# Java JIT コンパイラの高速化手法

首藤 一幸 †

Java JIT コンパイラには，生成コードの質だけでなく，コンパイル時間，メモリ消費量，Java 仮想マシン仕様への準拠といった，多くの相反する要求が課せられている．我々は，さらに開発コストをも考慮しつつ，研究基盤として有用なものを目指し，JIT コンパイラ shuJIT を開発してきた．本稿では，そこで用いてきたコード生成手法や様々な最適化手法とそれらの効果，またそこで得た知見を紹介する．

# Optimization Techniques for Java JIT Compiler

Kazuyuki Shudo†

## 1 はじめに

Java バイトコードの実行時 ( Just-in-Time , JIT ) コンパイラには，一般の，実行前にコンパイル処理を完了するコンパイラとは異なる様々な要求が課せられている．単に生成するコードの質が良ければよいというものではなく，コンパイルによる性能上の利得がコンパイルによって消費される時間やメモリ消費量に見合ったものでなければ，コンパイルするに値しないのである．また，Java 言語や仮想マシンの仕様 [2][4] は実行結果の高い再現性を狙った厳しいもので，その規定の一部は性能向上の妨げとなる．

相反する多くの要求がある状況では，それぞれの要求に特化した処理系の出現はごく自然なことであろう．例えば Sun 社の HotSpot Server VM はコンパイル時間に糸目を付けず，高い性能を得ることに特化した Java 仮想マシン ( JVM ) および JIT コンパイラである．逆に組み込み向けの処理系では，性能を多少犠牲にしてもフットプリントが小さいことを優先するかもしれない．

我々は，分散処理など他の研究の基盤として用いるべく，*shuJIT* という JIT コンパイラを開発してきた [6]．この開発には，次の方針で臨んだ．

- 研究基盤として利用しやすいものとする．
- 実用的なソフトウェアとする．
- 少ない労力で短期間で開発する．ひとりで，長くとも数ヵ月程度の期間で作る．

この方針に基づいて，相反する多くの要求のバランスをとり，多くの工学的選択を行ってきた．コンパイラの開発は，ソースプログラムのパーサ，中間表現の扱い，果ては最適化までかなりの労力を要する仕事である．そのため，開発においては，性能といった技術的要求にとどまらず，開発に要する労力という人的，工学的な要素もまた重要な指標である．

本稿では，これらの指標を考慮して開発，選択，実装してきたコンパイル手法，最適化手法を報告する．

## 2 shuJIT

shuJIT は我々が開発してきた Java バイトコードの JIT コンパイラである．Intel 社の *IA-32* ，いわゆる *x86* プロセッサを対象とし，OS は Linux と FreeBSD をサポートする．C 言語とアセンブリコードで記述してあり，Sun 社の Java 2 SDK などに含まれる *Classic VM* という JVM と共に動作する．

JIT コンパイラを利用した研究の基盤として利用しやすいものとするために，shuJIT が生成するコードを改変しやすい構造，コード生成手法を採った ( 第 3 章 )．その結果，作者自身が諸研究 [10][7] に利用してきただけでなく，他の研究者によっても研究の材料として利用されてきた [8]．

また，研究基盤としての利用し易さと同時に，日常利用する JIT コンパイラとしての実用性も目指してきた．この試みもおよそ成功し，1998 年 9 月の公開以来，2001 年 3 月までの 2 年半の間に，ソースコードは 7558 回，バイナリは 8476 回のダウンロードがあった．

†産業技術総合研究所　National Institute of Advanced Industrial Science and Technology
<*shudo@ni.aist.go.jp*>　http://www.shudo.net/

