis is a simple technique that turns out to be quite effective. It improves the execution time from 2% to 28% (with an average of 8%). In particular, it is effective for compress and mpegaudio that frequently access array elements. The optimizations of exception checks using forward and backward dataflow analysis including PRE are effective for compress, mpegaudio, and mtrt. Loop versioning is effective only for mpegaudio.

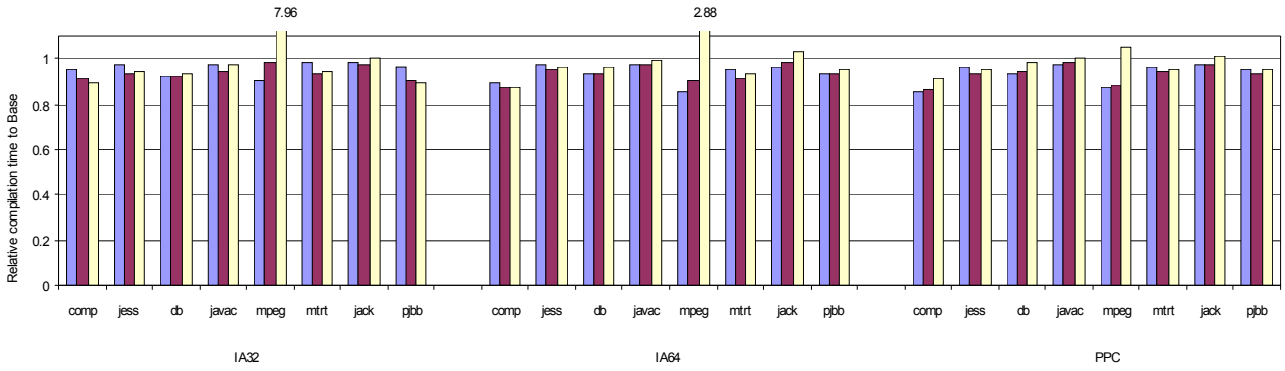
On IA-32, explicit nullcheck instructions are not generated for array and field accesses because of the utilization of hardware traps. Therefore, the degradations are smaller overall except for mtrt and mpegaudio for all of these optimization settings. The direct devirtualization newly introduces an explicit nullcheck since it removes a memory access to a receiver object. There are many opportunities for direct devirtualization in mtrt. Since more explicit nullchecks are generated than in the original program, the degradation is larger. In mpegaudio, there are many array bound checks that are explicitly generated.

On PowerPC, it takes only one cycle to execute an exception check using a special compare and branch instruction (tw/twi instructions). Therefore, the degradations are smaller overall for all of these optimization settings.

Eliminating exception checks using forward dataflow analysis has little effect on the compilation time. It even decreases the compilation time for jess, db, javac, mpegaudio, and jack. This is because the elimination reduces the size of the IR and thus decreases the time needed for other optimizations. When no elimination is performed, there is a remarkable increase of the compilation time for mpegaudio. The compilation time is about 8.0 times longer on IA-32 and 2.9 times on IA-64. This is due to the fact that the compilation times for optimizations on the DAG were increased drastically by disabling exception check eliminations using forward dataflow analysis. It is about 24.0 times on IA-32 and 5.7 times on IA-64. This is because many edges exist in the DAGs for q.m and tb.???[4]. Loop versioning increases the compilation time by up to 14%.



a) Relative execution times (Taller bars are better).



b) Relative compilation times (Shorter bars are better).

The color of the bar	Exception check eliminations using loop versioning	Exception check eliminations using PRE	Exception check eliminations using forward dataflow analysis
Base	ON	ON	ON
	OFF	ON	ON
	OFF	OFF	ON
	OFF	OFF	OFF

Figure 12. Measurements on exception check optimizations.

[4] This cannot be represented using ASCII characters.

4.2.3 Scalar Replacement

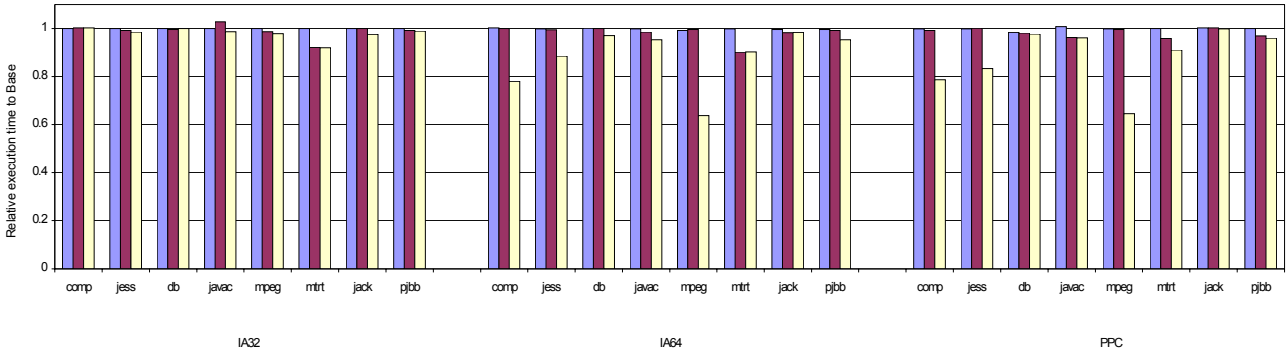
As we described in Sections 3.2 and 3.3, three types of scalar replacements are performed: scalar replacement using loop versioning, scalar replacement using escape analysis, and scalar replacement using dataflow analysis. Figure 13 shows the relative execution times and compilation times by selectively disabling the scalar replacement optimizations as in the table.

