計算機アーキテクチャ 120－7
（1996．10．31）

Java™ JIT コンパイラの試作

志村浩也† 木村康則†

Java 言語は，仮想計算機のバイトコードに変換され動作する．インタプリタ方式での実行は遅いため，実行時にバイトコードをネイティブの機械コードにコンパイルする JIT(Just In Time) コンパイル方式が提案されている．米国 Sun 社が提供している JDK-1.0.2 に SPARC V8 のコードに変換を行なう JIT コンパイラを組み込んだ．予備評価結果では，単純なループプログラムの場合はインタプリタと比較して速度で最大 38 倍の性能を得ることができた．しかし，インタプリタより遅くなるプログラムもあり，メソッド呼び出しがボトルネックであることが分かった．生成される機械コードは元のバイトコードの 6 倍程度に収まった．

Experimental Development of Java JIT Compiler

KOUYA SHIMURA† and YASUNORI KIMURA†

The Java language is compiled into the bytecode of the Java virtual machine (VM). This VM is usually implemented by software and thus the execution speed is very slow. One solution is to use a JIT(Just In Time) Compiler that generates native machine code at the execution time. We implemented a JIT Compiler on JDK-1.0.2 that is distributed by Sun Microsystems Inc. Our JIT Compiler generates SPARC V8 code. In the best case, the speedup of a simple loop program is 38 times faster than interpreted code. While the speedups of some other programs become worse. We found that method invocation is a bottleneck. The size of generated machine code is almost 6 times larger than original bytecode and that is permissive.

1. はじめに

Java 言語はオブジェクト指向型の言語であり，プラットホームを選ばないことや，Web ページ上でのプログラミングが行なえるなどの理由により急速に普及しつつある．しかし，インタプリタ形式で実行されるため実行性能は低く，計算パワーが必要とされるアプリケーションを実行するには問題がある．この問題を解決するために，実行時に Java の VM(Virtual Machine:仮想マシン) のバイトコードをネイティブのマシンコードにコンパイルする JIT(Just In Time) コンパイル方式が提案されている．

RISC プロセッサ向きに実装された JIT コンパイラの例は少なく，今回 SPARC Version 8 プロセッサのための JIT コンパイラを試作した．米国 Sun 社が提供している JDK(Java Developers Kit)1.0.2 に組み込む形で開発を行なった．

この報告では，JIT コンパイルの方針，Java の VM のアーキテクチャと最適化手法について説明する．また，SPARC Version 8 プロセッサ向けに実装した JIT コンパイラの性能評価結果を報告する．

† (株) 富士通研究所
FUJITSU LABORATORIES LTD
Java は米国 Sun Microsystems, Inc の登録商標です．

2. JIT コンパイルの方針

JIT コンパイラを試験的に開発するにあたり，次のような方針を立てた．

(1) 命令スケージュリングは行なわない．

基本的に，バイトコードの命令順序を保ったまま機械コードに変換する．実行時に動的にコンパイルするので，コンパイルに時間を要する命令スケジューリング等は行なわない．このような最適化は Java からバイトコードへのコンパイラが行なうべきである．

(2) 大域的最適化は行なわない．

一般に C 言語等では，大域変数をレジスタに割り付ける等の最適化を行なう．Java 言語においても static で宣言されるクラス変数に同様の最適化を行なうことは可能である．しかし，Java 言語はマルチ・スレッドをサポートしており，レジスタを使った最適化を行なうとメモリの変更がなされず，インタプリタとの動作が異なってしまう可能性がある．また，このような最適化は処理に時間がかかることもあるため，大域的な最適化は行なわない．

(3) 繁雑な命令はインライン展開しない．

基本的に，VM の 1 命令をターゲットの数命令にインライン展開することでコンパイルを行なう．しかし，VM の命令の中には展開すると何十命令となるような

繁雑な命令もある．実行速度の観点からすればインライン展開を行なうことが望ましいが，繁雑な命令は生成する機械コードの爆発を防ぐためにインライン展開は行なわない．このような命令には long(64bit) の演算命令やオブジェクトの生成，マルチ・スレッドの排他制御などがあり，展開する代わりにサブルーチンコールで対処する．

