厳密な浮動小数点演算セマンティクスの Java 実行時コンパイラへの実装

首藤 一幸 † 関口 智嗣 † 村岡 洋一 ††

IA-32 プロセッサは，IEEE 754 準拠であるにも関わらず，ある浮動小数点演算に対して他のプロセッサとは異なる結果を返す．この問題への対処を Java Just-in-Time コンパイラに実装した．いくつかの実装方法について性能へ影響を調べたところ，我々の方法ではペナルティを約 40%に抑えられることが判った．

Efficient Implementation of Strict Floating-Point Semantics

Kazuyuki Shudo† Satoshi Sekiguchi† Yoichi Muraoka††

IA-32 processors yield different results of floating-point operations from other processors, even though they are all compliant with IEEE 754. We implemented the floating-point semantics of the other IEEE 754 compliant processors on a Java Just-in-Time compiler. Performance evaluation demonstrates that our implementation method reduced the penalty to 40%.

1 はじめに

最近のほとんどのプロセッサは IEEE 754 [2] に準拠している．これは浮動小数点演算の規格で，浮動小数点数の表現形式や，四則演算，平方根などの基本的な操作，また丸め，例外の規定を含んでいる．事実，SPARC，Alpha，PowerPC，MIPS，IA-32（通称 x86）はすべて IEEE 754 準拠である．それにも関わらず，IA-32 は，ある演算に対して上記の他のアーキテクチャとは異なる結果を返す．IA-32 のこの特殊な仕様によって，演算を行う計算機によって結果が異なるという，計算の再現性の問題が起こり得る．また，異機種混在環境での並列，分散計算では，サブタスクの各プロセッサへの割り当てがどうなされるかによって結果が異なるという問題が起こる．

処理結果の再現性を重視している Java 言語，Java 仮想マシン [4] にとっても IA-32 の仕様は問題であった．異機種間での再現性を保証するため，Java 言語仕様 [1] は SPARC などと同じセマンティクスを浮動小数点演算に要求していた．IA-32 でこのセマンティクスを達成するには演算以外の補整処理が必要であり，IA-32 用の Java 処理系は性能上のペナルティを強いられていた．とはいえ，現実には各処理系ベンダはこの仕様を実装せず，Intel 社，IBM 社他の要求によって仕様の側が緩和されるに至った．

Java 2 より，言語仕様に `strictfp` という，クラス，メソッド，インタフェース修飾子が導入された．`strictfp` のレキシカルスコープ内では従来通りのセマンティクスが要求される反面，スコープ外，つまり `strictfp` が指定されない限りは，IA-32 そのままのセマンティクスも許容されるようになった．それでも，`strictfp` の文脈では補整が必要である．

`strictfp` のためのこの補整処理にはいくつかの技法がある．我々はそれらを Java バイトコードの実行時（JIT）コンパイラ [8][6] に実装し，それぞれの性能への影響を調べた．また，特定の JIT コンパイラへの影響だけでなく技法それ自身のペナルティを調べる目的で，Java 言語で記述したものと同等のベンチマークプログラムをアセンブリコードでも記述した．

本報告では，まず，問題となる IA-32 独特の仕様を説明し，その上で，Java 言語の `strictfp` が規定する他のプロセッサと同じセマンティクスを達成する手法を述べる．続いて，いくつかの実装方法を提案し，それらの効率を比較する目的で行った性能評価の結果を示す．

2 IA-32 の特徴的な仕様

IEEE 754 では，正規化数（normalized number）は次のように表現される．

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

ここで，それぞれの記号の意味は次の通りである．

s	:	符号ビット（0 か 1）
b_i	:	仮数部中の 1 ビット（0 か 1）
p	:	仮数部の精度 [bit]
E	:	指数部

†産業技術総合研究所 National Institute of Advanced Industrial Science and Technology
††早稲田大学 理工学部 School of Science & Engineering, Waseda University

	単精度	倍精度	拡張精度
全体の長さ [bit]	32	64	80
仮数部の精度（p） [bit]	24	53	64
指数部（E）の精度 [bit]	8	11	15
E の最大値	+127	+1023	+16383
E の最小値	-126	-1022	-16382

