Java JIT コンパイラの性能評価*

6Z－1

志村 浩也　　木村 康則†

(株) 富士通研究所‡

e-mail: kouya@flab.fujitsu.co.jp

1 はじめに

Java のバイトコードを高速に実行するための手法として，実行時に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) コンパイル方式がある．JDK1.1 の上で動作する SPARC Version 8 プロセッサのための JIT コンパイラを開発した．[1)]

単純なベンチマーク・プログラムではインタプリタに比べ 4～5 倍程度の速度向上が見られるのに対し，javac のような実用プログラムにおいては 1.9 倍程度しか性能が向上しなかった．Java の性能を向上させるために何が重要かを考察する．

2 性能測定

JDK1.1 では命令デコードがインライン展開されており，JDK1.0 のインタプリタと比較して，2～3 倍高速になった．JDK1.1 のインタプリタと JIT を使ったときの速度比で性能測定を行なった．測定に使用したマシンは SS-20(SuperSparc 75MHz, キャッシュ 1MB, 主記憶 64MB, Solaris2.4) である．

2.1 CaffeineMark3.0 による評価

CaffeineMark[2)]は一定時間内に処理した回数により性能を測定するベンチマークで，大きな値ほど性能が良い．この評価では同様の処理を C++で記述したものとも比較した (図 1)．C++のコンパイルには gcc 2.7.2 を用い，コンパイルオプションとして最適化なしの場合と-O2 でコンパイルした場合を測定した．ただし，String ベンチマークは単純に C++に変換できなかったので割愛している．

JIT はインタプリタと比較して概ね 4～5 倍程度の性能が出ている．Logic は peephole 最適化が効いて約 17.5 倍速くなった．Method は再帰呼出による SPARC の register window overflow/underflow により性能が出ない．C++に比較すると JIT は 0.3～0.6 倍程度である．

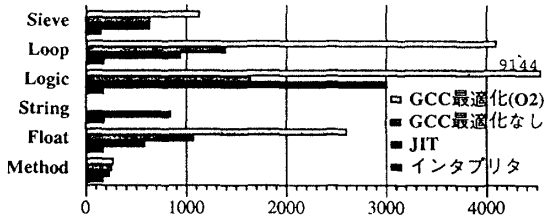

図 1: CaffeinMark3.0 の測定結果

2.2 javac の性能評価

JDK1.1 に付属の javac(java → bytecode コンパイラ) で Main.java のコンパイル (469 ファイル, 約 10 万行) の実行時間 (user time) を測定した．JIT はインタプリタに比べ約 1.9 倍しか速度が向上しない．

図 2: javac の測定結果

3 性能に関する考察

3.1 ヒープ領域のデータ構造

オブジェクトが割り付けられるヒープ領域の構造は，JDK1.1 では Handle 領域と Object 領域の 2 つに大きく分割されている．これは GC を容易に行なうためと推察される．構造を図 3 に示す．

オブジェクト変数を参照するとき，オブジェクトへのポインタから一旦 Handle 領域を読み出し，それを基に実際のデータをアクセスする必要がある．一般に load の実行レイテンシは数サイクル要するため，このオーバヘッドは大きい．配列のアクセスについても同様である．

javac の実行を測定した結果，バイトコードの命令に換算して，約 3 億命令，全実行の 23%の命令が Handle 領域のアクセスを行なっている．SuperSPARC 75MHz ではキャッシュが全てヒットした場合でもこのアクセスに 16.4 秒を損失する．

---

* Performance Evaluation of Java JIT Compiler

† Kouya SHIMURA　Yasunori KIMURA

‡ FUJITSU LABORATORIES LTD.

