ハイパフォーマンス
コンピューティング 82−21
(2000. 8. 4)

Javaクラスファイルの実行時ループ最適化手法

山崎泰伯[†] 窪田昌史[††] 津田孝夫[††]

数値計算プログラムでは、実行時間の多くがループに費やされる事が多い。しかし、ループ最適化のためのパラメータは計算機のキャッシュ容量、メモリなどにより異なるため、コンパイル時に決定するのは困難である。

そこで、本研究では、パラメータを指定することによりJavaのclassfileから直接ループアンローリングされたclassfileを生成するツールを作成した。それを用いて実行時に得られるプロファイル情報を用い、複数のパラメータから最適なものを選び出し、それを元にJavaクラスファイルを直接書き換える手法を提案する。

また、実際に行列積とJavaのベンチマークであるSciMark2.0への適用を行なった。その結果オーバーヘッドは無視できる大きさまでおさえる事ができ、実行時ループ最適化の効果が確認できた。

Runtime loop optimization techniques for Java classfiles

YASUNORI YAMASAKI, ATSUSHI KUBOTA
and TAKAO TSUDA

Most of execution time of many numerical software is spent on loops. However, it is difficult to decide parameters used by loop optimizations in advance because they depend on the capacity of the cache and the memory of computers.

This paper proposes techniques that select the optimal parameter from several candidate parameters using profiles obtained at runtime, and rewrite Java class files directly. In order to realize these techniques, we implemented systems which apply loop unrolling.

These techniques are applied to matrix multiplication and SciMark2.0, which is a Java benchmark. As a result, we confirmed that these techniques reduced the overhead and the techniques had the large effect.

1. はじめに

近年、大規模数値計算の高速化の研究が様々な分野で行なわれている。なかでも実行の高速化を目的とした様々な最適化技法が提案されている。数値計算のプログラムには繰り返し回数の多い多重ループが存在するものが多い。この多重ループがプログラム全体の実行時間の大半を占める場合多重ループに最適化技法を適用する事により実行が高速化が期待できる。

これらの最適化技法のなかには、コンパイラによって自動的に適用されるものもあるが、ソースプログラムの書き換えによって適用できるものも多い。さらにこれらの適用の可否は実行される計算機の性能、メモリやキャッシュの容量などの実行環境によっても異なる。

このような最適化技法には、関数のインライン展開、ループの交換、ループアンローリング、ループのブロック化アルゴリズムがある。インライン展開を行なったとしても必ずしも実行が高速化されるとは限らない。ループアンローリングでは実行時の環境や実行するプログラムが違えばアンローリング段数は変わってくるため容易には求められない。ATLAS[1]ではすべてのアンローリング段数の実行時間を計り、実行したマシンにおいて最適だと思われるアンローリング段数を求めているが、これには膨大な時間がかかる。さらに、計測したマシンで実行するすべてのプログラムにおいて求めたアンローリング段数が最適であるとは保証されない。ループの交換も、交換すべきかどうかが容易には決定できない場合がある。

そのため、現状では様々な最適化技法をプログラマが書き換えて適用したプログラムを実際に実行し、最適化技法を適用すべきかどうか、アンローリングの段数やブロックサイズをいくつにするかなどを決定している。

実行時に高速化をはかる手法[2]では実行時にプログラムの一部を実行し、最適化技法の可否や、アンローリング段数などのパラメータを求めているが、これは、一度ソースファイルを作り出し、実行中にそれをコンパイル

† 広島市立大学大学院 情報科学研究科
Graduate School of Information Sciences, Hiroshima City University Graduate School
†† 広島市立大学 情報科学部
Faculty of Information Sciences, Hiroshima City University

しているため、オーバーヘッドが大きい。

我々はバイトコードを直接変換することで、このオーバーヘッドを小さく手法[3]を提案している。本研究では、実行時に得られるプロファイル情報を用い、複数のアンローリング段数候補から最適と思われる物を選びだしそれを元にクラスファイルを直接書き換えるツールの作成を行なった。

実行時に書き換えたプログラムを利用する為、リフレクション機能を持つ Java を用いた。

クラスファイルの変換を実現するため、クラスファイルの中の変換対象部分をソースコードの行番号で指定した。Java のクラスファイルにはソースファイルの行番号とそれに対応するインストラクションコードの番地が記されているため、クラスファイルの変換が可能となっている。そのため、クラスファイルの変換場所の指定をソースコードの行番号で行うことができ、比較的容易にクラスファイルの変換を行うことができる。