表 1: 各精度の諸元

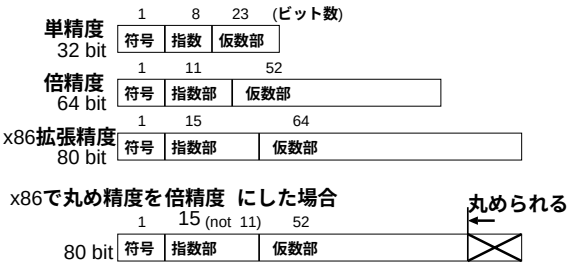


図 1: 浮動小数点数の表現形式と IA-32 の丸め精度

IA-32 は浮動小数点数の表現形式として，IEEE 754 が定める単精度（32bit），倍精度（64bit）の他に，独自の拡張倍精度（80bit）を持つ（表 1，図 1）．IEEE 754 はこのような拡張形式を許している．

IA-32 プロセッサで x87 浮動小数点演算命令を用いる場合，プロセッサ内では常に，浮動小数点数は 80 bit の拡張精度で保持される．さらに，他のアーキテクチャとは異なり，x87 命令には精度ごとの演算命令がない．例えば SPARC であれば `fmuls`，`fmuld` 命令を，MIPS であれば `mul.s`，`mul.d` 命令を持っている．その代わりに，丸め精度を単精度，倍精度，拡張精度のいずれかに設定可能であり，この設定に従って演算結果が丸められる．ここで IA-32 を特徴付けているのは，丸め精度が仮数部にしか影響を与えないという仕様である．丸め精度が何であれ，指数部は常に拡張精度と同じ 15 bit で表現される（図 1）．他のアーキテクチャでは，単精度数の演算であれば 8 bit に，倍精度であれば 11 bit に丸められる．ところが IA-32 内部では指数部は常に 15 bit なので，他のアーキテクチャでは起こる溢れが発生しない場合がある．例えば，次の演算をレジスタ上で倍精度で行った場合，他のアーキテクチャでは途中でオーバフローが起こりレジスタの値は $+\infty$ となるが，IA-32 で x87 命令を用いるとオーバフローは起こらず，2^{1023} が得られる．

$$\begin{aligned} \text{レジスタ} &= 2^{+1023} \\ \text{レジスタ} &+= 2^{+1023} \\ \text{レジスタ} &-= 2^{+1023} \end{aligned}$$

`strictfp` の文脈では IA-32 であっても他のアーキテクチャと同じ挙動が要求されるので，何らかの工夫が必要となる．

ここで述べた IA-32 の仕様は，Intel 社以外の互換プロセッサ，例えば AMD 社の K6 や Athlon，Transmeta 社の Crusoe にも共通のものである．

3 Golliver の手法

IA-32 上で，`strictfp` の要求する他のアーキテクチャと同じセマンティクス（FP-strict [7] と呼ぶ）を達成する手法が Golliver によって提案されている [3]．これは，現在知られている最も効率の良い手法で，Java Grande の Numerics Working Group も推奨している．本稿は，この手法の様々な実装方法を比較した結果を報告するものである．この手法は，*store-reload* と *scaling* という 2 つの部分から成る．

3.1 store-reload

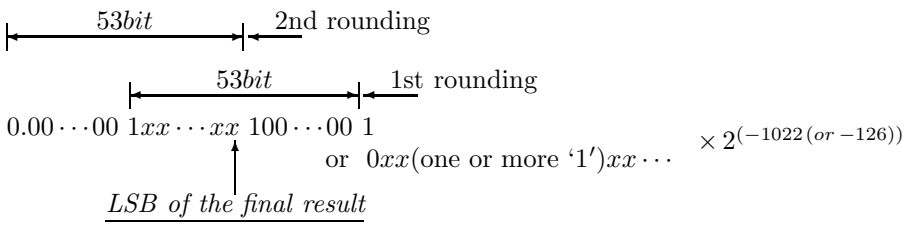
まず，仮数部の溢れを適切に起こすことを考える．レジスタ上では常に 15 bit である仮数部を，倍精度なら 11 bit，単精度なら 8 bit で溢れさせる．これは，適切な精度でレジスタからメモリへストアし，再びレジスタにロードすることで達成できる．仮数部は，メモリへのストア時に適切に丸められる．この操作は，加減乗除のすべてで必要である．