一方，コンパイラが一旦 load した Handle をレジスタに保持するコードを生成することにより，Handle 領域のアクセスを減らすことが可能である．

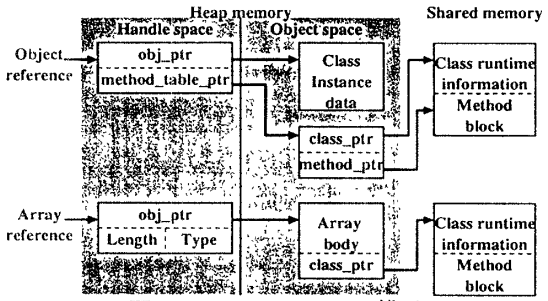

図 3: JDK1.1 のメモリ構造

3.2 配列の境界チェック

Java では実行中に配列の要素をアクセスするとき，必ず境界内であることをチェックする必要がある．これも C,C++に比べて性能上大きなオーバヘッドとなる．このチェックに JDK1.1 の場合 (図 3 を参照)，(1)Handle 領域の load. (2) シフト (Length の取り出し)，(3) 比較, (4) 分岐　の 4 命令を要する．

CaffeineMark で配列を使ったベンチマークについて，配列境界チェックを省いたときの性能を測定した (図 4)．このチェックは実行時間の 1 ～ 3 割を占めている．また，javac で測定した場合，14 秒 (実行時間の 6.5%) を費やしていた．特に数値計算では配列を多用するので，このオーバヘッドは深刻となる．

一方，コンパイラがプログラムを解析することにより，このチェックは省略することが可能である．我々の JIT コンパイラは，非常に簡単な配列境界チェックを省く最適化を行なう．CaffeineMark ではこの機能は働いていないが，javac では約 4 秒短縮できた．

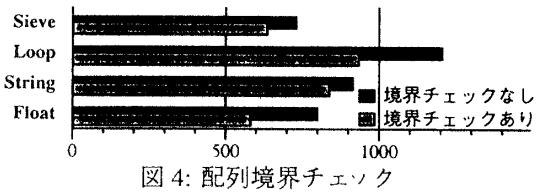

図 4: 配列境界チェック

3.3 GC(Garbage Collection)

JDK1.1 では GC としてマーク＆スイープ方式を用いているが，プログラマが明にオブジェクトの解放を指定する C や C++よりオーバヘッドが大きい．javac の実行において GC の処理時間を測定した所，約 90 秒であった．これは JIT コンパイラで実行したときの全実行時間の 43%を占める．

3.4 同期機構の処理時間

Java の実行中にはプログラマが指定した同期以外にも，Shared 領域の更新において，同期処理が実行される．Java では同期は monitor を使って行なわれるが，この処理に結構時間を取られることが分かった．

javac の実行中の同期を行なった回数を測定すると約 87 万回あった．同期だけをこの回数行なうプログラムを実行させた所，同期に 17.5 秒かかった．実際には monitor の衝突，ハッシュ値の衝突などがあるため，これ以上の時間がかかることが予想される．

4 まとめ

javac の実行時間の内訳は図 5 のようになる．5 割以上を占める GC と monitor 同期はコンパイルによる速度の向上が望めない．一方，12.5%を占める Handle 参照，配列境界チェックはコンパイラによって理想的には完全に取り除くことが出来る．

Java の処理速度を向上させるためには，最適化でその他の部分を高速することも重要であるが，それ以上に配列境界チェックをできるだけ省略することと Handle 参照を減らすことが重要である．また，C++並の処理速度にするためには GC と monitor 同期をいかに高速化するかが鍵となる．

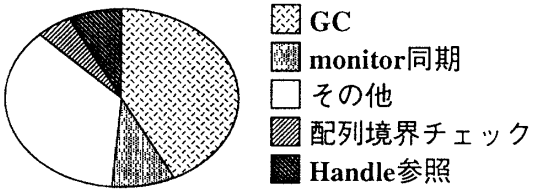

図 5: javac の実行内訳

参考文献

1)志村他, Java JIT コンパイラの試作, 情報処理学会研究報告 96-ARC-120, pp.37-42, Dec 1996

2)Pendragon Software Corp.: CaffeineMark 3.0, http://www.webfayre.com/pendragon/cm3

3)C.-H.A. Hsieh, M.T. Conte, T.L. Johnson, J.C. Gyllenhaal, W.W. Hwu, Optimizing Net Compilers for Improved Java Performance, IEEE Computer Vol 30 Number 6, 1997