Overview of the IBM Java Just-in-Time Compiler

by T. Suganuma
T. Ogasawara
M. Takeuchi
T. Yasue
M. Kawahito
K. Ishizaki
H. Komatsu
T. Nakatani

We present the design and implementation of several optimizations and techniques included in the latest IBM Java™ Just-in-Time (JIT) Compiler. We first discuss some of the modifications we have applied to Sun Microsystems' reference implementation of the Java Virtual Machine (JVM™) Specification to increase the performance, including a change in the object layout. We then describe each of the optimizations, referring to what had to be taken into account because of both the just-in-time nature of the compiler and the requirements of the Java language specification, such as exception checking. We also present code generation techniques targeting Intel architectures, describing the register allocation schemes, exception handling, and code scheduling. Finally we report on the performance of the IBM JIT compiler, showing both the effectiveness of the individual optimizations and the competitive overall performance of the JIT compiler in comparison with a competitor, using industry-standard benchmarking programs. All the techniques presented here are included in the official product (JIT Compiler version 3.0), which has been integrated into the IBM Developer Kit for Windows™, Java Technology Edition, Version 1.1.7.

The Java** language[1] has rapidly been gaining importance as a standard object-oriented programming language since its advent in late 1995. Java source programs are first converted into an architecture-neutral distribution format, called Java bytecode, and the bytecode sequences are then interpreted by a Java virtual machine (Jvm)[2] for each platform. Although its platform-neutrality, flexibility, and reusability are all advantages for a programming language, the execution by interpretation imposes an unacceptable performance penalty, mainly on account of the run-time overhead of the bytecode instruction fetch and decode. One means of improving the run-time performance is to use a just-in-time (JIT) compiler, which converts the given bytecode sequences "on the fly" into an equivalent sequence of the native code of the underlying machine. It significantly improves the performance, but the overall program execution time, in contrast to that of a conventional static compiler, now includes the compilation overhead of the JIT compiler. It is therefore very important for the JIT compiler to be fast and lightweight, as well as to generate high-quality native code.

There are several reasons for the poor execution performance of programs written in the Java language.

First of all, because of the standard object-oriented programming practices, there tend to be many relatively small methods that lead to more frequent method invocations than is the case in other languages. For example, there may be a method solely for accessing a private field variable. An object constructor method is also automatically created in Java even if it is not explicitly written in the program, and consequently there are many empty object constructor methods. Polymorphic method invocations[3] create another performance problem because of the overhead of indirectly calling through a dynamic method table lookup. This problem is aggravated by the reusability of object-oriented code, since programmers are encouraged to incorporate general class libraries or existing frameworks designed for general use into new programs to increase their productivity, and this practice in general has resulted in the creation of many polymorphic method invocation sites. Even if the class is actually not overridden in a particular program, the polymorphic method invocation overhead is still imposed; it likely occurs in many cases. All these problems are inherent in object-oriented programming and can be an important factor in the overall performance.

Second, the Java language specification requires run-time exception checking to ensure the validity of accesses to objects and arrays. If a reference or an operation is invalid, an exception must be thrown, and the environment, such as local variables, has to be preserved and made available to an exception handler that catches the exception. This condition imposes a large penalty in program execution and prevents the application of conventional loop optimization techniques. The type inclusion test is another factor in the run-time overhead of checking type conformance, which not only results from an explicit request by the programmer using instanceof operations, but is implicitly required by a statement assigning an object to another type.

Third, the synchronized methods or synchronized blocks, which are used to ensure the atomicity for a set of operations in a region, cause run-time overhead by locking a given object for the duration of the execution. Again, the reusability of object-oriented code makes this problem worse, since general-purpose class libraries are designed for use in the multithreaded environment, and synchronized methods or blocks are heavily used where thread safety must be guaranteed. When these libraries are used by single-threaded programs, there will be substantial performance degradation caused by unnecessary synchronization.

We address all these problems in the IBM JIT compiler by optimizing the original bytecode sequences and applying various code generation techniques, such as method inlining, loop versioning, fast type inclusion testing, and others, while keeping the original program semantics. Special care must be taken to ensure that any optimization here allows any exceptions that would have been thrown in the original program to still be thrown in the optimized code. That is to say, the exceptions must be thrown in precisely the same order with respect to the rest of the program execution.

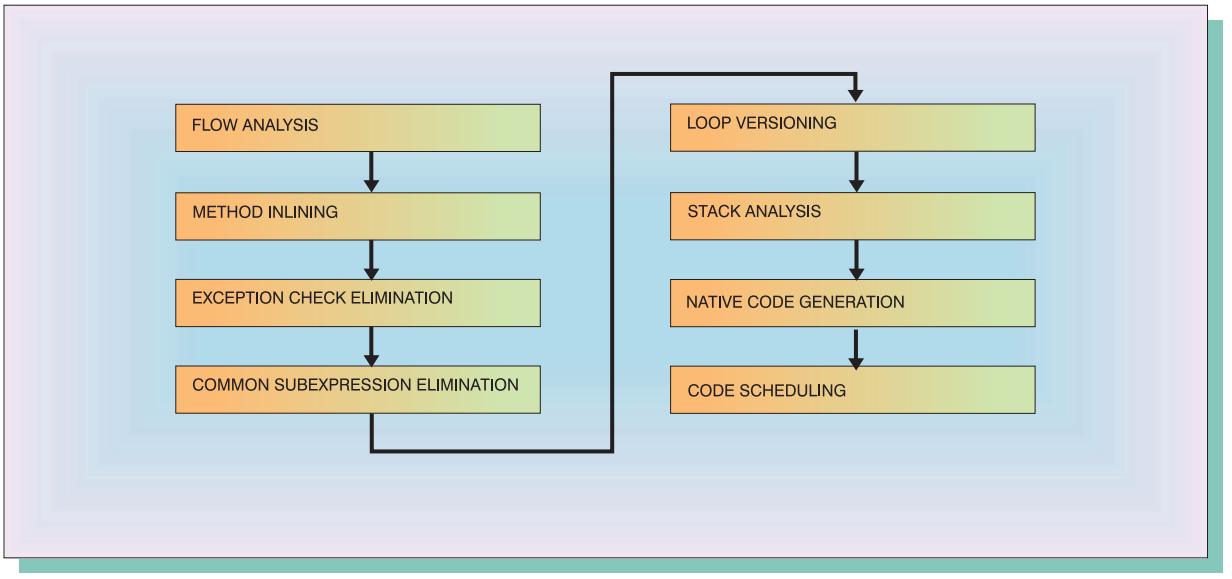
The next section of this paper describes the overall structure of the IBM JIT compiler and some of the modifications we applied in Sun Microsystems' reference implementation of the Java Virtual Machine (JVM**) Specification. The third section discusses each of the bytecode-level optimizations. The code generation technique, including the register allocation scheme, is described in detail in the fourth section. The fifth section presents some measurements obtained by using some industry-standard benchmarking programs and shows both the effectiveness of each of the optimizations presented in this paper and the overall performance of the compiler in comparison with those of a competitor on platforms based on Intel processors. The sixth section summarizes related works, and the final section concludes the paper.

Overview

We first present an overview of the JIT compiler before some of the details are described.