3.2 scaling

しかし *store-reload* だけでは完全ではない，演算時とメモリへのストア時の 2 回，値が丸められて，FP-strict な結果とは異なってしまうことがある．すなわち，アンダーフローが起きて，倍精度としては非正規化数（denormalized number）として表されるべき値が，指数部が 15 bit あるためにレジスタ上では正規化数（normalized number）として表されてしまい，ストア時になって初めて非正規化数に変換される場合である．これが起きると，演算時に一度で非正規化数に丸められた場合とは結果が異なってしまうことがある．この二度丸めの問題は単精度演算では起きないため，ここでは倍精度演算についてのみ考えればよい．その理由は第 4.2 節で述べる．

図 2 は，この二度丸めによって，FP-strict ではなくなってしまう演算結果の形式である．この形式の値は倍精度では最終的に非正規化数として表されるが，指数部が 15 bit ある IA-32 のレジスタ上では正規化数として表現できてしまう．つまり，演算時には正規化数に丸められる．その後，メモリへストアする際に非正規化数に変換され，ここで二度目の丸めが起きる．このように二回丸められた場合，最終結果の最下位ビット（LSB）は 0 となるが，本来の FP-strict な演算では 1 でなければならない．例えば，この問題によって次の演算の結果が変わってしまう．

$1.112080544714844E-308 \times 1.0000000000000002$
$(0x0008008000000000 \times 0x3FF0000000000001)$
$2.225073858507201E-308 \div 0.9999999999999999$



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

図 2: 二度丸めの影響を被る値

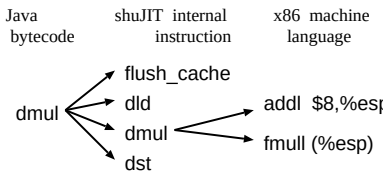


図 3: shuJIT のコード生成手法

(0x000FFFFFFFFFFFF ÷ 0x3FEFFFFFFFFFFFF)

この二度丸めを防ぐためには，演算の時点で適切にアンダーフローが起き，非正規化数が得られればよい．そのためにオペランドと演算結果の *scaling* を行う．一方のオペランドにある定数を乗じておき，演算結果にその定数の逆数を乗じる．これによって，アンダーフローが起きる境界を調節できる．この定数は $2^{-16382-(-1022)}$ である．この手法は，非正規化数による漸進的な（gradual）アンダーフローという IEEE 754 の規定も満足させる．

4 JIT コンパイラへの実装

第 3 章で説明した Golliver の手法を，Java バイトコードの実行時（JIT）コンパイラである shuJIT [8][6] に実装した．JIT コンパイラが生成するネイティブコードは次の処理を行う必要がある．このためにどのような実装を行ったのかを述べていく．

- 丸め精度を設定する．
- 指数部の溢れを適切に起こす（漸進的アンダーフローを含む）．

4.1 JIT コンパイラの変更

shuJIT は IA-32 用の JIT コンパイラで，OS として FreeBSD と Linux をサポートする．図 3 のように，コンパイル処理の最初に，Java バイトコードの命令列を内部命令に変換する．それに対していくつかの最適化を施し，最後に内部命令をネイティブコード列に置換することでコード生成を行う．ただし，ひとつの内部命令に対するネイティブコード列を最大 5 通り用意することで，複数のレジスタを活用できるようにしている [8]．コード長に対する計算量のオーダが高い処理を行わないために，コンパイル処理が軽いこと，つまり，コンパイル時間に対する性能比がよいことが特徴のひとつである．

今回，Golliver の手法のためにいくつかの内部命令を追加した．追加したのは，scaling を行う命令と，scaling に使う定数をレジスタへプリロード（後述）する命令，レジスタを解放する命令である．内部命令列を生成する際，これらを生成するようにした．例えば，乗除算の前後には scaling 命令を，プリロードを行う際はメソッドの先頭と末尾にプリロードおよび解放命令を生成する．