Since scalar replacement is a technique to replace global variables that are reused frequently with scalar temporaries to promote their allocation to registers, it is particularly effective on the IA-64 and PowerPC architectures that have many registers. As shown in the graph, this improves the execution time of mpegaudio by 35% on both platforms. It also improves the execution times of compress, mtrt, and jess by 1% to 22% on both platforms.

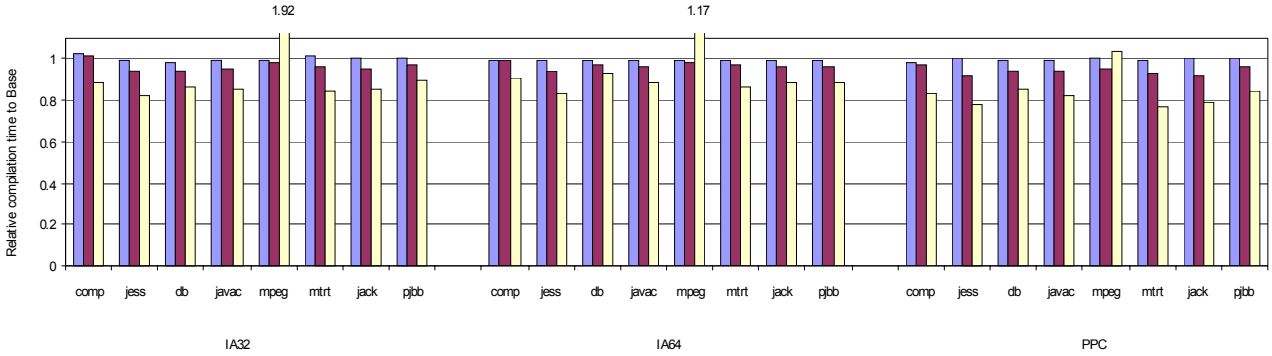
Scalar replacement using escape analysis improves the execution time only for mtrt, by 10.0% and 4.1% on IA-64 and PowerPC, respectively. This is because it replaces the fields of an object allocated in the method OctNode.Intersect with scalar temporaries. It also improves the execution time of mtrt on IA-32. This is because the escape analysis frees up a register that was used for pointing to an object header. Scalar replacement using loop versioning has little effect on the execution time of these programs.

Scalar replacement using dataflow analysis increases the compilation times by about 10% for all programs except mpegaudio. When no scalar replacement is performed, there is a remarkable increase of the compilation time for mpegaudio. The compilation time is about 1.9 times on IA-32 and 1.2 times on IA-64. This is due to the fact that the compilation times for the optimizations on the DAG were increased drastically by disabling scalar replacement using dataflow analysis. The compilation time for the optimizations on the DAG was increased about 4.3 times on IA-32 and 1.6 times on IA-64. This is because many edges exist in the DAG for q.m and tb.???.

Scalar replacement using escape analysis increases the compilation times by 1% to 9%. Scalar replacement using loop versioning has little effect on the compilation times.



a) Relative execution times (Taller bars are better).



b) Relative compilation times (Shorter bars are better).

The color of the bar	Scalar replacement using loop versioning	Scalar replacement using escape analysis	Scalar replacement using dataflow analysis
Base	ON	ON	ON
	OFF	ON	ON
	OFF	OFF	ON
	OFF	OFF	OFF

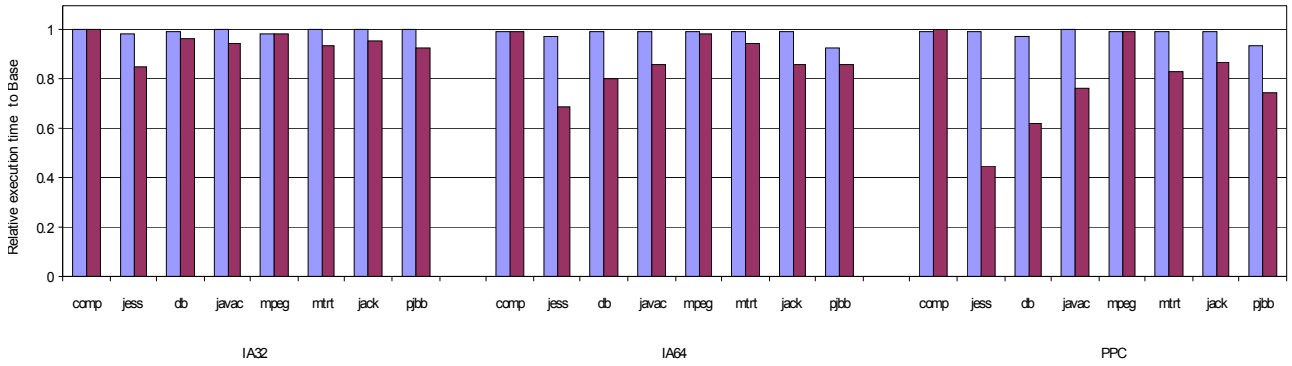
Figure 13. Measurements on scalar replacement algorithms.