JIT compiler structure. The overall flow diagram of the IBM JIT compiler is shown in Figure 1. The structure is basically similar to that used in typical static compiler environments. First, in the flow analysis phase, the basic blocks and the loop structures are generated by performing a linear-time traversal of the given bytecode sequences. The given bytecode is then converted into an internal representation, called extended bytecode, with some new opcodes introduced to represent operations resulting from the optimizations. An example of the new opcodes, which are produced by common subexpression elimination, is one to obtain the array object interior pointer for a common effective address. It allows consecutive array elements to be accessed by simple

Figure 1 JIT compiler overall structure



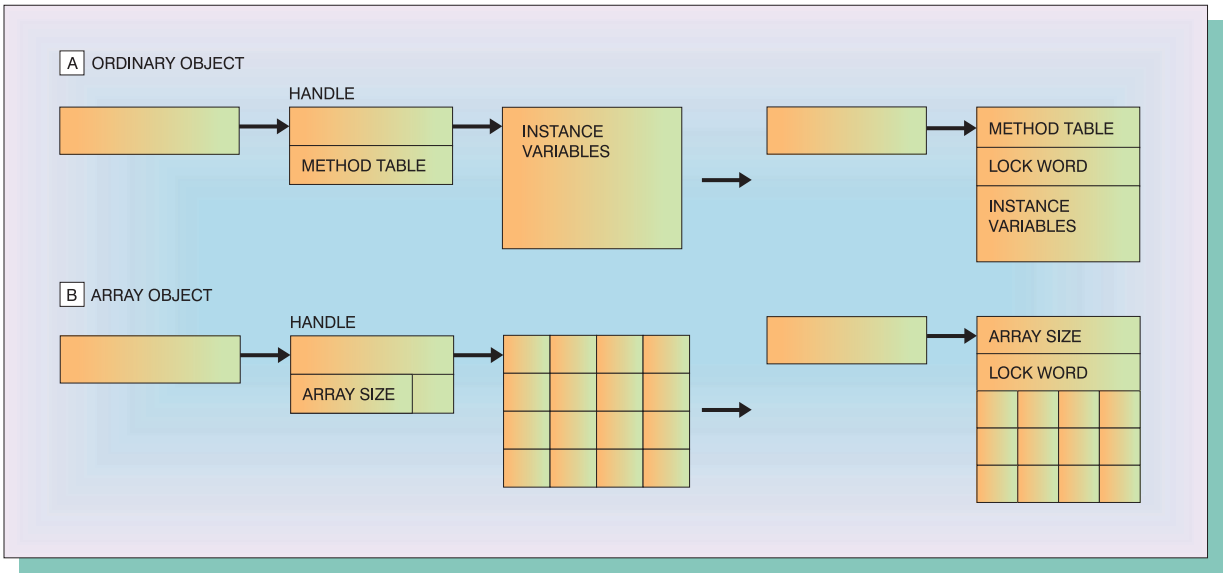
load-and-store instructions without indexing each element.[4] The extended bytecode, where stack semantics are still retained as in the original bytecode, is chosen as the internal representation in order to avoid the conversion cost in the compilation process. The optimization phase then starts, and several techniques are applied to this internal representation: method inlining, exception check elimination, common subexpression elimination, and loop versioning, in that order (all described in detail in the next section). Optimizations based on dataflow analysis, such as constant propagation and dead code elimination, are also applied to the internal representation. The variables in stack computation are then separately mapped to each of the logical registers for integer and floating-point calculation by traversing bytecode stack operations. The region for register allocation of local variables is also defined, and the number of accesses of local variables in each region is counted.

Finally, native code is generated on the basis of the optimized sequences of extended bytecode by allocating physical registers for each stack and local variable. To reduce the compilation time, there is no independent single pass for the register allocation; that is, the register allocation runs synchronously with the code generation. Simple code scheduling within a basic block is applied to reorder the instructions so that they best fit the requirements of the underlying machine. The JIT compiler has a potential advantage over the traditional compilation technique in that it can identify the type of machine it is running on, and we make use of this information in both code generation and code scheduling.

Base Jvm modifications. Among a number of enhancements included in the IBM Jvm that have been applied to Sun's reference implementation, two are major changes introduced to improve the overall performance of the JIT compiler: a change in the object layout and the execution of the static initializer.

Figure 2 shows the change in the object layout for both ordinary objects and array objects. An object is originally pointed to through a handle, causing an extra level of indirection every time object fields and array elements are accessed. In the new object layout, handles are removed, and each object has a two-word header instead of a handle. With this change, we can directly access instance fields simply by adding an extra offset to the object pointer. Furthermore, the element size now occupies the first word of the array object header, which allows the JIT compiler to generate array index out-of-bounds checking code by issuing just one instruction, whereas several instructions were required, including load and shift operations, with the original object layout. This is a

Figure 2 Change in object layout



great advantage in terms of code generation efficiency, since the array bound exception checking has to be done every time an array element is accessed, unless it has been proved to be safe. The space overhead per object is unchanged by this modification, two words for a header and one word for heap maintenance, and the space reserved for the handle array is totally unnecessary now.

The other important change we have made to the reference implementation of the JVM specification is to separate the resolution of a class from the execution of its static initializer. In the original Jvm implementation, whenever a class is resolved, an attempt is always made to run its static initializer. This action is good for the interpreter execution, because the attempt to resolve a class is always made at run time, which is exactly the time at which the static initializer is run. But from the standpoint of the JIT compiler, a class needs to be resolved at compile time as much as possible in order to obtain the addresses of methods and object fields and thus to generate better code. Otherwise, code has to be generated for calling the run-time class resolution routine, and dynamic code patching is required at run time. Since running a static initializer entails a side effect, it cannot be run speculatively at compile time. By separating the class resolution and the execution of its static initialization, the JIT compiler has more opportunity to generate faster code, using run-time calls if necessary to run the static initializer. Thus, the use of class resolution without static initialization is a very important modification for high-performance JIT compilers.

Other enhancements in the IBM version of the Jvm include object allocation, monitor optimization,[5] and class libraries. The IBM JIT compiler also takes advantage of these enhancements. The monitor optimization dramatically improves performance for the majority of object synchronization operations, namely locking an unlocked object or locking an object already locked by the current thread a small number of times (shallow nested locking), by attaining the operation with just a few machine instructions. This optimization takes advantage of the new object layout just described and uses a portion of the second word of the object header for encoding various information, such as the thread identifier, the nested lock count, and the inflation bit. The inflated locks support the general case of heavyweight locks, where contention can occur between multiple threads.

Selective compilation. Since JIT compilation occupies a part of the application run time, it is not necessarily beneficial to compile all the methods being invoked. For example, when a method is executed

only once and does not contain any loops, the overall performance might be degraded if it is JIT-compiled. The cost of the JIT compilation needs to be offset by the performance gain achieved by running the native code in terms of both time and space. The IBM JIT compiler and Jvm allow efficient mixed-mode execution of interpreter and JIT-generated code by sharing the execution stack and exception-handling mechanism. By adopting an appropriate way of identifying and choosing "hot" methods that deserve JIT compilation, it is expected to achieve high performance in running real applications as well as benchmarking programs.

The notion of mixed execution of interpretation and compiled code was considered as a continuous compiler or smart JIT approach in Plezbert and Cytron.[6] In deciding when a given compilation unit should be translated to native code, they propose a model for making the choices on the fly based on some experiments. Our current strategy for selecting hot methods to be compiled is quite simple. A method invocation count is provided for each method and initialized as a certain threshold value. Whenever the method is executed by the interpreter, the invocation count is decremented. When the count reaches zero, it is determined that the method has been invoked frequently enough, and JIT compilation starts for the method. If the method includes a loop, however, it is considered to be very JIT-sensitive and handled in a different way. When the interpreter detects a loop backedge, it "snoops" the loop iteration count on the basis of a simple bytecode pattern-matching sequence, such as iload/sipush/if_cmp_lt, and then adjusts the amount by which the invocation count is decremented, depending on the iteration count. In the extreme case, where the iteration count is large enough, the control is immediately transferred to the JIT compiler from the partially interpreted code by dynamically changing the frame structure for the JIT use and jumping to a specially generated padding code. The JIT compilation for such methods can be done without sacrificing any optimization features.

The run-time trace information is another benefit of mixed mode execution for the JIT compiler. For any conditional branches encountered for the first time, the interpreter keeps information on whether the branch is taken or not taken to provide the JIT compiler with a guide for what direction to take at basic block boundaries. The branch instruction is then converted to the corresponding quick instruction so that this operation, including the detection of the loop backedge, is not executed a second time to minimize the performance penalty. The trace information is used by the JIT compiler for ordering basic blocks so that the code can be generated in a straightline manner between basic blocks guided by the taken branch. This information also makes a difference in how a program is cut into tiles, in which registers are allocated to variables, as described in detail later.

