PAPER

# Cost-Effective Compilation Techniques for Java Just-in-Time Compilers

**Kazuyuki SHUDO**[†], **Satoshi SEKIGUCHI**[†], ***Nonmembers*, *and*** **Yoichi MURAOKA**[††], ***Fellow***

**SUMMARY** Java Just-in-Time compilers have to satisfy a number of requirements in conflict with each other. Effective execution of a generated code is not the only requirement, but compilation time, memory consumption and compliance with the Java Virtual Machine specification are also important. We have developed a Java Just-in-Time compiler keeping implementation labor little. Another important objective is developing an adequate base of following researches which utilize this compiler. The proposed compilation techniques take low compilation cost and low development cost. This paper also describes optimization methods implemented in the compiler, for instance, instruction folding, exception handling with signals and code patching.
***key words:*** *Runtime compilation, Java Virtual Machine, Stack caching, Instruction folding, Code patching*

## 1. Introduction

Just-in-Time (JIT) compilers for Java bytecode have to satisfy a number of requirements, which are different from those for ordinary compilers. Effective execution of a generated code is not the only requirement, but the time and memory consumed by compilation should worth performance gain because the compilation takes place while the target program is running. Java bytecode JIT compilers also suffer relatively strict specifications of Java language and Java Virtual Machine (JVM). The rules in the specifications yield high-reproducibility of execution results of Java programs on different platforms. But part of the rules limit a class of optimizations and performance improvement by the compilers.

Because of conflicting requirements for Java runtime, a number of different runtimes have naturally appeared and even an individual runtime takes different options on its behavior according to characteristics of user programs. For instance, Sun Microsystems' HotSpot Server VM has a JIT compiler specialized to computation-intensive application. The compiler spends much time on compilation of code segments which have been recognized as "hot spot", a code segment expected to run many times. Oppositely, a runtime for embedded application tends to save power consumption rather than performance improvement.

We developed a JIT compiler along the following policies.

1. Ease of use as a base of researches.
2. Cost-effective development. Less labor and relatively much effect.
3. Adequate quality and performance for practical use.

Compiler development involves much work on a parser, intermediate representations and a number of optimizations. Because of it, we have to consider those human and engineering factor seriously in addition to technical requirements like performance. Our plan on the development of the JIT compiler was to have a practical compiler with work several man-month. Our another goal was specifically having a research base on which we do following researches with less labor while developing it with less work.

It is known that development efforts need vast work when extreme high performance is set as one of the goals. We do not head such a goal and set our line to baseline compiler, which saves compilation time and memory.

In this paper, we present cost-effective code generation and optimization methods we have implemented in the JIT compiler and their effects. The code generation technique is template connecting. The code generator basically connects given templates of native code corresponding to internal instructions. In addition to the technique, stack caching [1] was implemented in the compiler and the technique makes use of multiple registers over templates. There have been a JIT compiler which caches only the top of stack on a register and a bytecode interpreter of Sun Microsystems' Classic VM which does dynamic stack caching. But there has been no JIT compiler stack caching is applied to and the JIT compiler we have developed is the first case. This technique, template connecting with stack caching achieved utilization of multiple registers with less compilation cost. Furthermore the compiler became easy in use as a base of researches because the template connecting technique allows us to modify native code generated by the compiler directly as mentioned in **2**.

In the next section, we describe an overview of the JIT compiler shuJIT and present the structure of the compiler and the code generation method. And we discuss how they affect easiness in use as a research base and the compilation cost. In **3**, pros and cons of the

[†]National Institute of Advanced Industrial Science and Technology, Tsukuba Central 2, 1-1-1 Umezono, Tsukuba-shi, 305-8568 Japan

[††]Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan

System and Computers in Japan, John Wiley & Sons, Inc., Volume 35, Issue 12, pp.10-24, Nov. 2004

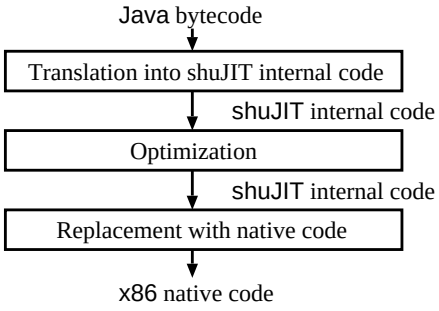



**Fig. 1** Structure of shuJIT.

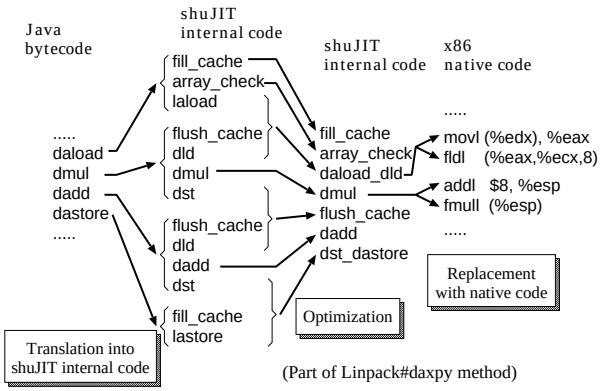



**Fig. 2** An example of compilation.

stack caching technique in this compiler are discussed. After effects of optimizations implemented in the compiler are shown in **4**, two important factors which affect usability, peak performance and invocation time are evaluated in **5**. We conclude with **6**.

## 2. Overview of the JIT Compiler

We have been developing and distributing shuJIT, a Java bytecode Just-in-Time (JIT) compiler. The compiler supports Intel's IA-32 processors, known as x86, and Linux, FreeBSD and NetBSD OSes. Except templates of generated native code, which are written in assembly code for x86, the compiler is in C language. The compiler works with a Java Virtual Machine (JVM) Classic VM, which are distributed with Sun Microsystems' Java 2 Platform, Standard Edition (Java 2 SE) and Java Development Kit. ShuJIT is expected to work on PC or more rich environments as declared with supporting architecture and OSes.

Practicality, stability and reliability for daily use were also our goals while ease of use as a research base is one of the goals. Compliance with the JVM specification [2] is, of course, one of the important goals. If a compiler does not achieve one of these goals, derived researches from the compiler ought to lack reality. We could have a certain number of users of shuJIT as the compiler achieved those goals. There were over 7500 downloads of the source code and about 8500 downloads of the binary for 2 and a half years since its first release in September of 2001.

Fig. 1 shows an overview of the compiler. First, the compiler translate Java bytecode instructions in a given method to shuJIT internal instructions. The compiler then applies optimization techniques to the internal instructions. The techniques described in **4** are instruction folding (**4.2**), inlining (**4.5** and **4.6**) and direct invocation (**4.1**). Finally the compiler translates the internal code to x86 native code and resolve function calls. Fig. 2 is an example of compilation by shuJIT.

The intermediate representation, shuJIT internal code is extended Java bytecode and has peculiar instructions. Translation process from Java bytecode to the internal code is just one-to-one or one-to-many replacement.