4.2.4 Optimizations for TIC

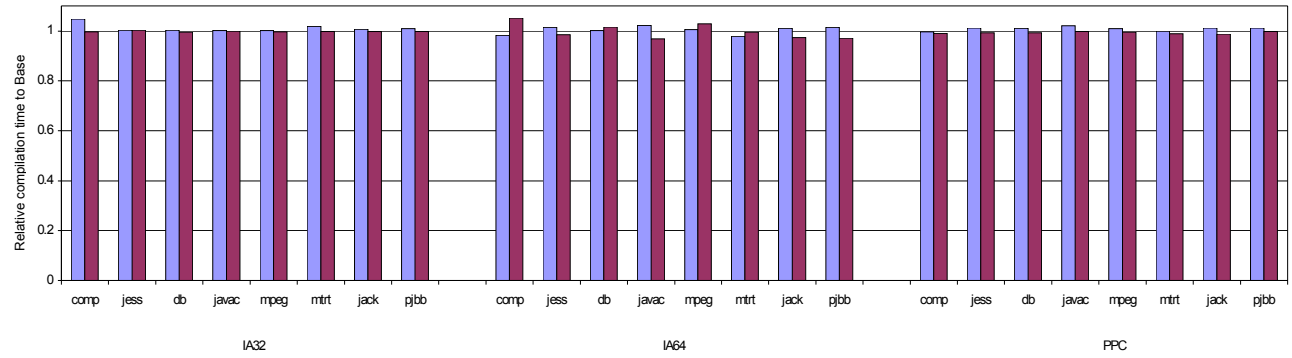
As we described in Sections 3.1 and 3.2, two kinds of optimizations for TIC are performed: elimination of redundant TICs and inlining of TICs. Figure 14 shows the relative execution times and compilation times by selectively disabling the optimizations of TIC as in the table.

Inlining of TICs improves the execution time from 0% to 55% (with an average of 14%). It is effective for jess, db, javac, jack, and pjbb that have more complicated class hierarchies than the other programs. It is most effective on the PowerPC, and least effective on IA-32. This is due to the difference in the overhead of calling a C routine. On IA-32, the overhead is small since calling C is very simple. On IA-64, the overhead is also small because of the register stack engine. On the PowerPC, there is some overhead, such as saving and restoring the non-volatile registers. Eliminating redundant TICs does not affect the execution time for these programs except for pjbb.

Inlining and eliminating redundant TICs has little effect on the compilation times. In the exceptional cases of javac and jack on IA-64, the compilation time increases by 4%. This is due to the fact that the compilation times for register allocation were increased by inlining of TICs. We suspect that this is due to the increase in the number of BBs.



a) Relative execution times (Taller bars are better).



b) Relative compilation times (Shorter bars are better).

The color of the bar	Elimination of redundant TICs	Inlining of TICs
Base	ON	ON
	OFF	ON
	OFF	OFF

Figure 14. Measurements on TIC optimizations

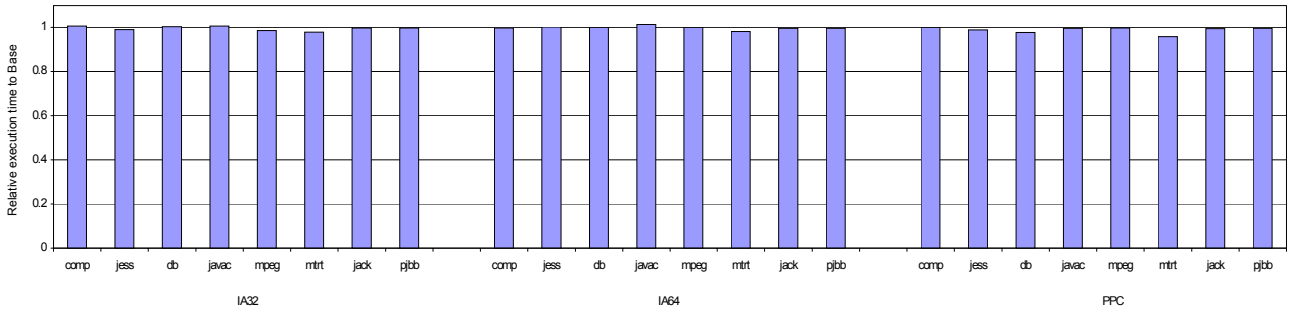
4.2.5 Elimination of merge points in the CFG

As we described in Section 3.2, eliminating merge points in the CFG, called frequency-directed splitting, is performed. Figure 15 shows the relative execution times and compilation times by disabling the elimination of merge points in the CFG as in the table.

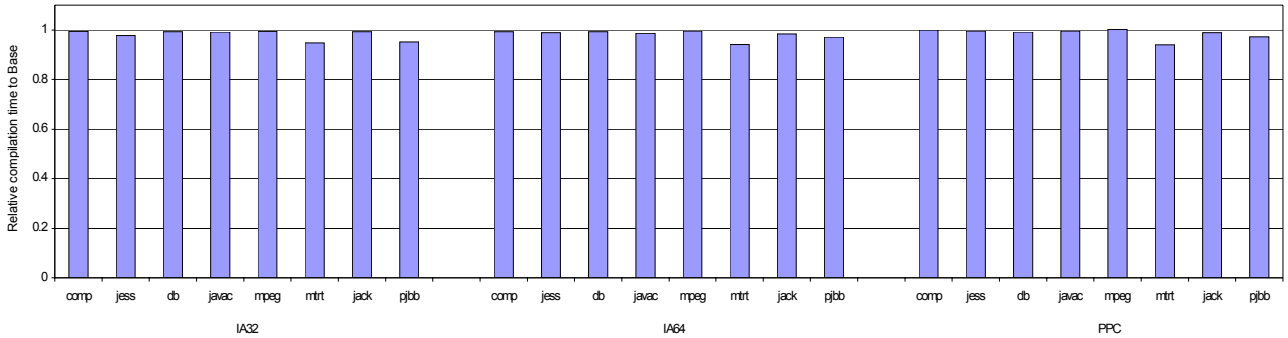
It improves the performance by 2% and 4% for mtrt on the IA-64 and PowerPC, respectively. In addition, it is effective for other programs on PowerPC. Since it improves the precision of dataflow analysis, dataflow optimizations such as common subexpression elimination and scalar replacement can be performed more effectively. Therefore, it is more effective on the architectures with many registers such as IA-64 and PowerPC.