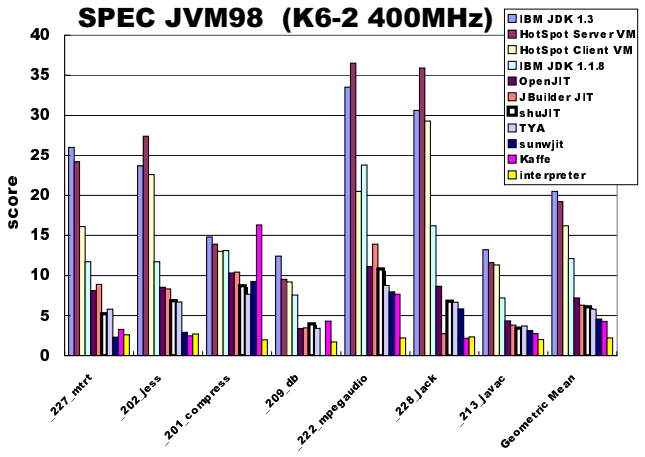


図 1: SPEC JVM98 の結果

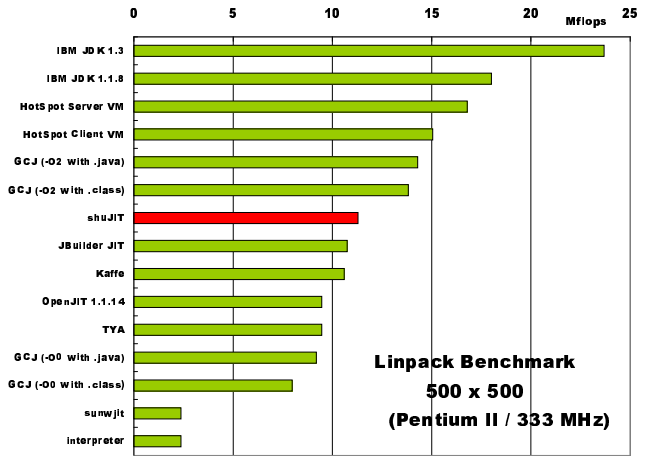


図 2: Linpack Benchmark の結果

性能は図 1 と図 2 に示す通りである．計測には SPEC JVM98 と Linpack Benchmark を用いた．いずれも，高い値が良い結果を表す．

shuJIT の特徴を以下にまとめる．

- コストが低く，かつ研究の基盤として活用し易いコード生成手法を採用（第 3 章）．
- `strictfp` に対応している [7]．
- コンパイル結果のネイティブコードをファイルに保存，再利用できる．このために，実行前に行える処理と，実行が始まって初めて行える処理を峻別している．
- 再コンパイルなしに各種の挙動を制御できる．環境変数 `JAVA_COMPILER_OPT` でオプション，パラメータを指定する．
  - コンパイル結果を保存，再利用するか否か．
  - 生成されたコード量を記録するか否か．
  - メソッドを何度呼ばれた際にコンパイルするか．
  - インライン展開するメソッドの最大コード長．
  - クラスのロード時にメソッドをコンパイルしてしまうか，呼び出された際に初めてコンパイルするか．
  - スレッドのスケジュールを POSIX.1b の `SCHED_FIFO`，`SCHED_RR`，`SCHED_OTHER` のいずれで行うか．

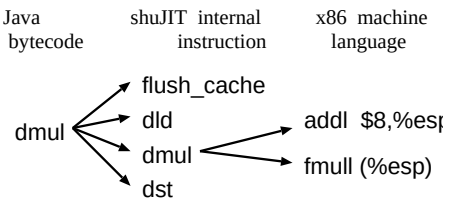


図 3: コード生成手法

## 3 コード生成手法

多くのコンパイラは，ソースプログラムを RTL といった中間表現に変換し，その上で各種最適化を施し，ターゲットプログラムに変換する．しかし shuJIT では，一般的なこの方式を採らなかった．JIT コンパイラを開発する側であらかじめ生成コードの断片を用意しておき，それらをつなぎ合わせることでコード生成を行う，という方式を採用した．これは，次の 2 点を優先した結果の選択である．

- 開発コストの低減
- 生成コードの改変し易さ

具体的には，図 3 に示すように，Java バイトコード命令を shuJIT の内部命令に変換し，それを，用意しておいた x86 ネイティブコード片に置き換える．

これにより，JIT コンパイラ内にアセンブラを持つ必要がなくなり，その分の開発コストを抑えることができた．ネイティブコード片は，JIT コンパイラ自体を C コンパイラでコンパイルしておく際にアセンブルしておくことができるからである．また，このコー

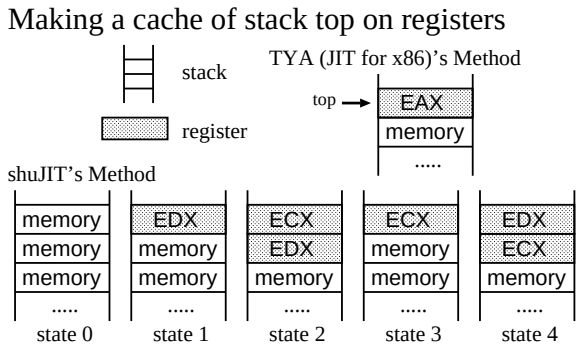


図 4: スタック状態

ド生成方式では，JIT が生成するネイティブコードを JIT 開発者自身が手で記述できるため，生成するコードを思いのままに改変することができる．その結果，生成コードを改変する研究の基盤として利用しやすいものとできた．

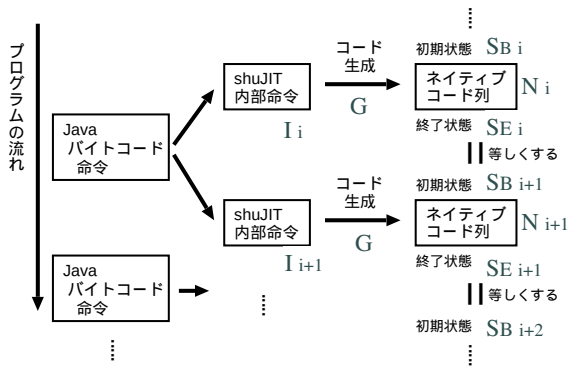


図 5: スタック状態の整合

スタック状態　このコード生成方式の大きな問題は，レジスタ割り付けを行えないことである．近年のコンピュータでは，アクセス遅延が相対的に非常に大きくなってしまったメインメモリをいかにアクセスしないかが高性能達成の鍵である．上述のコード生成手法はネイティブコード片をまたいで複数のレジスタを活用することが困難であり，性能上問題である．この難点を緩和するため，我々はスタック状態（図 4）という考え方を導入した．