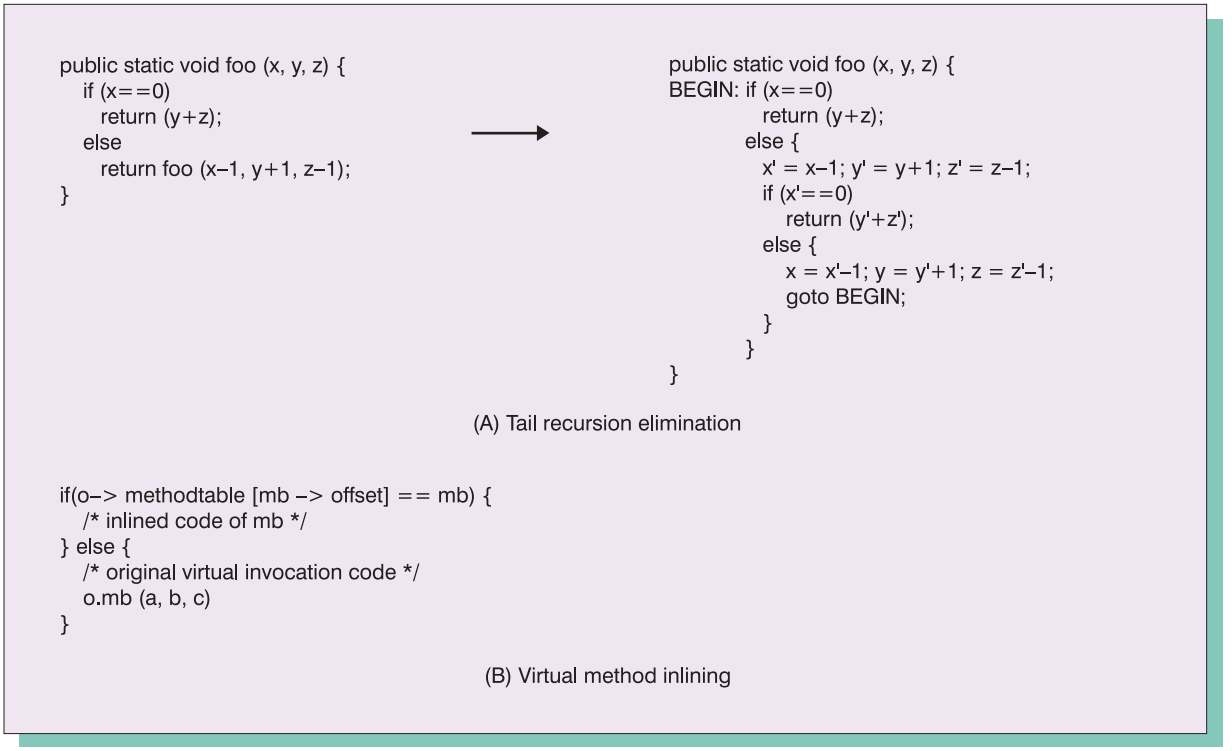
Optimizations on extended bytecode

In this section, we describe each of the optimizations for transforming the extended bytecode, namely, the internal representation. All the optimizations applied at this level will be transparent for the final code-generation phase.

Method inlining. As explained earlier, the method invocation overhead is one of the major reasons for the poor performance of Java. We deal with this problem by method inlining (which replaces calls to methods by copies of their bodies). At the bytecode level, the interpreter in Sun's Java Development Kit (JDK**) reference implementation does inline some simple methods, if the bytecode they contain fits into the space for method invocation or converts the calls to empty constructor methods to invokeignored_quick instruction. The JIT compiler can apply the technique in a more aggressive way. However, excessive application of this optimization causes an unacceptable level of code expansion, which can itself degrade the performance because of the instruction cache inefficiency. Our basic strategy is to apply method inlining only to hot spots to reduce the invocation overhead while avoiding explosive growth of the code size. By a hot spot, we mean a method invocation within a loop or an indirect method invocation from a loop.

In the case of static and nonvirtual method invocations, the problem is typically caused by empty methods, many of which come from object constructors, and small access methods, where the invocation and frame allocation costs outweigh the execution time of the method body. Since inlining these methods does not cause a code size problem, they are always inlined regardless of the hot spot. In other cases, several conditions are set as inlining criteria on the basis of (1) the nest level of invocations in the inlining analysis, (2) the number of total bytecode instructions to be inlined, (3) the number of local variables and stack variables increased in the inlining method, (4) the existence of exception tables in the inlined method, and (5) no native or virtual methods found

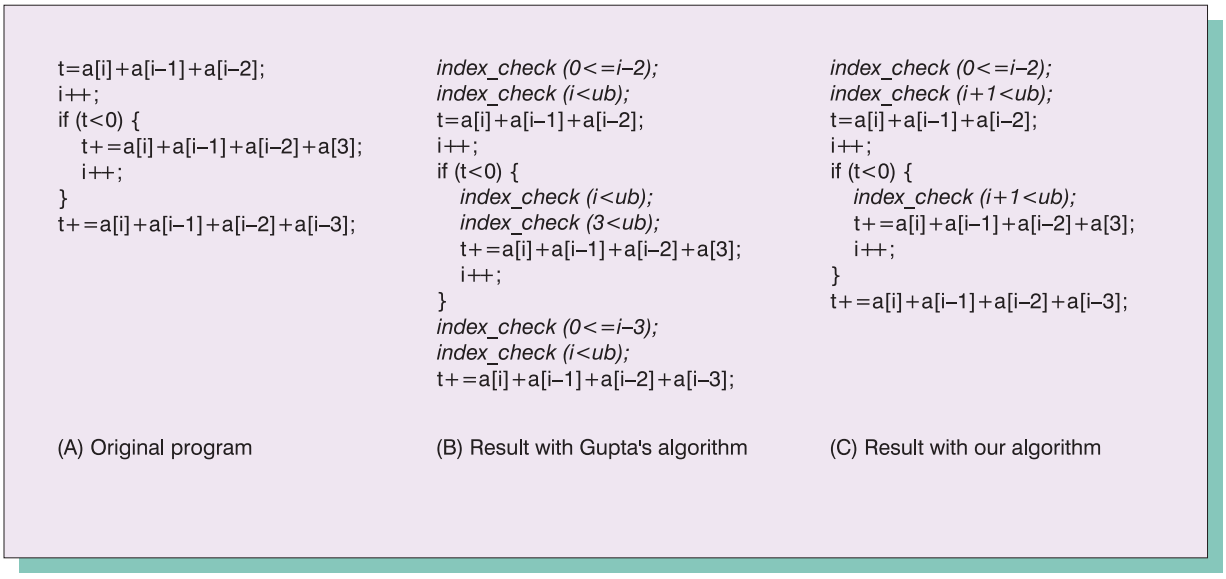
Figure 3 Examples of method inlining



in the call tree in the inlining analysis. We also optimize the recursive method invocation. A method call is replaced by the body of the method itself, reducing the number of method calls and the frame allocation cost. The tail recursion elimination, a special case of tail call optimization, converts the method invocation into the branch to the beginning of the method, eliminating the method call overhead. An example is shown in Figure 3A.

For virtual method invocations, we take advantage of the fact that the target class is frequently not overridden and that some virtual call sites actually execute only one method, namely monomorphic rather than polymorphic. We inline the target virtual method by versioning the fast path of the inlined code and the slow path of the original virtual method call in order to decrease the indirect call overhead through method table lookup, as illustrated in Figure 3B. The run-time checking code is provided to guard the inlined code to ensure that it is correct to execute for the current receiver object. Namely, the fast path inlined code is executed if the method in the receiver object method table matches the target method inlined, otherwise normal virtual invocation is performed. If the target method is only one for the virtual call site at compile time, and the class is not overridden during the course of execution, then the fast path inlined code can always be executed. The guard code is generated to test against the target method, rather than the target class, as a result of its efficiency. This method is used because the test can cover even those receiver objects whose classes are an extension of the target class, if the method inlined is not overridden by those classes. The detailed discussion and evaluation between method tests and class tests for guarding the inlining of virtual invocations is provided in Detlefs and Agesen.[7] We also optimize the interface method invocation, which is for classes with multiple inheritance. Again, we take advantage of the fact that the number of inherited classes is only one in many cases, and therefore we can skip searching the table for the implemented class. When the inlined fast path is executed, the method invocation cost for the interface class will

Figure 4 Example of exception check elimination



be almost the same as that for virtual method invocations.

Exception check elimination. One of the performance problems in executing Java programs results from the requirement of run-time exception checking to ensure the safety of program execution. Exception checking, both null pointer checking and array index out-of-bounds checking, can be eliminated if it can be proved that an exception cannot occur, namely, that the exception has already been tested somewhere along the data flow, or that the array access can be guaranteed within its bound. For the Intel-based platform, the null pointer exception can be detected by a hardware trap, as described in the later subsection on exception handling, for most of the instructions required; however, for those bytecode instructions where the object will not be touched in the generated code, such as invokenonvirtual and athrow, the explicit null pointer checking code still has to be generated for dynamic checking for the current object. Thus the elimination of null pointer checking is beneficial for Intel-based platforms as well.

