Efficient Implementation of Strict Floating-Point Semantics

Kazuyuki Shudo Yoichi Muraoka

School of Science and Engineering, Waseda University,
3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, JAPAN
{shudoh,muraoka}@muraoka.info.waseda.ac.jp

Abstract. The keyword '`strictfp`' has recently been introduced into the Java language and virtual machine specifications. '`Strictfp`' is a method, class, and interface modifier. In its syntactical scope, all floating-point operations must keep the strict semantics defined by the language specification regardless of the processor on which a Java virtual machine is running. The Intel x86 processor architecture, however, is not naturally compliant with the semantics, although almost all other processors are compliant. We have implemented the semantics on a Just-In-Time compiler which translates Java bytecode to the x86 native code. In the process we have investigated, developed, and compared efficient methods of achieving the `strictfp` semantics. This paper reports these methods and experimental results, and evaluates their performance. It also addresses problems that remain despite the introduction of '`strictfp`'.

Keywords: floating-point semantics; x86; Java; `strictfp`; Just-In-Time compiler

1 Introduction

Most recent processor architectures are compliant with IEEE 754 [1], which is a standard prescribing floating-point arithmetics, including the format of a floating-point number and basic operations like addition, rounding, and so on. In fact, the SPARC, MIPS, PowerPC and Intel IA–32 (the x86), all comply with IEEE 754. Nevertheless, the x86 gives different results than the other processors listed above for certain floating-point operations. This special behavior of the x86 can cause problems in distributed and parallel computation exploiting heterogeneous computers. Computation results can vary depending on how sub-tasks are assigned to the computers.

The Java language specification (JLS)[5] required the same floating-point semantics as the SPARC. Therefore, Java systems for the x86[1] such as Java virtual machines, just-in-time (JIT) compilers, and ahead-of-time compilers, had to spend processor cycles to preserve the semantics and thus suffered a performance penalty. Since Java systems which implement the semantics cannot exploit the full capacity of these processors, Intel, IBM, and others requested Sun Microsystems (Sun) to modify the JLS so as to make full use of the x86 and the PowerPC processors. The activities of the Numerics Working Group of the Java Grande (JGNWG) resulted in changes to the floating-point semantics of Java language and the introduction of the '`strictfp`' keyword [2][9][3].

The modifications of the JLS eliminated almost all the performance disadvantages of the x86. Sun introduced '`strictfp`' as a class, method, and interface modifier of the Java language. The original strict semantics were preserved in syntactical and static scope of the modifier, while the default semantics outside the context of `strictfp` was relaxed slightly.

[1]These semantics also affect the IBM PowerPC and other processors. We concentrate on the x86 architecture in this paper.