JVM のスタックは基本的にメインメモリ上に設けるのだが，スタックトップ付近はレジスタでキャッシュする．そこで，スタックトップやその下をどのレジスタでキャッシュしているのか，という状態をいくつか定義する．shuJIT では，2 つのレジスタをキャッシュに用いることとして，図 4 に示す 5 状態を定義した．それぞれの shuJIT 内部命令について，5 状態に応じたネイティブコード片を用意しておき，ネイティブコード片の実行終了時の状態に応じて，次のコード片を選んでつないでいくのである（図 5）．

この手法により，shuJIT では 2 つのレジスタを，コード片をまたいで活用することが可能となった．2 つとは少なく聞こえるかもしれない．しかし x86 はもともと 8 つの汎用レジスタしか持たない上，スタックポインタとベースポインタも汎用レジスタに含まれる．残りの 4 レジスタも，ひとつは JVM の局所変数のベースアドレスをキャッシュする目的に利用している他，その他のレジスタもネイティブコード片の中では活用している．x86 ではたいていの RISC プロセッサよりレジスタが少ないために，レジスタ割り付けを行うか否かの差が小さいのである．また，Pentium Pro 以降の x86 プロセッサは実際には内部に数十の物理レジスタを持ち，8 つの汎用（論理）レジスタをそれらに割り付けている（レジスタリネーミング）．一般の RISC ではレジスタの活用は完全にコンパイラの仕事だが，x86 ではその多くをプロセッサが担っているのである．

このコード生成手法は x86 に向いている．

# 4　高速化手法

shuJIT は機能面では第 2 章で述べた特徴を持っている．純粋に使用感の向上を求める一利用者の視点から見ると，短いコンパイル時間にも関わらず，インタプリタの数倍という実用的な性能向上が得られることが特徴である．つまり，コストパフォーマンスの良さである．

これらの特徴を維持するためには，最適化を実装するか否か，施すか否かを決定する際に，得られる性能向上だけでなく，コストも考慮しなければならない．コストとは処理に費される時間やメモリを指す．

本章では，shuJIT に実装してきた各種の低コストな最適化手法を紹介する．

## 4.1　生成コード間の直接呼び出し

ひとつの JVM 内には，インタプリタで実行されるバイトコードのままのメソッド，C，C++言語で記述されたネイティブメソッド，すでに JIT コンパイラに

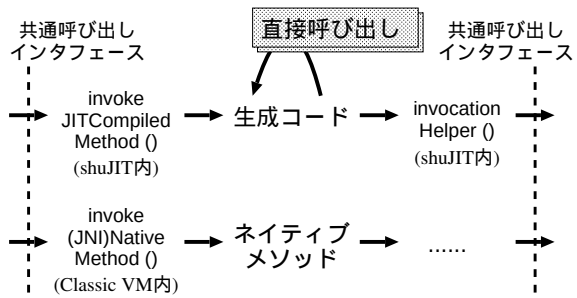


図 6: 直接呼び出し

よってコンパイルされたメソッドなど，多種類のメソッドが混在する．これらはお互いを相互に呼び出すので．そのために共通の呼び出しインタフェース，すなわち引数の型の並びが決められている ( 図 6 )．shuJIT が対象とする Sun の Classic VM では次の通りである．

```
(JHandle *o, struct methodblock *mb,
 int args_size, ExecEnv *ee)
```

図 6 中 `invokeJITCompiledMethod()` は shuJIT が用意している関数で，JIT コンパイラが生成したコードを呼び出すいわゆるラッパ ( wrapper ) である．生成コードの呼び出しインタフェースも各種メソッドの共通インタフェースと同じにしてあるのだが，生成コードと他種のメソッドではスタックの使い方，成長方向が異なるため，引数の積み直しを行うために同関数を用意してある．

つまり，JIT 生成コードどうしの呼び出しでは，`invokeJITCompiledMethod()` の呼び出しは不要であり，これを省くことで関数呼び出しのオーバヘッドを削減することができる．そこで，JIT 生成コードどうしは，ラッパを介さずに直接呼び出すという高速化を施した．

JVM では動的ディスパッチがあって，例えば JIT コンパイルされたメソッドがネイティブメソッドによってオーバライドされることもあり得るため，callee が JIT 生成コードなのかどうかを caller 側で判定しなければならない．この判定コストを削減するため，呼び出し先が事前に一意に決まる場合，つまり static，private または final メソッドである場合には，callee が JIT 生成コードか否かの判定も省くという高速化も施した．この高速化のためには callee が確実に JIT コンパイルできること，何らかの理由でコンパイルに失敗したりしないことを確認しなければならない．callee がまだコンパイルされていない場合，eager にコンパイルしてしまい，確実に JIT コンパイルできることを確認するようにした．

これらの高速化の結果，CaffeineMark 3.0 の Method ベンチマークのスコアが 1812 から 441 に，つまり 4 倍強に向上した．これは，333 MHz の Pentium II と JDK 1.1.8，Linux の組み合わせでの結果である．

おそらく最も多く用いられているベンチマークである SPEC JVM98，特にその一部である_227_mtrt ( マルチスレッド レイトレーシング ) はメソッド呼び出しの性能に敏感なので，その高速化は非常に有効である．

## 4.2 末尾再帰の除去

メソッドの末尾での再帰呼び出しはメソッド先頭へのジャンプに置き換えられることが知られている．この置き換えは末尾再帰除去 ( tail recursion elimination ) と呼ばれ，再帰呼び出しに伴うスタックの消費や，スタックポインタ，ベースポインタの操作をなくせるという利点がある．