The JIT compiler can eliminate null pointer checking by solving the simple data flow. Whenever an object is null-pointer-checked, regardless of whether the checking is done explicitly or implicitly in the actual code generation, a flag indicating that it has already been tested is simply propagated along the data flow. For array bound checking, in contrast, we developed a new algorithm by extending Gupta's algorithm[8] and eliminated a broader range of cases of array access. Gupta's method basically propagates the checked index set forward and backward, using data flow analysis; however, when an index variable is updated, it is treated as KILL,[9,10] and hence, the checked index set for the variable has to be reset. In the algorithm we developed, the exact range of the checked index set can be determined, including the maximum and minimum constant offset from the index variable, and this set of information is then propagated along the data flow. Even when the access index is updated by adding or subtracting a constant, it is not treated as KILL, but the checked index set is updated by offsetting the constant value. Consequently, the new algorithm can eliminate more exception checks for array accesses, especially for those with a constant index value. The example in Figure 4 shows one advantage of the algorithm, where the array index exception check required is explicitly indicated by *italicized statements*.[11] The number of exception checks, which was originally 11 (the number of array accesses), was reduced to six by the existing method and is further reduced to only three by the new method. The detailed algorithm and ex-

Figure 5 Example of loop versioning (pseudocode)

```
if ((array !=NULL) && (start>=0) && (end<=array.length) {
   /* safe loop (eliminate all array index exception checks) */
    for (i=start; i<end; i++) {
        array[i]=array[i]+const;
   }
} else {
   /*unsafe loop (original loop) */
}
```

perimental result will be reported in a future publication.

Common subexpression elimination. We apply three techniques for common subexpression elimination (CSE) to reduce the overhead of array and instance variable accesses: scalar replacement, common effective address generation, and partial redundancy elimination. Other techniques such as global value numbering,[12,13] which heavily relies on the static single assignment (SSA) form, are considered too expensive in our just-in-time compilation environment.

Scalar replacement replaces subscripted variables by local variable references and makes them available for register allocation when the array object and the index variable are not updated in the scope in which it is being applied. Common effective address generation produces an instruction pointing to the internal element of the array for consecutive array references, which typically appears in many sorting programs. This method is quite effective for reducing the register pressure, but the array object pointer needs to be retained so that it is not subject to garbage collection. In both cases, the code can be moved out of the loop if it is loop-invariant. Since a dependence DAG (directed acyclic graph) has to be constructed in order to apply both of these techniques, and this is quite expensive, we limit their application to loops, in view of the costs and benefits.

Partial redundancy elimination[14] is also used to reduce the number of accesses to instance variables. It eliminates all but one identical access on some execution path through a flowgraph and also moves invariant accesses out of loops. The instance variable accesses are then mapped to local variables, which can be allocated to physical registers in the code generation phase. An example of how redundancy elimination is applied is given later.

Loop versioning. Loop versioning is a technique for hoisting an individual array index exception check outside a loop by providing two copies of the loop: the safe loop and the unsafe (original) loop. The exception checking code is first created at the entry of the loop by examining the whole range of the index against the bound of the arrays accessed within the loop. Two versions of the target loop are then generated: one for the safe loop, which no longer requires exception checks within the loop, and the other for the original unsafe loop, with all exception checks retained, as shown in Figure 5. Depending on the result of the index range test at the time of entry, either the safe loop or the unsafe loop is selected for execution at run time. For the application of this transformation, the order of any exceptions that might be raised during the loop execution and any side effect that might occur at the time of an exception must be guaranteed to be unchanged. The current criteria for applying this optimization are as follows: (1) no method invocation exists within the loop; (2) there is a loop induction variable whose initial value and final value are both loop-invariant and whose stride is constant; and (3) for all the array accesses within the loop, the array object must be a local or a class variable that is loop-invariant, and the array indices must be constants, loop-invariant variables, or loop induction variables with constant offsets. Application of this optimization is also limited by the size of the target loop body, in view of the increase in code size.

The safe loop created by this optimization not only eliminates the exception checking code itself but imposes fewer constraints in the work on register allocation and code scheduling, thus producing better code. Also, the safe loop will allow the JIT compiler to apply other loop optimizations in the future, because the array exception checking and the array of array implementation for multidimensional arrays in Java are considered two major factors preventing the application of traditional loop optimizations and thus slowing down the execution of loops. Therefore, this technique, together with the use of rectangular multidimensional dense arrays, can pave the way for more aggressive loop optimizations, such as loop unrolling, loop interchange, and loop blocking, which should be effective for programs with numerically computation-intensive loops.[15]

Code generation details

Prior to the code generation phase, the given program is divided into several pieces, called tiles, each of which is used as a block for allocating registers. The tile cutting is done on the basis of the loop structure existing in the program; a loop is identified as a tile, and code between loops is treated as another tile. This heuristic is simple and appropriate for slicing a program in the absence of relative strength of edges between basic blocks. Information on local variable usage is then collected within each tile, and a local variable table sorted in decreasing order of access count is generated. For a nested loop, a local variable table is prepared for each level of loop nest, so that the local variables with the highest access counts in each nest level can be cached on registers.

In the code generation, several modes of addressing register operands, memory operands, and immediate operands provided by the IA32 architecture[16] are exploited. Different machine instructions are selected, depending on the underlying processor type for some bytecode instructions. In general, memory operand instructions are preferentially used with the Pentium Pro** family of processors to exploit its capability of out-of-order instruction execution and hardware register renaming. The instruction cache efficiency is also taken into account by generating a slow and rare path at the bottom of the code and compacting the fast execution path as much as possible.

Register allocation. As explained earlier, no independent single pass is used for the global register allocation in order to reduce the compilation time. We consider expensive register allocation algorithms, such as graph coloring,[17] to be inappropriate, owing to the just-in-time nature of the JIT compiler. Instead, we use a simple and fast algorithm for allocating registers that does not require an extra phase in the compilation process.

The register allocator, better called the register manager, allocates registers for stack variables as a first priority and then allocates the remaining available registers to local variables based on the usage count within each tile. During the course of the code generation, other local variables may be left unchanged in the registers by assigning stack computation results to them. Thus we have three types of registers: stack variable registers, permanent cached local variable registers, and temporary cached local variable registers. In deciding which of several available registers to allocate, the register manager adopts the circular allocation policy, returning the least recently allocated register. This policy likely results in the code sequences being less dependent on the surrounding code with respect to register usage and, hence, creates more opportunities for code scheduling work. When the code generator requires an additional register and none is available, the register manager searches for one, first in temporary local variable registers, and then in permanent local registers and stack variable registers, in that order. This simple heuristic is based on the assumption that the stack variables are generally short-lived and therefore most likely have the shortest interval to the next references. It is used to avoid the expensive computation needed to choose the optimal spill candidate.

There are some conventions regarding register usage for method call argument and return value passing. In order to prevent unnecessary copy instructions from being generated, designated registers can be allocated for those instructions that set method parameters or return values. Liveness information on local variables, obtained by data flow analysis, is also used to avoid generating unnecessary spill codes. Whenever a reference to a cached local variable is known to be the last use, the register is invalidated immediately after generation of the relevant code and is given back to the register manager to be added to the list of available registers.

Idioms in bytecode sequences. We provide a table of frequently appearing bytecode sequences as idioms to mitigate the inefficiency in code generation caused by stack semantics. The purpose of the id-

Figure 6 Fast type inclusion test for checkcast instruction (pseudocode)

```
Type to=(Type) from;

(A) Java source
```

```
if (from==NULL) then;
else if (last_succ==class_of (from)) then;
else if (call C function) then last_succ=class_of (from)
else throw classcast_exception;

(B) Resulting pseudocode
```

Table 1 Examples of bytecode idioms

Bytecode Idiom	Corresponding Java Source Statement
dup / getfield / iload / iop2 / putfield	Obj.field += var;
dup2 / iaload / iload / iop2 / iastore	Array[i] += var;
dupx1 / putfield	if(Obj.field = val) . . .
aload / iload / iinc / iaload	. . . = Array[i++] + . . .

iom table is to locally eliminate the stack semantics and to exploit the local variable cache registers by avoiding unnecessary move instructions between local and stack variables. Many of the idioms work around the stack manipulation bytecode instructions to reduce the stack height of expression evaluation, thus reducing the number of stack variable registers needed and allowing more local variables to be cached. The total number of idioms provided is more than 80, and some examples of these idioms and their corresponding Java source statements are shown in Table 1. A similar approach is discussed in Proebsting,[18] where frequently appearing bytecode patterns are synthesized as superoperators and where a heuristic method for inferring a good set of superoperators is proposed.