また，コンパイラの挙動を変えるオプションを追加した．ひとつは `frcstrictfp` で，これを指定すると，あらゆる浮動小数点演算を FP-strict だとみなしてコンパイルする．もうひとつは逆の働きをする `ignstrictfp` で，これが指定された場合 `strictfp` 修飾子は無視される．`frcstrictfp` オプションは，これを指定することで既存のアプリケーションを変更なしに FP-strict だとして実行できるため，実装方法の性能評価に有用であった．

4.2 丸め精度の設定

IA-32 は精度に応じた x87 演算命令を持たず，あらゆる演算に影響を与える丸め精度を設定できるのみである（第 2 章）．FP-strict な演算を行うためには，この丸め精度の設定にも気を配らねばならない．

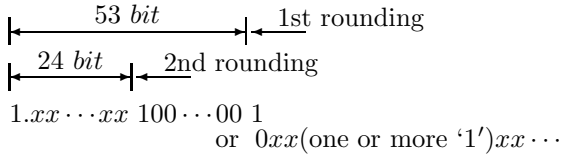
最も素朴なのは，演算の前にその演算に応じた精度に設定するという方法だろう．しかし，丸め精度の設定にはオーバヘッドが伴うため，設定の回数は極力減らしたい．実は，もし，常に store-reload（第 3 章）を行うなら，丸め精度は倍精度に設定したままでよい．そのまま単精度演算を行っても，第 3.2 節で述べた二度丸めの問題は起き得ないため，scaling は不要である．

丸め精度を倍精度に設定したまま単精度演算を行った状況を考える．まず，指数部の扱いはもともと丸め精度の設定に影響されない（第 2 章）ので問題ない．仮数部は，次の通り二回丸められる．

1. 演算時に倍精度（53 bit）に．

2. store-reload でのメモリへのストア時に単精度（24 bit）に．

正規化数（*normalized* number）の場合:



非正規化数（*denormalized* number）の場合:

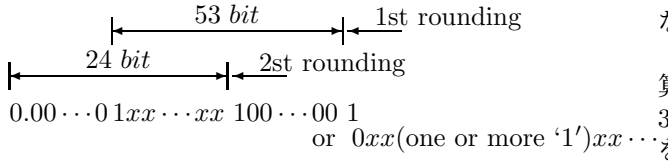


図 4: 二度丸めで値が変わる仮数部

ここでもし，このように 2 段階にわたって丸められた結果が一度で単精度まで丸められた結果と異なることがあるならば問題である．この二度丸めで問題が起きる演算結果は，図 4 の形式に限られる．

しかし，演算結果はこの形式をとり得ないため，丸め精度を倍精度としたまま単精度演算を行ってもまったく問題はない．理由を以下に示す．乗算，加減算，除算と 3 つの場合に分けて考える．単精度数の仮数部は 24 bit 幅なので，乗算の真の（丸め前の）結果はたかだか 48 bit 幅である．それに対して，図 4 の形式は最低でも 54 bit 幅である（最上位と最下位の '1' が 53 bit 離れている）．積は図 4 の形式をとり得ない．

加減算の結果は非正規化数にはなり得ないため，図 4 中の正規化数の形式をとり得るかを考える．図 5 は単精度数どうしの加減算について仮数部の処理を筆算で行っている状況を表している．二度丸めが問題となる正規化数の形式では，最上位ビットから数えて 54 bit 目以降に '1' がある．加減算の結果で 54 bit 目以降が '1' となるためには，図 5 に示しているように，2 つのオペランドの桁が最低でも 30 bit は離れている必要がある．このとき，破線で囲んだ部分が問題のあるビットパターン（1 に続いて 5 bit 以上の 0）をとることはあり得ない．ゆえに加減算の結果は図 4 の形式をとり得ない．