末尾再帰を積極的に使う Scheme 言語のプログラムならともかく，JVM 上で実行するプログラムにおいて現実にこの最適化が効奏する局面は少ない．しかし，最適化のコストはとても小さいので，Java プログラムにどの程度末尾再帰が現れるのか調べる目的もあり，直接末尾再帰の除去を shuJIT に実装した．

**実装**　メソッド呼び出しを行うバイトコード命令は数種類ある中で，直接末尾再帰呼び出しを行い得る命令は `invokespecial` と `invokestatic` である．バイトコード命令を shuJIT 内部命令に変換する際に，末尾再帰である場合は `invokerecursive` という内部命令に変換する．これが最終的に，メソッド先頭へのジャンプに変換される．

IBM 社が配布している Java 2 SDK 付属の JIT コンパイラ *JITC* もこの最適化を行う．JITC は devirtualization という手法 [3] を用いて一部の動的呼び出しを静的呼び出しに変換できるため，devirtualize された動的呼び出しに対しても末尾再帰除去を施している．

この最適化はコストこそ小さいものの，実際の効果も大きいものではない．SPEC JVM98 くらいの大きさをもったプログラムであっても，直接末尾再帰を除去できる箇所はせいぜい数箇所であり，ベンチマーク

結果への影響は有意なものではない．

失敗談　末尾再帰除去の最初の実装には不具合があった．当初，直接再帰呼び出しの直後の return を末尾再帰であるとみなしていたが，この単純な判定方法には欠陥があった．aMethod() というメソッド中に次のコードがあった場合を考える．

```
try {  return aMethod()  }
catch (Exception e) {  ...  }
```

try 節中の aMethod() の呼び出しは一見末尾再帰に見えるが，実はそうではない．aMethod() の呼び出しで例外が発生した場合，呼び出しから戻った後，return はせずに catch 節に制御が移る．この恐れがある限り，この呼び出しを末尾再帰としては扱うわけにはいかない．もし aMethod() が例外を throw し得ないなら末尾再帰として扱えるのだが，JVM ではメモリ不足や他スレッドによる強制停止 ( java.lang.Thread#stop() ) によって非同期に例外が発生し得る．つまり，呼び出し後に catch 節に制御が移る可能性のある呼び出しは，末尾再帰としては扱えないのである．

この問題報告を受け，try 節中のメソッド呼び出しを末尾再帰として扱わないように判定方法を修整した．

## 4.3　シグナルの活用

例外の捕捉　他の多くの JIT コンパイラと同様に，以下の通り，shuJIT も OS のシグナルを例外を捕捉するために活用している．

- SIGSEGV で NullPointerException を検出．
- SIGFPE で ArithmeticException を検出．
- SIGSEGV で StackOverflowError を検出．

Classic VM 中で JVM の null は 0 で表現されているので，それをオブジェクトとしてアクセスすると SIGSEGV が発生する．つまり，NullPointerException を検出できる．また，ゼロ除算では SIGFPE が発生し，ArithmeticException を検出できる．StackOverflowError を検出するためには，メソッド呼び出しの直前にわざとスタックの成長方向の少し先，数百から数千番地先をアクセスする．もしアクセス先が有効なアドレスでなければ SIGSEGV が発生し，スタックの溢れを検出できる．

シグナルを用いることで条件分岐なしに上記の例外を検出できる．条件分岐，すなわち正常系のオーバヘッドをゼロにできるのである．

```
void signal_handler(int sig, ...) {
  struct sigcontext *sc = ...
        シグナルコンテクストを取得;

  switch (sig) {
    シグナルの種類に応じた処理（例外の throw など）
  }

  シグナルコンテクスト中の
  プログラムカウンタを書き換える;

  return;
}
```

図 7: シグナルハンドラの処理

シグナルハンドラ中で行う処理を図 7 に示す．まず，例外が発生したスレッドのコンテクスト ( struct sigcontext 型 ) を取得する．続いて，シグナルの種類とコンテクストに基づいて発生した例外の種類を判定し，その例外を throw する．最後に，シグナルハンドラから戻った後で適当な catch 節や return から実行が続くように，コンテクスト中のプログラムカウンタの値を書き換えて，シグナルハンドラを抜ける．

**Code Patching**　shuJIT は例外の捕捉以外の目的にもシグナルを利用する．生成したコードを x86 の INT 3 命令で上書きしておいて SIGTRAP を発生させ，シグナルハンドラ中である必要な処理を行った後，INT 3 命令の上に元の命令を書き戻す．この操作によって，その生成コードが初めて実行されたときのみ，ある必要な処理を行うことができる．これを生成コードの書き換えなしに達成するには条件分岐が必要であり，それによるオーバヘッドは避けられない．生成コードの書き換えについて，詳しくは第 4.4 節で述べる．

## 4.4　生成コードの書き換え

Java 言語仕様 [2] に準拠するためには，何通りかの局面で，あるバイトコード命令を初めて実行したときだけある処理を行う，という必要がある．その処理は，2 度目以降の実行では，行ってはならないか，または，性能のためにはできれば行いたくないものである．