The bytecode sequences generated for a specific code fragment in Java sources, of course, depends on the Java source compiler. A different Java compiler may produce different bytecode sequences for the same source programs. Therefore, some of the idioms provided may not be applicable to some class files if their underlying Java compiler for the class files is different.

Type inclusion test. The type inclusion test is a procedure for checking whether two given types are related by a subclass relationship and will result from either an explicit or an implicit requirement by programmers. Explicit type checking can be specified by using the construct instanceof in the source program, and the type conformance test is implicitly imposed for a statement assigning an instance to a different type of local variable or array element. This run-time overhead can be a significant performance bottleneck in many Java programs.

In the past, several techniques have been proposed for conducting efficient type inclusion tests[19] by encoding class hierarchies in a small table, which is referred to by the inlined checking code. The speed and space efficiency varies depending on the encoding type of the subclass relations. However, new classes and interfaces can be loaded in Java at arbitrary points in the program execution, and the table has to be recomputed for the new encoding when classes are loaded dynamically. Although we did not explore the possibility of this scheme in terms of performance benefit and the recomputation cost in our environment, we chose to implement a simple and effective caching mechanism for type inclusion testing and inlined the fast path of the code as shown in the pseudocode for the checkcast instruction in Figure 6.

First, the given object is tested to determine whether it is null. Since a null object can be cast to any type

of object, no operation is required in this case. The object class is then compared with the value of the cache that holds the successful class from the previous test. If these tests fail, then the Jvm-supplied run-time library is called to traverse the class hierarchy to check subclass relations, and if it succeeds, the cache value is updated with the new object class. In the code generation, only the first and second tests are inlined in the fast execution path, and the remaining operations are placed at the bottom of the code. We also hold another cache holding failure class for the instanceof instruction, and the object class is compared with the value of the cache as well. This action is taken because the instanceof instruction returns either true or false depending on the result of the test, instead of just throwing the class-cast exception as in the case where the checkcast instruction fails.

Exception handling. Since using exceptions can be one of the normal programming practices in Java, the exception-handling mechanism must be efficient in the JIT implementation. For example, a program may contain an infinitely running loop that exits only through an array index out-of-bounds exception. We rely on the hardware trap and system-level exception-handling mechanism to detect and handle some Java exceptions, namely, the null pointer exception, arithmetic (division by zero) exception, and stack overflow exception, to achieve high performance by eliminating the code for explicitly checking the exception condition. Once an exception of this type has been raised, the system-level exception handler directly calls the user-level handler, which is registered through the exception registration record. The exception registration record is created in the execution frame only for those methods that have an exception table, namely, a try-catch block, and is registered and deregistered to the system during the course of program execution. Thus an exception chain is maintained for each thread, always indicating the list of exception records with the topmost one as the anchor. The user-level handler will then traverse along this exception chain to find an appropriate address to be resumed, instead of unwinding the entire stack frame linearly.

The try region at the time of the exception has to be identified by the exception handler in determining the corresponding catch block where the execution is possibly resumed. This is originally given as the range in the bytecode sequences; however, it is not appropriate for the JIT compilation, since the code generation does not take place in the order of given bytecode sequences, but takes advantage of the basic block ordering that results from various bytecode-level optimizations and the run-time trace information from the mixed mode interpreter. We therefore provide an extra local variable to keep the current try region identification: Whenever the execution enters or exits a basic block and the corresponding try region identification becomes different, the local variable is updated so that the exception handler can locate the corresponding catch blocks correctly to check whether the execution can be resumed.

For those exceptions that require code to be generated explicitly for checking exception conditions such as array index out-of-bounds, we generate a conditional instruction to jump to the bottom of the code when the exception condition is met, so that the execution falls through the jump instruction in normal cases.[20] The actual exception-throwing instructions are placed at the bottom of the code. This placement is to avoid an instruction cache miss caused by the loading of an exception-throwing code that is unlikely to be executed, and also to take advantage of the processors' static branch prediction of any forward conditional branches not to be taken.[21]

The exception-handling mechanism must be efficient in the JIT implementation.

Code scheduling. Code scheduling is a well-known technique for optimizing code by scheduling or reordering given instructions to best fit the requirements imposed by the underlying machine architectural characteristics. Since many machines nowadays, including Intel processors,[21] have multiple pipelines and expose instruction-level parallelism to the users, it is essential that the code for such machines be organized in a way that takes best advantage of pipelines present in an architecture or implementation. The code scheduling technique was developed for static compilation, and therefore the compilation time was not a major consideration. Even with the simple list scheduling technique for the basic block range, it is necessary to construct a dependence DAG.

The worst-case running time is $O(n^2)$, where n is the number of machine instructions in the basic block.[6] In the IBM JIT compiler, we have implemented a different way of scheduling code within a basic block, running in $C * n$, where C is the size of the lookahead to be scheduled, in view of the compilation time constraint.

The code scheduler works synchronously with the code generator within a basic block range. A table, whose columns show the number of pipelines and whose rows show the number of clocks, is provided to serve as a buffer for reordering instruction sequences. When a native code is generated, it is given to the scheduler together with some attributes for the generated code, such as reference registers, an updated register, an address and the type of memory access, restrictions on executable pipelines, and a flag indicating whether an exception can be raised with this instruction, all represented by bit vectors. The scheduler then considers all the requirements and dependencies between the new generated instruction and existing instructions in the buffer, including the number of clocks for address generation interlock (AGI), and places the instruction at a slot with the earliest possible clock number in the buffer. When there are no available slots in the buffer or the code scheduling scope has ended, the instructions in the buffer are emitted into the actual code space in the scheduled order.

The type of memory access given to the scheduler is categorized on the basis of the language specification to ensure that there is no interference between those memory access instructions with different types. Even though the instructions have the same type of memory access, the different offset means the accesses are for different memory locations, since generated instructions should not have the interior pointer of an object, unless common subexpression elimination has been applied. Our code scheduler takes advantage of the fact to reorder instructions.

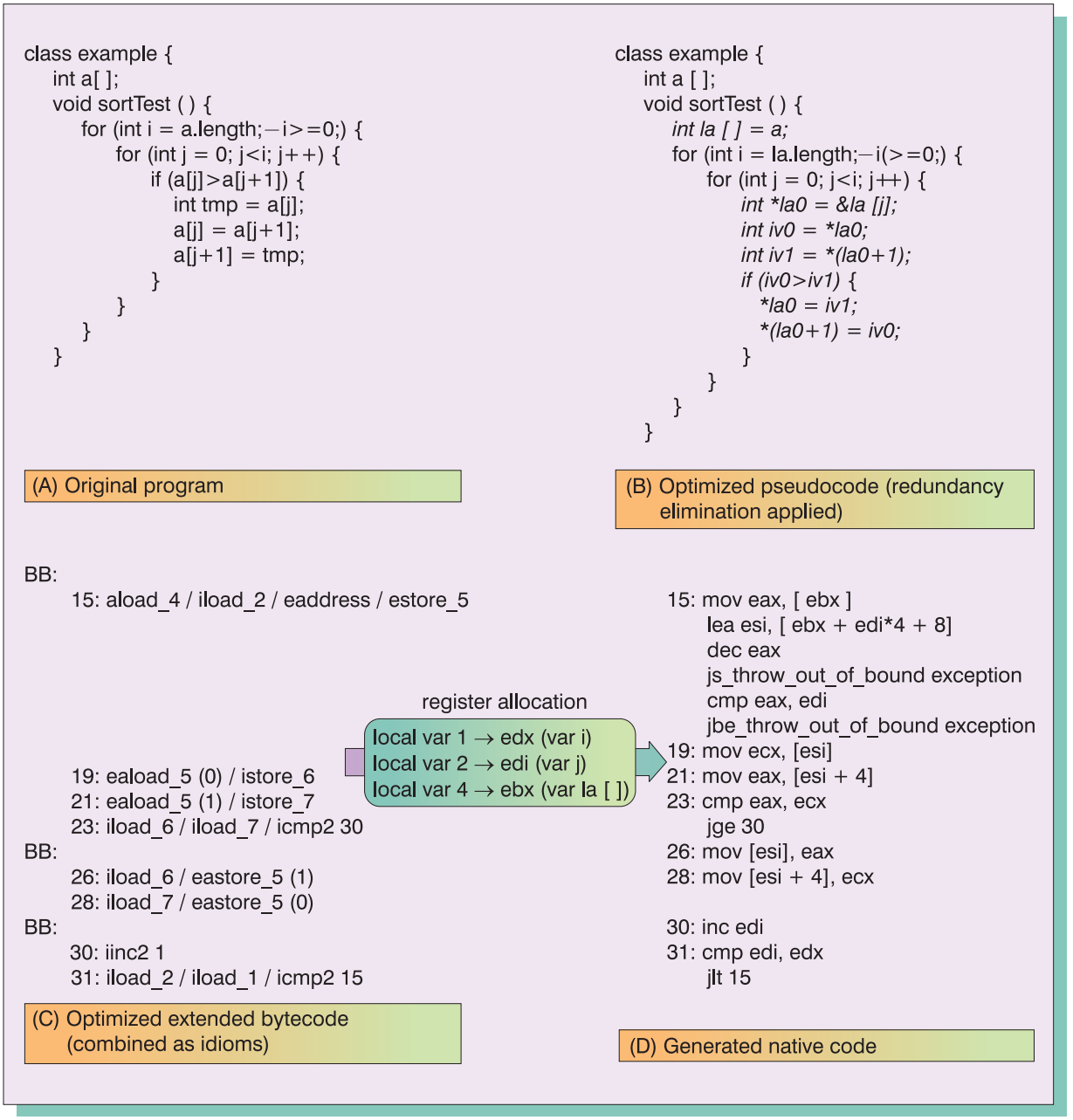
The scheduler cannot reorder any two instructions that may raise exceptions or produce side effects when executed, in order to guarantee that the correct exception can be thrown and the environment at the time of the exception is preserved. This is why the information indicating whether the given instruction can raise an exception or not is necessary for the scheduler. For array index out-of-bounds exception checking, we treat compare and jump instructions as a single complex instruction and include it in the scheduling scope, unlike the traditional basic block range scheduler, in order to make the scheduling scope long enough.