(4) JDK のデータ構造を利用する．

評価用に様々な実験ができるように，JDK のインタプリタと変換された機械コードの間で相互の呼び出しを可能にしたい．そのため機械コードが使うデータ構造は JDK のものと合わせる必要がある．また，開発を早めるために JDK のデータ構造を機械コードにおいても使用する．

3. VM のアーキテクチャとコードの最適化

Java VM のアーキテクチャについての簡単な説明と第 2 章で述べた方針に従って生成する SPARC コードの最適化手法について述べる．

3.1 レジスタ割り当て

VM は，次のようなレジスタを持つ．

pc プログラムカウンタ．
optop オペランド・スタックの先頭へのポインタ．
frame 実行中のメソッドの実行環境へのポインタ．
var 実行中のローカル変数領域へのポインタ．

基本的に VM はスタックマシンであり，引数の受け渡しや途中結果を保持するためにレジスタを使用しない．Intel 486 のような汎用的に使えるレジスタの数が少ないマシンでも効率よく実行できるようにしているためである．

しかし，一般の RISC プロセッサでは利用できる汎用レジスタは 30 個以上ある．VM が持つレジスタをネイティブのプロセッサのレジスタに割り付けただけでは，利用しないレジスタが多くあり，これを利用しない手はない．

加えて VM はスタックマシンにも関わらず，プログラムの正当性を確保するためにバイトコードにおいてスタックが動的に延び縮みすることを禁じている．これはベリファイアによってチェックされる．スタックの領域はメソッドが呼び出される時に必要な数だけ確保されるが，メソッド内において，VM のある命令がアクセスするスタックの位置は常に同じである．

従ってスタックをレジスタに静的に割り付けることが可能であり，これを使って高速化を目指すことにした．またローカル変数もレジスタに割り付けることが可能である．そこで，メソッド当りのスタックの消費量と使用するローカル変数の個数の調査を行なった．

3.1.1 スタック消費量と使用ローカル変数の個数

JDK に付属の全てのクラス・ライブラリを調査の対象とした．この中にはシステム・ライブラリや Java のコンパイラである javac，アプレットを実行する appletviewer などのアプリケーション・プログラムも含まれている．クラスの数は総勢 604 個，これに含まれるメソッドの数は，コンパイル対象とならない native 型と abstract 型を除くと全部で 4242 個あった．

これらのメソッドについて，実行時に必要とするスタック消費量と使用するローカル変数の個数を調べた．またスタックとローカル変数の両方をレジスタに割り付けることを考慮して，この 2 つを足したものも測定した．結果を図 1 に示す．横軸は，スタック消費量，使用ローカル変数の個数，およびその和であり，縦軸はその個数を使うメソッドがいくつあったかを表す．

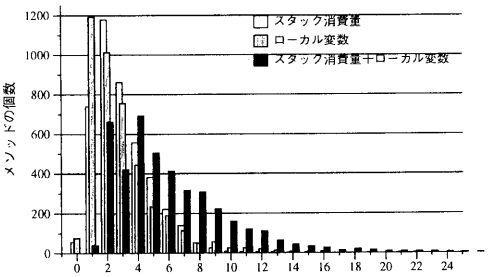

図 1 スタック消費量とローカル変数

スタック消費量と使用ローカル変数の和では 4 つのものが一番多く，最大でも 37 個であった．平均をとると 1 メソッド当りスタック消費量は 3.09 個，使用ローカル変数は 3.01 個，合わせると 6.1 個という結果になった．しかし，システム・ライブラリには数多くの小さなメソッドが含まれている．従って実際に VM が動作しているときに比べると小さな値となっている可能性がある．そこで，Java のコンパイラである javac を実際に動作させ，そのときの動的な個数を測った．結果はスタック消費量は 3.41，使用ローカル変数は 3.13 個，合わせると 6.54 個という結果になった．