Eliminating merge points on the CFG does not improve the execution times of most of the programs. This result contradicts other research work [25], which reported the effectiveness of splitting (a variation of eliminating merge points). This is due to the fact that they apply it with guarded devirtualization, while we apply it with direct devirtualization by code patching [6]. The overhead with guarded devirtualization is higher than that with direct devirtualization, and eliminating the merge points to reduce their overhead is more effective when the overhead is high.

With this optimization, the compilation time for mtrt is increased by 6%. This is due to the fact that the compilation times for optimizations on the QUADs were increased by disabling frequency-directed splitting. This is because the number of BBs is increased.



a) Relative execution times (Taller bars are better).



b) Relative compilation times (Shorter bars are better).

The color of the bar	Frequency-directed splitting
Base	ON
	OFF

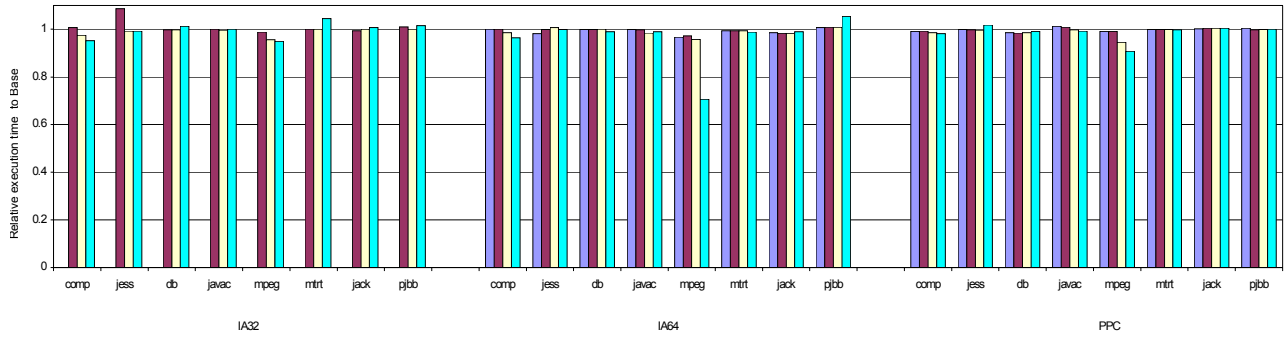
Figure 15. Measurements on elimination of merge points in the CFG.

4.2.6 Optimizations on DAG

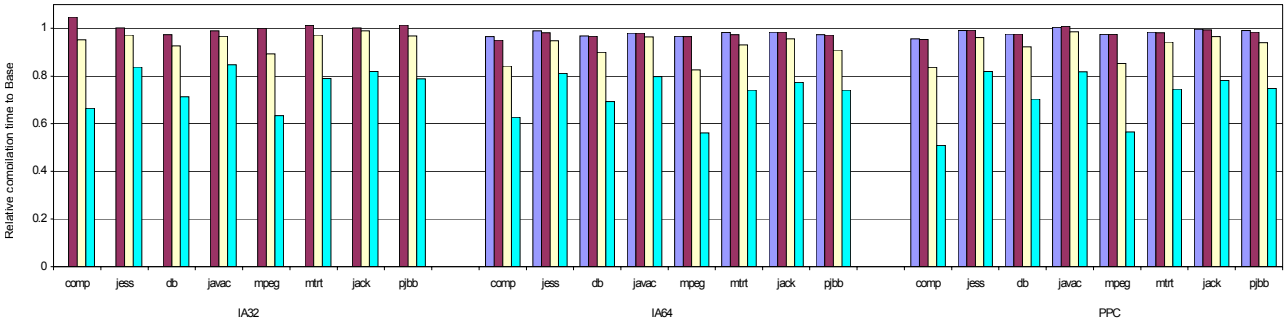
As we described in Section 3.3, four kinds of optimizations on the DAG are performed: generation of loops using a dedicated loop count register and instructions with updates, scalar replacement using loop versioning, exception check elimination using loop versioning, and pre-pass scheduling. Figure 16 shows the relative execution times and compilation times by selectively disabling optimizations on the DAG as in the table.

Pre-pass code scheduling is effective for compress and mpegaudio, which are loop-centric programs. In particular, it improves the execution time for mpegaudio by 25.1% on the IA-64. This is because the architecture can exploit higher instruction level parallelism available in a program than other architectures. Exception check elimination using loop versioning is effective for those programs that frequently access array elements, such as mpegaudio. There are few opportunities for scalar replacement using loop versioning in the programs selected for this experiment, but this optimization is intended to improve the execution times of the programs with frequent memory accesses to the same array elements, such as FFT. Except for mpegaudio on IA-64, it has little effect on the execution times to generate loops with a dedicated loop count register and instructions with updates.

