論　文

Java Just-in-Time コンパイラのためのコスト効率の良いコンパイル手法

首藤　一幸[†]　　関口　智嗣[†]　　村岡　洋一[††]

Cost-Effective Compilation Techniques for Java Just-in-Time Compilers

Kazuyuki SHUDO[†], Satoshi SEKIGUCHI[†], and Yoichi MURAOKA[††]

あらまし　Java Just-in-Time コンパイラには，生成コードの性能だけでなく，コンパイル時間の短さや Java 仮想マシン仕様への準拠といった相反する要求が課せられている．我々は，研究基盤として有用なものとすることを第 1 の目標として Just-in-Time コンパイラを開発してきた．コンパイル処理と開発のコストを抑えつつもベースラインコンパイラとして実用的な性能を達成した．本論文では，スタックキャッシュを応用したコード生成手法，命令畳込み，シグナルを利用した例外の捕そく法，性能上のペナルティなしに Java の仕様を厳密に守るためのコード書換え手法など，当該コンパイラに実装した最適化手法を述べ，その有効性を示す．

キーワード　実行時コンパイル，Java 仮想マシン，スタックキャッシュ，命令畳込み，コード書換え

1. まえがき

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（JVM）の仕様 [1], [2] は実行結果の高い再現性をねらった厳しいもので，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 がもつ JIT コンパイラは，Hot であると認識され選択されたメソッドについてはコンパイル時間に糸目をつけない，高い性能を得ることに特化した JIT コンパイラである．逆に組込み用途向けの処理系では，性能を多少犠牲にしてもメモリ消費量が少ないことを優先するかもしれない．

我々の場合は，次の方針に基づいて JIT コンパイラを開発してきた．

（1）　研究基盤として利用しやすいものとする．

（2）　開発コストを抑える．

（3）　実用に耐える品質と性能を達成し，多くの利用者を獲得する．

コンパイラの開発は，パーサ，中間表現の扱い，種々の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，性能に代表される技術的な要求にとどまらず，開発に要する労力という人的，工学的な要因もまた重要な指標となる．我々は，数人月程度の労力で，研究基盤として扱いやすいソフトウェアを作成することを第 1 の目標とした．つまり，JIT コンパイラ自体の開発コストを抑制しつつ，それを研究に利用する際のコストも低く抑えることをねらった．

コンパイラが生成するコードのピーク性能を追求すると，開発コストは性能以上に高くなってしまうことが一般的である．そのため我々は，ピーク性能ではなく，JIT コンパイラのもう一方の形態である，コンパイル処理のコストを抑えたベースラインコンパイラを開発するという方針をとった．

本論文では，当該 JIT コンパイラ向けに開発，実

† 産業技術総合研究所，つくば市
National Institute of Advanced Industrial Science and Technology, Tsukuba Central 2, 1-1-1 Umezono, Tsukuba-shi, 305-8568 Japan
†† 早稲田大学理工学部，東京都
Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan

装してきたコスト効率の良いコード生成手法と最適化手法，及びその効果を報告する．コード生成手法としては，あらかじめ用意したネイティブコード片をつないでいくというテンプレート方式を採用し，その上で，複数レジスタを活用するためにスタックキャッシュ（stack caching）[3] を応用した．スタックトップの値のみを常にレジスタに保持する JIT コンパイラ [4] や，動的スタックキャッシュを採用した Java バイトコードインタプリタ（Sun 社 Classic VM）はあるが，スタックキャッシュ，つまり複数のキャッシュ状態を JIT コンパイラに適用した例はない．この方式によって，開発コストとコンパイル処理のコストを低く抑えながらもテンプレートにまたがって複数のレジスタを利用できた．また，本コード生成方式は，**2.** で述べるように，開発者側で JIT コンパイラが生成するコードを直接的に改変できるため，研究の材料として利用しやすい．

最適化としては，命令畳込み，インライン展開といったよく知られているものに加えて，生成コード間の直接呼出しやシグナルの利用，コード書換えといった処理コストの低いものを実装し，本コンパイラのコード生成方式と組み合わせた場合の効果を調べた．

続く **2.** では開発した JIT コンパイラの概要を述べる．コンパイラの構成や，また，採用したコード生成手法が開発コスト，研究基盤としての利用しやすさ，コンパイル処理のコストに与える影響を論じる．**3.** では，より多くのレジスタを活用するために導入したスタックキャッシュの得失を述べる．**4.** で，各最適化手法について，その性能上の効果を述べた後，**5.** でピーク性能とアプリケーションの起動に要する時間という使用感を大きく左右する要因を評価し，**6.** でむすぶ．

2. JIT コンパイラの概要

我々は，shuJIT という Java バイトコードの実行時（Just-in-Time）コンパイラを開発，配布してきた．Intel 社の IA-32，いわゆる x86 プロセッサを対象とし，OS として Linux と FreeBSD，NetBSD をサポートする．生成コードのテンプレートがアセンブリコードで記述してある他は主に C 言語で記述されている．Sun 社の Java 2 Platform, Standard Edition（Java 2 SE）や Java Development Kit に含まれる Classic VM という Java 仮想マシン（JVM）とともに動作する．対象が x86，Java 2 SE，Linux や FreeBSD，NetBSD であることに表れているように，shuJIT は

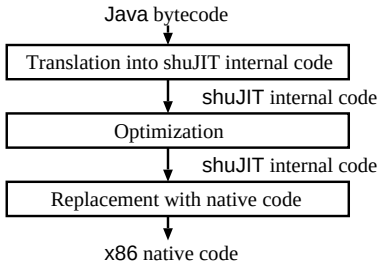


図 1　shuJIT の全体構成
Fig. 1　Structure of shuJIT.

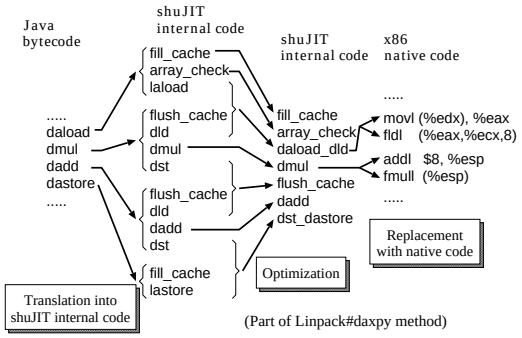


図 2　コンパイルの例
Fig. 2　An example of compilation.

PC 以上の性能，メモリ容量を想定している．

本 JIT コンパイラの開発においては，研究基盤としての利用しやすさだけでなく，日常の使用に耐える実用性や安定性，信頼性の達成を目標にすえた．また，JVM 仕様 [2] への厳密な準拠も指向してきた．これらの要件は，研究目的のソフトウェアであっても，それを利用した派生研究の現実性，例えば評価結果がどのくらい現実に則したものになっているか，を高くするために重要なものである．その結果，一定の利用者を得ることに成功し [5]，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，バイナリーは 8476 回のダウンロードがあった．

コンパイラの構成を図 1 に示す．コンパイル対象のメソッドを与えられると，まず，その Java バイトコード命令列を shuJIT の内部命令に変換する．続いて，得られた内部命令列に対して，**4.** で述べるいくつかの最適化，具体的には，命令畳込み（**4. 2**），インライン展開（**4. 5** 及び **4. 6**），直接呼出しへの変換（**4. 1**）を施す．最後に，内部命令列を x86 プロセッサのネイティブコードに置き換える．図 2 に，Java バイトコー

ド命令からネイティブコードが得られるまでの変換例を示す．

本コンパイラの中間言語である shuJIT 内部命令は，Java バイトコード命令に独自の命令を追加したものである．Java バイトコード命令からの変換処理は，1 対 1，または 1 対多の単純な置換えで済む．文脈によって，例えば `strictfp` [1], [6] であるか否かによって変換結果が変わることはある．