Native code generation in the last stage of compilation is achieved by replacement of the internal instructions with templates, which are pre-compiled pieces of native code. The compiler has the templates because compiler developers provided them. The prepared templates were written as simulating JVM stack with hardware-supported stack of the processor. Java bytecode instructions push a value on a JVM stack are basically translated to processor's `push` instructions.

The aims to adopt such a template connecting technique are as follows.

- Saving of development cost.
- Easy modification of generated code.
- Control of compilation cost.

The technique eliminates the need of assembler in the compiler and we could save the labor on its development. Assembler is not necessary because the prepared templates which have native code can be assembled while the compiler itself is compiled by a C compiler. Development cost of an assembler for x86 is relatively high compared with one for RISC processors because bit patterns of x86 machine instructions are not very regular.

48 days after the start of its development, the compiler started working and could compiled simple Java programs. It is difficult to compare the development cost with other software, but the cost seems to be very low as cost of JIT compiler development.

Generated native code can be directly modified by making changes on the templates because the templates appear in the generated code directly. It is a natural conclusion that the compiler is easy to be applied to researches which need modification of generated native code. The compiler has been utilized as a base of such researches [3]–[5] because of the property. Because usual compilers use a more fine-grain internal representation like GCC's RTL just before code generation, it is not possible to modify generated code directly and

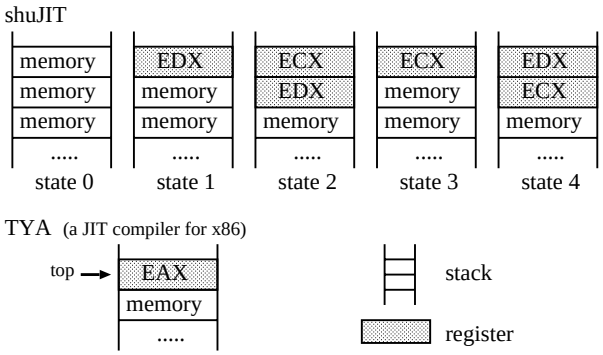



**Fig. 3** Cache states.

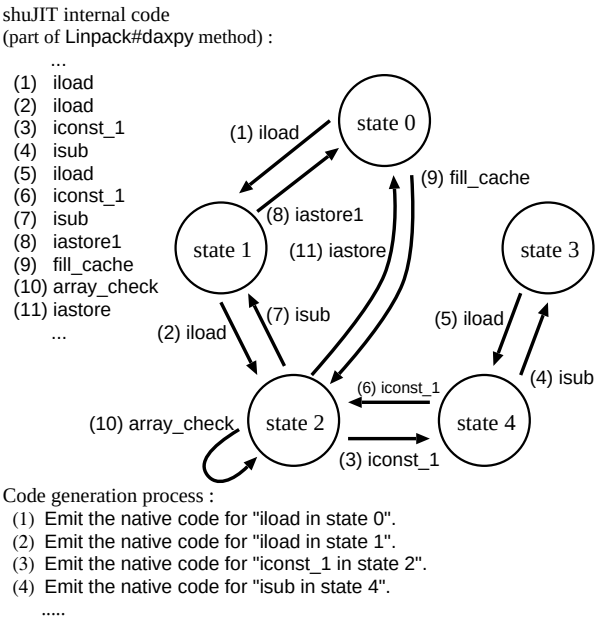



**Fig. 4** An example of state transition.

such a modification can be carried out in an indirect way. And further, such a modification often needs care to code elimination, order modification of machine instructions and so on. Such care includes suspension of optimizations and making temporary dependence between instructions.

On the other hand shuJIT's template connecting such a drawback as templates are dependent on one processor architecture like x86. Even if you make a change on templates for x86 processors, the change does not affect templates for SPARC processors. ShuJIT supports only x86 processors currently because one of the goals of the development is ease of development. Templates have to be re-implemented for other processor architectures to support them.

We expected a lower compilation cost of the template connecting technique. Code generation takes linear time proportional to the length of internal code sequence. Not only code generation, but calculation and space cost of entire compilation process are also limited to $O(n)$ not to spoil the compiler's merit as a baseline compiler.

## 3. Stack Caching

One significant problem of the template connecting technique is difficulty in register allocation. It is difficult to utilize multiple general-purpose registers over templates with the technique even though it is a key to high-performance to make good use of the registers. A Java bytecode JIT compiler TYA [6] also uses the template connecting technique and it is an instance of such a compiler which use only one general-purpose register over templates. Native code TYA generates caches a value on the top of the JVM stack on a register.

In order to improve register utilization, we implemented stack caching [1] technique in shuJIT. JVM stacks are basically placed on main memory but values around the top of the stacks are cached on registers. We defined 5 cache states corresponding to how stacks are cached as shown in Fig 3. In this case 2 registers can be exploited to cache stacks. 5 templates corresponding to 5 cache states are needed for 1 shuJIT internal instruction. When generating native code, next template is chosen according to the state in which the previous template ends its execution. An example of template connecting process is shown in Fig. 4.

Dynamic stack caching has been often implemented in a interpreter and there has been an implemented instance in JVM, The interpreter in Sun Microsystems' Classic VM. The implementation defined 3 cache states and dynamically choose native code implementing next bytecode instruction. But there has been no JIT compiler the stack caching is applied to and shuJIT is the first case.

Lazy code selection [7] is a code generation technique related to a technique of shuJIT. Both those techniques replace quasi-bytecode with native code directly and try keeping compilation process light. In Lazy code selection, generated native code can use 3 scratch registers to cache stack values. And a code generator provides and uses mimic stack to track where stack values are cached on during code generation. This tracking technique is corresponding to the technique of shuJIT in which a cache state is tracked. A code generator along the lazy code selection folds a load instruction and an arithmetic instruction into one machine instruction exploiting rich addressing modes of x86 ISA. In case of shuJIT, such folding is achieved by hand writing of templates and instruction folding during optimization. Points of difference include the number of registers for caching and register allocation to local variables by lazy code selection. Consequently, quality of generated code by lazy code selection is expected to be better than shuJIT and code generation cost of shuJIT will be

lower than lazy code selection because of simpleness.

It is difficult to measure effect in performance of stack caching added to the simple template connecting. It is due to lack of JIT compiler which implements both of simple template connecting and stack caching. But all scores of shuJIT are above those of TYA (see 5.1). TYA also implements a template connecting technique but no stack caching. Native code generated by TYA caches only the top of stack with EAX register (Fig reffig:states).

Stack caching can improve performance of generated native code. On the other hand, development work of a JIT compiler increases because a compiler developer has to provide some templates corresponding to each cache state. In case of shuJIT, 5 states were defined and 5 templates are needed. But the amount of work to provide 5 templates is not always 5-fold because 5 templates are almost same except register names in many cases. In compiled form 5 templates take 5 times as large as a template and the JIT compiler takes more space compared with a compiler without stack caching. But the the assembled template takes only about 30 kilobyte of disk and the amount of the space does not cause problems on PC.

Stack caching enabled the compiler to utilize multiple general-purpose registers. 2 as the number of registers which the compiler uses is not large number but the number of free registers on x86 is limited. X86 ISA originally provides only 8 general-purpose registers and 8 registers include a stack pointer and a base pointer. ShuJIT uses remaining 4 registers someway. One register is used to cache a base address of JVM local variables and other 3 registers work in a template.