次に図 1 の結果を累計して，レジスタの個数によってレジスタに完全に割り付け可能なメソッドの割合を測った．図 2 にその結果を示す．

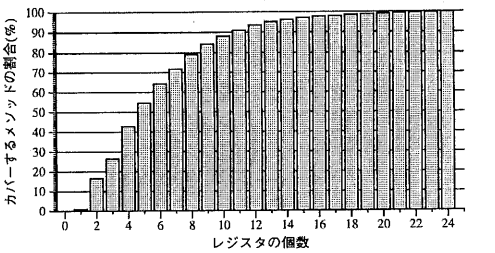

図 2 レジスタの個数とカバーするメソッドの割合

レジスタが20個で99%，30個で99.95%のメソッドが完全にレジスタ割り付け可能になる．一般のRISCプロセッサは30個以上のレジスタを持つので，ほぼ完全にレジスタ割り付けが可能と言える．

3.1.2 レジスタ割り付けと最適化の例

例えば，次のようなバイトコード（Javaで"a=b+c"を行なうコード）を機械コードに変換することを考える．

```
iload_1  // 変数1をスタックに積む
iload_2  // 変数2をスタックに積む
iadd     // スタックの上2つの値を加
            え，スタックに積む
istore_3 // スタック・トップを変数3
            に書き込む
```

スタックをレジスタに割り付けない場合，SPARCコードに変換すると次のようになる．

```
iload_1  -> ld  [vars + 0], %r1
            st  %r1, [optop + 0]
iload_2  -> ld  [vars + 4], %r1
            st  %r1, [optop + 4]
iadd     -> ld  [optop + 0], %r1
            ld  [optop + 4], %r2
            add %r1, %r2, %r3
            st  %r3, [optop+0]
istore_3 -> ld  [optop+0], %r1
            st  %r1, [vars + 8]
```

スタックにレジスタを割り付けると，次のようにコードを小さくし，速くすることができる．

```
iload_1  -> ld  [vars + 0], %r1
iload_2  -> ld  [vars + 4], %r2
iadd     -> add %r1, %r2, %r1
istore_3 -> st  %r1, [vars + 8]
```

更に，ローカル変数をレジスタに割り付けると次のようになる．実行サイクルを多く必要とするロード/ストア系の命令がレジスタ転送命令に置き換えられるため，実行速度が向上する．

```
iload_1  -> mov %l1, %r1
iload_2  -> mov %l2, %r2
iadd     -> add %r1, %r2, %r1
istore_3 -> mov %r1, %l3
```

このように，VMにおいてメモリに置かれるデータをレジスタ上に置くと，メソッド呼び出しの際にレジスタの内容をセーブする必要がある．しかし，SPARCの場合はレジスタウィンドウが扱えるため，このセーブの操作を省くことができ，更に高速化が望める．

以上の最適化は汎用レジスタを多く持つRISC向けの最適化手法である．Intel 486のようなマシンでは，自由に扱えるレジスタが少なくこのような最適化を行なうことは難しい．

3.2 peephole最適化

peephole最適化では，レジスタ割り付けを行なった結果，実行時に冗長な操作となる命令を取り除く．VMはスタックマシンであるため，バイトコードをレジスタを使った機械コードに変換したとき無駄なコードが生成されることが多い．

例えば，定数を演算に使う場合（Javaで"a=b+100"）に1対1に変換すると次のようになる．

```
iload_1  -> mov %l1, %r1
ldc 100  -> mov 100, %r2
iadd     -> add %r1, %r2, %r1
istore_2 -> mov %r1, %l2
```

ldc命令とiadd命令が基本ブロック内にあれば（2つの命令の間に分岐してくる命令がない）とき，iadd命令は必ず100を加える演算を行なうはずである．一般のマシンの命令では即値(Immediate Value)が使えるため，この場合最適化が可能である．SPARCの場合だと即値として13bitの符号付整数（-4096～4095）が扱え，次のように最適化できる．

```
iload_1  -> mov %l1, %r1
ldc 100  -> -----------------
iadd     -> add %r1, 100, %r1
istore_2 -> mov %r1, %l2
```