最後のコード生成，つまり shuJIT 内部命令から x86 ネイティブコードへの変換は，内部命令をネイティブコードの断片，つまりテンプレートに置き換えていくという方法で行う．テンプレートはあらかじめ JIT コンパイラ側で用意しておく．テンプレートとして記述されるネイティブコードは，JVM のスタックを，x86 プロセッサ自身のスタック操作機構，すなわち `push`, `pop` 命令を用いてシミュレートする．

このコード生成方式は，次の効果をねらったものである．

- 開発コストの低減
- JIT コンパイラが生成するコードの改変しやすさ
- コンパイル処理のコスト抑制

このテンプレート方式によって，JIT コンパイラ内にアセンブラをもつ必要がなくなり，その分の開発コストを抑えることができた．アセンブラが不要であるのは，JIT コンパイラ自体を C コンパイラでコンパイルする際に，テンプレートもアセンブルしておくことができるからである．x86 は命令のビットパターンがいわゆる RISC プロセッサとは違って規則的ではないため，アセンブラの開発コストが比較的高い．

コンパイラの実装は 1 人で行い，開発開始から 48 日後には簡単なプログラムを動作させるに至った．他のソフトウェアとの間で開発コストを比較することは困難であるが，JIT コンパイラとしてはかなり低い開発コストであろう．

このコード生成方式では，テンプレートに記述されたネイティブコードの列が JIT の生成コード中にそのまま現れるため，テンプレートに変更を加えて JIT コンパイラ自体を C コンパイラで再コンパイルすることで，生成されるコードを直接的に改変できる．その結果，生成コードの改変が必要な研究の材料として利用しやすいものとなり，各種研究の基盤として利用されるに至った [7]〜[9]．これに対して，多くのコンパイラは RTL といったプロセッサ命令に近い，より細かい粒度の中間表現を経由してコード生成を行う．こういったコンパイラで生成コードを改変するためには，中間表現の生成部を改変するといった直接的ではない方法をとらざるを得ない．更に，場合によっては，最適化によって生成したいコードが消えたり順序が崩れたりしないように配慮をする必要がある（ある種の最適化を止めたり，偽の依存関係を作っておくなど）．一方，本コード生成方式には，テンプレートが機種依存するため命令セットアーキテクチャ（ISA）ごとに用意する必要があり，テンプレートに施した変更は特定の ISA でしか機能しないという難点がある．例えば本コンパイラを SPARC プロセッサに移植し，x86 用テンプレートに変更を加えた場合，その変更は x86 上でしか機能しない．

本コンパイラが対象とする ISA は現在 x86 のみである．大きな労力を費やすことなく研究の材料を作成することが目標であったため（**1.** 参照），JIT コンパイラ自体の移植性よりも開発コストの抑制を優先した結果である．これをもし，他の ISA に移植しようとした場合，その ISA 向けにテンプレートを記述し直す必要がある．一方，機種非依存の中間表現をもとにコード生成，アセンブルを行うという一般的なコンパイラの場合は，対象 ISA 向けのアセンブラの作成と，対象 ISA 向けのパラメータ（レジスタ数など）調整などが必要となろう．どちらの手間が大きいかは場合によるが，本コンパイラのテンプレート方式の方が，コンパイラ中で異機種間で共通して使えるコードの割合は低いことは確かであろう．

テンプレート方式では，コード生成処理のコストが低いことも期待できる．もとのバイトコード命令列の長さに比例した時間で行える．本コンパイラの開発にあたっては，ベースラインコンパイラとしての価値を損なわないようにコンパイル処理を軽く保つため，コード長 n として計算量 $O(n)$ を超える処理は行わないようにした．

3. スタックキャッシュ

テンプレートを用いたコード生成方式の大きな問題は，レジスタ割付けを行いにくいことである．近年のプロセッサでは，レジスタの有効活用が高性能達成のかぎとなっているが，テンプレート方式では，テンプレートをまたいで複数のレジスタを活用することが困難である．現に，本コンパイラと同様にテンプレート方式を採っている JIT コンパイラ TYA [4] では，テン

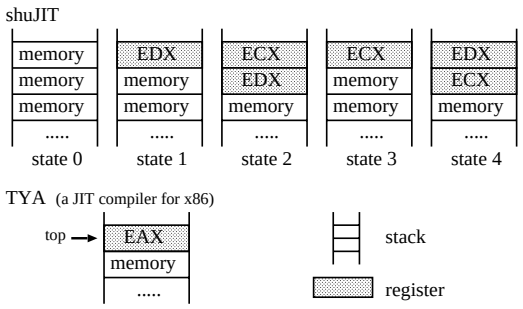


図 3　キャッシュ状態
Fig. 3　Cache states.

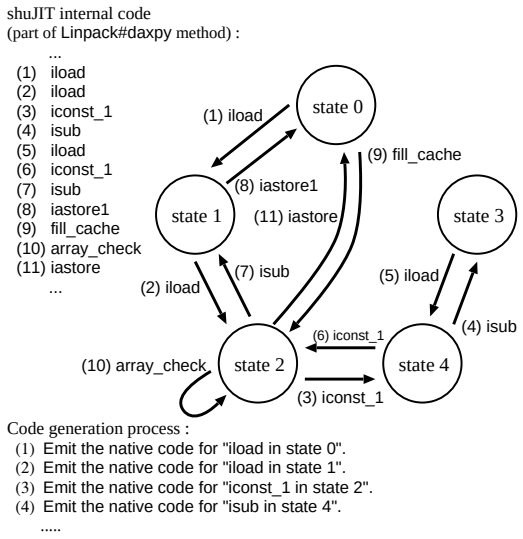


図 4　状態遷移の例
Fig. 4　An example of state transition.

プレートをまたいでレジスタに保持できるのはスタックトップの値のみである（図 3）.

この問題を緩和するため，スタックキャッシュ（Stack Caching）[3] を採用した．JVM のスタックは基本的には主記憶上に設けるが，スタックトップ付近はレジスタを用いてキャッシュする．図 3 のとおり，キャッシュに用いるレジスタ数は 2 とし，スタックトップ付近をどのレジスタでキャッシュしているのかというキャッシュ状態を 5 通り定義した．それぞれの shuJIT 内部命令について，5 状態に応じたテンプレート，すなわちネイティブコード片を用意しておき，各テンプレートの実行終了時の状態に応じて，次のテンプレートを選んでいく（図 4）.

動的スタックキャッシュはインタプリタによく用いられ，JVM でも，Sun 社 Classic VM のインタプリタの 3 状態という実装例がある．しかし，スタックキャッシュを Java JIT コンパイラに適用した例はない．

本コンパイラのコード生成手法に関連する手法として，lazy code selection [10] がある．両手法の共通点は，Java バイトコード命令を対応するネイティブコードに直接的に置き換えるというコード生成方法であり，これによってコンパイル処理を軽く保とうとしている点である．lazy code selection では，コード生成中，三つのスクラッチレジスタで JVM のスタックのどの位置の値をキャッシュしているかを mimic stack を使って追跡する．mimic stack は JVM スタックをシミュレートし，スタック上の要素がどこ（レジスタ，メモリ）に置かれているのかという情報を保持する．これは，本コンパイラがスタック状態を追跡することと類似している．また，lazy code selection では RISC と比較して多様である x86 のアドレッシングモードを利用して，レジスタへのロード命令と演算命令を 1 命令に畳込むことを試みる．本コンパイラでは，この畳込みはテンプレートを記述する時点で人間が行う．両手法の違いは，lazy code selection が局所変数へのレジスタ割付けを行う点や，スタックのキャッシュに用いるレジスタ数である．これらの違いによって，生成されるコードの質は lazy code selection の方がよいと予想できる．また，mimic stack を操作してアセンブルを行う lazy code selection よりも，状態番号に対応したコードを選ぶだけの本手法の方が，より処理が軽く開発コストが低いと予想する．

スタックキャッシュを用いたコード生成と単純なテンプレート方式の双方を実装している JIT コンパイラがないため，スタックキャッシュの効果を正確に測定することはできない．しかし少なくとも，SPEC JVM98 の全ベンチマークにおいて本コンパイラのスコアは TYA のものを上回っている（**5.1** 参照）．TYA は本コンパイラと同様にテンプレート方式を採用しており，JIT 生成コードの実行中は常に EAX レジスタでスタックトップの値のみをキャッシュする（図 3）.