クラスの初期化　static 変数アクセス，static メソッド呼び出し，インスタンス生成がクラスの初期化を

引き起こし得る．

インスタンス生成時の諸チェック クラスが interface や abstract クラスだったら `IllegalAccessError` を throw する．

**interface** 呼び出しの呼び出し先解決結果チェック メソッドが見付からなければ `IncompatibleClassChangeError` を throw する．

JIT コンパイラの実装者にとっては，これらの処理は JIT コンパイル時に行ってしまうという実装が容易なのだが，それでは JVM 仕様には沿えない．アプリケーションプログラムが期待された通りに動かないこともある．

Java 言語仕様第二版は，12.4.1 When Initialization Occurs にて，クラスが初期化されるタイミング，つまり static ブロックの実行と static 変数の初期化が行われるタイミングを厳密に定めている．JIT コンパイラにとって都合の悪いことに，static 変数へのアクセスと static メソッドの呼び出しがクラスの初期化を引き起こし得る．例えば，初期化されていないクラスの static 変数を読み出そうとした時点でそのクラスを初期化しなければならない．

これを素朴に実装してしまうと，static 変数アクセスと static メソッドの呼び出しのごとに対象クラスが初期化済みかどうかを判定する条件分岐を行うことになりかねない．条件分岐のオーバヘッドは極力避けたい．実際には，JIT コンパイルの時点でそのクラスが初期化済みであればこの判定処理を生成する必要はないため，これだけで条件分岐はかなり省ける．

この頻繁な条件分岐を避けつつ，2 度目以降の実行では無駄な処理を行わないためには次の方法がある．

- 初めての実行はインタプリタで行い，2 度目に呼び出された時点で初めて JIT コンパイルを行う．
- 初めての呼び出しで JIT コンパイルを行ってしまう．このときは，必要に応じてクラスの初期化を行えるコードを生成しておく．2 度目の呼び出しで再度コンパイルする．今度は，初期化済みを仮定してコンパイルできる．

しかし，どちらの方法も大きな問題を抱えている．前者の方法では，初めて呼び出されたメソッドの処理が多かった場合，例えば繰り返し回数の非常に多いループがあった場合に問題が起こる．重いメソッドが JIT コンパイルされずに性能に劣るインタプリタで実行され続けてしまう．後者の方法はひとつのメソッドを 2 度コンパイルするので，コンパイル処理に消費される時間が増えてしまう．

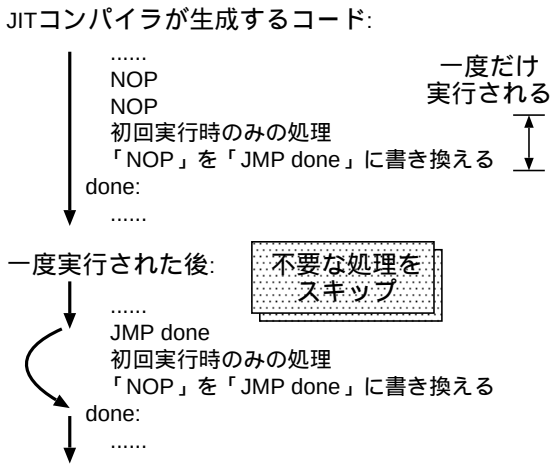


図 8: NOP 命令をジャンプ命令で上書きする方法

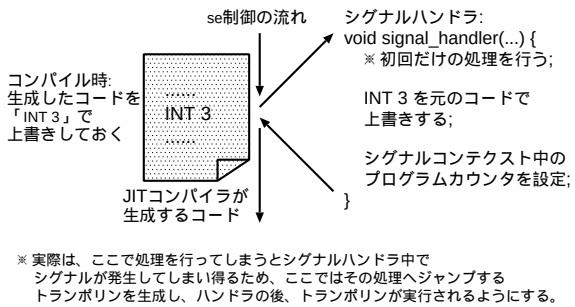


図 9: ソフトウェア割り込み命令を本来の命令で上書きする方法

**shuJIT に実装された 2 つの方法** shuJIT は，コード書き換えを行うことで，上述の問題を被ることなしに JVM 仕様に完全に準拠している．生成されて 1 度実行されたコードを上書きするのである．コード書き換え ( code patching ) の方法として以下の 2 通りを実装してあり，JIT コンパイラ自身を C コンパイラでコンパイルする時点で選択できるようにしてある．

**図 8:** 無操作命令 ( NOP ) をジャンプ命令 ( JMP ) で上書きする方法 ( ジャンプ命令法 )

**図 9:** ソフトウェア割り込み命令 ( INT ) を本来の命令で上書きする方法 ( 割り込み命令法 )

ソフトウェア割り込み命令としては，もともとデバッガのために用意してある唯一の 1 バイト命令である INT 3 ( 0xCC ) を用いた．JVM がマルチプロセッサで実行されることを考えると，複数バイトの書き換えは atomic に行わねばならない．atomicity の保証には最悪の場合メモリバスのロックが必要であり，JIT コンパイラが複雑になると共に性能上のペナルティもある．atomicity を気にせずに済むように，1 バイト命令を採用した．