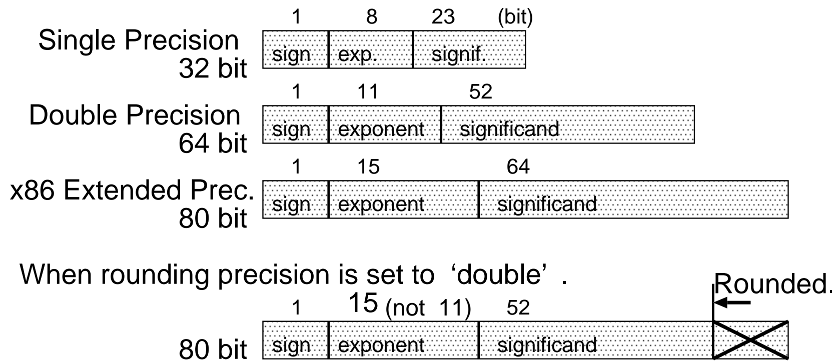


Figure 1: x86's floating-point value format

An efficient way of implementing the strict semantics has already been proposed by Roger A. Golliver [2] and the JGNWG has recommended it to Java system developers. Golliver's technique can be implemented in several different ways. We implemented them in a JIT compiler [7] to test their effectiveness and examine their performance. This study represents the first attempt to implement strict floating-point semantics on JIT compilers. To study not only performance drawbacks with JIT compilers but also the penalties imposed by the technique itself, we also developed benchmark programs in C, and modified the assembly code of the programs to satisfy the semantics. This paper describes the peculiar behavior of the x86 and Golliver's technique. We then addresses the implementation of '`strictfp`' to find a way to make JIT compilers compliant with 'strictfp'. We consider rounding precision and choices of scaling methods. The results of performance comparisons between the scaling methods are also shown.

2 Problem with the x86

This section explains the problem which the x86 floating-point unit imposes on developers of Java runtime systems. The x86 has a particular *extended precision* as a floating-point value format, in addition to single and double precision defined in IEEE 754 (Fig. 1). According to IEEE 754, normalized numbers are expressed in the following format:

$$(-1)^s 2^E (1.b_1 b_2 \cdots b_{p-1})$$

Here each symbol represents:

s	:	a sign bit (0 or 1)
b_i	:	a bit in significand (0 or 1)
p	:	precision of significand [bit]
E	:	exponent

The number of bits taken by the exponent and significand parts is shown in Table 1.

In the x86 processor, floating-point values are always held in the 80-bit extended precision. Additionally, the x86 does not have machine instructions corresponding to each precision, although almost all other processors have. For example, SPARC has `fmuls` and `fmuld` instructions, and MIPS has `mul.s` and `mul.d` instructions. Instead of instructions such as these, the x86 provides rounding precision bits in its status register. We can set the bits as one of the three precisions shown in Fig. 1, causing the results of floating-point operations to be rounded to that precision.

precision	single	double	extended
whole width [bit]	32	64	80
prec. of signif.(p) [bit]	24	53	64
width of E [bit]	8	11	15
max E	+127	+1023	+16383
min E	-126	-1022	-16382

Table 1: The properties of each precision of x86

Note, however, that only the significand part is rounded. The exponent part is not rounded and always has a 15-bit width as shown in Fig. 1. In other processors, the exponent is rounded to 8 bits by single-precision operations, and to 11 bits by double-precision operations. Consequently, the overflow and underflow that occur in other processors may not occur in the x86. For instance, the following operations in double precision result in $+\infty$ in the SPARC, but 2^{1023} in the x86:

$$
\begin{array}{rcl}
register & = & +1023 \\
register & += & +1023 \\
register & -= & +1023
\end{array}
$$

The proper overflow threshold must be fulfilled to satisfy the same strict semantics as SPARC. Additionally, we must also implement the proper gradual underflow behavior, which is prescribed in IEEE 754 [2].

The problem described here is common to all x86-compliant processors — even x86-compatible processors designed by companies other than Intel, such as AMD's K6 and Athlon.

3 Golliver's Technique

A technique developed by Roger A. Golliver achieves the same strict floating-point semantics for the x86 as with the SPARC and the other processors [2]. It is the best technique currently known and JGNWG strongly encourages developers of the Java runtime system based on the x86 to adopt it. The technique consists of two sub-techniques, which are *store-reload* and *scaling*.

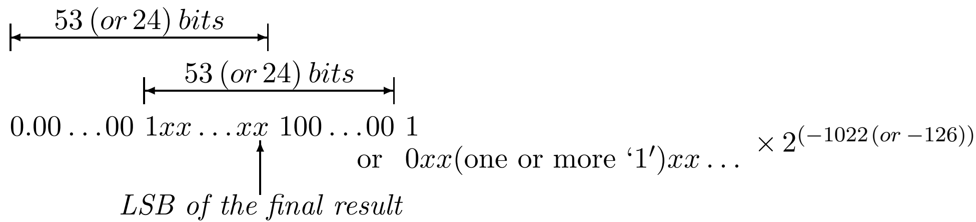
3.1 store-reload

The exponent part (by default, 15 bits) has to be rounded to the proper width (11 bits for double precision and 8 bits for single precision) for the strict semantics. We can achieve this rounding by storing the result to memory in the appropriate precision and reloading it to a floating-point register from memory. The exponent part of the results is also rounded when the result is stored to memory. This technique is called *store-reload*. All the four rules of arithmetic require it.