スタックキャッシュの導入によって性能向上が期待できる反面，JIT コンパイラの開発コストは増加する．状態数に応じたテンプレートを用意しなければならないためである．しかし，本コンパイラのようにキャッシュ状態を 5 通り定義したからといって，テンプレート記述の手間は 5 倍までは増えない．各状態に対応

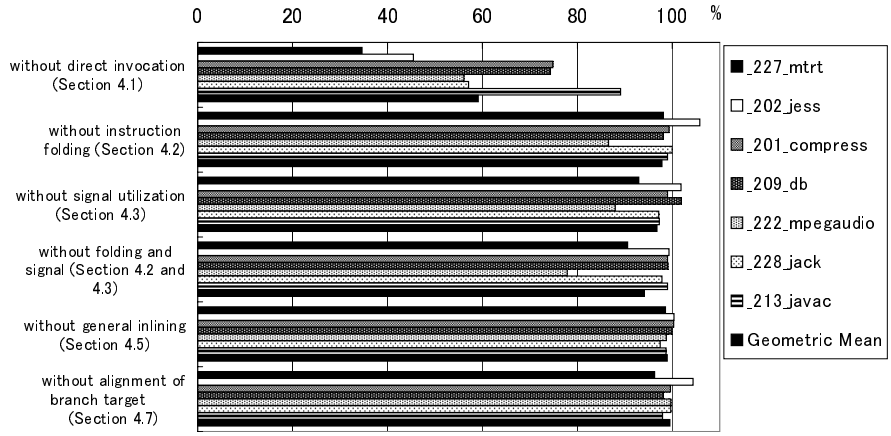


図 5　各最適化の SPEC JVM98 への影響
Fig. 5　The impacts of each optimization on SPEC JVM98 scores.

するテンプレートは，レジスタ名が変わるくらいで，処理内容はほぼ同じだからである．それでも，テンプレートのコード量，ひいては JIT コンパイラ自体のディスク占有量，メモリ占有量が増加することは避けられない．とはいえ，本コンパイラのテンプレートは，アセンブル後のオブジェクトファイルの状態でたかだか 30KByte 弱であり，PC 上で問題となるサイズではない．

スタックキャッシュによって，テンプレートを用いるコード生成でも，テンプレートをまたいで複数のレジスタを活用することが可能となった．2 とは決して多い数ではないが，x86 はもともと汎用レジスタを 8 しかもたない上，スタックポインタとベースポインタもこの汎用レジスタに含まれている．その他の 4 レジスタも，一つは 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おり，残り 3 レジスタもテンプレートの中では活用している．

4.　最適化手法

本コンパイラは，命令畳込み，末尾再帰除去，インライン展開といったよく知られた最適化に加えて，生成コード間の直接呼出し，OS のシグナルの利用，コード書換えといった処理コストの低い最適化手法をいくつか実装している．すべての処理はコード長 n として最大でも計算量 $O(n)$ である．

よく知られた手法であっても，適用対象が JVM であるために特に有効であったり，または，本コンパイラのコード生成方式を前提として高い効果を発揮するという場合がある．例えば，シグナルの利用は JVM 仕様を遵守するために有効であるし，命令畳込みは JVM がスタックマシンだからこそ，テンプレートを用いたコード生成方式に適用すると効果が高い．逆に，インライン展開のように，テンプレート方式ではその効果が最大限には発揮されない最適化手法もある．

本章では，本コンパイラに実装した各最適化手法について，その効果や，コード生成方式との関係，JVM 仕様との関係，メモリ消費量への影響を論じる．

各最適化が SPEC JVM98 のスコアに与える影響を図 5 に示す．これは，すべての最適化を施した場合のスコアを 100%として，ある最適化を行わなかった場合のスコアをそれぞれのベンチマークについて示したものである．値が小さいほど，その最適化を外した場合のスコア低下が大きいことを示す．つまり，最適化の効果が大きいということになる．

4.1　生成コード間の直接呼出し

一つの 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，既に JIT コンパイラに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．これら異なる種類のメソッドが相互に呼び出し合えるようにするためには，共通の呼出しインタフェース，すなわち引き数と返り値についての型の並びを規定するという方法が一般的である（図 6）．このインタフェースは，本コンパイラが対象とする Classic VM では次のように定められている．

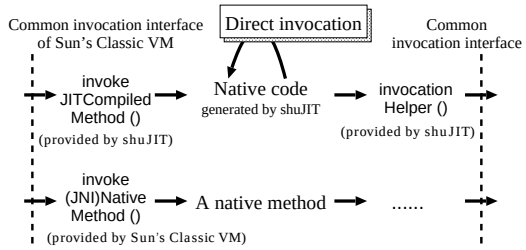


図 6 直接呼出し
Fig. 6 Direct invocation between compiled methods.

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

図 6 中の `invokeJITCompiledMethod()` は本コンパイラ側で用意している関数であり，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．JIT 生成コードと他種のメソッドでは，スタックのメモリ上の位置及び成長方向が異なるため，引き数の積換えを行うために同関数を用意している．

しかし，JIT 生成コード同士の呼出しでは，共通呼出しインタフェースを介す必要も，スタックを積み換える必要もない．ラッパの呼出しを省けば，関数呼出しとスタックの積換えのコストを削減できる．そこで，JIT 生成コード同士は，ラッパを介さずに直接呼び出すという最適化を施した．

それでも依然として，呼出し対象が JIT 生成コードかそれ以外の種類のメソッドかを実行時に判別する必要はある．動的束縛では，callee が JIT 生成コードか否かをコンパイル時に判別できるとは限らないからである．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でオーバライドされることもあるので，実行時のこの判別処理は避けられない．本コンパイラはこの判別を行うネイティブコードを生成する．

この判別処理は条件分岐を必要とし，実行時のオーバヘッドを伴う．そこで，判別を削減するというもう一つの最適化を施した．callee が static，private または final メソッドである場合には，上述の判別処理を省くのである．

とはいえ，JIT コンパイルは必ず成功するとは限らない．JIT コンパイルできなかったメソッドは，インタプリタで実行せざるを得ない．caller に対してここで述べた最適化を施す，つまり，callee が JIT 生成コードか否かを判別することなく，ラッパを介さずに直接呼び出すためには，callee を確実にコンパイルできることの確証が必要である．コンパイルの失敗は，メモリの確保失敗など不測の理由によって起こるので，確証を得るためには実際にコンパイルを済ませる必要がある．

そこで本コンパイラは，callee が static，private または final メソッドであり，caller のコンパイル時点で未コンパイルであった場合には，eager に，その時点でコンパイルしてしまう．これによってコンパイルの成否を確認でき，判別処理を省略できるか否かを判断できるのである．

しかし，この方法では，呼び出されないメソッドまで eager にコンパイルしてしまう可能性があり，メモリとコンパイル時間が無駄に費やされるおそれがある．このように得失の双方があるため，本コンパイラ自体を C コンパイラでコンパイルする際に，この最適化を行うか否かを選択できるようにしてある．また，eager なコンパイルを避けつつ判別処理を省くことができる別の手法もある．すなわち，callee は呼び出された時点で必要に応じてコンパイルし，コンパイル成功を確認できた時点で caller の JIT 生成コードを書き換えて最適化するのである．判別処理をスキップするようにコードを書き換え，callee のアドレスを書き込むのである．ただ，この方法では，caller 中には常に判別処理のためのコードが生成され，これを省略することはできない．つまり，eager なコンパイルによる余計なメモリ消費を抑えられる一方，若干，caller の JIT 生成コードが大きくなる．