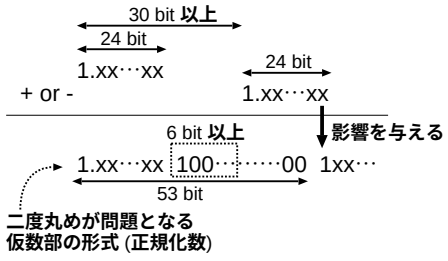


図 5: 単精度数どうしの加減算

最後に除算について考える．単精度数の演算 $a \div b$ の真の（丸め前の無限精度の）結果が c である場合，$c \times b = a$ が成立する．c が図 4 の形式をとると仮定する．図 4 の形式は最低でも 54 bit 幅である．54 bit 幅以上の c に 24 bit 幅以下の b（$\neq 0$）を乗じて 24 bit 幅以下の結果を得ることはできない．つまり，a が 24 bit 幅以下の単精度であるという仮定に反するため，除算の真の結果は図 4 の形式をとり得ない．b が 0 の場合は積 a も 0 となるが，元の除算において $b = 0$ はゼロ除算であり，その結果はやはり図 4 の形式をとり得ない．

以上により，丸め精度を倍精度としたまま単精度演算を行っても，store-reload に伴う二度丸めの問題（第 3.2 節）は起きないことが判った．つまり，store-reload を行うだけで FP-strict となる．

4.3 溢れ

JIT コンパイラが生成するコードは，指数部の溢れを適切に起こすために store-reload を行わねばならない．また，倍精度演算の場合は，store-reload に伴う二度丸めの問題を回避するために scaling を行わねばならない（第 3 章）．

scaling を行う方法は，IA-32 の x87 命令の場合，次の二通りが考えられる．

- **乗算命令**: 2^n を乗じる．
- `fscale` **命令**: 2 の巾による scaling 命令を用いる．

また，scaling に用いる定数をメモリからレジスタにロードするタイミングにもいくつかの選択肢がある．

- scaling の直前にロードする．
- プリロードしておく．
  - `strictfp` が指定されたメソッドの先頭．
  - （JIT コンパイラの初期化時.）

JIT コンパイラの初期化時にロードしてしまうと，プログラムの実行中は常にレジスタが定数に占有されてしまうので，プリロードする際はメソッドの先頭で行うようにした．

scaling を `fscale` 命令で行う場合，定数をメモリ上に置いておく形式として，整数，単精度数，倍精度数の 3 通りが考えられる．333 MHz の Pentium II プロセッサ上で，プリロードを行わずにそれぞれの方法を比較した．その結果，単精度数と倍精度数では同じ演算性能が得られたが，それに対して整数では 10^7 回の演算あたり 130 から 140 ミリ秒のオーバヘッドがあった．整数をレジスタ上の浮動小数点数に変換するため

のオーバヘッドだと考えられる．この結果に基づき，単精度数を採用した．

今回，2 通りの scaling 方法と，プリロードの有無を組み合わせた 4 通りの実装を行い，それぞれの性能を評価した．

5　性能評価

第 4.3 節で述べたいくつかの実装方法について性能を測定，比較した．比較対象として BulletTrain [5] 2.0.0b15 を用意した．BulletTrain はプログラム実行前にコンパイルを済ませてしまう Ahead-of-Time コンパイラであり，我々の知る限り shuJIT 以外で唯一 `strictfp` を正しく実装している IA-32 用処理系である．

実験環境として，2 種類の IA-32 プロセッサ，1.7 GHz の Pentium 4 と 650 MHz の Mobile Pentium III を用意した．OS は，BulletTrain での実験では Windows 2000 SP2，その他の実験では Linux を用いた．shuJIT と共に用いた Java 処理系は Blackdown JDK 1.2.2 FCS である．

5.1　乗除算

まず，store-reload と scaling の双方が必要となる，倍精度の乗算，除算の性能を測った．10 回乗算または除算を含むコードを 5×10^6 回繰り返して実行して時間を計測した．

C 言語の処理系に FP-strict な演算セマンティクスを実装した場合にはどの程度の性能が得られるのかも評価した．とはいえ，C コンパイラに実装したのではなく，Java 言語で記述したものと同じベンチマークプログラムを C 言語でも記述し，そこから生成したアセンブリコードに各種 scaling 方法（第 4.3 節）を実装することで上記の状況を模した．C コンパイラは GCC 2.95.2 を用いた．