3.2 scaling

It is possible, however, for the double-rounding problem to occur when the store-reload technique is applied alone. Thus, the store-reload technique alone cannot satisfy the semantics. In certain cases, an operation result is rounded twice and deviates from the desirable value rounded only once. An operation result can be a denormalized (subnormal) number in double or single precision but

that number is represented as a normalized number in the x86 extended precision with the x86's wider exponent range. In this case, the exponent of the operation result is first rounded to 15 bits and then rounded to 11 or 8 bits again, when store-reload is applied. This double-rounding can occur in multiplication and division, so these two types of operations have to be looked at more closely.



(note that '0...0' is a sequence of '0' and 'x' is a '0' or '1'.)

Figure 2: A binary number that is rounded twice.

Fig. 2 shows the format of values rounded twice in this manner. The following double-precision operation is an example affected by such double rounding:

$$1.1128085544714844E-308 \quad \times \quad 1.0000000000000002$$

(in IEEE 754 double precision: 0x0008008000000000 × 0x3ff0000000000001)

The least significant bit (LSB) of the rounded result should be incremented by carrying up to preserve the strict semantics. But the LSB is not incremented if it is rounded once with the 15-bit exponent and rounded again with the 11- or 8-bit exponent.

Scaling is a technique that prohibits this sort of double rounding. It causes rounding with the desirable threshold corresponding to a precision of the operation and preserves the gradual underflow behavior. Multiply one of the operands by a *scale*, which is a constant number, ahead of a floating-point operation (scale down or up). Next, multiply the reciprocal of the constant number by the result (scale up or down). Assume that to scale down the multiplicand or the dividend in advance, the scale should be $2^{-16382-(-1022)}$ in the case of double precision and $2^{-16382-(-126)}$ for single precision.

4 Implementation of Strict Semantics

We implemented Golliver's technique on shuJIT [7], which is a JIT compiler for the x86. JIT compilers that are compliant with the strict semantics have to generate native code that deals with the following:

- Rounding precision.
- Overflow and underflow (including gradual underflow),
  achieved with *store-reload* and *scaling*.

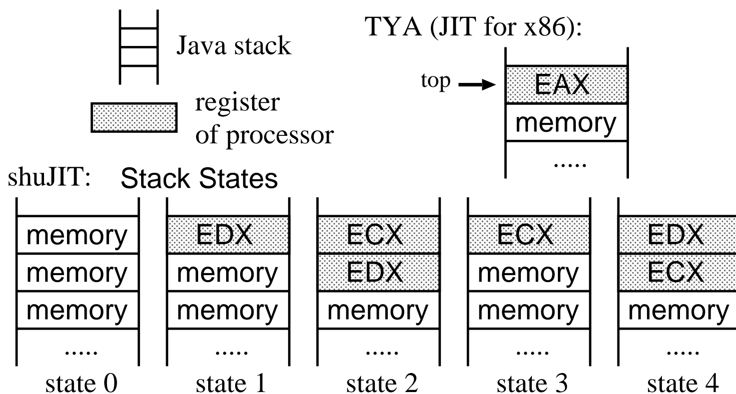


Figure 3: Five stack states introduced by shuJIT

This section explains how to make the JIT compiler compliant with the `strictfp` and discusses the efficient management of rounding precision and overflow and underflow. Several methods can be used to achieve the right overflow and underflow behavior. The performance of several implementations is evaluated in Section 5 because the choice affects the penalty introduced into floating-point operations.