これらの最適化の結果，CaffeineMark 3.0 の Method ベンチマークのスコアが 1780 から 7652 に，つまり約 4.3 倍に向上した．また，図 5 に示すとおり，SPEC JVM98 のスコアへの影響も非常に大きい．この最適化によって，総合スコアは 1.7 倍，最も影響の小さい _213_javac で 1.1 倍，最も影響の大きい _227_mtrt に至っては 2.9 倍となっている．これらの結果から，SPEC JVM98 のスコア，特に _227_mtrt に対してメソッド呼出しの効率が非常に大きく影響することがわかる．

4.2 命令畳込み

命令畳込み（instruction folding）は，複数の命令を同等なより少ない命令群に置き換えることで，命令数や処理コストを低減する最適化である．スタック操作のために命令数が増えがちであるスタックマシンにおいて効果が高いことが知られている [11]．picoJava [12]

を代表として，Java プロセッサへの実装例や検討例が多い [13]．本コンパイラは，JVM 上のスタック操作を x86 のスタック操作に置き換えるというコード生成を行うため（**2.** 参照），スタックマシンと同様に命令畳込みの効果が期待できる．

例えば，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場合，局所変数に対するポップ命令と局所変数からのプッシュ命令を続けて生成する．32 ビット整数であれば `istore`，`iload` という命令列である．この命令列をそのままコンパイルすると，ポップとプッシュという 2 度のメモリ操作を行ってしまう．これを，局所変数へのコピー命令に置き換えるのである．

本コンパイラは，内部命令列の上で，メモリアクセス，すなわち整数及び浮動小数点数のレジスタ–メモリ間コピーの削減を目的とした命令畳込みを行う．そのために，次に示す命令列を検出して，() 内の無駄なコピーを削減する．() 内のコピー処理を行わないような別種の内部命令に置き換えるのである．

（1） スタック → 局所変数（→ スタック）

（2） 浮動小数点レジスタ（→ スタック）→ 局所変数または配列

（3） 局所変数または配列（→ スタック）→ 浮動小数点レジスタ

（4） 浮動小数点レジスタ（→ スタック）→ 浮動小数点レジスタ

具体的には，例えば `istore`，`iload` という命令列を等価な内部命令 `istld` に置換する．この `istld` は，スタックトップから局所変数へのコピーだけを行う内部命令である．この畳込みは上記（1）の例である．

ここでいう局所変数とは Java 言語，JVM の局所変数であり，本コンパイラはこれをメモリ上に配置する．同様にスタックとは JVM のスタックであり，基本的にはメモリ上にあるが，**3.** で述べたとおり，スタックトップ付近は整数レジスタでキャッシュされる．

このように，整数レジスタは **3.** で述べたスタックキャッシュによって活用されるが，浮動小数点レジスタの効率的な利用は考えられていない．それを補い，浮動小数点数の無駄なコピーを削減することが，本コンパイラにおける命令畳込みの大きな目的である．命令畳込みはよく知られた最適化手法ではあるが，その効果と位置付けは本コンパイラ独特のものである．

現に，SPEC JVM98 のベンチマーク群のうち，浮動小数点演算の多い_222_mpegaudio に対する効果が特に高く，この最適化によってスコアが 1.16 倍に向上している（図 5）．同様に浮動小数点演算の多い Linpack ベンチマークでその効果を調べた．問題サイズは 500×500 である．

畳込みなし　14.042 Mflops
畳込みあり　19.083 Mflops

命令畳み込みによって 1.36 倍と大きく向上している．

もし，スタックキャッシュで整数レジスタだけでなく浮動小数点レジスタでもキャッシュできたとすると，本コンパイラが実装している上述の畳込み (1)〜(4) のうち，(2)，(3)，(4) は不要となる．しかしそのためには，スタック状態の数を増やしたり，整数，浮動小数点数の双方に用いられる型汎用のバイトコード命令（`pop`，`pop2`，`dup`，`dup2` など）に対して生成するネイティブコードを適切に選ぶためにスタックに積まれている要素の型を追跡する必要が生じる．

4.3 シグナルを利用した例外の捕そく

OS のシグナルを利用した例外の検出は，Java のインタプリタや JIT コンパイラでよく用いられる手法である．条件分岐なしにある種の例外を検出できるため，例外が発生しない場合，つまり正常系のオーバヘッドをゼロにできる．

具体的には，次の例外検出にシグナルを利用する．

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

`NullPointerException` と `ArithmeticException` の検出は多くの Java バイトコード処理系において一般的であるが，本コンパイラは `StackOverflowError`，すなわちスタックあふれの検出にもシグナルを活用する．

Java 言語の `null` は Classic VM 中ではアドレス `0` で表現されているため，それをオブジェクトとしてアクセスするとメモリ保護違反によって `SIGSEGV` が発生する．このシグナルをシグナルハンドラでとらえることで，`NullPointerException` を検出できる．また，整数のゼロ除算では `SIGFPE` が発生し，`ArithmeticException` を検出できる．

スタックあふれを検出するためには，メソッド呼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` が発生する．どの程度先をアクセスするかは，OS や libc の種類といっ

た環境に依存する．シグナルハンドラと例外発生処理を行うために充分なマージンをとっておく必要があり，必要なマージンは環境によるからである．

シグナルの利用によって例外検出のオーバヘッドをなくすことができる反面，例外が発生した場合の処理コストは一般に増加する．条件分岐を用いた検出よりも，シグナルの発生及び捕そくの方が重い処理だからである．このため，上記の例外が頻繁に発生するプログラムでは，この最適化によって性能が低下することもあり得る．

例外発生の頻度はプログラムによって様々であるため，この最適化の効果はプログラムの性質に依存する．しかし，例外は例外的な状況でのみ使用するという方針がプログラミングの良い方法だとみなされており，これに従っているプログラムでは例外発生の頻度は高くないことが期待できる．実際に，図 5 に示したとおり，SPEC JVM98 のスコアは向上している．

では，例外発生の頻度がどの程度であれば性能は向上，あるいは低下するのか．これを見積もるために，シグナルの利用による正常系のコスト削減量と例外発生時のコスト増加量を測定し，比較した．オブジェクトへの参照からインスタンス変数を読み出すという `NullPointerException` を発生し得る処理を繰り返すプログラムを用いて，参照が `null` の場合（例外発生）と非 `null` の場合（正常系），それぞれの場合について，シグナルの利用による実行時間の増減を測った．

1.7 GHz Pentium 4 では，正常系の実行時間が 10 億回当り 136 ミリ秒短縮され，逆に例外発生時の実行時間は 100 万回当り 12302 ミリ秒延びた．600 MHz Pentium III では，正常系の実行時間が 1 億回当り 334 ミリ秒短縮，例外発生時の実行時間は 10 万回当り 1521 ミリ秒延びた．繰返し 1 回当りで計算すると，正常系の実行時間減少に対する例外発生時の実行時間増加の比は次のとおりである．

1.7 GHz Pentium 4	90500 倍
600 MHz Pentium III	4600 倍

つまり，インスタンス変数アクセスといった `NullPointerException` を発生し得る処理において，Pentium 4 であれば，例外発生の頻度が約 9 万回に 1 度より低ければこの最適化が有効である，と見積もることができる．