こういった小規模で作為的なベンチマークプログラムを作成する際は，コンパイラによる最適化，つまり不要コードの除去や共通部分式の除去などの影響を考慮する必要がある．最適化によって意図したものとは異なる処理となってしまう場合がある．ここでは，上記の最適化を妨げるために，変数の使用を記述するなどの工夫を施し，コンパイラが生成したコードを見て目的のベンチマーク処理が行われることを確認した．また，乗除算に要する時間は演算のオペランドに依存するので，オペランドが 0 や ∞ になって演算時間が極端に短くなってしまいわないよう注意を払う必要もある．

図 6 から図 9 はそれぞれ，Pentium 4 での乗算，Pentium 4 での除算，Pentium III での乗算，Pentium III での除算に要した時間である．`fmul` 命令とは，scaling に乗算の `fmul` 命令を用いた場合を表している．4 通りの scaling 方法の他に，何も補整処理を行わない場合（raw），store-reload のみを行った場合も測定した．

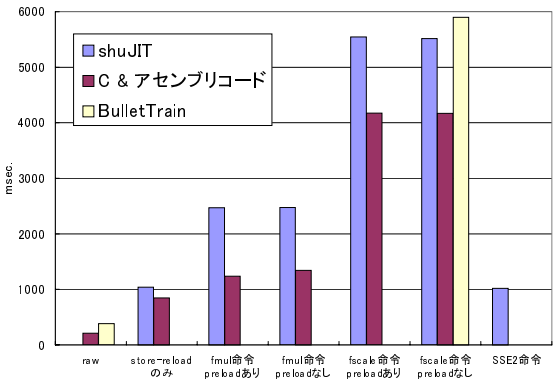


図 6: Pentium 4 での乗算

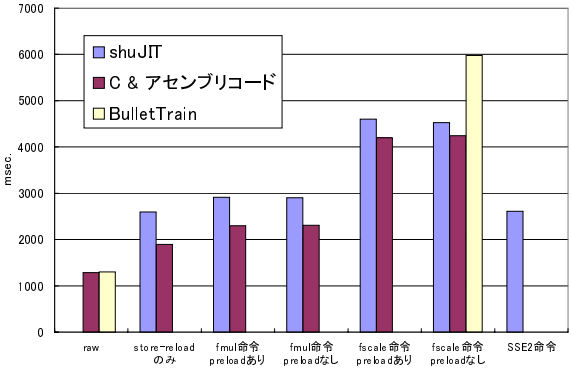


図 7: Pentium 4 での除算

この 2 者の演算方法は FP-strict ではない．

shuJIT での結果を見ると，scaling の方法としては `fscale` 命令を用いるよりも乗算（`fmul` 命令）を行った方が，約 2〜3 倍速い．Pentium 4 と Pentium III では乗算を用いた方が良いと言えるだろう．

表 2 に C 言語での FP-strict のペナルティを示す．補整処理なし（raw）の実行時間に対する，補整処理手法の中で最も速かったもの，つまり `fmul` 命令でプリロードありの場合の比である．Java Grande Numerics WG は FP-strict によるペナルティは 2 から 4 倍と予測しており，表 2 の最大約 6 倍という値はこれを逸脱している．しかし今回の実験方法では補整処理のため

演算	raw（ミリ秒）		`fmul` 命令，プリロード	
P4, 乗算	209	→	1238	5.92 倍
P4, 除算	1284	→	2299	1.79 倍
PIII, 乗算	603	→	1694	2.81 倍
PIII, 除算	3150	→	4233	1.34 倍

(P4: Pentium 4, PIII: Pentium III)