4.1 Modification of the Just-In-Time compiler

We chose shuJIT as the Java runtime system to be modified to implement the strict semantics. shuJIT is a Java bytecode just-in-time (JIT) compiler for a combination of the x86, Sun classic VM, and Linux/FreeBSD. It has been publicly available on the world wide web since September 1998. Its compilation method is simple and requires a low overhead. But it's performance is sufficient to make it a practical compiler. The compiler translates Java bytecode into the x86 native code almost one to one, like TYA [4], which is another JIT compiler. But, in contrast with TYA, a pre-assembled native code segment for a bytecode instruction has five variations corresponding to five stack states (Fig.3). Each code segment not only has a state ahead of the execution of the segment but also has a state behind. The compiler connects these code segments along the states.

As described above, the compiler does not perform the usual compilation techniques such as register allocation and instruction scheduling. Accordingly, code generated by shuJIT cannot run as fast as the compilers from IBM or Inprise. Nevertheless, this compiler is practical. But shuJIT is suitable for experiments involving modifications to the Java runtime system. It is easy to modify the behavior of JIT-compiled native code since we can change the native code directly by hand. Consequently, shuJIT has been used as the base compiler by researchers [10][8] investigating modifications to the Java runtime system.

We introduced additional internal instructions into shuJIT to implement `strictfp`. The compiler generates a sequence of internal instructions from the Java bytecode instructions, and then translates the internal instructions into x86 native code. The following internal instructions were added:

- *Pre-loading and post-releasing of scales:*
  `strictenter`, `strictexit`

- *Floating-point operations compliant with the strict semantics:*
  `fmul_strict`, `dmul_strict`, `fdiv_strict`, `ddiv_strict`

Java runtime systems compliant with its specification have to correctly handle the default semantics when `strictfp` is not specified, because the JLS also prescribes the default semantics. The proper default semantics requires the attention of Java runtime system developers. The default semantics can be achieved by such as controlling the rounding precision corresponding to each operation, or the store-reload technique. shuJIT satisfies the default semantics by setting the rounding precision as double and using the store-reload technique.

In this study, special internal instructions for the strict semantics are provided for only multiplication and division, not for addition and subtraction. The behavior of shuJIT results in this design. Multiplication and division need both *store-reload* and *scaling*, but addition and subtraction need only *store-reload* (Section 3). A strict version of addition and subtraction is unnecessary because shuJIT does *store-reload* whenever floating-point operations are performed. For other JVM's, it may be necessary to take specific actions to handle addition and subtraction semantics.

We provided two switches in shuJIT in order to control the behavior of the compiler. One, '`frcstrictfp`', which forces all floating-point operations to obey the strict semantics. The other, '`ignstrictfp`', instructs the compiler to ignore the `strictfp` modifier. These switches can be specified via the environment variable '`JAVA_COMPILER_OPT`'. They are useful in examining the performance of existing programs in the strict context.

4.2 Rounding Precision

Generated native code has to deal with the rounding precision bits in the control word of the x86 floating-point unit (FPU). It is clearly sufficient to set it as the precision corresponding to each operation, for example, single precision for a single-precision operation. But frequent re-setting is inefficient because setting the rounding precision involves memory access, which introduces large access latency if a cache miss occurs.

In fact it is safe to keep double precision as long as the *store-reload* technique applies to every single-precision operation. In other words, rounding precision can be double throughout the strict context, and setting a precision corresponding to each operation is not necessary. When a single-precision operation is performed with rounding precision as double, the significand of the result of the operation is rounded twice:

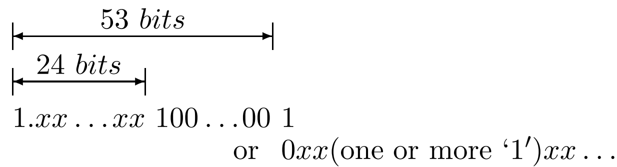
1. Rounded to double (53 bits) precision when it is stored in an FPU register.
2. Rounded to single (24 bits) precision when it is stored in memory (*store-reload*).