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．

```
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
Fig. 7　Signal handling for catching an exception.

4.4　コード書換え

Java 言語の仕様 [1] では，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が厳密に定められている．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 メソッドの呼出しがクラスの初期化を引き起こすことがある．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場合，その時点で初めてそのクラスを初期化しなければならない．

かといって，このために素朴に条件分岐を用いたのでは，static 変数アクセスと static メソッドの呼出しごとに条件分岐のオーバヘッドを被るはめになる．あるコードの 2 度目以降の実行では，初期化は確実に済んでいるためこの条件分岐は不要であり，極力避けたい．実際には，JIT コンパイルの時点で初期化済みのクラスについては初期化済みか否かの判定コードを生成する必要がないため，これだけで多くの条件分岐を省ける．しかし，JIT コンパイル時には未初期化だったクラスについても，条件分岐のオーバヘッドは極力小さくしたい．

2 度目以降の実行では無駄な条件分岐を避ける方法として，初めての呼出し時はインタプリタで実行しておき，2 度目に呼び出された時点で初めて JIT コンパイルを行うという方法がある．例えば，Borland 社の

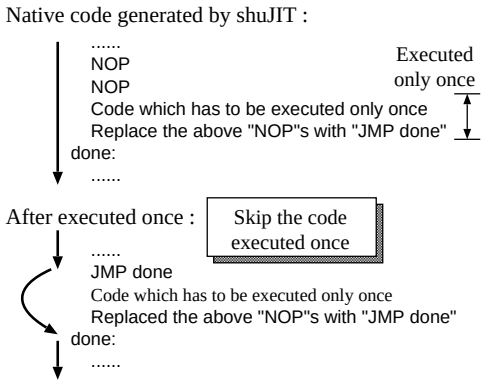
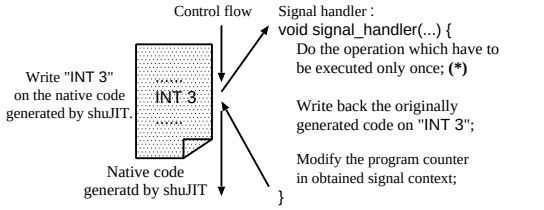


図 8　NOP 命令をジャンプ命令で上書きする方法
Fig. 8　Overwriting a jump instruction onto a NOP instruction.



(*) In reality, the signal handler do not execute the required operation by itself. Instead the signal handler generates a trampoline code which jumps to the operation and set the address of the trampoline into the obtained signal context. Hence the operation is executed after the signal handler returns. If the signal handler executes the operation by itself, more signals can occur in the operation even though the signal handler has already been running.

図 9　ソフトウェア割込み命令を本来の命令で上書きする方法
Fig. 9　Overwriting the original instruction onto an instruction to cause software interruption.

JBuilder Java 2 JIT がこの方法をとっている．しかしこの方法では，初めて呼び出されたメソッドの実行が長時間続いた場合，性能に劣るインタプリタで実行が続けられてしまう．この問題を回避するため，IBM 社の Java 2 SE や Sun 社の HotSpot VM は，いったんインタプリタで実行を始めたメソッドであっても，JIT コンパイル結果に実行を移す機構を備えている．本 JIT コンパイラは，これらとはまた異なる方法で問題を回避している．

4.4.1　shuJIT の方法

shuJIT は，コード書換えを利用して，上述の問題を被ることなく，Java 言語と JVM の仕様を完全に遵守している．すなわち，1 度実行した JIT 生成コードを，2 度とは実行しないようにコードを書き換えるのである．これによって，初めての呼出しの時点でメソッドをコンパイルしても，条件分岐によるペナルティを被ることなく，Java の仕様を守ることができる．処理の軽いベースラインコンパイラを指向する本コンパイラにとって，初回の呼出し時に問題なくコンパイルを行えることは重要な要件である．ゆえに，コード書換えは本コンパイラ向きの手法である．

コード書換えの方法としては以下の 2 通りが実装されており，JIT コンパイラ自体を C コンパイラでコンパイルする時点で選択できる．

- 図 8：無操作命令（NOP）をジャンプ命令で上書きする方法（ジャンプ命令法）
- 図 9：ソフトウェア割込み命令（INT）を本来の命令で上書きする方法（割込み命令法）

ソフトウェア割込み命令としては，デバッガの実装用に用意されている唯一の 1 バイト命令 `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慮すると，コード書換えは atomic に行う必要がある．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，処理が複雑になると同時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 バイト命令を利用した．

本コンパイラに実装されたジャンプ命令法は，2 バイトの書換えを行う．2 バイトを atomic に書き換えるためには，通常の MOV 命令ではなく，ORP [14] と同様に XCHG 命令を用いている．これによって atomicity が保証される [15]．ジャンプ命令法を 1 バイトの書換えで実装することも可能ではある．ジャンプ命令のジャンプ先のみを書き換えればよい．

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（無条件）ジャンプ命令は実行せざるを得ないが，割込み命令法では，2 度目以降は完全に無駄のないコードになるので，数度目以降の実行で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よる．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，生成コードだけを見ると割込み命令法が勝る．しかし，割込み命令法では，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り，そのための表が消費するメモリも考慮しなければならない．最終的にどちらが勝るかは，実装方法やアプリケーションプログラムに依存する．

4.4.2　考えられる他の方法

本コンパイラに実装されているのは上述の 2 通り

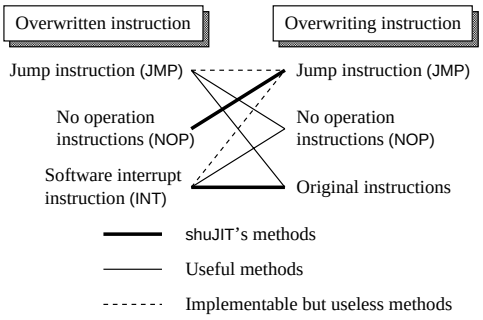


図 10 コード書換えを行う種々の方法
Fig. 10 Various implementation methods for code patching.

の方法であるが，コード書換えの方法はほかにも考えられる．書換え前の命令と書換え後の命令の組合せによって，図 10 に示す数通りの方法があり得る．本来の命令で上書きする方法には，数度目以降の実行でペナルティが皆無になるという利点があり，無操作またはジャンプ命令で上書きする方法には，本来の命令を保存しておくためのメモリが必要ないという利点がある．

4.5 インライン展開

インライン展開は，メソッド呼出しコストの削減と，手続き間解析及び最適化の適用範囲拡大につながる，よく知られた最適化手法である．

本コンパイラは，callee をコンパイル時に特定できる場合，つまり，対象が static，private または final メソッドである呼出しについて，ある条件が満たされていればインライン展開を行う．より具体的には，バイトコード命令 `invokespecial` と `invokestatic` での non-virtual な呼出しや，`invokevirtual` での final メソッドの呼出しについて，次の条件が満たされていれば展開を行う．

- ジャンプ命令を含まない
- catch 節を含まない
- 長さが shuJIT 内部命令で *20* 以下
- caller と callee の双方とも `strictfp` [1], [6] ではないか，または，双方とも `strictfp` である

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，実行時に環境変数として指定できる．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あふれたり，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．