The DAG-based optimizations increase the compilation times significantly (by 10% through 44%), but they are effective only for a few programs such as compress and mpegaudio. In particular, they greatly increase the compilation times for these two programs (18% for compress and 32% for mpegaudio). In general, despite the large compilation time, their effectiveness is limited to the loop-intensive programs, and therefore it is important to select the target methods carefully when they are applied.



a) Relative execution times (Taller bars are better).



b) Relative compilation times (Shorter bars are better).
Since IA32 has no update instruction or loop count register, the left most bars are omitted for IA32.

The color of the bar	Generation of loops using a dedicated loop count register and instructions with update	Scalar replacement using loop versioning	Exception check elimination using loop versioning	Pre-pass scheduling
Base	ON	ON	ON	ON
	OFF	ON	ON	ON
	OFF	OFF	ON	ON
	OFF	OFF	OFF	ON
	OFF	OFF	OFF	OFF

Figure 16. Measurements on optimizations on DAG.

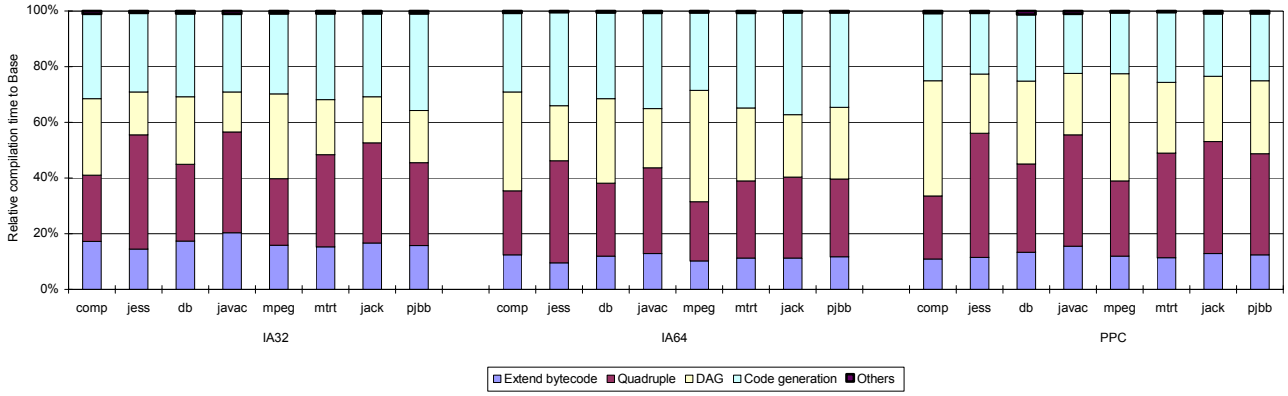


Figure 17. Breakdown of the compilation times with all of the optimizations enabled (Base).

4.2.7 Breakdown of compilation times

Figure 17 shows the breakdown of the compilation times for each program with all the optimizations enabled on every platform. Roughly speaking, the compilation time is spent in a ratio of 15:25:30:30, in corresponding to each of the four phases: the optimizations on the EBC, those on the QUAD, those on the DAG, and the code generation, respectively. As exceptional cases, the optimizations on the DAG take much longer for compress and mpegaudio, but they are quite effective for these two programs.

In general, the effectiveness of each optimization varies, but its compilation time has a similar overhead across all the platforms. For example, scalar replacement is effective on IA-64 and PowerPC, but it is not effective on IA-32. Nevertheless it incurs almost equal compilation time on all of the platforms.

Finally, register allocation in the code generation phase accounts for 10% and 4% of the total compilation time on IA-64 and PowerPC, respectively, though it is not shown in Figure 17. This is quite different from the HotSpot Server Compiler [3], whose graph coloring register allocator reportedly accounts for 49% of the total compilation time. We suspect that this is due to the fact that our JIT compiler employs more time-consuming optimizations such as those on the DAG.

4.2.8 Selected optimizations aiming at lightweight compilation

In practice, it is important to select a small set of the optimizations most effective for general programs with the shortest compilation times. To this end, method inlining of tiny methods, exception eliminations using forward dataflow analysis, scalar replacement using dataflow analysis, and inlining of TICs are the most effective ones, which can achieve 80% of the peak execution time, at the expense of the overhead of the compilation time by up to 10%.

Some optimizations are effective for particular programs at the expense of a small overhead in compilation time. For example, eliminating merge points in the CFG is effective for mtrt, while its compilation time overhead is less than 5%. Eliminating redundant TICs is effective for jess, db, and pjbb, while its compilation time overhead is small. Eliminating exception checks using PRE is also effective for compress, mpegaudio, and mtrt, while its compilation time overhead is limited to 3%.

On the other hand, some optimizations are effective for particular programs at the expense of a large overhead in compilation time. For example, escape analysis is effective for mtrt, while its compilation time overhead is up to 9%. Optimizations on the DAG are effective for mpegaudio, while the compilation time overhead is nearly 30%. Method inlining with static heuristics is effective for compress, jess, mpegaudio, mtrt, and pjbb, while its compilation time overhead is up to 68%.