更に，以上の命令が基本ブロック内であれば，この命令列はローカル変数の1番に100を加えローカル変数の2番に結果を書き込めば良いことが分かる．結局，レジスタ間の転送命令（mov）は不要となり，以下のように最適化できる．

```
iload_1  -> ------------------
ldc 100  -> ------------------
iadd     -> add  %l1, 100, %l2
istore_2 -> ------------------
```

このような最適化は，実行速度の向上に大きく寄与する．ただし，第2章の方針に従って，冗長な命令を取り除くことのみ行ない，バイトコードの実行順序を変更するような命令スケジューリングは行なわない．この程度であれば，最適化に要する時間もあまり必要とせず，その場で（Just In Time）行なっても実行時間にはほとんど影響はないと判断した．

3.3 動的リンク

Javaのバイトコード内では，クラスやインスタンスの変数，メソッドは全てシンボルで参照されている．従って実行の際はシンボルから実際のアドレスを検索する必要がある．検索処理にはクラスのロードの処理などが含まれており，非常に時間を要する．

JDKのVMの実装ではこのシンボル検索のオーバヘッドを無くすために，一度検索を行なったバイトコード中の命令をquick命令と呼ばれる命令に変更（自己修正）する．シンボル検索を必要とする命令は，最初に実行される時は非常に繁雑な検索処理が必要であるが，2回目以降の実行ではこの処理を省略するquick命令に置き換えられており，高速に実行できるようになっている．

JITコンパイラで機械コードに変換するときに，この動的リンクを効率よく行なうことは難しい問題である．対処方法としては次のような方法が考えられる．

(1) 命令毎に実行済みフラグを設ける

動的リンクが必要な命令毎にフラグを設け，実行時にこのフラグを見て実行したことがなければ，動的リンクを行なう機械コードを生成する．問題は，フラグのための余分な記憶領域が必要なこととフラグのチェックのオーバヘッドが大きいことである．

(2) 参照しているクラスをロード

JITコンパイル時にシンボル検索を行なってアドレスを解決する．この方法だと，再帰的にシンボル検索，ロード，コンパイルが繰り返され，全てのクラスがロードされてしまいJust In Timeの意味がない．

(3) 自己修正コードを生成

インタプリタでは命令を変更することによって処理内容を変更することは容易である．しかし，生成した機械コードが処理内容を変更するように自己修正することは困難で，普通は無駄なnop命令や分岐命令の挿入が必要である．

結局，次のように自己修正コードを生成することによって対応した．実行効率を上げるため，シンボル検索処理を呼び出す命令を生成しておき，検索後は判明したアドレスをレジスタにセットする命令に置き換える．具体的には以下のように自己修正する．

```
call resolve -> sethi %o0,%hi(ad)
mov  #5,%o0  -> or %o0,%lo(ad),%o0
```

movは遅延スロットの命令で，#5はシンボル表の5番を参照することを表す．呼び出されたresolveルーチンが，シンボル表を調べリンクを行なった後，呼び出し元の命令を修正する．この方法により，インタプリタがquick命令に置き換えるのと同等の効果を無駄なコードなしに実現することができた．ただし，最初の実行時に命令書き換え後に命令キャッシュをフラッシュする必要があり，実行速度を低下させる要因となる．

4. JITコンパイラの実装

米国Sun社が提供しているJDK(Java Developers Kit)1.0.2にSPARC version 8のマシンコードへ変換を行なうJITコンパイラを組み込んだ．このシステム上でのJavaのソースから実行されるまでの様子を図3に示す．

Javaのソース・プログラムはJavaコンパイラによってJavaのバイトコードに変換される．このときバイトコードはクラス単位に分割される．実行時にバイトコードはクラス・ローダによってファイル・システムまたはネットワークから必要に応じてロードされ，バイトコード・ベリファイアによって正当性が検証される．その後，インタプリタがバイトコードを逐次解釈・実行する．