ジャンプ命令や catch 節を含むメソッドをインライン展開しないという方針は，これを行うとジャンプ先の解決処理が煩雑になり，コンパイル処理が重くなることが予想されるためである．Sun 社の ExactVM も同じ方針を採用しており [16]，また，経験則どおり多くのアクセサが分岐や catch 節を含まないならば不利益は小さい．

図 5 のとおり，SPEC JVM98 に対するこのインライン展開の効果は _228_jack にて 3%，総合スコアでは 1%と決して高くはない．適用対象を non-virtual な呼出しに限定していることに加えて，テンプレートを用いたコード生成に原因があると考えている．本コンパイラのコード生成方式（**2.** 参照）では，インライン展開によって呼出しコストは削減されても，メソッド間最適化の機会拡大にはつながらない．そもそもそういった最適化を行わないからである．

4.6 特定メソッドのインライン展開

上で述べた一般のインライン展開に加えて，特定のメソッドを特別扱いしてあらかじめ用意したネイティブコードに展開するというインライン展開を試みた．本コンパイラへの実装は非常に容易である．特定メソッドに対応する shuJIT 内部命令とテンプレートを用意し，Java バイトコード命令を内部命令に変換する際に当該内部命令に変換すれば済む．

浮動小数点演算のために用意されている `java.lang.Math` クラスに着目し，`Math` クラスがもつメソッド群の浮動小数点演算命令への置換えを試みた．対象は次のメソッドである．

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

展開によってセマンティクス，つまり演算結果が変わってしまうことを確認できているメソッド，例えば `exp` については，展開を行わない．

この展開を行うことで，次の二つの理由で性能向上が得られる．

- ソフトウェアの数値演算ライブラリ `fdlibm` ではなく，浮動小数点演算命令が使われる．

表 1　sqrt の呼出し 10,000,000 回に要した時間
Table 1　Execution times to invoke the sqrt method 10,000,000 times.

	展開なし	展開あり	高速化率
Pentium 4	15535	851	18.3 倍
Pentium III	34959	1567	22.3 倍

（ミリ秒）

表 2　sin の呼出し 10,000,000 回に要した時間
Table 2　Execution times to invoke the sin method 10,000,000 times.

	展開なし	展開あり	高速化率
Pentium 4	4242	1567	2.71 倍
Pentium III	8021	2226	3.60 倍

（ミリ秒）

- ネイティブメソッドの呼出しが行われなくなる．

Java 2 には，全く同じメソッド群をもった二つのクラス `java.lang.Math` と `java.lang.StrictMath` が用意されている．`StrictMath` の仕様は，どんな環境でも Freely Distributable Math Library（`fdlibm`）というソフトウェアと全く同じ結果を返すというものであり，本論文で実験に用いている Sun 社，IBM 社の Java 2 SE は `StrictMath` のメソッド群を実装するために `fdlibm` を用いている．一方，`Math` クラスのメソッド群では，複数の環境で厳密に同じ結果を返すことが要求されていない．そのため，環境に応じた方法で計算を行って `fdlibm` を使うよりも高い性能を得る余地がある．しかし，Sun 社，IBM 社の Java 2 SE のクラスライブラリは，shuJIT が対象とする Classic VM も含めて，`Math` クラスであっても `StrictMath` と同様に `fdlibm` を呼び出すことで演算を行うように実装されている．そのため，JIT コンパイラが `Math` クラスを特別扱いしない限りは，どんなにインライン展開しようとも演算には `fdlibm` が使われてしまう．性能に勝る浮動小数点演算命令が使われることは期待できないのである．`Math` クラスによる浮動小数点演算をハードウェアで行うためには，本手法のように，JIT コンパイラが `Math` クラスを特別扱いすることが必須なのである．

本手法によって性能向上が期待されるもう一つの理由は，ネイティブメソッドの呼出しを減らせることである．Java 2 SE では，`sqrt` など `Math` クラスのメソッドを呼び出すと JNI に準拠したネイティブメソッドが呼び出される（以下，JNI 呼出しと呼ぶ）．JNI 呼出しは JVM の種類によらず重い処理であることが知られている．これには，引き数のスタック上レイアウトを JNI 規約に合わせるためのスタックへの積直しが避けられない，といった原因がある．`Math` クラスのメソッドを単にインライン展開しても JNI 呼出しは避けられない．そこで，本手法のように特別扱いして置換することで JNI 呼出しまで削減できるのである．

`sqrt` と `sin` について性能上の効果を調べた．1.7 GHz Pentium 4 と 600 MHz Pentium III での結果を表 1 と表 2 に示す．`sin` でも 3 倍前後，`sqrt` に至っては 20 倍前後という大きな効果が得られている．とはいえ，この種の最適化はその特定のメソッドを頻繁に呼び出すプログラムに対してのみ有効である．例えば SPEC JVM98 では効果は全く期待できず，この最適化を行わずともスコアの有意な低下は見られなかった．しかしそれでも，特定の場合にはその効果は非常に大きい．

この最適化処理はコンパイラに埋め込まれているため，実装が容易とはいえ，対応メソッドを増やすためにはコンパイラ自体に手を加えなければならない．ソフトウェアのよい設計という観点からは，本来は，OpenJIT 2 の Compilet 構想 [17] のように最適化処理をコンポーネントとして JIT コンパイラに対してプラグインできる構造が望ましい．

4.7　ループ先頭の整列

本コンパイラ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 バイトずつフェッチするので，分岐先を 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．もっとも，Pentium 4 プロセッサでは，1 次命令キャッシュは，x86 命令から変換された後のマイクロオペレーション（μop）を保持する方式となったため，この 1 次キャッシュから命令を取得できる限りは分岐先が整列されているか否かは性能に影響しない．分岐先を整列することの重要性は Pentium III 以前より下がった．

この最適化によって，必ずしも性能が向上するとは限らない．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ようにしている．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．また，生成コード量も増加するため，命令キャッシュの活用に悪い影響を及ぼし得る．とはいえ，この最適化によって，SPEC JVM98 の総合スコ

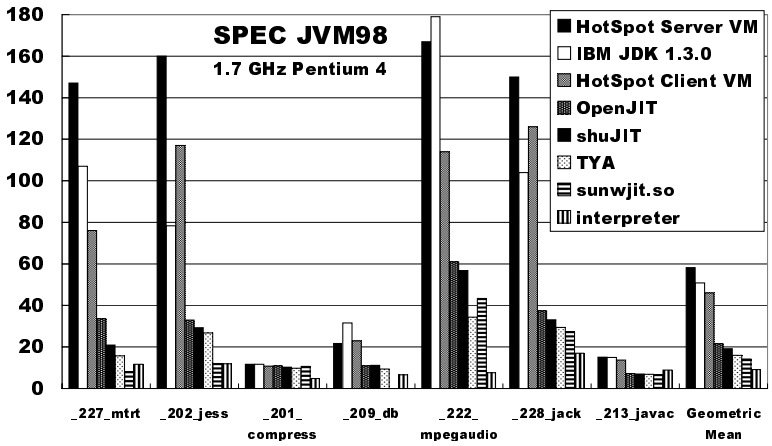


図 11　SPEC JVM98 の結果
Fig. 11　The result of SPEC JVM98.

アはわずかに向上している（図 5）.

5. 性能評価

以下では，PC 用の他の JIT コンパイラとの間でピーク性能とアプリケーション起動時間を比較することで，shuJIT の性能上の位置付けを明らかにする.

5.1 ピーク性能

本コンパイラを含めた種々の JIT コンパイラの性能を SPEC JVM98 と Linpack ベンチマークで計測した．結果を図 11 と図 12 に示す．実行方法は，SPEC に報告可能な（reportable）結果を得る方法，すなわち，SPEC JVM98 のクラスはウェブサーバからロードし，application としてではなく applet として問題サイズ 100 で Auto Run ボタンを押すことで実行した．調整が許されているパラメータは，SPEC JVM98 に含まれているプロパティファイル（`props/user`）中の値をそのまま用いた．つまり，以下に述べるとおりのパラメータである．SPEC JVM98 に含まれる各ベンチマークは，最低 2 回（`automin=2`），最高で 5 回（`automax=5`）繰り返して実行し，最良の結果を採用する．ただし，前回の結果を最低でも 3%（`percentTimes100=300`）上回ることができなかった場合は，その時点で繰返しを打ち切る．ベンチマークの各実行の間には，500 ミリ秒（`autodelay=500`）の停止時間を設ける．また，各実行の間には `System.gc()` と `System.runFinalization()` を呼び出す（`autogc=true`）.

SPEC JVM98 は，実アプリケーションをもとにした 7 種類のベンチマークプログラムを実行し，ある環境での結果に基づいて実行時間を正規化し，その逆数をスコアとする．総合スコアはそれらの相乗平均である．Linpack Benchmark はガウスの消去法で連立 1 次方程式を解き，結果は Flops で得られる．いずれも高い値が良い結果を表す.