If the double-rounded result can be different from the number rounded only once to single precision, keeping the rounding precision as double causes a problem, and cannot be done. In reality, there is no problem. The results of single-precision addition and subtraction are never affected by the former rounding because the significand of the result reaches only 25 bits at most. The product reaches 48 bits at most. The quotient can reach 54 bits or more but it cannot take the form which suffers double rounding as shown in Fig. 4.

4.3 Overflow and Underflow

A combination of *store-reload* and *scaling* techniques satisfies the strict floating-point semantics completely (Section 3). In this study, we do not have to implement the store-reload technique anew because shuJIT always does it. However, scaling must be implemented and we have several choices

In the case of a *normalized* number:



In the case of a *denormalized* number:

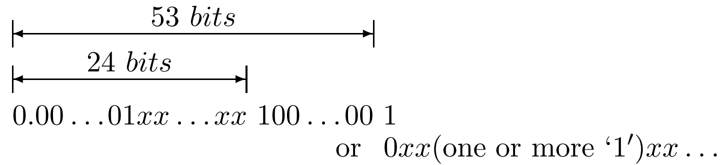


Figure 4: The significand that suffers the double rounding.

when implementing it: a way of scaling, the timing of loading scales, the selection of pre-loading scales, and the format of scales in memory.

First of all, there are two methods that operate to scale a floating-point number:

1. Multiply by the n-th power of 2 (with the `fmul` instruction).

2. Use the `fscale` instruction of x86.

`fscale` is a machine instruction for scaling by the n-th power of 2.

Secondly, we can choose the timing to load scales into FPU registers:

1. When scaling is carried out.

2. When a `strictfp` method is called (*pre-loading*).

3. When the JIT compiler is initialized.

There is a trade-off between these choices. Trying to save memory accesses with *pre-loading* causes the FPU registers to be occupied for a long time. We implemented and compared the first and second of these three choices.

Next, which scale do we load into the FPU register? Because half the scales are reciprocals of the remaining half, half can be calculated on demand even if they are not loaded from memory. Therefore, we have the following choices:

- All four scales:
  - For double precision: $2^{-(16383-1023)}$ and $2^{(16383-1023)}$.
  - For single precision: $2^{-(16383-127)}$ and $2^{(16383-127)}$.

- Two scales:
  $2^{-(16383-1023)}$ and $2^{-(16383-127)}$. When post-operation scaling, divide the operation result by these scales instead of multiplying the result by the reciprocal provided as a scale in advance. Or calculate the reciprocal and multiply it.

- Only the necessary scales:
  Load only scales for single or double. Because all four scales for both single and double precisions are not always necessary, we can omit unnecessary scales.

We chose simple implementations as follows:
When scaling is performed with . . .

- *Multiplication*: pre-load all four scales.
- `fscale` *instruction*: pre-load two scales ($-(16383 - 1023)$ and $-(16383 - 127)$).

This decision is based on the cost of calculating the reciprocal number. Scaling with multiplication requires the real reciprocal, while scaling with the `fscale` instruction takes the exponent of the reciprocal. Therefore, in the case of `fscale`, calculating the reciprocal is only turning over the sign of the exponent. It is achieved with the `fchs` machine instruction and the cost of the instruction is relatively low compared with the cost of calculating the real reciprocal. Consequently, we decided to pre-load all four scales for scaling with multiplication and two scales for the `fscale` method.

We can also choose the format of scales in memory. When a scale is loaded into an FPU register, it is translated from the format in memory to extended precision format in a register. The performance of single- and double-precision floating-point formats and a 32-bit integer format was compared, noting that the integer format is allowed only with the `fscale` scaling. The format would affect traffic across the memory hierarchy from the memory to the registers and the translation costs from the format to the extended precision format. However, no difference was observed between double and single floating-point formats. The integer format needed additional 130 or 140 milli-seconds for 10^7 times translations on a 333-MHz Pentium II processor. Hence, we adopted single-precision format which occupies less memory than double.