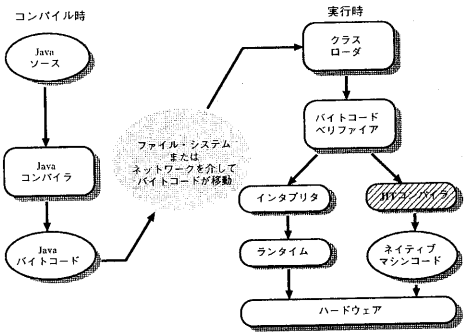

図3 処理の流れ

JITコンパイラはバイトコード・ベリファイアを通過したバイトコードに対して働く．現在の実装では組み込みを簡単にするため，ベリファイアの直後にJITコンパイラを置いた．JITコンパイルの単位はクラスである．JDKの動的リンク機構により，動作中に必要なクラスがロードされるときに，そのクラスに含まれるメソッド全てのコンパイルを行なう．性能の観点からすると，一度も実行されないメソッドもコンパイルされるので無駄がある．最終的には実行直前にメソッド単位にJITコンパイルする方式に変更する予定である．

コンパイルされた機械コードはメモリ中に保存され，一度コンパイルされた機械コードは再利用される．

4.1 選択的コンパイル

デフォルトでは全てのメソッドがコンパイル対象となる．しかし，起動するときのオプションによって，どのメソッドを機械コードに変換するかユーザが指定できる．この指定により，インタプリタで実行されるメソッドとハードウェアで実行されるメソッドが混在できる．またJavaシステムではクラスがパッケージ化されているが，クラス単位やパッケージ単位での指定もできる．

JITコンパイラにより変換を行うと，インタプリタと比較して次の点で不利になる．

- コンパイルに時間を要する．
  一度しか実行されないようなメソッドの場合，コンパイルした方が遅くなる可能性がある．
- メモリ領域を消費する．
  Javaのバイトコードを機械コードに変換するとその大きさは元の数倍の大きさとなる．

従って，全てのメソッドをJITコンパイラで変換するのは得策とは言えない．頻繁に使われる，または処理がループしており実行時間のかかるようなクラスについてのみJITコンパイラを用いるのが望ましい．しかし，このような判断を自動で行なうのは難しいため，現在はユーザに任せている．

4.2 最適化レベルの指定

JIT コンパイラの最適化のレベルとして次の 3 つを用意した．このレベルは java を起動するときのオプションとして与えることができる．

O0 レジスタ割り付け，最適化を全く行なわない．

O1 スタックとローカル変数をレジスタに割り付ける．

O2 O1 に加え peephole 最適化を行なう．

4.3 問 題 点

現在は試作段階であり，まだ次のような問題を残している．

(1) 浮動小数点

浮動小数点の変数，スタック操作，演算についてはレジスタの割り付け，最適化ともに行なっていない．

(2) GC(Garbage Collection)

JDK の GC はマーク＆スィープ方式である．このため，スタックやローカル変数上に存在するオブジェクトへのポインタがレジスタに割り付けられた場合，マークが行なわれず必要なオブジェクトまで回収されてしまう．GC が起きたときの動作は保証できない．

(3) 例　外

メソッドの呼び出しに C 言語と同じインタフェースを使用しており，C 言語のスタックとレジスタウィンドウを例外ハンドラのレベルに戻すのが難しい．setjmp, longjmp のシステムコールを使って実現可能だが，オーバヘッドが大きいので，もっと良い解決策を考案中である．

5. 性 能 評 価

最適化レベルを変えて実行速度，生成コードサイズ，JIT コンパイル速度について性能評価を行なった．測定を行なったマシンは，SparcStation 20(SuperSPARC 75MHz，外部キャッシュ1MB)，OS は Solaris2.5 である．ここではその結果と考察を示す．