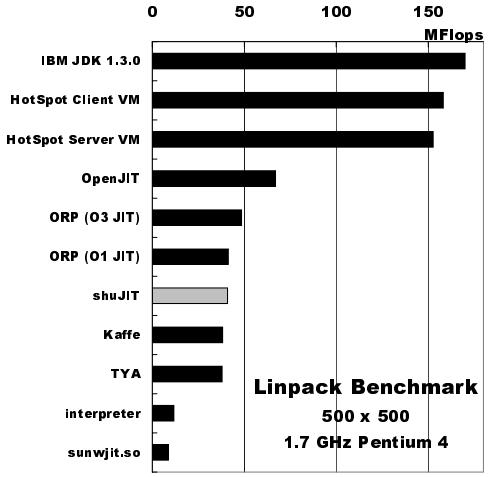


図 12　Linpack ベンチマークの結果
Fig. 12　The result of Linpack Benchmark for Java.

比較対象として，shuJIT と同じく PC 以上の規模の性能，メモリ容量を想定している JIT コンパイラを集めた．すなわち，Sun 社の Java 2 SE 1.3.1_02 に含まれる HotSpot Server VM 及び Client VM，IBM 社の Java 2 SE 1.3.0 [18] [19]，Intel 社の ORP（Open Run-

time Platform）[14]，Transvirtual 社の Kaffe 1.0.6, TYA 1.7v3 [4]，OpenJIT 1.1.16 [20]，Sun 社が Java 2 SE 1.2.2 とともに配布していた `sunwjit.so`，そして HotSpot Client VM のバイトコードインタプリタである．いずれの処理系でも 1.7 GHz Pentium 4 上で OS として Linux 2.4.18-pre3 を用いている．以下，特に断りのない限り，実験結果はこの環境でのものである．

ただし，ORP と Kaffe は SPEC に報告可能な方法で SPEC JVM98 を実行することができない．必要なライブラリが欠けているためである．このため，ORP と Kaffe の結果は図 11 には含まれていない．

本コンパイラのピーク性能は，インタプリタの数倍は達成しているものの，他のコンパイラと比べて高い方ではない．しかし，本コンパイラが指向する，処理が軽くコストパフォーマンスが高いコンパイル手法は依然重要である．例えば，ORP [14]，Jikes RVM [21] といった JVM は Java バイトコードのインタプリタを持たず，その代わりとして，ベースラインコンパイラと呼ばれる処理の軽い JIT コンパイラをもっている．また，良いピーク性能を示している HotSpot VM や IBM 社の Java 2 SE は，コンパイル対象とするメソッドを厳しく限定している．未コンパイルのメソッドはまずインタプリタで実行され，呼出しや後方ジャンプの回数が 1000〜10000 に達したメソッドに限ってコンパイルする．あらゆるメソッドを時間をかけてコンパイルするわけにはいかない以上，インタプリタやベースラインコンパイラの効率は重要である．

5.2 アプリケーションの起動に要する時間

5.1 の性能評価結果に表れているのは JIT コンパイラが生成したコードのピーク性能である．これらのベンチマークはピーク性能の測定を目的としており，一度コンパイルされたメソッドが頻繁に実行されるため，実行時間に対するコンパイル時間は相対的に小さくなり，結果に表れない．特に SPEC JVM98 では同じベンチマークプログラムが複数回実行されて最良の結果が採用されるので，コンパイルに要する時間は隠れがちである．

そこで，コンパイル処理に要する時間の比較を試みた．評価の方法として，アプリケーションの起動に要する時間を JIT コンパイラ間で比較するという方法を採った．アプリケーションが起動するまでの間は，メソッドの初めての呼出しが頻繁に起こるため，JIT コンパイルも頻繁に起きる傾向がある．特に，1 度目の呼出しの際にそのメソッドを JIT コンパイルしてしまうという設定では，起動中に呼び出されたすべてのメソッドがコンパイルされるため，起動時間に占めるコンパイル時間の割合が大きくなる．そして，コンパイル処理に要する時間の長さが起動時間に大きく反映されることが期待できる．もし可能ならば，むしろコンパイル時間を直接計測すべきだが，コンパイラのソースコードが手に入らない場合，計測のための時刻取得処理を追加できない．また，時刻の取得は行えたとしても，OS によるプリエンプティブなプロセス切替や，Java ではごく当たり前であるスレッド切替が行われる状況で，コンパイル時間を直接測定することは困難である．また，直観的には，アプリケーションの起動時間は利用者が感じるストレスを左右する大きな要因なので，実際の使用感の指標ともなるだろう．

評価に使用するアプリケーションとしては，文書作成ソフトウェア一太郎 Ark 1.1 と，統合開発環境 NetBeans 3.3.1 を用いた．一太郎 Ark は，クラス数が 904，圧縮後のコードサイズ（JAR ファイル）が 1652 キロバイトであり，NetBeans は圧縮後のコードサイズが 2200 キロバイトという一定規模の実用ソフトウェアである．また，起動の完了を目視で判断しやすいことからこれらを選んだ．起動に要する時間は，ストップウォッチを用いて手で計測した．

比較の対象は，JIT コンパイルを行うまでのメソッド呼出しの回数を指定できる JIT コンパイラ，すなわち，IBM 社の Java 2 SE 1.3.0 に含まれる JIT コンパイラ JITC 3.6 と Sun 社の HotSpot Client VM と HotSpot Server VM である．shuJIT を含めて，これらの JIT コンパイラは，メソッドごとにカウンタをもち，メソッド呼出しなどの際にこれを減じて，0 となった際にそのメソッドをコンパイルする．この呼出しカウンタの初期値は，特に指定しない場合，IBM の Linux/x86 用 JITC 3.6 では 2000，HotSpot Client VM で 1500，HotSpot Server VM で 10000 となっている．ベースラインコンパイラには，呼び出された全メソッドをコンパイルするに耐えるだけの処理の軽さが要求されるため，shuJIT ではこの既定値は 0 としている．

呼出しカウンタの初期値を，0，3，2000，10000 と設定して，それぞれ，起動に要した時間を計測した．一太郎 Ark での結果を表 3 に，NetBeans での結果を表 4 に示す．∞ とあるのは，JIT コンパイルを禁止して完全にインタプリタのみで実行した場合の結果で

表 3　一太郎 Ark の起動に要する時間
Table 3　Start-up time of Ichitaro Ark.

カウンタの初期値	shuJIT	IBM JITC	HotSpot Client VM	HotSpot Server VM
0	4.4 *	22.1	計測不能	計測不能
3	4.0	14.1	5.9	32.1
1500	4.3	6.3	3.8 *	6.8
2000	4.4	5.7 *	3.7	6.6
10000	4.5	5.2	3.8	5.4 *
∞	3.9	5.0	4.0	4.1

(秒)

* カウンタ初期値の既定値

表 4　NetBeans の起動に要する時間
Table 4　Start-up time of NetBeans.

カウンタの初期値	shuJIT	IBM JITC	HotSpot Client VM	HotSpot Server VM
0	8.9 *	43.3	計測不能	計測不能
3	8.4	28.6	12.0	166.6
1500	10.3	10.8	8.6 *	20.7
2000	10.7	10.6 *	8.5	20.9
10000	13.2	10.5	8.9	17.2 *
∞	12.2	14.0	13.0	13.4

(秒)

* カウンタ初期値の既定値

ある．二つの HotSpot VM では，カウンタの初期値を 3 未満とした場合，試したどの Java プログラムも起動しなかった．

表 3，表 4 の結果は，厳密には，JIT コンパイラ間で公平に比較することができない．なぜなら，JVM やインタプリタの実装がそれぞれ異なり，呼出しカウンタを減じるタイミングも全く同じではないからである．例えば，shuJIT が用いる JVM は Sun 社の Classic VM であり，IBM JITC はそれを改造した JVM を用いている．HotSpot VM はまたそれらとは異なる JVM である．カウンタを減じるタイミングは，shuJIT ではメソッド呼出し時のみ，IBM JITC では呼出し及び後方ジャンプ時である．HotSpot VM も呼出し及びジャンプ時だが，ジャンプの方向が関係するか否かは公表されていない．

それでも，カウンタの初期値が 0 の場合は確実に全メソッドがコンパイルされ，カウンタの減じ方は結果に影響を与えない．また，カウンタの初期値が小さい場合に HotSpot Server VM での起動時間が 30 秒以上と非常に大きくなっており，JIT コンパイルに時間が費やされていると考えることが自然である．処理が重い JIT コンパイラほど起動に時間がかかるという傾向が表れており，shuJIT のコンパイル処理が軽いことを確認できた．

6. むすび

本論文では，コンパイル処理及び開発のコストを抑えつつ，研究基盤としての扱いやすさと高いコスト効率を達成した Java Just-in-Time コンパイラについて，コード生成手法と各種最適化手法を述べた．全メソッドをコンパイルするベースラインコンパイラとしての使用に耐えるように，コンパイル処理のコストを低く保ちながら，インタプリタの数倍という実用的な性能を達成できることを示した．同時に，開発コストも低く抑えたことで，1 人で 50 日前後の開発で，簡単なプログラムを動作させるに至った．