5 Experimental Performance Results

Scaling method	Pre-load	shuJIT		C & assembly code		BulletTrain	
		time (msec)	rate	time (msec)	rate	time (msec)	rate
raw (not strict)		N/A	N/A	3463	1.00⋆	3290	1.00⋆
only store-reload (not strict)		3683	1.00⋆	3530	1.02	N/A	N/A
`fscale` insn.	yes	5631	1.53	5521	1.59	N/A	N/A
`fscale` insn.	no	5733	1.56	5656	1.63	6980	2.12
multiplication	yes	9843	2.67	9663	2.79	N/A	N/A
multiplication	no	10698	2.90	10344	2.99	N/A	N/A

⋆: The standard of rates.

Table 2: Performance of multiplication

Test programs were developed to compare the efficiency of the different methods in order to implement the strict floating-point semantics. We implemented not only test programs in Java language, but also C programs that does the same test. Then we modified the assembly code derived from the C program in order to implement the strict semantics in the several ways stated in Section 4. In addition to our implementations, BulletTrain[6] 1.4.0 was evaluated. It is an ahead-of-time compiler of Java bytecode which supports the `strictfp`.

Scaling method	Pre-load	shuJIT		C & assembly code		BulletTrain	
		time (msec)	rate	time (msec)	rate	time (msec)	rate
raw (not strict)		N/A	N/A	8489	1.00⋆	330†	1.00⋆
only store-reload (not strict)		8460	1.00⋆	8527	1.00	N/A	N/A
`fscale` insn.	yes	25167	2.97	24982	2.94	N/A	N/A
`fscale` insn.	no	25248	2.98	25024	2.95	14440†	43.8
multiplication	yes	23705	2.80	23489	2.77	N/A	N/A
multiplication	no	24388	2.88	23951	2.82	N/A	N/A

⋆: The standard of rates.

†: These measured times do not make sense because the calculation results deviate from SPARC's and shuJIT's.