## 4. Optimization Techniques

ShuJIT implements not only well-known code optimization techniques such as instruction folding, tail recursion elimination and inlining but cost-effective techniques. They include direct invocation between compiled methods, exception handling with OS signal and code patching. All implemented optimizations take $O(n)$ time and space cost at most assuming that $n$ is the length of bytecode.

Even a well-known technique possibly shows different effects depending on the code generation method of shuJIT and the property of JVM. For example, OS signal can be exploited to obey the JVM specification without a performance penalty. Instruction folding is further effective for the template connecting code generation because a JVM is a stack machine. On the other hand the ability of inlining is not fully exploited for lack of inter-method optimization in the compiler.

In this section, we report optimization techniques implemented in shuJIT, their effects in performance and memory consumption, implications in the code generation method and the JVM specification.

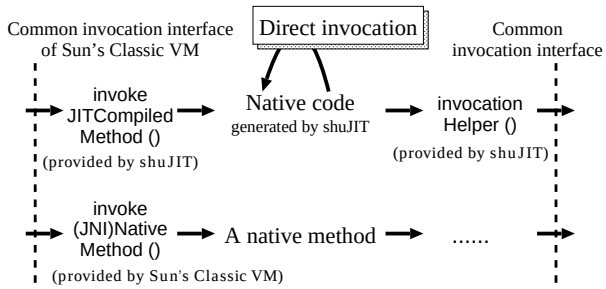



**Fig. 6** Direct invocation between compiled methods.

Fig. 5 shows the effects on the scores of SPEC JVM98 benchmarks on 1.7 GHz Pentium 4. The scores in case all optimizations are applied are represented by 100 %. Therefore a lower number means more decline of the score and the optimization technique was effective.

### 4.1 Direct Invocation between Compiled Methods

In a JVM, various kinds of methods co-exist simultaneously, such as methods left as bytecode and executed by an interpreter, native methods written in C or C++ language, methods compiled by a JIT compiler. There needs agreed calling convention to call each other between different kinds of methods. One of popular ways to achieve this purpose is having the single common calling convention (Fig. 6). Classic VM, a JVM which shuJIT works on, defines the common interface as follows.

```
bool_t <method name>
(JHandle *o, struct methodblock *mb,
 int args_size, ExecEnv *ee)
```

The `invokeJITCompiledMethod()` in Fig.6 is a wrapping function shuJIT provides to adjust difference between the common interface and calling convention of the compiled native method. There are differences in stack growing direction and place of stack in memory. Because of it the wrapping function re-stack values on the stack for the caller onto the stack for the callee.

But, in case that a compiled method calls another compiled method such a call to the wrapping function is not necessary because the common interface does not has to be involved and re-stacking is not needed. The JIT compiler then omit a call to the wrapping function if it is allowed. A compiled method can call directly another compiled method without an overhead of a call to the wrapping function if the call is omitted.

Even in case of the call omitted, dynamic identification of the callee is still needed. When a compiled method calls another method, the caller has to identify in runtime whether the callee is a compiled method or not because the callee cannot always be identified when the caller is compiled. For example even if the callee

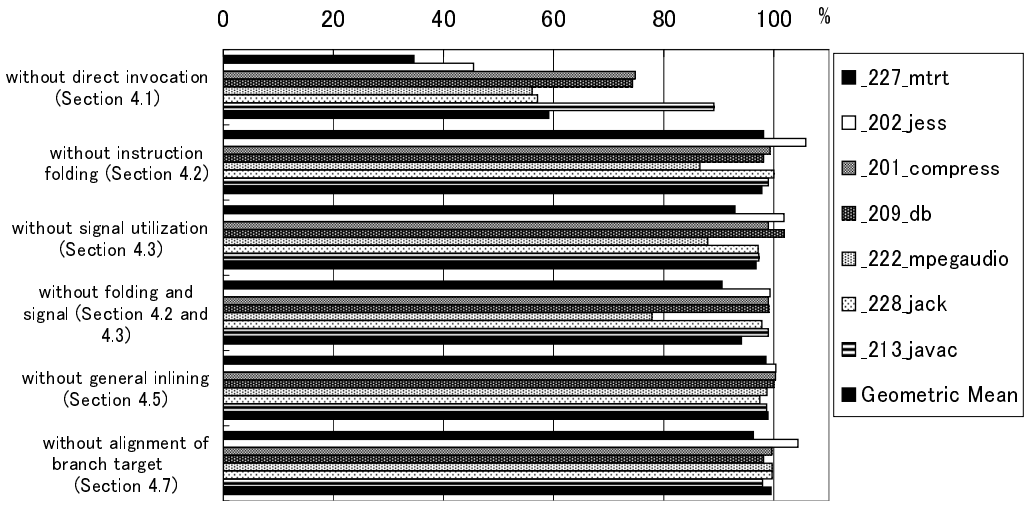



**Fig. 5** The impacts of each optimization on SPEC JVM98 scores.

is a compiled method when the caller is compiled, the callee is possible to be overridden by another kind of method like a native method. The compiler generates such a code which perform the identification.

The identification naturally needs a conditional branch and the branch involves a runtime overhead. We then implemented another optimization which eliminates the identification. The identification can be eliminated if the callee is a static, private or final method and the compiler do it.

We have to notice that JIT compilation of a method is not always completed. Even when a method cannot be compiled for some reason, the method is still executed by an interpreter. It is better than termination of program execution. The compiler has to confirm it can compile the callee successfully before it eliminates the identification in the caller. To confirm it, the compiler has to actually compile the callee because compilation failure occurs for unexpected reasons.

The compiler then eagerly compiles the callee if it is a static private or final method and the callee has not yet been compiled. The compiler determines whether it can do the optimization according to the result of the compilation.

One drawback of this optimization is possibly wasteful compilation. If callees compiled for the confirmation have never been called at last, the time and the space consumed by the compilation are wasted. Because this optimization have pros and cons as explained here, it can be disabled when the compiler is compiled by a C compiler.

Another possible way to avoid such wasteful compilation is code patching technique on the caller. In the case the compiler compiles a callee lazily when it is just called and modifies the call in the caller in runtime. But this way is complicated and another drawback. There always needs a code for the identification in a callee and it cannot be omitted. There is a trade-off as the technique will needs more space in a caller and can save eagerly wasteful compilation of callees.

As a result of the optimization described here, the score of Method benchmark in CaffeineMark 3.0 was improved from 1780 to 7652. The ratio of the improvement is about 4.3. The impact of the optimization on the scores of SPEC JVM98 is much as shown in Fig. 5. The synthesized score of SPEC JVM98 rose to 1.7 times as the original score.

The effect on _213_javac benchmark is the least and the score of it became 1.1 times as the original score. As the best case the score of _227_mtrt benchmark rose to 2.9 times. These results also show _227_mtrt is very sensitive to efficiency of method call.