Since all the requirements and dependencies among instructions are solved by using bit vectors, an appropriate slot in which to place an instruction can be found quickly. The limitation of our approach is that there may be more appropriate instructions to be placed among the succeeding instructions that will be generated, since the first-fit strategy is used to place the given instruction in the buffer; that is to say, it is placed in the available slot with the earliest possible clock number. The architectural characteristics are largely different between the Pentium** processor and Pentium Pro processor family (Pentium Pro, Pentium II, and Pentium III processors), and the code scheduling is available only for Pentium processors in the current version of the JIT compiler. It will be enabled for the Pentium Pro processor family as well in the next version of the JIT compiler.

Example. Figure 7 shows an example from the bubble sort sample program. We use this example to illustrate common subexpression elimination and its resulting extended bytecode, native code generation using the bytecode idioms, and register allocation. Through bytecode-level optimization, common subexpression elimination is applied to the original program (Figure 7A), and it is transformed into the equivalent sequence of extended bytecode, for which the pseudocode can be shown in Figure 7B. The instance variable access for the array object is localized and moved out of the loop. An effective common address is then generated for the array accesses with consecutive indices. The italicized statements in Figure 7B correspond to the extended bytecode added or modified through this optimization.

The resulting extended bytecode for the innermost loop is shown in Figure 7C with some labels showing the basic block boundary. Each instruction is numbered with its index in all the sequences of the extended bytecode. New bytecode instructions, such as eaddress, eaload, and eastore, are added in our internal representation to express the operations applied by redundancy elimination. These instructions denote generating an effective address, loading a value from the specified effective address, and storing a value to the specified effective address, respectively. The number in parentheses in the figure for eaload and eastore indicates the offset from the effective address to be accessed.

Figure 7 Examples of code generation and optimizations



The extended bytecode sequences in Figure 7C are combined by idioms, based on which native code generation is performed. When the innermost loop is entered, the local variables i, j, and la[ ] are assigned permanent local registers according to the access counts for those variables. Numbers shown in generated native codes (Figure 7D) correspond to the numbers in the extended bytecode (Figure 7C). In the code generation for eaddress (numbered 15), array index out-of-bounds checking code is produced

for the range of indices accessed through the effective address generated; that is, two checks are generated for the lower and upper offsets of the interval against the lower and upper bounds of the array respectively. In this case, the interval [0, 1] is given as the index offset to be accessed from the effective address, and two checking instructions are generated without changing the index variable in the register edi.

Experimental results

To evaluate the effectiveness of the optimizations and techniques presented above, we performed two experiments of different kinds: one that focused on individual optimizations to check their benefits and contributions to the speedup, and another that measured the total performance to examine the overall competitiveness of the IBM JIT compiler relative to those of a major competitor.

Evaluation for individual optimization. We chose three industry-standard client benchmarking programs for the evaluation of individual optimization: CaffeineMark** 3.0,[22] JMark** 2.0,[23] and SPECjvm98.[24] For JMark 2.0, only the six computation-intensive tests were selected, since the other tests in the benchmark mainly focus on the performance related to AWT (Abstract Window Toolkit), which is not very JIT-sensitive. The measurement for SPECjvm98 was conducted in the test mode with a count of 100 in this set of experiments by running the test as a local application, not through the Web server as specified in the SPEC-compliant mode. Since these experiments are to measure the effectiveness of the JIT optimizations, the selective compilation switch is turned off; that is, all the methods are JIT-compiled (the switch is a development and tuning aid and not made public). The JIT compilation time is or is not included, depending on how each test of benchmarks is organized. For SPECjvm98, we took the best time for each test that resulted from autorun; the compilation times were not counted in this comparison. All the experiments described below were conducted on a Pentium II 375 MHz processor with 256 MB of RAM, running Windows NT** 4.0 Service Pack 3.

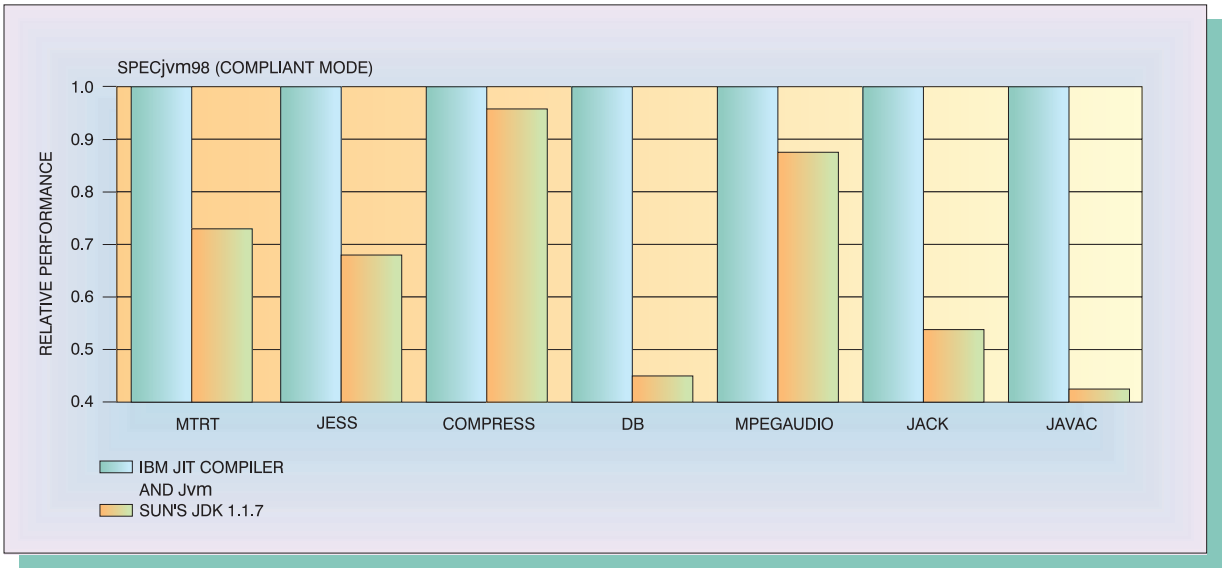
Figure 8 shows the relative performance when we turn off one of the optimizations described above to examine its effectiveness in each of the benchmark tests. The bar graphs for each test show how much performance is lost when we turn off each optimization—method inlining (NO INLINE), common subexpression elimination (NO CSE), loop versioning (NO LOOPVER), exception check elimination (NO EXC), fast type inclusion testing (NO TYPE), and idioms in bytecode sequences (NO IDIOM), respectively, from left to right—compared with the performance when we run the compiler fully optimized. As a reference, the performance with all the optimizations disabled is also shown as the right-most bar graph (NO OPT).

