Java 実行時コンパイラへの strictfp の実装 *

首藤 一幸　村岡 洋一
{shudoh,muraoka}@muraoka.info.waseda.ac.jp
早稲田大学 理工学部

SPARC と通称 x86 はどちらも浮動小数点演算規格 IEEE754 に準拠したプロセッサアーキテクチャである。しかし SPARC と x86 はある同じ演算に対して異なる結果を返す。

Java 言語と仮想マシン (JVM) の仕様は SPARC と同じ挙動を要求してきたので、x86 上の JVM が仕様を満たすためには工夫が必要であり、ある程度のオーバヘッドは避けられなかった。そこで Intel 社や IBM 社などが Sun 社に出した要求がきっかけとなり、Java 言語の浮動小数点演算についてのセマンティクスが変更され、メソッド、クラス、interface に対する修飾子 `strictfp` が導入された。`strictfp` のスコープ、文脈ではそれまでの言語仕様通りのセマンティクスであり、どの JVM も同じ演算結果を返すべきであるが、`strictfp` ではない文脈では x86 が SPARC などに対して不利にならないよう、言語仕様が緩められた。

x86 の挙動は `strictfp` の要求と異なるため、JVM、Just-In-Time コンパイラ (JIT) など Java 実行系は `strictfp` を実装する必要がある。性能のロスが比較的小さい手法が提案され [1]、Java Grande Forum の Numerics Working Group(JGNWG) もそれを推奨している。しかし実装例がほとんどない。商用 Ahead-of-Time コンパイラ [2] がひとつ実装しているのみである。Sun(JDK 1.3beta) も IBM(JDK 1.1.8) も実装していない。

そこで、手法の有効性を検証しオーバヘッドを調べるため、JIT コンパイラ shuJIT[3] に `strictfp` を実装した [5]。本稿では、x86 の挙動と `strictfp` の実現手法を説明し、`strictfp` 実装の際の種々の選択、また、それらの比較結果を報告する。

1　x86 の挙動

x86 プロセッサは浮動小数点数の表現形式として、IEEE754 が定める単精度 (32bit)、倍精度 (64bit) の他に、独自の拡張精度 (80bit) を持つ (図 1)。IEEE754 はこのような独自の拡張形式を許している。

x86 内部で浮動小数点数は常に拡張精度 (80bit) で保持される。そして x86 は各精度に応じた演算命令を持たない。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、それに従って演算結果が丸められる。ここで x86 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ことである。丸め精度の設定が何であれ、指数部は常に 15bit で表現される。SPARC や MIPS など他の多くのアーキテクチャは精度

*An Implementation of '`strictfp`' Semantics in Java Just In Time Compiler
Kazuyuki SHUDO, Yoichi MURAOKA
School of Science and Engineering, Waseda University

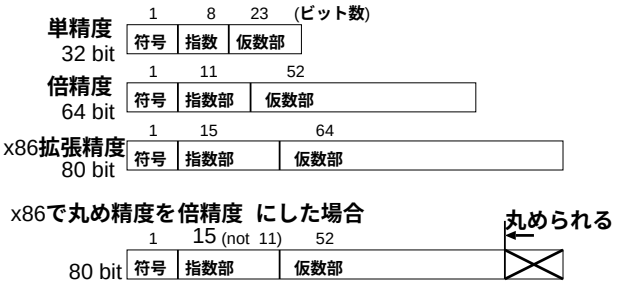

図 1: 浮動小数点数の表現形式と x86 の丸め精度

に応じた演算命令を持ち (例:fmuls と fmuld、mul.s と mul.d)、仮数部だけでなく指数部も倍精度なら 11bit、単精度なら 8bit に丸められる。ところが x86 内部では指数部は常に 15bit 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。例えばレジスタ上で $register = 2^{1023}$,$register+ = 2^{1023}$,$register- = 2^{1023}$ という倍精度数の演算を行った場合、SPARC では途中で overflow が起こり $register$ の値は $+\infty$ となるが、x86 では overflow が起こらずに 2^{1023} が得られる。

`strictfp` の文脈では SPARC などと同じ挙動が要求されるので、x86 用 Java 実行系では工夫が必要となる。

2　手法

x86 上で SPARC 同様のセマンティクスを実現する手法として Roger A. Golliver が提案し、JGNWG が推奨している手法がある [1]。

store-reload　まず、適切に溢れを起こすために、加減乗除算のたびに結果をレジスタからメモリに書き戻し、再びレジスタにロードする。これにより、指数部が倍精度数なら 11bit、単精度数なら 8bit に丸められる。

scaling　しかしこれ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、SPARC などとは結果が異なってしまうことがある。すなわち、倍 (単) 精度数としては非正規化数 (denormalized number) として表されるべき値が 15bit 指数部では正規化数 (normalized number) として表現されてしまい、ストア時になってはじめて非正規化数に変換されて 2 度目の丸めが起きる場合である [4][1]。これを防ぐために、演算の時点で、レジスタ上で適切に underflow が起きるようにしておき、非正規化数が得られるようにする。