### 4.2 Instruction Folding

Instruction folding is a well-known optimization technique by which multiple instructions are replaced to a less number of instructions which equal the original sequence of instructions. It is known that the technique is effective in a stack machine in particular. Application of the technique to Java processors has been often considered and carried out [8]. The picoJava [9] is an instance of such implementations. The technique is expected to have much effects similar to Java processors in shuJIT because the compiler yields many stack operations due to its code generation method (see **2**).

For example, a Java compiler like `javac` emits a `pop` instruction and a following `push` instruction to copy the top of stack to a local variable. The bytecode sequence is `istore` and `iload` if the value is 32 bit integer. ShuJIT's code generator naively translates this sequence to machine instructions as a `pop` and a `push` if any optimization is not performed. It is not desirable that the resulted sequence involves 2 memory

operations. Instruction folding technique replaces this sequence with a single copy instruction.

ShuJIT performs instruction folding on its internal instructions. The main purpose of the optimization is to reduce memory accesses. Value copies between memory and register are reduced by the optimization. The following operations are targets of the optimization to be replaced with lighter operations. The optimizer finds these sequence of internal instructions and replaces the sequence with another internal instructions which do not perform the copies in parentheses.

1. Stack → local variable (→ stack)
2. Floating-point register (→ stack) → floating-point register
3. local variable or array (→ stack) → floating-point register
4. floating-point register (→ stack) → floating-point register

For instance, a sequence of `istore` and `iload` is replaced with an alternative internal instruction `istld`, which copy a value only once. This folding is an example of 1 on the above list.

Local variable in the above list means a construct of Java language and JVM and shuJIT places them on memory. Similarly, stack is a construct of Java language and JVM and basically placed on memory by the compiler. But values around the top of stack are cached on integer registers as described in **3**.

In this manner, the code generator of shuJIT exploits integer registers but effective use of floating-point registers are not taken into account. ShuJIT compensate it by instruction folding and reduce wasteful copies of floating-point values. Such a purpose and effects are peculiar to shuJIT even though instruction folding itself is a well-known technique.

In reality, the effect on the _222_mpegaudio benchmark which has many floating-point operations is highest among SPEC JVM98 benchmarks. The technique improved the score to 1.16 times as the original score as shown in Fig. 5. We also examined the effect on Linpack benchmark. Almost all its operations are floating-point arithmetic. The problem size is $500 \times 500$ in this case.

| | |
|---|---|
| Without instruction folding | 14.042 Mflops |
| With instruction folding | 19.083 Mflops |

The score was much improved by the technique to 1.36 times as the original.

If the compiler can cache floating-point values with the stack caching technique (**3**), cases from 2 to 4 in the above list will be treated by the technique and not needed to be folded. But it will take much cost to increase the number of cache states And the compiler will have to track and identify the type of a value on stack to choose an appropriate template for a bytecode instruction which operates on both of integer and floating-point, such as `pop`, `pop2`, `dup` and `dup2`.

### 4.3 Exception Detection with Signal

Exception detection in Java runtime is often achieved utilizing a signal mechanism of UNIX-like OSes. A runtime can detect several kinds of exception without any conditional branch and it does not suffer any overhead if an exception is not thrown.

ShuJIT utilizes the signal mechanism as follows.

- Detects `NullPointerException` with `SIGSEGV`.
- Detects `ArithmeticException` with `SIGFPE`.
- Detects `StackOverflowError` with `SIGSEGV`.

Detection of the first 2 kinds of the exceptions with signal is common to Java runtimes. In addition to them, shuJIT also uses signal to detect `StackOverflowError` which means a stack overflow.

Because the `null` in Java language and JVM is represented with address 0, an access to it violates memory protection and causes `SIGSEGV`. The runtime can detect `NullPointerException` by catching the signal `SIGSEGV`. Besides, `ArithmeticException` can be detected with `SIGFPE` thrown in case of zero division.

To detect a stack overflow, A method compiled by the compiler accesses a further address in a stack growing direction. Memory protection mechanism of the processor causes `SIGSEGV` if the stack will soon run up. The actual address to be accessed in this time depends on the platform, as the kind of libc and OS. A margin of stack space needed to execute a signal handler and handle the thrown exception is dependent on the platform.

Exception detection with signal does not suffer any performance penalty in case an exception is not thrown. But a cost to handle an exception increases because the process for a preparation and a catch of signal is heavier than detection with a conditional branch. For that reason, the performance of a program involving many exception handling can fall down by such signal utilization.

Effects of the signal utilization depend on behavior of a program, especially frequency of exceptions. We can expect a good program does not throw exceptions very often because it is recognized as a good practice to use an exception only in exceptional case. In reality the scores of the SPEC JVM98 benchmark set improve with signal utilization as shown in Fig. 5.

How much frequency of exceptions does the technique tolerate? To estimate the threshold, we measured and compared the amount of positive effect on the normal case with no exception thrown and the amount of negative effect on the opposite case. The benchmark program we provided reads an instance variable via an object reference. The program throws a `NullPointerException` if the reference is `null` (exceptional case) and does not throw if the reference is not `null` (normal case). As to both cases, we measured

```
void signal_handler(int sig, ...) {
  struct sigcontext *sc = ...
        <obtain the signal context>;

  switch (sig) {
    <transact the received signal>
    (i.e. throwing an exception);
  }

  <modify the program counter
   in the signal context obtained above>;

  return;
}
```

**Fig. 7** Signal handling for catching an exception.

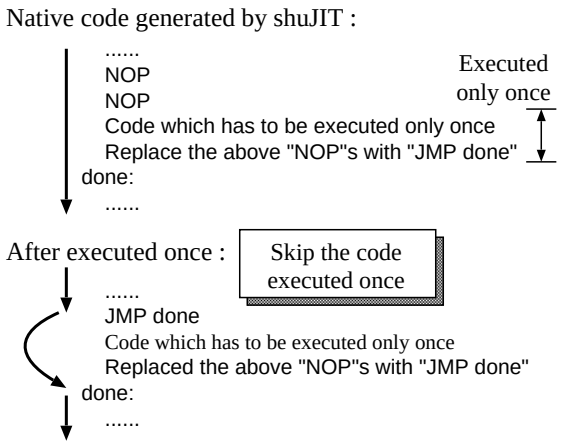



**Fig. 8** Overwriting a jump instruction onto a NOP instruction.

how much effect the signal utilizing technique made on performance.

On 1.7 GHz Pentium 4, execution time in the normal case reduced by 136 milliseconds per a billion times reads. Oppositely, the time in the exceptional case increased by 12302 milliseconds per a million times reads. On 600 MHz Pentium III, the time in the normal case reduced by 334 milliseconds per 10 million times reads and the time in the exceptional case increased by 1521 milliseconds per 100 thousand times reads. The following numbers are the ratios of the increased time in the exceptional case to the reduced time in the normal case.

| | |
|---|---|
| 1.7 GHz Pentium 4 | 90500 times |
| 600 MHz Pentium III | 4600 times |

Therefore we can estimate that the signal utilizing technique is worth being applied in case the frequency of exceptions thrown is lower than once a 90500 executions of such an instruction which possibly throw the exception.