どちらの方法にもそれぞれの利点がある．ジャンプ命令法では，2 度目以降の実行でも毎回ジャンプ命令を実行せざるを得ないが，割り込み命令法では，2 度目以降は完全に無駄のないコードになるので，数度目以降にはペナルティは皆無となる．一方，メモリ消費の少なさでどちらが勝るかは実装方法に依る．ジャンプ命令法では生成コード中に初回のみの処理が何度もコピーされてしまうため，生成コード量の少なさは割り込み命令法が勝る．しかし，割り込み命令法では，上書きする本来の命令を JIT コンパイルの時点でどこかに保存しておく必要があるため，その表が消費するメモリを考慮しなければならない．最終的にどちらがメモリ消費の少なさで勝るかは，表管理の実装方法やアプリケーションプログラムに依存するので，一概に断ずることはできない．

メモリ消費量　shuJIT ではどちらの方法がメモリ消費量において勝るのか，Linux 上で Java 2 SDK 1.2.2 を用いて調べた．アプリケーションプログラムとしては，ウィンドウをひとつ開いて文字列を表示するだけのごく単純な Java アプレットを用いた．JIT コンパイラが生成したコード量の合計は次の通りで，当然ながら割り込み命令法の方が少なかった．

| | |
|---|---|
| ジャンプ命令法 | 1516 KB |
| 割り込み命令法 | 1408 KB |

しかし，JIT コンパイラが生成するのは直接実行されるコードだけではない．shuJIT は，ネイティブコードと同時に，シグナル発生時にプログラムカウンタの値から例外の種類やバイトコード上のプログラムカウンタを得るための例外表を生成する．shuJIT の割り込み命令法の実装では，この例外表の各要素に，上書きする本来の命令を保存するので，割り込み命令法を用いると例外表のメモリ消費量は大きくなってしまう．生成コードだけでなく，この例外表が消費しているメモリも含めて消費量を計測した結果が次の通りである．

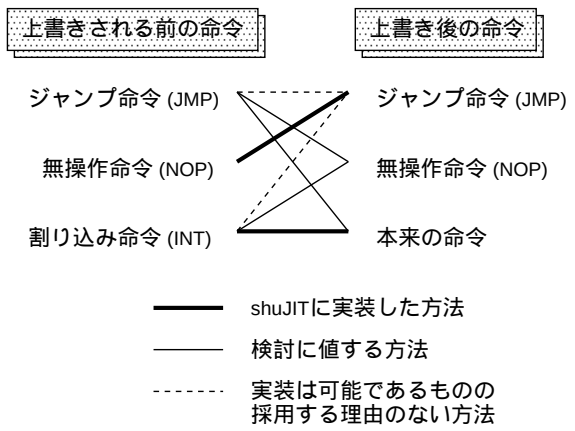


図 10: コード書き換えの方法

| | |
|---|---|
| ジャンプ命令法 | 1682 KB |
| 割り込み命令法 | 1773 KB |

結局，割り込み命令法の方がメモリを多く消費していることがわかった．

考えられる他の方法　shuJIT に実装されているのはこの 2 つの方法だが，コード書き換えの方法は他にも考えられる．書き換え前の命令と書き換え後の命令の組み合せによって，図 10 に示す数通りの方法があり得る．本来の命令での上書きには数度目以降の実行ではペナルティを皆無にできるという利点があり，無操作またはジャンプ命令での上書きには，本来の命令を保存しておく必要がないという利点がある．また，ときに，上書き前の命令として割り込み命令を用いないことの利点もある．例えば，ソフトウェア割り込みやシグナル関連の挙動や API はどうしてもプラットフォーム依存になりがちなので，それらを利用すると JIT コンパイラ自体の保守性や移植性は低下しがちである．

## 4.5　インライン展開

インライン展開 ( inlining ) はメソッド呼び出しコストの削減と手続き内解析および最適化の適用範囲拡大につながるポピュラーな最適化である．shuJIT ではそのコード生成手法の性質上，最適化の機会拡大にはつながらないが，メソッド呼び出しの高速化だけでも恩恵は大きいため，実装している．

callee をコンパイル時に特定できる場合，つまりバイトコード命令の `invokespecial` か `invokestatic` での呼び出しで，ある条件が満たされている場合に，インライン展開を行う．条件は次の通りである．

- synchronized メソッドではない
- catch 節を含まない
- ジャンプ命令を含まない
- 長さが shuJIT 内部命令で *20* 以下
- caller と callee の双方とも `strictfp` ではないか，または，双方とも `strictfp` である

これらをすべて満たすメソッド呼び出しがインライン展開される．あるメソッドに対して展開処理を *2* 回施すので，展開されたメソッド中の呼び出しが展開されるというインライン展開のネストは 2 重までである．

上述の *20*，*2* という値はどの程度インライン展開を行うかというパラメータであり，JIT に対して外から指定できる（第 2 章）．インライン展開は，過度に施してしまうと，そのコード自体がキャッシュから溢れたり，他のコードをキャッシュから追い出すなど，性能低下の原因ともなり得る．20，2 というパラメータは，インライン展開の効果が大きいアクセサ（getter，setter メソッド）が展開されることを目標として決めた経験的な値である．

その他の条件は，インライン展開の処理を軽いものに抑えるためと，実装を複雑にしないために設けたものである．これらの条件が理由でアクセサを展開できないことは少ないため，性能への影響は大きくないだろう．