In summary, we can categorize our optimizations into four classes:

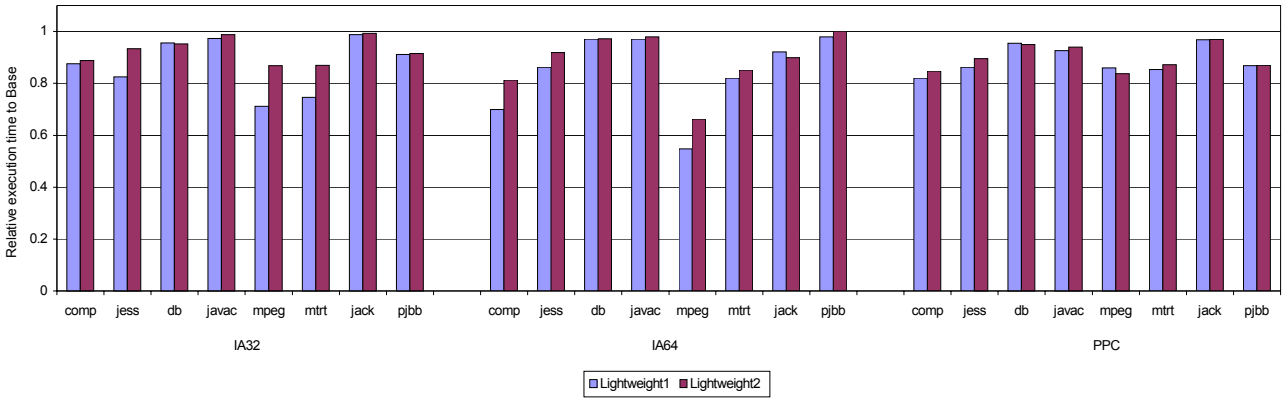
(a) Generally effective optimizations with small compilation overhead:
- Method inlining for tiny methods
- Exception check eliminations using forward dataflow analysis
- Scalar replacement using dataflow analysis
- Inlining of TICs

(b) Occasionally effective optimizations with small compilation overhead:
- Exception elimination optimizations using PRE
- Elimination of redundant TICs
- Elimination of merge points in the CFG

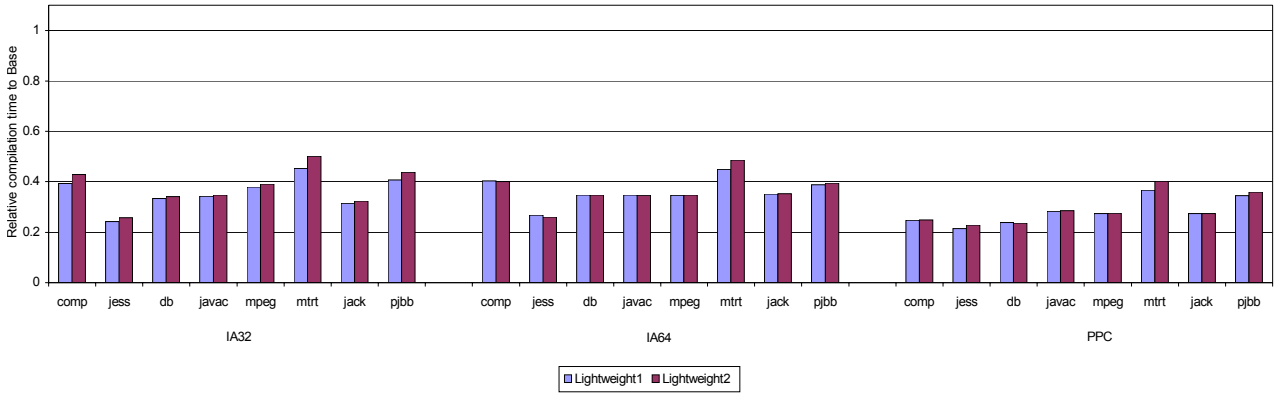
(c) Occasionally effective optimizations with large compilation overhead:
- Method inlining with static heuristics
- Scalar replacement using escape analysis
- Optimizations on the DAG

(d) Others (ineffective).

We created two sets of optimizations, **Lightweight1** and **Lightweight2**, corresponding to the optimization class (a) and the optimization classes (a) and (b), respectively. The results are shown in Figure 18. **Lightweight1** achieved 86% of the peak execution time of **Base** (all the optimizations enabled), while it only took 33% of the compilation time of **Base**. **Lightweight2** achieved 90% of the peak execution time of **Base** (all the optimizations enabled), while it only took 34% of the compilation time of **Base**.



a) Relative execution times (Taller bars are better).



b) Relative compilation times (Shorter bars are better).

Figure 18. Selected optimizations.

For achieving the best execution times on benchmark programs, it is important to enable all of the optimizations. Yet, for practical purposes, it is more effective to enable only a few important optimizations such as those selected for **Lightweight1** and **Lightweight2**. Although challenging, it would be ideal if the JIT compiler could automatically choose a customized set of those optimizations, which are most effective for the target program, by analyzing the characteristics of the target program and its profiling information.

5. RELATED WORK

There are quite a few Java runtime environments available today. Sun's HotSpot and IBM DK are the two major production runtime environments, while IBM's Jikes Research Virtual Machine (RVM) [41] and Intel's Open Research Platform (ORP) [42] Java Virtual Machine are the two major research runtime environments. Interestingly, these two production environments employ an interpreter and optimizing compilers, while these two research environments employ only compilers.

The HotSpot Virtual Machine includes an interpreter that supports mixed execution and the Java HotSpot compiler that supports the IA-32, the IA-64, and the 32/64-bit SPARC architectures. The paper [3] described the detailed implementation of the compiler, and evaluated the effectiveness of some optimizations on the IA-32 and SPARC architectures. It uses the DAG representation based on SSA throughout the optimizations and register allocation. It uses BURS [43] for portable code generation. It performs a single level of optimizations, including method inlining, global code motion, and local code scheduling.