Fig. 7 shows the process of the signal handler for catching an exception. The handler first obtains the context of the thread (`sigcontext` type) in which the exception is thrown. Next, it identifies the kind of the thrown exception according to the kind of the signal and the context. It then throws the exception. It finishes after it updates the program counter in the context whereby following execution starts at the appropriate catch clause or a return instruction.

### 4.4 Code Patching

The Java language specification [10] strictly prescribes the timings at which a class is initialized, as its static block is executed and static variables are initialized. The rule is cumbersome for a JIT compiler as a mere access to a static variable and a call to a static method can bring about a class initialization. For example, a class has to be initialized immediately and first when a static variable of the class is accessed and the class has not been initialized yet. It is easy for a JIT compiler to initialize the class when compiling an accessing method but it is not allowed.

A conditional branch is a naive means to implement such a just-in-time initialization. But it is not very efficient because it involves overhead to execute the branch every time even the second time and later. the branch is not necessary to the second and later execution of the access to the static variable. It is desirable to avoid the overhead. In many actual cases the compiler can eliminate the conditional branches if the accessed class has been already compiled. But it is better to be able to eliminate the branches in other cases.

A way to avoid such conditional branches is deferment of compilation, which shuJIT did not adopt. The way is not to compile a method until all accesses to static variables and calls to static methods in the method are executed at least once. Cumbersome just-in-time initialization of a class does not occur after all touches to static variables and static calls are once executed. Borland's JBuilder Java 2 JIT adopted this way but it is incomplete. The JIT compiler compiles a method on its second invocation. It is incomplete because the first execution of the method cannot certainly pass through all the touches. One of serious problems of this means is that once an interpreter starts executing a method, the execution can last long. It is a problem because an interpreter is usually inferior to a JIT-compiled method in performance.

The method of shuJIT is different from the deferring means and the compiler ensures the just-in-time initialization even though the compiler has once compiled the accessing method.

#### 4.4.1 Method of shuJIT

ShuJIT uses code patching technique to ensure the rule without suffering the above-mentioned interpreter

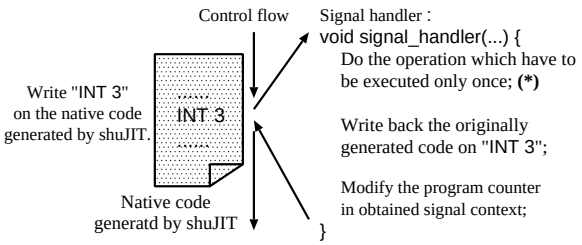



**Fig. 9** Overwriting the original instruction onto an instruction to cause software interruption.

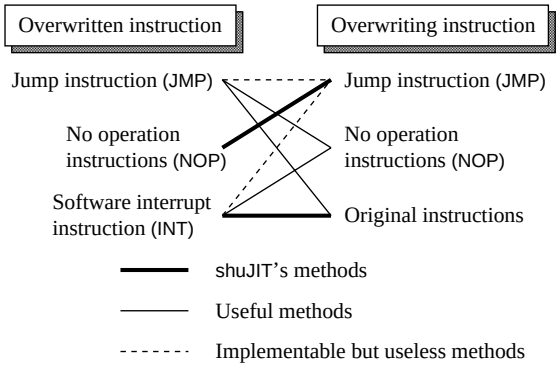



**Fig. 10** Various implementation methods for code patching.

problem. That is to rewrite part of a compiled code just after its first execution to prevent its second execution. The technique enables the compiler to compile the method on its first invocation without any performance penalty a conditional branch introduces. It is an important requirement to be able to compile a method on its first invocation because the compiler should be a baseline compiler. A baseline compiler should be lightweight and complements other kinds of compilers exploiting a peak performance. It can be even an alternative to an interpreter and in that case it compiles a method on its first invocation. Code patching technique for the just-in-time initializations is suitable to a baseline compiler because it enables compilation on the first invocation without any performance penalty in later executions.

We implemented two methods and can choose one of then when compile the compiler itself.

- Fig. 8: overwrite a jump instruction onto no operation instructions (NOP) (jumping method)
- Fig. 9: overwrite the original instructions which should be there onto a software interrupt instruction (INT) (interrupting method)

The compiler uses the only one-byte interrupt instruction `INT 3` (0xCC) as the software interrupt instruction. Code patching like these has to be carried out atomically to avoid inconsistent code sequence expected to other processors and threads. Lock of a memory bus is necessary to assure the atomicity in a case. The lock makes the patching complicated and involves a performance penalty. In contrast to that, rewriting 1 byte is always atomic without any lock on x86 and other modern processors.

The jumping method shuJIT implements rewrites 2 bytes of code. The compiler uses XCHG instruction, not usual MOV instruction, to ensure the atomicity while patching 2 bytes. It is the same method as ORP [11]. Note that it is also possible to implement a kind of jumping method with 1 byte patching. It can be achieved by rewriting only the target of the jump instruction.

Both of the jumping method and the interrupting method have their own advantages. With the jumping method, a jump instruction has to be executed in second and later executions. On the other hand, the interrupting method can eliminate such performance penalties completely. But we should take account of the amount of memory consumption. It depends on the implementation and application programs. The size of compiled can be smaller with the interrupting method. But the interrupting method needs a table holding the original instructions which is overwritten onto a software interrupt instruction.

#### 4.4.2 Other Possible Methods

There could be other methods to rewrite code. As shown in Fig. 10, there are several ways to implement code patching corresponding to pairs of the kind of overwritten instructions and the kind of overwriting instructions. The ways to overwrite the original instructions eliminate any performance penalty in second and later execution. On the other hand, the ways to overwrite no operation instructions or jump instructions do not need a table to hold the original instructions.

### 4.5 Inlining

Inlining is a well-known and frequently implemented optimization technique, which reduces the number and cost of method calls and introduces more opportunities of inter-procedual analysis and optimizations.

ShuJIT inlines a method if the compiler can identify which method is actually called and some conditions are satisfied. In other words, in case the method being called is a static, private or final method the compiler do it. More specifically, the targets of inlining are non-virtual calls with `invokespecial` and `invokestatic` bytecode instructions and a call to a final method with `invokevirutal` instruction. In addition to the kinds of a call, when the following conditions are satisfied the compiler do the inlining.

- The callee does not have a jump instruction.
- The callee does not have a catch clause.
- The length of the callee is equals to or less than *20*.
- Both of the caller and the callee are `strictfp` [5], [10] or both are not `strictfp`.

ShuJIT applies the inlining to a method twice and the nesting level of inlining, in which a method is inlined into an already-inlined method, is *2*.

The above magic numbers, *20* and *2* are parameters which determines how often inlining happens. We can specify them when invoking a JVM. Inlining can make performance worse by filling up instruction cache if it is applied excessively. We determined those default parameters 20 and 2 heuristically to inline accessors (getter and setter methods) while preventing over-inlining. Inlining of accessors is certainly cost-effective because they are small and the call cost is relatively high compared to the execution cost.

We determined not to inline a method having a jump or a catch clause because such an inlining will take more compilation cost including resolution of jump targets. Sun Microsystems' ExactVM took the same policy.