**特定メソッドのインライン展開**　これまで述べた一般のインライン展開の他にも，shuJIT はある特定のメソッドを特別扱いして既定のコード列への展開を行う，具体的には，`java.lang.Math` クラスの次のメソッドを x86 の浮動小数点演算命令に置き換える．

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

sqrt と sin について，この高速化の効果を調べた．表 1 と表 2 は，Linux 上で Java 2 SDK 1.3.1 を用いて 1.7 GHz の Pentium 4 と 600 MHz の Pentium III で計測した結果である．

| | 高速化なし | 高速化あり | 高速化率 |
|---|---|---|---|
| Pentium 4 | 15535 | 851 | 18.3 倍 |
| Pentium III | 34959 | 1567 | 22.3 倍 |

（ミリ秒）

表 1: sqrt の呼び出し 10,000,000 回に要した時間

| | 高速化なし | 高速化あり | 高速化率 |
|---|---|---|---|
| Pentium 4 | 4242 | 1567 | 2.71 倍 |
| Pentium III | 8021 | 2226 | 3.60 倍 |

（ミリ秒）

表 2: sin の呼び出し 10,000,000 回に要した時間

sin でも数倍，sqrt に至っては 20 倍前後の効果が見られた．この種の高速化はその特定のメソッドを利用するプログラムに対してのみ有効であるが，その効果は大きい．shuJIT ではこの高速化処理が JIT コンパイラに埋め込まれているので，分離や他のメソッドへの拡張は容易には行えない．本来は，最適化処理は OpenJIT 2 [9] の Compilet のようにコンポーネントとして提供され，JIT コンパイラに対してプラグインできることが望ましい．

### 4.6　ピープホール最適化

shuJIT は，picoJava プロセッサ [5] の命令フォールディングに近いピープホール最適化を用いて，メモリアクセスや浮動小数点数のレジスタ，メモリ間コピーを削減している．

Java コンパイラは，JVM のスタックトップを局所変数にコピーするために，局所変数に対してポップする命令と局所変数からプッシュする命令を続けて生成する．操作対象が 32 ビット整数であれば `istore`，`iload` という命令列となる．shuJIT 内部命令は Java バイトコード命令に非常に近いので，この命令列をそのままネイティブコードに変換してしまうと，ポップとプッシュという 2 度のメモリ操作を行うはめになってしまう．スタックトップから局所変数へのコピーを一度行えば済むにも関わらずである．

そこで shuJIT は，こういった簡略化が可能な命令列を検出して，別の，より処理が軽い内部命令に置き換える，というフォールディングを行う．`istore`，`iload` という命令列であれば，これと等価な内部命令 `istld` に置換する．同じ方法で，浮動小数点数のレジスタとメモリ間のコピーも削減し，浮動小数点演算が続く場合には値をレジスタに載せたまま演算を続けられるようにしている．

この高速化は処理コストが低いながら，プログラムによっては非常に効果が大きい．問題サイズ $500 \times 500$ の Linpack ベンチマーク [1] を 600 MHz の Pentium，Java 2 SDK 1.3.1，Linux で実行した場合の結果を次に示す．

| | |
|---|---|
| フォールディングなし | 14.042 Mflops |
| フォールディングあり | 19.083 Mflops |

フォールディングの有無で 1.36 倍の差が開いている．

## 4.7 コードデータベース

実行時にコンパイルを行う JIT コンパイラに対して，実行前にコンパイルを済ませておく Ahead-of-Time (AOT) コンパイラは，コンパイルに費される時間が実行時間に入ってこない点では JIT コンパイラより有利だと言われている．

そこで，JIT コンパイラでも，生成したネイティブコードを保存しておいて，後日 JVM が起動された際にそのコードを再利用するという方法が考えられる．この仕組みが shuJIT に実装してあり，環境変数経由でオプションを指定することで利用できる (第 2 章)．JIT コンパイルすることを決定したメソッドについて，コンパイルに取り掛かる前にコンパイル済みコードが保存されていないかどうか，コードデータベースを検索する．もし見付かったなら，コードデータベースからコードを引き出し，それをコンパイル結果とする．

**リンク処理**　しかし実はコードを引き出しただけでは済まない．JIT コンパイル結果のコード中には，クラスオブジェクトやメソッドのアドレスなど，プログラムを実行するたびに変わり得る値が含まれている．ロード後，これらを解決しなければならない．

- 各種アドレスの解決:
  static 変数，メソッド，C の関数，...
- 実行時になって作られる表への書き込み:
  メソッドの例外表

shuJIT の中では，これら実行のたびに必要な解決処理とそれ以外の静的な処理が峻別してあり，静的な処理が済んだ時点でコードを保存する．コードデータベースからロードしたコードについては，動的な解決処理のみを行う．

この解決処理のコストは見落とされがちである．生成コードを保存，再利用する仕掛けを用意しても，JIT コンパイルのコストは皆無にはならないのである．

また，生成コードの保存には別の難点もある．保存しようとすると，たとえ JIT コンパイル時にあるクラスが初期化済みであっても，それが初期化済みであることを仮定したコンパイルを行えなくなる．生成したコードを再利用する際にそのクラスが未初期化かもしれないからである．クラスが初期化済みであることを仮定できないと，コード生成手法によってはコード生成量や性能上のペナルティを被ることがある (第 4.4 節)．もっとも，shuJIT で割り込み命令法を用いた場合はどちらのペナルティもない．また，コードデータベースから引き出したコードはそのメソッドの 2 度目以降の実行から使うようにすることで，保存，再利用する場合でもあらゆるクラスが初期化済みであることを仮定できる．