IBM DK optionally enables a dynamic optimization framework [2]. It can trigger recompilation with specialization using instrumentation code to improve the performance. Some study has been conducted with various policies for profile-directed method inlining to improve the performance and reduce the compilation time [18], as one of the promising approaches for choosing a customized set of those optimizations that are most effective for the target program.

Jikes RVM supports the IA-32 and PowerPC architectures with multiple optimization levels. It uses three IRs, which are all register-based. The high-level intermediate representation (HIR) adopts almost the same set of opcodes as Java bytecode, and it explicitly represents PEIs unlike Java bytecode. Using HIR, the compiler performs simple optimizations such as local optimizations within an extended BB, flow-insensitive optimizations, and method inlining. Then, the low-level intermediate representation (LIR) expands the HIR into operations that are specific to the RVM object layout and conventions. The LIR is still independent of the target machine architecture. Here,

SSA-based optimizations such as common subexpression elimination and scalar replacement are performed as part of machine-independent optimizations. Finally, a machine-specific intermediate representation (MIR) is generated from the LIR. The MIR depends on the target machine architecture. Here, register allocation is performed as in our system. The native code is generated using BURS. The approach using different IRs is similar to that of our system. It uses a compiler with multiple optimization levels to reduce the overhead of the compilation time by triggering recompilation using a polling-based profiler. The parameters for some optimizations can be tuned using feedback-based optimizations to increase the performance of the target program [44].

The Intel ORP is also a research virtual machine with two compilers. One is the simple code generator (known as the O1 JIT [45]) that produces native code directly from the Java bytecode with lightweight optimizations. The other is the optimizing compiler (known as the O3 JIT [46]) that translates the Java bytecode to an IR that can be used for time-consuming optimizations such as method inlining, global dataflow optimizations, and loop versioning. When a method is invoked, the O1 JIT compiles the method with counter-based instrumentation code. When the counter for a method reaches a predetermined threshold, the O3 JIT recompiles that method with predetermined optimizations, including feedback-based optimizations. For example, if the profiling data shows that a loop is not iterated frequently, loop versioning is not performed against that loop.

The Intel JIT shipped with Intel VTune also supports the IA-32 architecture. It employs only a compiler with a single level optimization, since it was released some times ago. The paper [45] evaluated the performance and the compilation time with this compiler by selectively disabling optimizations. These optimizations without an explicit IR are lightweight and designed to have short compilation times since all methods are compiled.

6. CONCLUSION

We described the system overview of our Java JIT compiler, which is the basis for the latest production version of the IBM Java JIT compiler that supports a diversity of processor architectures, including both 32-bit and 64-bit modes, CISC, RISC, and VLIW architectures. In particular, we focused on the design and evaluation of the cross-platform optimizations that are common across different architectures. We studied the effectiveness of each optimization by selectively disabling it in our JIT compiler on three different platforms: IA-32, IA-64, and PowerPC. Based on the detailed statistics, we classified our optimizations and identified a small set of the most cost-effective ones in terms of the performance improvement as the benefit and the compilation time as the cost. In summary, we demonstrated that, with a selected set of optimizations, we can achieve 90% of the peak performance for SPECjvm98 and a special version of SPECjbb2000 at the expense of only 34% of the total compilation time in comparison to the case in which all of the optimizations are enabled. In the future, we plan to study a new, dynamic compilation strategy to automatically choose a customized set of those optimizations, which are most cost-effective for the target program, based on the structure of its methods and the online-profile information collected on the fly for that program.

ACKNOWLEDGEMENTS

We would like to thank the IBM Software Group in Toronto for taking over and continuing the development and the services of the IBM Java JIT compiler. We are grateful to the former members of our team, including Hiroyuki Momose, Kazuhiro Konno, and Kunio Tabata, for their various contributions. We are also grateful to John Whaley for prototyping escape analysis, Janice Shepherd for experimenting with the enhanced memory system, Arvin Shepherd for implementing a linear scan register allocator, Koblets Gita for implementing the optimizations for loop striding and countdown loops, Hiroshi Dohji for implementing an MMI for IA-32, Masao Nishimoto for implementing an MMI for S/390, Akio Watanabe for implementing some optimizations for S/390, Masashi Doi for implementing a register allocation of FPRs for IA-32, and Akihiko Togami for providing excellent services for problem determination. We also thank Shannon Jacobs for his editorial assistance. Finally, we appreciate the insightful comments from the anonymous reviewers.

REFERENCES

[1] K. Arnold and J. Gosling. Java Programming Language, Addison-Wesley, 1996.

[2] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler, In *the Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 180-194, 2001.

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

[4] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal*, Vol. 39, No. 1, pp. 175-193, 2000.

[5] M. Kawahito, H. Komatsu, and T. Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap, In *the Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems*, pp. 139-149, 2000.

[6] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler, In *the Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2000.

[7] The Standard Performance Evaluation Corp., SPECjvm98 Benchmarks, available at http://www.spec.org/osg/jvm98/.

[8] The Standard Performance Evaluation Corp., SPECjbb2000 Benchmarks, available at http://www.spec.org/osg/jbb2000/.

[9] T. Onodera and K. Kawachiya. A Study of Locking Objects with Bimodal Fields, In *the Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 223-237, 1999.