5.1 CaffeinMark による性能評価

性能測定には Java のベンチマークとして有名な CaffeinMark(2.01a)☆を使用した．まだ GC に対応できていないこともあり，CaffeinMark のクラスのうち，BenchMark と StopWatch クラスだけを JIT コンパイルして計測した．また，CaffeinMark は 9 個のベンチマークからなるが，Image, Graphics, Dialog の 3 つはグラフィックスやウィンドウシステムの処理性能に依存することもあり省略している．測定した結果を図 4 に示す．インタプリタ (JIT なし) のときの値を 1 として，JIT コンパイラの最適化レベルを変えたときに CaffeinMark の値が何倍になったかを表している．なお，JIT コンパイラは時間の計測前に働いており，コンパイルに要した時間はこの結果には含まれていない．

Loop において，O2 のとき最大で 38 倍の性能を得ることができた．Sieve や Logic で O2 の伸びが小さいのは，基本ブロックが小さく peephole 最適化があまりできなかったためと考えられる．String の性能向上が見られないのは，システムクラスを呼び出しており，ほとんどインタプリタで実行されているためである．Method はインタプリタより性能が劣る結果となった．この原因は Method が再帰呼び出しを行なっており，SPARC のレジスタウィンドウの overflow/underflow が原因と考えられる．

☆ http://www.webfayre.com から入手可能

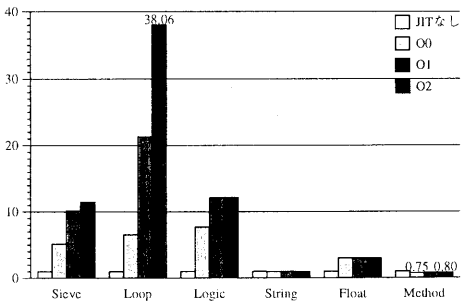

図 4 CaffeinMark による実行速度向上比

5.2 javac による性能評価

CaffeinMark のようなベンチマークはコードサイズも小さく，またオブジェクト指向の特徴である仮想関数の呼び出しやオブジェクト変数のアクセスも頻繁に行なわれていないので，実際のアプリケーションにおける性能を如実に表していると言えない．そこで Java のコンパイラ (javac) が自分自身 (ソースファイルの数は 110 個，全部で 18076 行) をコンパイルする時の CPU 時間を測定した．結果を図 5 に示す．O2 のときでも 1.9 倍程度しか速くならなかった．この時間には JIT コンパイルの時間も含まれているが，5.5 節で述べるようにその影響はほとんどない．CaffeinMark の結果から推測すると，原因はメソッド呼び出しやオブジェクト変数のアクセスにあると考えられる．

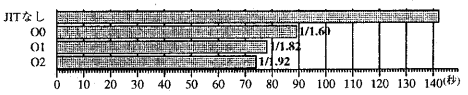

図 5 javac による性能評価

5.3 オブジェクト指向関連命令の処理時間

前節での javac の性能があまり上がらない原因が，メソッド呼び出しやオブジェクト変数のアクセス等の命令であることを検証するため，インスタンスとクラスについて，

- メソッド呼び出し
  引数なしで何も行なわないメソッドの呼び出し．
- オブジェクト変数のアクセス
  変数の値を 1 増やす (読出，加算，書込)．

の処理を一回行なうのにかかる時間を計測した．結果

を図6に示す．インタプリタとO2を比較すると，インスタンス変数のアクセスでは約16倍，クラス変数の場合は約13倍とまずまずの性能である．しかしメソッド呼び出しでは1.8倍程度にしかならず，これがネックとなっていることが分かった．

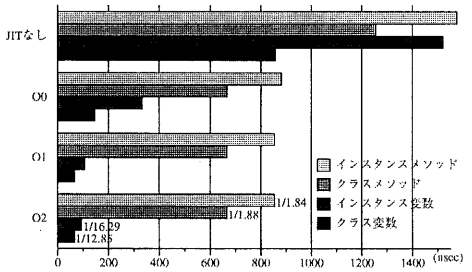

図6 オブジェクト指向命令の処理時間

更に現在のJITコンパイラ実装ではインスタンスメソッドの呼び出しにSPARCのレジスタウィンドウを3枚消費する．一方，インタプリタの場合，メソッド呼び出しはレジスタウィンドウを消費しない．この影響を調査した結果，測定したマシンでは，メソッド呼び出しを3段行なうとレジスタウィンドウのオーバフローを起こし，その処理に1.38 μ秒を要することが分かった．これに呼び出し時間を加えると2.23 μ秒かかることになり，インタプリタでの呼び出し時間が1.58 μ秒であるので，コンパイルした方が約1.4倍遅くなる．これは性能に深刻な影響を与える．