## 4.8 その他小手先の高速化

**ループ先頭の整列**　shuJIT はループの先頭をメモリの 16 バイト境界に整列する．Pentium Pro，Pentium II，Pentium III プロセッサは命令を 16 バイトずつ 1 クロックに最大で 3 命令デコードするので，ジャンプ先を 16 バイト境界に整列することで，デコーダの能力を最大限に活用できる．

整列によって命令列にすき間ができる．このすき間は無操作命令 (NOP) で埋めるか，長さが一定以上であればジャンプ命令でスキップする．無操作命令やジャンプ命令にはわずかなオーバヘッドがあり，整列による利得の方が必ず大きいとは限らない．そのため，速くなるとは限らない．

**ジャンプ命令の縮小**　x86 の無条件相対ジャンプ命令には，5 バイト命令と 2 バイト命令の 2 通りの形式がある．

```
E9 XX XX XX XX
EB XX
```

もしジャンプのオフセットが 1 バイトで表現可能ならば，生成コード量とデコーダが扱える命令数の観点から，後者の形式を用いた方が良い．shuJIT があらかじめ用意しているネイティブコード片の中ではジャンプ

命令は常に 5 バイト命令であるが，可能であれば 2 バイト命令に変換する．

**配列の境界チェックの改良** JVM は配列に対する境界を越えたアクセスに対して `ArrayIndexOutOfBoundsException` を throw しなければならない．これはもちろん JIT コンパイラが生成するコードでも同様である．

shuJIT は初め，次のコードに相当するネイティブコードを生成していた．

```
if (index < 0)  throw;
if (index >= length)  throw;
```

この処理には 2 度の条件分岐が含まれている．後に，次の処理で境界チェックを行えることがわかった．

```
if (((unsigned)index - length) < 0)
  throw;
```

インデックスを符号なし整数とみなす，つまり，除算の後，キャリーフラグを検査するのである．これによって，1 度の条件分岐で境界チェックを行える．

# 5 おわりに

本稿では，shuJIT に実装してきた様々な低コスト高速化手法とその効果，また，そこで得た経験を紹介した．

言語処理系は楽しい．shuJIT の設計，様々な最適化の実装，性能測定，JVM 仕様への追従，利用者とのインタラクション，デバッグなど，3 年間に渡ってたっぷりと楽しんできた．本稿がどなたかの知的楽しみの一助となることを願って止まない．

# 謝辞

性能評価の環境を提供して下さった関口様をはじめとする産業技術総合研究所の皆様と早稲田大学の村岡洋一教授，第 4.2 節の失敗談で述べた欠陥を指摘して下さった前田修吾様，第 4.8 節で述べた配列境界チェック法を教示して下さった石崎一明様，shuJIT の動作報告，問題報告を下さった利用者の皆様に感謝致します．

# 参考文献

[1] Jack J. Dongarra and Reed Wade. Linpack benchmark — Java version. http://www.netlib.org/benchmark/linpackjava/.

[2] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *Java Language Specification, Second Edition*. Addison Wesley, 2000.

[3] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, , and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proc. of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, October 2000.

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

[5] Harlan McChan and Mike O'Connor. PicoJava: A direct execution engine for Java bytecode. *COMPUTER*, Vol. 31, No. 10, pp. 22–30, October 1998.

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

[7] Kazuyuki Shudo and Yoichi Muraoka. Efficient implementation of strict floating-point semantics. In *Proc. of 2nd Workshop on Java for High-Performance Computing (in conj. with ICS'00)*, pp. 27–38, May 2000.

[8] 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.

[9] 丸山冬彦. JIT コンパイラ向けアプリケーションフレームワークの設計と実装. Master's thesis, 東京工業大学, February 2001.

[10] 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.

## 質疑応答

敬称は略します．

**Q** 田中 シグナルまわりは OS によっていろいろと異なりがちで，FreeBSD で版による違いで困った経験がある．シグナルで困った経験はないか．

**A** Linux ではシグナルコンテクスト ( `sigcontext` ) を容易に取得できずにフレームを辿ってスタック上を探すというアクロバティックなことをしている．

**Q** 河野 コード書き換えでは命令キャッシュに書き換え前の内容が残るなどの問題が起きがち．何か配慮をしているか．

**A** 一般の RISC プロセッサでは命令キャッシュとパイプラインに書き換え前の内容が残るという問題に対処しなければならないが，x86 はそのあたりは勝手によきに計らってくれる．プログラマはサボれる．

**Q** 和田 コード書き換えの際，NOP 命令をジャンプ命令で上書きしているが，それはジャンプ命令を本来の命令で上書きするべきなのではないだろうか．

**A** 何か理由があっての選択だったと思うのだが，理由が思い浮かばない . . .
( プロシンの時点では第 4.4 節で言うところのジャンプ命令法のみが実装されていた．本来の命令で上書きするには，その内容をどこかに保持しておかねばならないため，それを嫌って NOP をジャンプで上書きする方法を採っていた .)

池田駅

会場の池田町営田園ホール

ワイン城での夕食