The effects on SPEC JVM98 are not significant as shown in Fig. 5. The effect on _228_jack is the largest and 3 % and the effect on the synthesized score of SPEC JVM98 is 1 %. One possible reason is that the target of the inlining is limited to non-virtual calls. Another one is template connecting code generation. With the code generation technique of shuJIT, inlining does introduce more opportunities of inter-procedual optimizations and its merit is limited.

### 4.6 Special Inlining of Specific Methods

We also tried another type of inlining. The technique just replaces a specific method with already-provided native code. We provided the native code by hand. Its implementation in shuJIT is fairly simple and does not much work. We just provided a internal instruction and templates for the target method and the compiler replaces a call to the method with the internal instruction when parsing bytecode.

We choose the methods of `java.lang.Math` class as the first target of this technique. The target has to be a static or final method to be inlined without any runtime decision and the target methods are static. And further, they are native methods written in C or C++ language, which are especially expected to be improved for the reason we describe later.

> `sqrt`, `sin`, `cos`, `tan`, `atan2`, `atan`, `log`, `floor`, `ceil`

We decided not to apply the special inlining to methods known as its return value changes by this technique, such as `exp`.

**Table 1** Execution times to invoke the sqrt method 10,000,000 times.

| | Pentium 4 | Pentium III |
|---|---|---|
| Without special inlining | 15535 | 34959 |
| With special inlining | 851 | 1567 |
| Improvement ratio | × 18.3 | × 22.3 |

(milliseconds)

Performance of calls to those methods improves for the following two reasons.

- Execution of the method body become faster. Floating-point instructions can be used instead of the software implementation of arithmetics `fdlibm`.
- Calls to native methods are reduced.

There are two classes which have the same set of methods, `java.lang.Math` and `java.lang.StrictMath`. The special inlining is necessary to exploit performance of the `Math` class. `StrictMath` has to yield the exactly same results as Freely Distributable Math Library (`fdlibm`), which is a software-implemented mathematics library. The specification of `StrictMath` prescribes it. The Sun Microsystems' and the IBM's JVM which we use in this paper use the `fdlibm` itself. In contrast to `StrictMath`, `Math` is not required to produce the same result as the `fdlibm`. Because of it `Math` class can do calculation for its methods in more efficient way than `StrictMath` and `fdlibm`, for example, using floating-point instructions of hardware. But both of Sun's and IBM's `Math` class is just a proxy for `StrictMath` class, a call to `Math` class is redirected to `StrictMath` and the calculation is carried out by `fdlibm`. For that reason, those JVM always use `fdlibm` unless the JVM can recognize and deal with calls to `Math` class. To exploit more efficient ways than `fdlibm`, special handling of `Math` class by JVM like shuJIT's special inlining is necessary.

The other reason why performance improvement is expected is reduction of calls to native methods. The `Math` methods listed above are native methods compliant with JNI, the convention for call between Java and native methods. A call to a JNI native method is known as a heavy process. One of the reasons of it is re-stacking of stack values. When Java code calls a JNI native code, it is usually unavoidable to stack values on a Java stack onto a stack for the native method. General inlining technique cannot eliminate such a re-stacking unless the stack layout of JIT-compiled code is the same as JNI's one. It is unlikely for performance reason. Only special handling like shuJIT's special inlining has an effect on calls to native methods.

We measured the effects on performance with calls to `sqrt` and `sin` methods. Table 1 and 2 show the results on 1.7 GHz Pentium 4 and 600 MHz Pentium III respectively. These results demonstrate effectiveness of

**Table 2** Execution times to invoke the sin method 10,000,000 times.

| | Pentium 4 | Pentium III |
|---|---|---|
| Without special inlining | 4242 | 8021 |
| With special inlining | 1567 | 2226 |
| Improvement ratio | × 2.71 | × 3.60 |
| | | (milliseconds) |

the technique as 20 times in case of `sqrt` and 3 times even in case of `sin`. We have to notice that such a technique is effective on a limited number of programs which call specific methods frequently. For example, we could not see any effect on the scores of SPEC JVM98. But it is still very effective in special cases.

The special inlining of shuJIT needs modification to the compiler itself to support more methods even though the modification is easy. It is a better design to support a software component implementing an optimization. Such a component is attached to and detached from the compiler and the addition does not require any modification to the compiler itself.

### 4.7 Memory Alignment of Loop Heads

ShuJIT aligns the loop heads with on a 16 byte boundary. This is a known technique effective on Pentium Pro, Pentium II and Pentium III processors, which fetch 16 bytes from the instruction cache once. Though the effect on Pentium 4 is different from them. L1 instruction cache of Pentium 4 is a trace cache and it is not affected by memory alignment of instructions as long as instructions are found in the trace cache.

This technique does not always improve performance. The alignment naturally yields wasteful spaces between machine instructions. The spaces is filled with no operation instructions (NOP) or a jump instruction in case that the space is large enough to be skipped. Those no operation and jump instructions involve a bit of overhead and performance gain is not always larger than the overhead. And the size of generated code increases and it is possible to have a bad effect. However, the synthesized score of SPEC JVM98 on Pentium 4 improves a little as shown in Fig. 5.

## 5. Performance Evaluation

In this section, we compare shuJIT with other JIT compilers in peak performance and application startup time to clarify its position.

### 5.1 Peak Performance

We adopted SPEC JVM98 benchmark suite and Linpack benchmark to measure peak performance of JIT compilers including shuJIT. The benchmark results are shown in Fig. 11 and Fig. 12. All runs of SPEC JVM98 reported in this paper are compliant with the

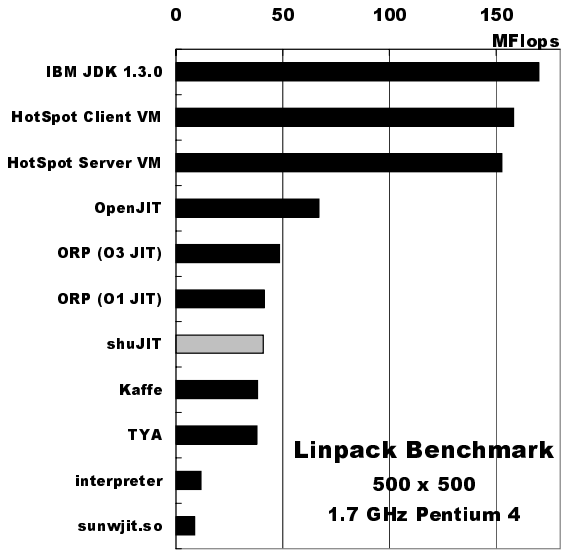



**Fig. 12** The result of Linpack Benchmark for Java.

run rules prescribed by the SPEC, the benchmark development and maintenance body. All codes of the SPEC JVM98 were loaded from a web server. They were started as an applet by the "Auto Run" button with benchmark size 100. Even adjustable parameters are left as the default values in the property file distributed with SPEC JVM98 (`props/user`). In detail, each benchmark runs from 2 times at least (`automin=2`) to 5 times at most (`automax=5`) and the best result is chosen. But the iterative runs are discontinued unless a score exceeds the score of the previous run by 3 % (`percentTimes100=300`). The iterative runs take intervals of 500 milliseconds (`autodelay=500`) and a call to `System.gc()` and a following call to `System.runFinalization()` (`autogc=true`).