演算前にオペランドに定数を乗じ、演算後にその定数の逆数を乗じる。適切な定数を用いることで、指数部

が 11(8)bit である場合と全く同じ条件で underflow を起こすことができる。倍精度の場合は $2^{-16382-(-1022)}$、単精度の場合は $2^{-16382-(-126)}$ がその定数である。この 2 度丸めは加算、減算では起き得ないので、この処理は乗除算で行えば充分である。

3 実装

この手法を JIT コンパイラ shuJIT に実装した。JIT が生成するネイティブコードは次の処理を行う。

- 丸め精度を設定する。
- 適切に指数部の溢れを起こす。

x86 は精度に応じた演算命令を持たないとはいえ、演算に応じてたびたび丸め精度を設定し直す必要はない。store-reload(第 2節) を行うなら、単精度数の演算であっても倍精度に設定しておいたままで問題は起きない。理由は次の通りである。まず、指数部の扱いは丸め精度の設定に影響されない (第 1節) ので問題ない。仮数部はレジスタ上で倍精度に、メモリにストアされる際に単精度に丸められる。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に丸められた結果と異なるとすれば問題だが、加減乗除算のいずれでもそれは起こらない。

指数部の溢れを適切に起こすために、store-reload および scaling を行う。store-reload は shuJIT では常に行ってしまっているので今回考慮する必要はない。scaling についてはいくつかの選択肢がある。

- scaling 方法:
  2^n を乗じるか x86 の `fscale` 命令 (2 の巾による scaling 命令) を用いるか。
- レジスタに scale をロードするタイミング:
  乗じる時点 (事前ロード無し), メソッドの先頭 (事前ロード有り) など。メモリアクセスの頻度とレジスタ消費量 × 時間のトレードオフ。

それぞれを実装してオーバヘッドの大きさを比較した。

scale をレジスタに事前ロードしておく場合は、4 つ ($2^{-(16383-1023)}$,$2^{16383-1023}$,$2^{-(16383-127)}$,$2^{16383-127}$) の scale すべてをロードしておくか、または事前ロードは 2 つにしておいて除算を行うか必要なときに逆数を生成するか、といった選択がある。scaling を乗算で行う場合は 4 つすべてを事前ロード、`fscale` 命令を用いる場合は必要に応じて逆数を生成することにして事前ロードする定数を 2 つに抑えた。fscale 命令を用いる場合は逆数の算出コストが低い (`fchs` 命令:符号の逆転) ためである。

また scaling を `fscale` 命令で行う場合、メモリ上に scale を置いておく際の形式として倍精度数、単精度数、整数の 3 通りが考えられる。次節の条件 (Pentium II) で実験した結果、単精度数と倍精度数では同じ演算性能が得られたが、それと比べて整数では 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。この結果に基づき単精度数を採用した。

4 実験

`strictfp` の文脈にない演算と `strictfp` の何通りかの実装法について、乗算 (除算)10 回を含むコードを 10^6 回繰り返し実行して時間を計測した。用いたプロセッサは MMX Pentium 233MHz と Mobile Pentium II 333MHz 、OS は Linux 2.2.13pre7、Java 実行系として JDK 1.1.7v3 と 1.2 pre-release 2 を用いた。

scaling 方法, 事前ロード	乗算 (時間, 比)		除算 (時間, 比)	
非 strictfp	3701	1.00	796	1.00
fscale 命令, 有	5598	1.51	2590	3.25
fscale 命令, 無	5680	1.53	2671	3.36
乗算命令, 有	9855	2.66	1097	1.38
乗算命令, 無	10674	2.88	1085	1.36

表 1: Pentium II, JDK 1.2 での結果 (ミリ秒)

scaling 方法, 事前ロード	乗算 (時間, 比)		除算 (時間, 比)	
非 strictfp	4004	1.00	903	1.00
fscale 命令, 有	5648	1.41	2749	3.04
fscale 命令, 無	5727	1.43	2836	3.14
乗算命令, 有	10434	2.61	1117	1.24
乗算命令, 無	11712	2.93	1976	2.19

表 2: Pentium, JDK 1.1.7 での結果 (ミリ秒)

JGNWG は `strictfp` によるペナルティは 2 倍から 4 倍と予想した [1]。実験では、fscale 命令使用、事前ロード無しという条件で 1.43 から 3.36 倍という結果を得た。ただし計測されたペナルティは scaling によるものだけであり、store-reload のものを含んでいない。非 `strictfp` でも store-reload は行われているためである。

元にした JIT コンパイラの最適化能力は高いとは言い難いため、結果は参考値ととらえるのが妥当であろう。しかし JGNWG 予想の妥当性が確認できた。

5 まとめ

浮動小数点演算で環境に依らずに同一の結果を得るための予約語 `strictfp` を Java バイトコードの実行時コンパイラに実装した。`strictfp` の実現手法は Java Grande Forum が推奨するものを採用し、何通りかの実装を行ってそれぞれの演算性能を比較した。

参考文献

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

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

[3] Kazuyuki SHUDO. shuJIT—JIT compiler for Sun JVM/x86. http://www.shudo.net/jit/.

[4] Sun Microsystems, Inc. Differences among IEEE 754 implementations, 1997. http://www.validgh.com/goldberg/addendum.html.

[5] 首藤一幸. strictfp の実装, 1999. http://www.shudo.net/java-grandprix99/strictfp/.