5.4 コードサイズ

Javaがバイトコードを選んだ理由の一つにコードを小さくすることがあり，JITコンパイラが生成するコードのサイズは一つの評価基準となる．そこで，5.2節と同じ条件でjavacについて生成されるコードのサイズを調べた．表1にバイトコードのサイズと生成されたSPARCのコードサイズを示す．その結果，O2では6倍弱になった．命令数で比較するとO2のとき3倍以下である．JITコンパイラの生成するコードはメソッド呼び出しの引数の受渡しにメモリを使っており，受渡し操作の命令増加を予想していたが，意外に小さく収めることができた．

表1 コードサイズの比較

	コードサイズ	命令数
バイトコード	75KB (1.0)	38841 (1.0)
O0	773KB(10.30)	193197(4.97)
O1	522KB (6.96)	130529(3.36)
O2	459KB (6.11)	114680(2.95)

5.5 JITコンパイル速度

5.2節と同様の条件でjavacのコンパイル時間を計測した．OSの計時精度(10msec単位)では正確な時間を測定できなかったので，JITコンパイル時に1000回同じことを繰り返すようにして計測した．この時間はキャッシュ等の影響を考慮すると実際の値よりも少なめと考えられる．結果を表2に示す．

表2 コンパイル時間（単位は秒）

	実行時間	コンパイル時間	毎回コンパイル
O0	89	0.216(0.24%)	3.86(4.16%)
O1	78	0.260(0.33%)	4.69(5.67%)
O2	74	0.369(0.49%)	6.53(8.11%)

全実行時間中にJITコンパイル処理が占める割合が一つの評価基準となる．当然ではあるが，最適化を行なう程実行時間が短くなり，コンパイル時間が増すので，コンパイルに要する時間の比率が大きくなる．どこまで最適化を行なうかは，どれだけコードが再利用されるかにかかっている．この場合はO2でも0.5%以下であり，コンパイル時間は無視できる程度と言える．

しかし，メソッドやループによって繰り返し同じコードが使われる程この割合は小さくなる．そこで最悪の場合として，メソッドが呼び出される直前にそのメソッドを毎回JITコンパイルした場合に，JITコンパイルにかかる時間を計測した．表中の毎回コンパイルはこの時間を表す．実行時間にこの時間を加えた場合でもO2が最小となった．これは一度しか実行されないコードばかりの条件下でもO2が最高性能を引き出せることを意味する．現在の実装ではまだ最適化の余地は残されていると言える．

6. おわりに

JDKにSPARC用JITコンパイラを組み込み，その性能評価を行なった．単純なループプログラムの場合にインタプリタと比較して最大38倍速度が向上した．しかし，javacのようなプログラムでは1.9倍程度の速度にしかならず，メソッド呼び出しがボトルネックとなっていることが分かった．生成されるSPARCのコードはバイトコードの6倍程度に収めることができた．JITコンパイルの速度は全実行時間の0.5%以内と高速である．

今後の課題として，GCや例外処理への対応，浮動小数点処理の最適化，メソッド呼び出しの高速化などを行なう予定である．

参考文献

1) Sun Microsystems, Inc: The Java Virtual Machine Specification, ftp://javasoft.com/docs/vmspec.ps.Z
2) Lemay L, Perkins, C. L: Teach Yourself Java in 21 days, Sams.net(1996) (武舎博之，久野禎子，久野靖訳：Java言語入門— アプレット,AWT, 先進的機能，株式会社トッパン (1996))