SPEC JVM98 benchmark suite has 7 benchmark programs and the final score is synthesized from their execution times. The score is the geometric mean of reciprocal numbers of normalized execution times. Linpack benchmark solves a dense system of linear equations and its score is represented in Flops, floating point number operations per second. In both benchmarks, higher means better.

Investigated JIT compilers target PC or more rich environments like so-called server as well as shuJIT. They are HotSpot Server VM and Client VM of Sun Microsystems' Java 2 SE 1.3.1_02, IBM Java 2 SE 1.3.0 [12], Intel ORP (Open Runtime Platform) [11], Transvirtual Kaffe 1.0.6, TYA 1.7v3 [6], OpenJIT 1.1.16 [13], `sunwjit.so` distributed with Java 2 SE 1.2.2 by Sun Microsystems, and a bytecode interpreter of HotSpot Client VM. The PC on which all results are produced has a 1.7 GHz Pentium 4 and runs Linux 2.4.18-pre3 if any notice is not given.

Note that ORP and Kaffe could not produce SPEC

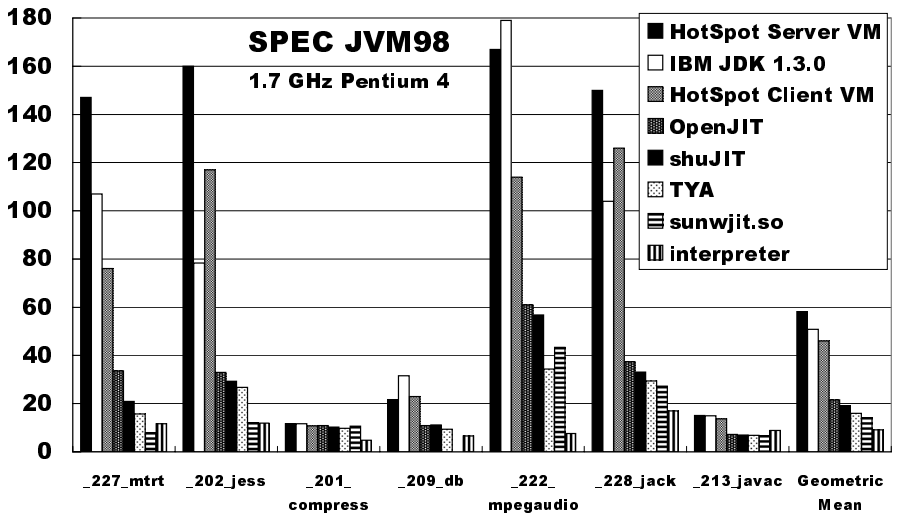



**Fig. 11** The result of SPEC JVM98.

JVM98 results observing the run rules because they did not have necessary libraries. The results of them are not given in Fig. 11 because of it.

Several JIT compilers showed better results than shuJIT even though the results of shuJIT could be several times faster than the interpreter. One reason is that Classic VM, the target JVM of OpenJIT, TYA, `sunwjit.so` and shuJIT is not originally very performance-oriented. Another reason is that shuJIT had been developed as a baseline compiler. A baseline compiler should be light-weight and it is used even as an alternative of an interpreter. For example, ORP [11] and Jikes RVM [14] have a baseline compiler instead of an interpreter. Performance-oriented JIT compilers of Hot Spot VM and IBM Java 2 SE have to limit target methods their compilation strictly because the compilation takes much time and much compilation hurts performance oppositely. Those JVMs first use an interpreter to run a method if it has not been yet compiled and compile it when the total number of call to it and backward jump in the method reaches much number as 1000 or 10000. Efficiency of an interpreter and a baseline compiler is still important a performance-oriented compiler cannot compile all methods.

### 5.2 Time to startup a Java Program

The results shown in **5.1** are peak performance of JIT compiled code. Those benchmarks do not show JIT compilation time because they are designed to measure peak performance. Already-compiled methods are executed many times and compilation time takes a slight time relatively in overall execution time. Otherwise only execution time of compiled methods is measured. For example, in SPEC JVM98 a benchmark runs several times and the best result is adopted.

We tried comparison of JIT compilation time. The criteria we adopted is application startup time. JIT compilation frequently happens while many methods are called for the first time during application startup process. Compilation time especially takes much in the startup time in a setting that a method is compiled on its first invocation because all called methods are compiled.

Such a comparison should involve direct measurement of JIT compilation time. But direct measurement is almost impossible if source code is not available and we cannot have it for several JIT compilers. Further, preemptive context switch by OS and Java thread switch also make it difficult. Application startup time is an index of usability of the application program and also is a usability factor of a JVM. Its reduction is still important however much JIT compilation time is involved.

Application programs used here are a word processor Ichitaro Ark 1.1 and an integrated development environment (IDE) NetBeans 3.3.1. They are practical software with a certain amount of code, as Ichitaro Ark has 904 classes, the size of compressed code (JAR file) is 1652 kilobyte and the size of NetBeans is 2200 kilobyte. It is another reason to choose them that we can easily confirm the completion of their startup by our own eyes. We measured the startup time with a stop watch by hand.

The target JIT compilers are JITC 3.6 of IBM Java 2 SE 1.3.0, HotSpot Client VM, Server VM and shuJIT. They are chosen because we can specify the threshold. It is the number of invocation of a method in which the method is compiled. These JIT compilers provide an invocation counter for each method, decrease it on its invocation and start compiling the method if the counter reaches 0. The initial number of the counter depends on a JIT compiler. It is 2000 in IBM JITC 3.6 for Linux/x86, 1500 in HotSpot Client VM and 10000

**Table 3** Startup time of Ichitaro Ark.

| Initial number of counter | shuJIT | IBM JITC | HotSpot Client VM | HotSpot Server VM |
|---|---|---|---|---|
| 0 | 4.4 * | 22.1 | N/A | N/A |
| 3 | 4.0 | 14.1 | 5.9 | 32.1 |
| 1500 | 4.3 | 6.3 | 3.8 * | 6.8 |
| 2000 | 4.4 | 5.7 * | 3.7 | 6.6 |
| 10000 | 4.5 | 5.2 | 3.8 | 5.4 * |
| $\infty$ | 3.9 | 5.0 | 4.0 | 4.1 |

(second)

∗ The default of the initial number

**Table 4** Startup time of NetBeans.

| Initial number of counter | shuJIT | IBM JITC | HotSpot Client VM | HotSpot Server VM |
|---|---|---|---|---|
| 0 | 8.9 * | 43.3 | N/A | N/A |
| 3 | 8.4 | 28.6 | 12.0 | 166.6 |
| 1500 | 10.3 | 10.8 | 8.6 * | 20.7 |
| 2000 | 10.7 | 10.6 * | 8.5 | 20.9 |
| 10000 | 13.2 | 10.5 | 8.9 | 17.2 * |
| $\infty$ | 12.2 | 14.0 | 13.0 | 13.4 |

(second)

∗ The default of the initial number

in HotSpot Server VM. In shuJIT it is 0 because the compiler is a baseline compiler which has to possibly compile all called methods.