表 2: C 言語でのペナルティ

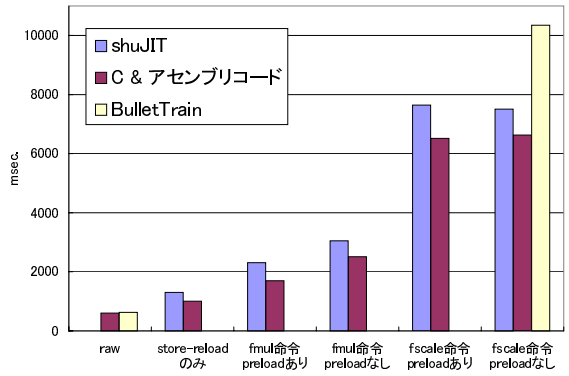


図 8: Pentium III での乗算

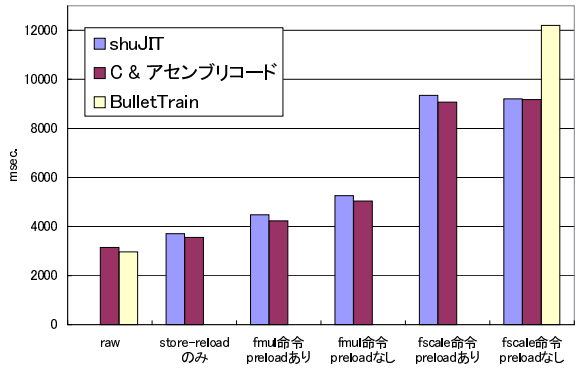


図 9: Pentium III での除算

に追加した命令はコンパイラによる命令スケジューリングの対象となっていないため，実際に C コンパイラに実装した場合は，オーバヘッドはもう少し小さくなるであろう．

5.2 SPEC JVM98 と Linpack Bench.

図 10，図 11 に，Pentium 4 での SPEC JVM98 と Linpack ベンチマークの結果を示す．大きい値が良い結果である．shuJIT で `fmul` 命令を用いた場合に着目して，FP-strict の場合に補整処理なしの場合と比較してどれだけの性能となっているかを算出すると，SPEC の mpegaudio で 58.2%，Linpack で 60.1%となっている．浮動小数点演算が中心のプログラムでは約 60%の性能となることが判る．

6 まとめ

IA-32 で他の IEEE 754 準拠アーキテクチャとまったく同じ演算結果を得る手法を，Java JIT コンパイラに実装した．オーバヘッドを伴う丸め精度の設定は倍精度に固定しておいてよいこと，scaling の方法は `fscale` 命令よりも乗算の効率が良いこと，浮動小数点数が中心のプログラムでペナルティは約 40%に抑えられることが判った．

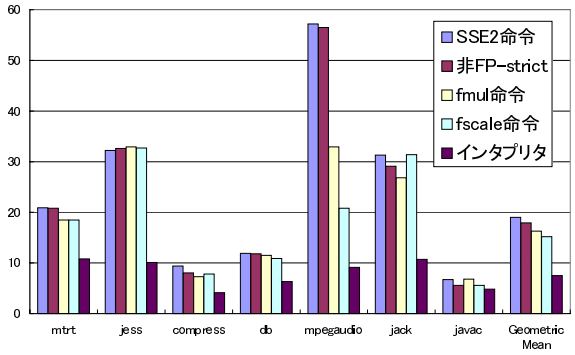


図 10: Pentium 4 での SPEC JVM98

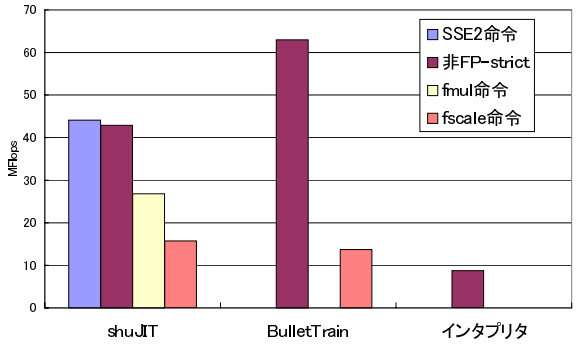


図 11: Pentium 4 での Linpack Benchmark

参考文献

[1] Ken Arnold, James Gosling, and David Holmes. *Java Language Specification, Third Edition*. Addison Wesley, 2000.

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

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

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

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

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

[7] 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.

[8] 首藤一幸, 根山亮, 村岡洋一. プログラマに単一マシンビューを提供する分散オブジェクトシステムの実現. 情報処理学会論文誌:プログラミング, Vol. 40, No. SIG 7, pp. 66–79, August 1999.