As we can easily see, method inlining is the most effective in the CaffeineMark 3.0 method, the JMark 2.0 processor, and several SPECjvm98 tests. Recursive method inlining is the biggest contributor to improving the method test score, whereas virtual method inlining is actually effective for improving the processor and SPECjvm98 mtrt test performance. Common subexpression elimination is effective when applied to the CaffeineMark 3.0 loop and float and to the JMark2.0 fast Fourier tests, all of which have array-manipulation-intensive loops. Loop versioning also makes a modest contribution in the same set of tests. These two optimizations have almost no effect on any tests in SPECjvm98, probably because in these tests there is no single hot spot where the optimizations applied for some loops contribute to the total performance. Exception check elimination seems to be effective only for the CaffeineMark 3.0 logic and string and for SPECjvm98 mtrt and mpegaudio tests in this limited set of benchmark tests. Fast type inclusion testing is quite effective when applied in several SPECjvm98 tests and contributes the improvement in performance, up to 20 percent for the jess and db tests. Idiom-based code generation is widely effective for many of the benchmark tests.

Total performance evaluation. Figure 9 shows the results of the overall performance measurements for SPECjvm98, comparing the IBM Developer Kit for Windows, Java Technology Edition, Version 1.1.7, which includes the JIT compiler we developed with Sun's reference implementation of JDK 1.1.7 (JDK 1.1.7B_003 for Windows),[25] which features the Symantec JIT compiler. This measurement was conducted in the SPEC-compliant mode on a machine with a Pentium II 350 MHz processor, 512 MB memory, running Windows NT Server 4.0 Service Pack 4 and the Apache Web server 1.3.4. The relative performance is computed from the best data (elapsed time) produced by the autorun, so the higher bar means the better performance.

Figure 8 Effectiveness of individual optimization

Figure 9 Total performance comparison



The IBM Jvm and JIT compiler outperform Sun's JDK 1.1.7 in all of the tests in this benchmark, and for some tests our score is more than double. An attempt to run the latest version of Microsoft Java execution environment Software Development Kit (SDK) 3.2 (build 3186, released August 24, 1999)[26] was also made on the same machine environment, but the valid result could not be obtained in the SPEC-compliant mode. In the comparison done under the test mode, the IBM Jvm and JIT compiler perform better than SDK 3.2, although the differences are smaller than those against Sun's JDK 1.1.7. Unfortunately, the comparison between different Java environments based on the measurement in test mode cannot be published by the term of our SPEC membership. Overall, the IBM Jvm and JIT compiler can be considered a top performer among major Java environments available for platforms based on Intel processors.

Related work

There have been several reports of optimizations and code generation techniques for fast and efficient compilation in product-level Java compilers. The Intel JIT compiler[20] exploits lazy code selection as a fast and effective way of folding Java stack operands by propagating information about operands via an auxiliary data structure called the mimic stack. Since it took a lightweight approach to having no internal representation, the optimization features are limited to extended basic blocks. Both the Microsoft JIT compiler[26] and CaCao[27] have their own internal representations for optimizations, but the details of their JIT optimizations are not clearly described. For static compilers, JOVE,[28] TowerJ,[29] and HPCJ[30] are now products and available, and they all exploit traditional optimization techniques such as static single assignment (SSA) representation, SSA-dependent optimizations, and global register allocation based on graph coloring. Marmot[31] is a complete system of a native compiler and run-time system, implementing standard scalar and object-oriented optimizations, such as call binding based on class hierarchy analysis. Ninja[15] is another static compiler, a research prototype that addresses optimizations for technical computing to make Java competitive with FORTRAN and C++.

Jalapeno[32] is a Jvm implemented in Java itself, designed to satisfy some critical requirements for servers. It takes a compile-only approach for program execution with three different dynamic compilers, instead of providing both interpreter and a JIT compiler as in most other Jvms. Some interesting optimizations, such as profile-directed method inlining and escape analysis, are being implemented in the most aggressive version of the compiler.

A fast and lightweight register allocation method was proposed in Poletto and Sarkar[33] and Traub et al.[34]; it consists of scanning all the live ranges of local variables and allocating registers to them in a greedy fashion. It is reported that the compile time speed is several times faster than the fastest graph coloring method, yet the quality of the generated code is comparable. Since the register allocation should be the key to improving JIT compiler performance further, it may be worth investigating a similar idea or an extension of this idea for the IBM environment.

The problem of unnecessary synchronization in Java is addressed in Bogda and Hölzle[35] by removing synchronizations for those objects reachable from only a single thread. The analysis first identifies objects that will be local to a thread by dereferencing the field of the thread local object, if necessary, and then transforms the program by introducing nonsynchronized versions of optimizable classes. This work is in contrast to our effort of alleviating the synchronization cost by speeding up the majority of synchronization cases. Although this work is done for a static environment where the whole program is available to be analyzed, the idea behind the work will be applicable in the IBM environment. The analysis can also be used for determining objects that can be allocated on the stack, instead of heap, which will reduce the cost of both object allocation and garbage collection.

Concluding remarks

In this paper, we have presented the design and implementation of the IBM Java JIT Compiler version 3.0 for platforms based on Intel processors. Following an overview of the overall structure and the base Jvm modification, we discussed each of the optimizations and the code generation techniques included in the JIT compiler in detail, and then presented a performance comparison using three industry-standard client benchmarking programs. It was shown that each of the optimizations is very effective for some type of program, and that overall the IBM JIT compiler combined with IBM's enhanced Jvm is one of the top performing Java execution environments on Intel-based platforms.

The discussion in this paper has been centered around an Intel-based platform; however, most of the optimizations and techniques are common to other IBM platforms as well, such as those based on the PowerPC* and S/390*, which the JIT compiler supports.[36] Specifically, the bytecode level optimization features described in the section on optimization and the basic idea of register allocation and idiom-based code generation in the section on code generation are all applicable as cross-platforms. Also, all the techniques presented here can be applicable regardless of the version of Java implementation; namely not only for version 1.1.7 of Java as made generally available, but also for the Java 2 implementation and beyond.

Since the JIT compiler is a critical component of the Jvm for achieving high performance, we will continue to improve the JIT compiler to stretch the limit of Java performance by balancing the compilation time requirement and the quality of code generation. In the next version of the JIT compiler, we will completely eliminate the stack semantics from the internal representation and will move into a register-based representation for further improvement in the quality of JIT-generated code. We also plan some object-oriented specific optimizations, including class hierarchy analysis and its use in method inlining and direct binding for virtual invocation sites.[36]

Acknowledgments

We would like to thank Duc Vianney and Akihiko Togami for their cooperation in measuring performance of several Jvms for SPECjvm98 by SPEC-compliant mode. We also thank the IBM Network Computing Software Division System Performance group in Austin for their helpful discussion and analysis of possible performance improvements.

Cited references and notes

1. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
3. A given call site may invoke several different actual methods over the course of program execution, depending on the dynamic type of receiver object.
4. The concrete example appears in the fourth section in the subsection labeled "Example."
5. D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin Locks: Featherweight Synchronization for Java," *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation* (1998), pp. 258–268.