また，本コンパイラに実装した処理コストの低いいくつかの最適化について，実装方法，性能上の効果，コード生成手法との関係，Java 言語や仮想マシン仕様を遵守するためのペナルティ削減効果を論じた．メソッド呼出しの性能が多くのベンチマークの結果に大きく影響すること，本コード生成手法に対して命令畳込みの効果が高いこと，シグナルを利用した例外検出の効果を示した．また，仕様遵守に伴う性能上のペナルティをコード書換えで回避するというベースラインコンパイラ向きの手法を提案した．

派生研究の材料として扱いやすいものとするという目標も達成され，本コンパイラの開発者だけでなく，他の研究者によっても研究基盤として利用されるに至った．

今後の課題としては，ベースラインコンパイラとインタプリタの，性能，使用感，開発コストについての比較，呼出しカウンタのしきい値といったコンパイル戦略の構成法，それと関連して，メモリ消費量を考慮したコンパイル済みコード破棄戦略などが挙げられる．

文　献

[1] J. Gosling, B. Joy, G. Steele and G. Bracha: “Java Language Specification, Second Edition”, Addison Wesley (2000).
[2] T. Lindholm and F. Yellin: “The Java Virtual Machine Specification”, Addison Wesley (1997).
[3] M. A. Ertl: “Stack caching for interpreters”, Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 315–327 (1995).
[4] A. Kleine: “TYA”. http://sax.sax.de/~ adlibit/.
[5] 首藤一幸, 村岡洋一：“Java Just-in-Time コンパイラ実用化と配布の経験”, Proc. of 第 4 回 プログラミングおよび応用のシステムに関するワークショップ (SPA2001) (2001).
[6] 首藤一幸, 関口智嗣, 村岡洋一：“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”, 情報処理学会研究報告, 2001-ARC-144, pp. 99–104 (2001).

[7] M. Welsh and D. Culler: “Jaguar: Enabling efficient communication and I/O from Java”, Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications (1999).

[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, **40**, no.SIG 7（PRO 4）, pp. 66–79 (1999).

[9] K. Shudo and Y. Muraoka: “Efficient implementation of strict floating-point semantics”, Proc. of 2nd Workshop on Java for High-Performance Computing (in conj. with ICS'00), pp. 27–38 (2000).

[10] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh and J. M. Stichnoth: “Fast, effective code generation in a Just-In-Time Java compiler”, Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), pp. 280–290 (1998).

[11] 重田大助, 小川洋平, 山田克樹, 中島康彦, 富田眞治：“命令畳み込み，データ投機および再利用技術を用いた Java 仮想マシンの高速化”, 情報処理学会論文誌, **41**, No.SIG05（HPS 1）, pp. 28–38 (2000).

[12] H. McChan and M. O'Connor: “PicoJava: A direct execution engine for Java bytecode”, COMPUTER, **31**, 10, pp. 22–30 (1998).

[13] L.-R. Ton, L.-C. Chang, M.-F. Kao, H.-M. Tseng, S.-S. Shang, R.-L. Ma, D.-C. Wang and C.-P. Chung: “Instruction folding in Java processor”, Int'l Conference on Parallel and Distributed Systems, pp. 138–143 (1997).

[14] M. Cierniak, G.-Y. Lueh and J. Stichnoth: “Practicing JUDO: Java under dynamic optimizations”, Proc. ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI) (2000).

[15] Intel Corporation: “IA–32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide” (2001).

[16] 松岡聡：“Java とかの高速化技法”, 並列処理シンポジウム JSPP2000 チュートリアル資料, pp. 1–37 (2000).

[17] 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”, 修士論文, 東京工業大学 (2001).

[18] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu and T. Nakatani: “Overview of the IBM Java Just-In-Time compiler”, IBM Systems Journals, Java Performance Issue, **39**, 1 (2000).

[19] 石崎一明, 川人基弘, 今野和浩, 安江俊明, 竹内幹雄, 小笠原武史, 菅沼俊夫, 小野寺民也, 小松秀昭：“Java Just-In-Time コンパイラにおける最適化とその評価”, 電子情報通信学会技術研究報告, CPSY99-64, pp. 17–24 (1998).

[20] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda and Y. Kimura: “OpenJIT: An open-ended, reflective JIT compile framework for Java”, Proc. of 14th European Conference on Object-Oriented Programming (ECOOP '2000) (2000).

[21] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan and J. Whaley: “The Jalapeño virtual machine”, IBM Systems Journals, Java Performance Issue, **39**, 1 (2000).

（平成 14 年 4 月 10 日受付，8 月 19 日再受付）

首藤　一幸

1996 早大・理工・情報卒．1998 同大メディアネットワークセンター助手．2001 同大大学院理工学研究科博士後期課程了．同年産業技術総合研究所入所．現在に至る．博士（情報科学）．分散処理方式，プログラミング言語，言語処理系，情報セキュリティ等に興味を持つ．IEEE–CS，ACM 各会員．

関口　智嗣

1984 工業技術院電子技術総合研究所入所．以来，データ駆動型スーパーコンピュータ SIGMA-1 の開発，ネットワーク数値ライブラリ Ninf，クラスタコンピューティング，グリッドコンピューティング等に関する研究に従事．2001 独立行政法人産業技術総合研究所に改組．2002 年 1 月より同所グリッド研究センター長．市村賞，情報処理学会論文賞受賞．グリッド協議会会長．情報処理学会，日本応用数理学会，日本計算工学会，SIAM，IEEE，つくばサイエンスアカデミー各会員．

村岡　洋一　（正員）

1965 早大・理工・電気通信卒．1971 イリノイ大学電子計算機学科博士課程了．Ph.D. この間，Illiac-IV プロジェクトで並列処理ソフトウェアの研究に従事．同学科助手の後，日本電信電話公社（現 NTT）電気通信研究所に入所．1985 より早稲田大学理工学部教授．現在同大学メディアネットワークセンター所長．並列処理，マンマシンインタフェース等に興味を持つ．「コンピュータアーキテクチャ」（近代科学社）など著書多数．