Table 3: Performance of division

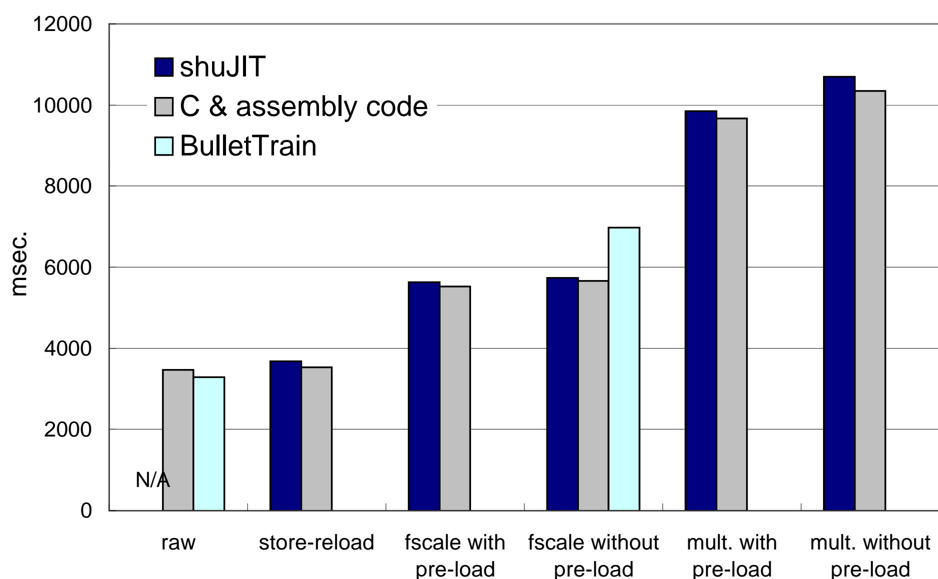


Figure 5: Performance of multiplication

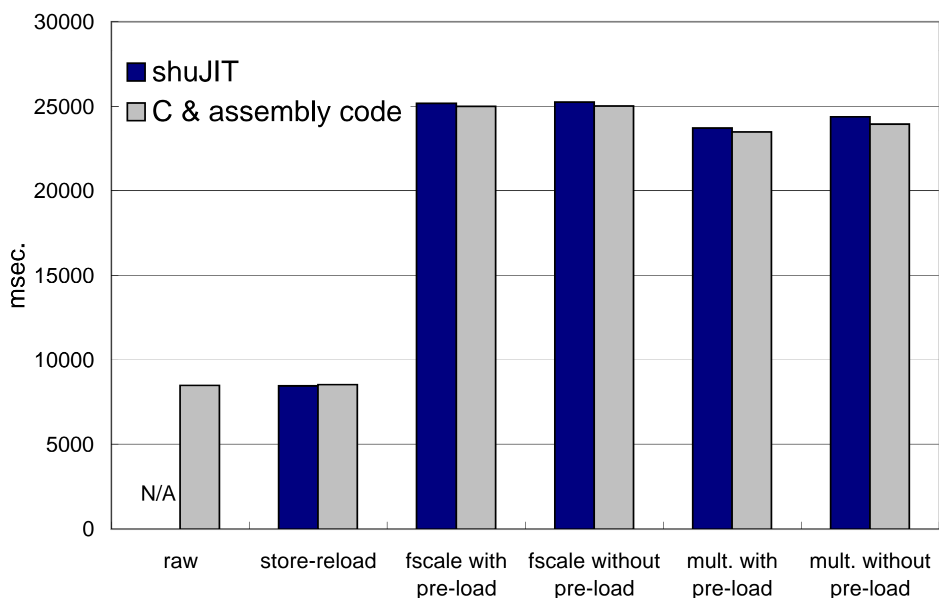


Figure 6: Performance of division

The test programs repeat a loop (which has 10 multiplications or divisions in its body) 10^6 times, so the number of operations is 10^7. In the C program, each operation takes one operand from an FPU register and the other from memory: almost all memory accesses will hit data cached in a processor. We used '1.23456789012345' as the operand of all operations. We should take care of the operands of multiplication and division because the latency of these operations depends strongly on its operands. The number we chose involved a relatively large latency.

The amount of time taken by all 10^7 operations was measured. All tests were carried out on a Mobile 333-MHz Pentium II, Linux 2.2.15-pre15, and Blackdown Java Development Kit 1.2.2 RC4. Table 2 and Fig. 5 show the results when multiplications were repeated. Table 3 and Fig. 6 present the division results.

The results of shuJIT and the results of native code are almost identical. This indicates that shuJIT introduces almost no overhead in such small benchmark programs. In case of multiplication, scaling with the `fscale` instruction shows better performance than scaling with multiplication. S-caling with multiplication introduces additional two multiplying operations to one original operation and the number of operations becomes three times as many as in the original raw code. Therefore, an observed performance decline (2.77∼2.88) seems to be appropriate. Regarding division, the two scaling methods in shuJIT incur roughly the same overhead. Scaling with multiplication is slightly better than the `fscale` method.

Consequently, scaling with the `fscale` method suits for multiplication and division prefers the multiplication method. The effect of pre-load in performance is small in comparison with the overhead introduced by the scaling itself. Although these results are useful for examining methods of implementing strict semantics, we should keep in mind that they are somewhat limited. We did only a simple permutation of native code in this study. A floating-point operation was expanded into a fixed native code sequence that complies with the semantics. If a generated code sequence is combined with surrounding other code and then some optimizations are applied, different results may be obtained.