[10] D. F. Bacon, R. Konuru. C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java, In *the Proceedings of ACM SIGPLAN '98 Conference on*

Programming Languages Design and Implementation, pp. 258-268, 1998.

[11] R. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, 1996.

[12] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko. A Parallel, Incremental and Concurrent GC for Servers, In *the Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, pp. 129-140, 2002.

[13] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. ACM Transactions on Computer Systems, Vol. 11, No. 4, pp. 376-408, 1993.

[14] C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano, and H. Srinivasan. Dependence analysis for Java, In *12th International Workshop on Languages and Compilers for Parallel Computing* (LCPC'99), Volume 1863 of LNCS, Springer-Verlag, pp. 32-52, 1999.

[15] J.-D. Choi, D. Grove, M. Hind, V. Sarkar. Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs, In ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99), pp. 21-31, 1999.

[16] J. Palsberg and M. I. Schwartzbach, Object-Oriented Type Inference, In *the Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 146-161, 1991.

[17] E. M. Gagnon, L. J. Hendren, and G. Marceau. Efficient Inference of Static Types for Java Bytecode, In S*tatic Analysis 7th international Symposium* (SAS'00), Volume 1824 of LNCS, Springer-Verlag, pp. 199-219, 2000.

[18] T. Suganuma, T. Yasue, and T. Nakatani. An Empirical Study of Method Inlining for a Java Just-In-Time Compiler, In *USENIX 2nd Java Virtual Machine Research and Technology Symposium*, pp. 91-104, 2002.

[19] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy, In the proceed*ings of the 9th European Conference on Object-Oriented Programming - ECOOP*, Volume 952 of LNCS, Springer-Verlag, pp. 77-101, 1995.

[20] D. Detlefs and O. Agesen. Inlining of Virtual Methods, In the proceed*ings of the 13th European Conference on Object-Oriented Programming - ECOOP*, Volume 1628 of LNCS, Springer-Verlag, pp. 258-278, 1999.

[21]R. Gupta. Optimizing array bound checks using flow analysis, *ACM Letters on Programming Languages and Systems*, Vol. 2, No. 1-4, pp. 135-150, 1993.

[22] M. Kawahito, H. Komatsu, and T. Nakatani. Eliminating Exception Checks and Partial Redundancies for Java Just-in-Time Compilers. IBM Research Report RT0350, 2000.

[23] A. Aho, V. Jeffrey, and D. Ullman, Principles of Compiler Design. Addison-Wesley, 1977.

[24] C. Chambers and D. Ungar. Making pure object-oriented languages practical, In *the Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 1-15, 1991.

[25] M. Arnold, B. G. Ryder. Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading, In the *Proceedings of the 16th European Conference on Object-Oriented Programming - ECOOP*, Volume 2374 of LNCS, Springer-Verlag, pp. 498-524, 2002.

[26] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs, In *the Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 187-206, 1999.

[27] J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion, *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 5, pp. 777-802, 1995.

[28] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, T. Nakatani. Design, Implementation, and Evaluation of Optimizations in a Java(tm) Just-In-Time Compiler, *Concurrency: Practice and Experience*, Vol. 12, No. 6, pp. 457-475, 2000.

[29] B. Alpern, A. Cocchi, and D. Grove. Dynamic type checking in Jalapeno, In *USENIX 1st Java Virtual Machine Research and Technology Symposium* (JVM'01), pp. 41-52, 2001.

[30] T. Ogasawara, H. Komatsu, and T. Nakatani. A Study of Exception Handling and Its Dynamic Optimization in Java, In *the Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 83-95, 2001.

[31] 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 Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 451-490, 1991.

[32] 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 Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 451-490, 1991.

[33] V. C. Sreedhar, R. D. Ju, D. M. Gillies, and V. Santhanam. Translating Out of Static Single Assignment Form, In *Static Analysis 6th international Symposium (SAS'99)*, Volume 1694 of LNCS, pp. 194-210, 1999.

[34] J. R. Allen and K. Kennedy and C. Porterfield and J. Warren. Conversion of control dependence to data dependence, In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, pp. 177-189, 1983.

[35] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock, In *the proceedings of the 25th International Symposium on Microarchitecture*, pp. 45-54, 1992.

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

[37] M. Kawahito, H. Komatsu, and T. Nakatani. Effective Sign Extension Elimination, In *the Proceedings of SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 187-198, 2002.

[38] M. Poletto and V. Sarkar. Linear scan register allocation, *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 5, pp. 895-913, 1999.

[39] G. J. Chaitin. Register allocation and spilling via graph coloring, In *the proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No. 6, pp. 257-265, 1982.

[40] D. Stefanovic, M. Hertz, S. M. Blackburn, K. S. McKinley and J. E. B. Moss. Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine, ACM SIGPLAN Workshop on Memory System Performance, pp. 25-36, 2002.

[41] 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. Implementing Jalapeno in Java, *IBM System Journal*, Vol 39, No 1, pp. 211-238, 2000.

[42] M. Cierniak, B. T. Lewis, and J. M. Stichnoth. Open Runtime Platform: Flexibility with Performance using Interfaces, In *the Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pp. 156-164, 2002.

[43] E. Pelegri-Llopart and S. L. Graham. Optimal code generation for expression trees: an application of the BURS theory, In *the Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 294-308, 1988.

[44] M. Arnold, M. Hind, and B. G. Ryder. Online Instrumentation and Feedback-Directed Optimization of Java, In *the Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 111-129, 2002.

[45] A.-R. A.-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parakh, and J. M. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler, In *the Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 280-290, 1998.

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