6. M. P. Plezbert and R. K. Cytron, "Does 'Just in Time' = 'Better Late Than Never'?" *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), pp. 120–131.
7. D. Detlefs and O. Agesen, "Inlining of Virtual Methods," *Proceedings of the 13th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 1628, Springer-Verlag (1999), pp. 258–278.
8. R. Gupta, "Optimizing Array Bound Checks Using Flow Analysis," *ACM Letters on Programming Languages and Systems* **2**, No. 1–4, 135–150 (1993).
9. KILL is a term used in data flow analysis. If an instruction redefines a variable, it is said to kill the definition, which means the collected information regarding the variable cannot be preserved after the point in the flow analysis.
10. A. V. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., Reading, MA (1986).
11. Please note that *index_check* ($3 < ub$) in Figure 4B can be eliminated in Figure 4C as a result of the fact that checking ($0 <= i - 2$) and ($i + 1 < ub$) at the top results in ($3 <= i + 1 < ub$).
12. C. Click, "Global Code Motion, Global Value Numbering," *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation* (1995), pp. 246–257.
13. S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan-Kaufmann Publishers, San Francisco, CA (1997).
14. J. Knoop, R. Oliver, and S. Bernhard, "Lazy Code Motion," *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation* (1992), pp. 224–234.
15. Ninja: Numerically Intensive Java, IBM Corporation, available at http://www.research.ibm.com/ninja/.
16. *Intel Architecture Software Developer's Manual*, Order Number 243191-001, Intel Corporation, Santa Clara, CA (1997).
17. F. Chow and J. Henessey, "The Priority-Based Coloring Approach to Register Allocation," *ACM Transactions on Programming Languages and Systems* **12**, No. 4, 501–536 (1990).
18. T. A. Proebsting, "Optimizing an ANSI C Interpreter with Superoperators," *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), pp. 322–332.
19. J. Vitek, R. Horspool, and A. Krall, "Efficient Type Inclusion Test," *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '97* (1997), pp. 142–157.
20 A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth, "Fast, Effective Code Generation in a Just-in-Time Java Compiler," *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (1998), pp. 280–290.
21. *Intel Architecture Optimization Manual*, Order Number 242816-003, Intel Corporation, Santa Clara, CA (1997).
22. CaffeineMark3 Benchmarks, Pendragon Software Corporation, Libertyville, IL, available at http://www.pendragon-software.com/pendragon/cm3/info.html.
23. JMark2.0 Benchmarks, Ziff-Davis, Inc., New York, available at http://www.zdnet.com/zdbop/jmark/jmark20/applet/jmdocs/jmarkdoc.htm.
24. SPECjvm98 Benchmarks, Standard Performance Evaluation Corporation (SPEC), Manassas, VA, available at http://www.spec.org/osg/jvm98.
25. JDK1.1.7B for Win32, Sun Microsystems, Inc., Palo Alto, CA, binary available at http://java.sun.com/products/jdk/1.1/index.html.
26. MS SDK for Java 3.2, Microsoft Corporation, Redmond, WA, binary available at http://microsoft.com/java/vm/dl_vm32.htm.
27. A. Krall and R. Grafl, "CACAO: A 64-bit Java VM Just-in-Time Compiler," *Proceedings of the ACM PPoPP '97 Workshop on Java for Science and Engineering Computation* (1997).
28. *JOVE Technical Report*, Instantiations Inc., Tualatin, OR, available at http://www.instantiations.com/jove/jovereport.htm.
29. TowerJ, Tower Technology Corporation, Austin, TX, available at http://www.towerj.com.
30. High Performance Compiler for Java, now integrated in VisualAge for Java, IBM Corporation, available at http://www.software.ibm.com/ad/vajava/.
31. Marmot: an Optimizing Compiler for Java, Microsoft Corporation, Redmond, WA, available at http://www.research.microsoft.com/apl.
32. M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeno Dynamic Optimizing Compiler for Java," *Proceedings of the ACM SIGPLAN Java Grande Conference* (1999), pp. 129–141.
33. M. Poletto and V. Sarkar, "Linear Scan Register Allocation," *ACM Transactions on Programming Languages and Systems* (1999).
34. O. Traub, G. Holloway, and M. D. Smith, "Quality and Speed in Linear-Scan Register Allocation," *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (1998), pp. 142–151.
35. J. Bogda and U. Hölzle, *Removing Unnecessary Synchronization in Java*, Technical Report TRCS99-10, Department of Computer Science, University of California, Santa Barbara, CA (1999).
36. K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani, "Design, Implementation, and Evaluation of Optimizations in a Just-in-Time Compiler," *Proceedings of the ACM SIGPLAN Java Grande Conference* (1999), pp. 119–128.

Accepted for publication September 15, 1999.

Toshio Suganuma *IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan (electronic mail: suganuma@jp.ibm.com).* Mr. Suganuma joined IBM in 1992 as a research member at the Tokyo Research Laboratory, and since then he has worked on compiler optimizations and code generation for the High Performance FORTRAN (HPF) compiler and IBM Java Just-in-Time Compiler projects. His research interests are in the area of code optimization, parallel processing, and instruction scheduling. He received the B.E. and M.E. degrees, both in applied mathematics and physics from Kyoto University in 1980 and 1982, respectively, and received the M.S. degree in computer science from Harvard University in 1992. He is currently a research staff member in the Network Computing Platform group.

Takeshi Ogasawara *IBM Research Division, Tokyo Research Laboratory, 1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502 Japan (electronic mail: takeshi@jp.ibm.com).* Mr. Ogasawara works in the areas of optimizing compilers. He designed and implemented the type inclusion test optimization, the method invocation optimization, and the exception-handling mechanism for the IBM Java Just-in-Time Compiler in the IA32 architec-

ture. He also contributed to the fixed-size buffer management of the just-in-time compiler for Java-based thin clients. He joined IBM in 1991 at the Tokyo Research Laboratory after receiving the B.S. and M.S. degrees in computer science from the University of Tokyo. His research interests include code optimization and memory management.

Mikio Takeuchi *IBM Research Division, Tokyo Research Laboratory, 1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502 Japan (electronic mail: mtake@jp.ibm.com).* Mr. Takeuchi joined IBM as a researcher at the Tokyo Research Laboratory in 1993. He has been a member of the IBM Java Just-in-Time Compiler project since 1996, where he has been working on the platform-independent fast register manager and the code generator for platforms based on Intel processors. For this work, he received a Division Award in 1998. His current research interests are the implementation of dynamic languages (especially optimizing compilers and programming environments) and object-oriented technology (especially languages, tools, frameworks, and design patterns). He received the B.E. and M.E. degrees in mathematical engineering and information physics from the University of Tokyo in 1990 and 1993, respectively.

Toshiaki Yasue *IBM Research Division, Tokyo Research Laboratory, 1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502 Japan (electronic mail: yasue@jp.ibm.com).* Mr. Yasue received the B.S. and M.S. degrees from the School of Science and Engineering, Waseda University, in Tokyo in 1989 and 1991, respectively. He joined IBM in 1995 and is currently a research member in the Network Computing Platform group at the Tokyo Research Laboratory. His primary research interests include compiler optimization and parallel processing.

Motohiro Kawahito *IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (electronic mail: JL25131@jp.ibm.com).* Mr. Kawahito received the B.S. degree from Waseda University in 1991. He joined IBM in 1991 as a member of the AIX Systems group at the IBM Yamato Laboratory. He developed the PC simulator, 5080 emulator, WABI, CMDS, and CATIA Viewer. He has been a research member at the Tokyo Research Laboratory since 1997, where he has been working on the exception check elimination, constant propagation, dead store elimination, and class variable privatization for the Java JIT compiler project.

Kazuaki Ishizaki *IBM Research Division, Tokyo Research Laboratory, 1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan (electronic mail: ishizaki@trl.ibm.co.jp).* Mr. Ishizaki received the B.S. and M.S. degrees, both in computer science from Waseda University in 1990 and 1992, respectively. Since joining IBM in 1992 at the Tokyo Research Laboratory, he has worked on the High Performance FORTRAN (HPF) compiler. He is currently involved with the IBM Java Just-In-Time Compiler. His research interests include optimizing compiler and processor architectures.

Hideaki Komatsu *IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (electronic mail: komatsu@jp.ibm.com).* Dr. Komatsu received the B.S. and M.S. degrees in electrical engineering from Waseda University in 1983 and 1985 and the Ph.D. in computer science from Waseda University in 1998. Since joining IBM in 1983 at the Tokyo Research Laboratory, he has carried out research activity in the areas of the optimizing compiler, Prolog compiler, data flow compiler, FORTRAN 90 compiler, High Performance FORTRAN compiler, and the IBM Java Just-in-Time Compiler. His research interests include compiler optimization techniques (register allocation, code scheduling, and loop optimizations) for instruction-level parallel architecture and loop optimizations for massively parallel computers. Dr. Komatsu is currently a research staff member in the Network Computing Platform group.

Toshio Nakatani *IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan (electronic mail: nakatani@jp.ibm.com).* Dr. Nakatani received the B.S. degree in mathematics from Waseda University in 1975, and the M.S.E., M.A., and Ph.D. degrees in computer science from Princeton University, in 1985, 1985, and 1987, respectively. He joined the Tokyo Research Laboratory as a research staff member in 1987 and is currently manager of the Network Computing Platform group. His research interests include architecture, compilers, and algorithms for parallel computer systems.

Reprint Order No. G321-5722.