We measured startup time for each initial numbers as 0, 3, 2000 and 10000. The results of Ichitaro Ark is shown in Fig. 3 and Fig.4 is for NetBeans. $\infty$ means prohibition of JIT compilation. An interpreter executes an entire application in that case. HotSpot VMs could not run all Java programs if the initial number is set as less than 3. In Fig. 3 and Fig.4, N/A reflect it.

For the results in Fig. 3 and Fig.4, comparison between different JIT compilers cannot be strictly fair, because they have different interpreters and JVMs and the timing in which invocation counters are decrement varies according to the compilers. ShuJIT works with the Classic VM and IBM JITC uses a modified version of the JVM. HotSpot VM is a completely different JVM from them. The timing of shuJIT is method invocations. IBM JITC takes backward branches in addition to method invocations. HotSpot VM also takes branches and invocations but it is not clear whether the JVM takes account of the direction of a branch.

An exception is the initial number 0. In that case all called methods become targets of JIT compilation and all compilers can be compared impartially. In other cases JIT compilation time seems to affect startup time much. HotSpot Server VM is known to take much time to compile a method and the results show it takes over 30 seconds with the initial number 3. It is natural to take the result as startup time reflects JIT compilation time well. Heavier JIT compilers tend to take more time and The results in Fig. 3 and Fig.4 back up lightness of shuJIT's JIT compilation.

## 6. Conclusion

A Java Just-in-Time compiler shuJIT achieved ease of use as a research base and light-weight compilation. In this paper, we described code generation method and optimization techniques implemented in the compiler. The compiler demonstrates practical performance as several times faster than an interpreter while its compilation takes smaller amount of time. These properties are sufficient to be a baseline compiler which is possibly used even as an alternative of an interpreter. The compiler also takes relatively small amount of development work as it starts working in 50 days after the start of its development.

For the light-weight optimization techniques, we discussed implementation, effects on performance, implication in the code generation method and the Java specifications. We could have the following results. Efficiency of method invocation affects many benchmarks much. Instruction folding is effective to the code generation method of shuJIT. Handling exceptions with signal takes much time in case an exception is thrown but the cost can be compensated by a certain number of normal executions. And we showed a code patching technique whereby we can eliminate a performance penalty caused by the Java specifications. The technique is suitable for a baseline compiler because it enables JIT compilation on the first invocation of a method.

Another goal of the development of shuJIT was to be a base of derived researches which modify JIT-compiled code. The compiler has been used as a base by other researchers, not only us and the goal has been well achieved.

Future work includes comparison between interpreter and baseline compiler in performance, usability and development cost. Compilation strategy and its construction methodology are also worthwhile. The strategy will involve disposal strategy of JIT compiled code considering memory consumption and performance gain.

### References

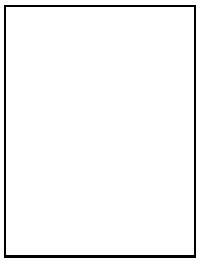
[1] M.A. Ertl, "Stack caching for interpreters," Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp.315–327, 1995.

[2] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Addison Wesley, 1997.

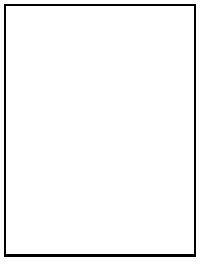
[3] M. Welsh and D. Culler, "Jaguar: Enabling efficient communication and I/O from Java," Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications, Dec. 1999.

[4] K. Shudo and Y. Muraoka, "MetaVM: A transparent distributed object system supported by runtime compiler," Proc. of 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), VOL II., pp.879–882, June 2000.

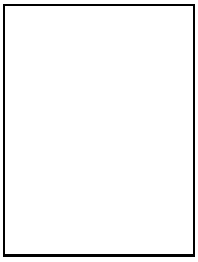
[5] K. Shudo and Y. Muraoka, "Efficient implementation of strict floating-point semantics," Proc. of 2nd Workshop on Java for High-Performance Computing (in conj. with ICS'00), pp.27–38, May 2000.
[6] A. Kleine, "TYA." http://sax.sax.de/~ adlibit/.
[7] A.R. Adl-Tabatabai, M. Cierniak, G.Y. Lueh, V.M. Parikh, and J.M. Stichnoth, "Fast, effective code generation in a Just-In-Time Java compiler," Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), pp.280–290, June 1998.
[8] L.R. Ton, L.C. Chang, M.F. Kao, H.M. Tseng, S.S. Shang, R.L. Ma, D.C. Wang, and C.P. Chung, "Instruction folding in Java processor," Int'l Conference on Parallel and Distributed Systems, pp.138–143, 1997.
[9] H. McChan and M. O'Connor, "PicoJava: A direct execution engine for Java bytecode," COMPUTER, vol.31, no.10, pp.22–30, Oct. 1998.
[10] J. Gosling, B. Joy, G. Steele, and G. Bracha, Java Language Specification, Second Edition, Addison Wesley, 2000.
[11] M. Cierniak, G.Y. Lueh, and J. Stichnoth, "Practicing JUDO: Java under dynamic optimizations," Proc. ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI), June 2000.
[12] 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 Journals, Java Performance Issue, vol.39, no.1, Feb. 2000.
[13] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura, "OpenJIT: An open-ended, reflective JIT compile framework for Java," Proc. of 14th European Conference on Object-Oriented Programming (ECOOP '2000), June 2000.
[14] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño virtual machine," IBM Systems Journals, Java Performance Issue, vol.39, no.1, Feb. 2000.



**Kazuyuki Shudo** Kazuyuki Shudo received BE, ME and PhD in computer science from Waseda University in 1996, 1998 and 2001, respectively. He worked as a Research Associate at the same university from 1998. Since 2001, he is a Research Scientist at National Institute of Advanced Industrial Science and Technology. His research interests include distributed computing, language systems and information security. He is a member of IEEE Computer Society and ACM.



**Satoshi Sekiguchi** Satoshi Sekiguchi is the director of Grid Technology Research Center, AIST. He received BS from the Department of Information Science of the University of Tokyo, MS from University of Tsukuba in 1982 and 1984, respectively. He joined Electrotechnical Laboratory, Agency of Industrial Science and Technology, MITI, Japan in 1984, and he served as the deputy director of Research Institute of Information Technology, AIST. Since 2002, he is the founding director of Grid Technology Research Center, AIST. He is a member of IEEE CS and SIAM.



**Yoichi Muraoka** Yoichi Muraoka is a Vice President of Waseda University. He received his BE degree from Waseda University in 1965, and a PhD degree in computer science from the University of Illinois in 1971. Then he worked as a Research Associate at the same university and Nippon Telegraph and Telephone Public Corporation. Since 1985, he is a professor at the School of Science and Engineering. His research interests include parallel processing and human-machine interface.