6 Remaining Problems

The 'strictfp' keyword and its semantics have been incorporated into the Java language specification (JLS). Nevertheless, in reality a few problems still remain. This section lists these problems and shows what Java runtime system developers should take care of.

6.1 Lack of Implementation

Implementations of the strictfp and its strict floating-point semantics on the x86 are rare. Even Sun has not implemented it in their Java virtual machines, although even though they published the language specification describing the strictfp. There have been only two implementations. One is BulletTrain [6]. This is an ahead-of-time compiler, which translates Java class files that contain bytecode into the x86 native code. The other is shuJIT, on which we implemented strictfp.

6.2 Java Compiler Support of strictfp

Most Java compilers calculate numerical expressions at compile time as much as they can. Because the calculations have to be carried out in an appropriate context — default or strict — Java compilers must be aware of strictfp. The Sun's Java compiler javac takes no account of the context. It always assumes the default context. By running a Java compiler on shuJIT, we can confirm the compliance of the Java compiler with strictfp.

6.3 Single-precision Operations in Default Context

The floating-point semantics in the default context were relaxed by the introduction of the strictfp into the JLS. But as before, there are still some constraints even in the default context. The width of exponent bits was relaxed but the significand has to comply with the precision of each operation, 24 bits for single operation and 53-bit for double. Developers of the Java runtime system should note this specification.

To maintain the appropriate width of significand bits, the default semantics needs accurate control of the rounding precision or the store-reload technique. If the underlying processor supports the Streaming SIMD Extensions (SSE) instructions, which is an instruction set implemented in PentiumIII and newer Celelon, we can utilize certain instructions of SSE for single-precision operations to preserve the appropriate width. This is more efficient way than controlling the rounding precision, and the store-reload technique because the SSE instructions do not have to access memory.

7 Conclusion

In this paper, we have addressed the x86's problematic specification in relation to floating-point operations and Golliver's solution. We have developed several methods of implementing Golliver's solution and discussed their characteristics. These methods were implemented in a JIT compiler called shuJIT and their performance was compared. This is the first attempt to implement strict floating-point semantics in a JIT compiler. These experiments yielded guidelines for implementing the strictfp keyword and its semantics. We also addressed problems remaining despite the fact that the strictfp keyword has been introduced into the Java language specification.

Programming languages other than Java do not have any language factor like the strictfp. It is interesting to incorporate the strict semantics in other languages including C and Fortran in terms of language design and implementation. The incorporation is in demand for clustering of

various computers and distributed, global, and heterogeneous computing. Experiments on other x86-compatible processors such as AMD's Athlon and Pentium are interesting because they have different architectures from the Pentium II used in this study.

References

[1] IEEE standard 754-1985 for binary floating-point arithmetic, 1985.

[2] Java Grande Forum Numerics Working Group. Improving Java for numerical computation, October 1998. http://math.nist.gov/javanumerics/.

[3] Java Grande Forum Numerics Working Group. Recent progress of the Java grande numerics working group, June 1999. http://math.nist.gov/javanumerics/.

[4] Albrecht Kleine. TYA Archive. http://sax.sax.de/˜ adlibit/.

[5] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison Wesley, 1997.

[6] NaturalBridge, Inc. BulletTrain optimizing bytecode compiler, 1998. http://www.naturalbridge.com/bullettrain.html.

[7] Kazuyuki Shudo. shuJIT—JIT compiler for Sun JVM/x86, 1998. http://www.shudo.net/jit/.

[8] Kazuyuki Shudo and Yoichi Muraoka. MetaVM: A transparent distributed object system supported by runtime compiler. In *Proc. of 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, June 2000. to appear.

[9] Sun Microsystems, Inc. Updates to the Java language specification for JDK release 1.2 floating point, 1999. http://java.sun.com/docs/books/jls/strictfp-changes.pdf.

[10] Matt Welsh and David Culler. Jaguar: Enabling efficient communication and I/O from Java. In *Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications*, December 1999.