以下、本研究では、2 章で作成したプログラムを説明し、実行時の流れを説明する。そして、3 章では実際のプログラムに適用し、実行した結果の評価をする。最後に 4 章で総括を行ない、今後の展望や将来性を述べる。

2. 実行時最適化手法の実装

2.1節で本研究で実行時に適用するループ最適化技法として取り上げるループアンローリング、ループの除去、多重ループのブロック化について説明し、2.3節でクラスファイルの構造の説明をした後、2.4節から2.6節でそれぞれの技法を実行時に適用する手法について説明する。

2.1 ループの最適化技法

ループの最適化技法には様々なものがあるが今回は以下の 3 つについて実行時の最適化を行なった。

- ループアンローリング
- ループの除去
- 多重ループのブロック化

以下に上記 3 つの内容を記す。

2.1.1 ループアンローリング

ループアンローリングとは、ループの中身を展開する事である。この最適化技法を適用することによって、繰り返し回数を減らすことができ、ループのオーバーヘッドを少なくすることが出きる。図 1にループアンローリングを適用した例を示す。

LoopUnrolling 適用前

```
for( i=0; i<ary; i++ ){
  c[i] = ary / i;
}
```

LoopUnrolling 適用後

```
for( i=0; i<ary%2; i++)
  c[i] = ary / i;

for( i=ary%2; i<ary; i+=2)
  c[i] = ary / i;
  c[i+1] = ary / (i+1);
}
```

図1 ループアンローリング

2.1.2 ループの除去

ループの除去は完全にループを展開してしまってループを取り去ってしまうことである。ループアンローリングを適用する際に、繰り返し回数が少なく、また適用時に繰り返し回数が確定していれば、こちらの手法を適用する。ループに適用することでループのオーバーヘッドを取り除くことが出きる。図 2にループの除去を適用した例を示す。

Loop除去 適用前

```
for( i=0; i<5; i++ ){
  c[i] = ary / i;
}
```

Loop除去 適用後

```
c[i] = ary / i;
c[i+1] = ary / (i+1);
c[i+2] = ary / (i+2);
c[i+3] = ary / (i+3);
c[i+4] = ary / (i+4);
```

図2 ループの除去

2.1.3 多重ループのブロック化

多重ループのブロック化とは、ループにストリップマイニングを適用することで、ループの入れ子が参照するメモリをブロック化する、メモリ参照の一つの最適化技法である。これによりキャッシュミスを少なくすることで高速化をはかることができる。図 3に多重ループのブロック化を適用した例を示す。

ブロック化適用前

```
for(k=0; k<ary; k++)
 for(i=0; i<ary; i++)
  for(j=0; j<ary; j++)
   c[i][j] += a[i][k]*b[k][j];
```

ブロック化適用後

```
for(kk=ary%26; kk<ary; kk+=26)
 for(ii=ary%26; ii<ary; ii+=26)
  for(jj=ary%26; jj<ary; jj+=26)

   for(k=kk; k<min(kk+26,ary); k++)
    for(i=ii; i<min(ii+26,ary); i++)
     for(j=jj; j<min(jj+26,ary); j++)
      c[i][j] += a[i][k]*b[k][j];
```

図3 多重ループのブロック化

2.2 Java の実行形態

Java は 1995 年に SunMicrosystems 社が発表したオブジェクト指向言語である。Java 言語によって書かれたソースファイルをコンパイルするとバイトコードと呼ばれる中間言語で記されたプログラムが一度作られる。Java は Java 仮想マシン (Java Virtual Machine、以下 JVM と略す) が動きさえすればプラットフォームを選ばず実行することができる。JVM はバイトコードを解釈し、インタプリタ形式で実行していく。これが Java の一般的な実行形態である。図 4に実行の様子を示す。

2.3 クラスファイルの内部構成

本節ではクラスファイルの内部構成の簡単な説明を行ない、ソースファイルの行番号より、ループの場所を指定できる事を示す。

図 5にクラスファイルの構成を示す。クラスファイルの主要部分はコンスタントプール、インタフェース、フィールド、メソッドの 4 つがあり、メソッドの中にイ

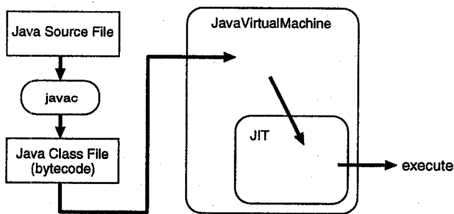

図4 Javaの実行形態

ンストラクションコード、ラインナンバーテーブルが収められている。ラインナンバーテーブルは、ソースファイルの各行とそれに対応するインストラクションコードの番地を格納している。次節以降で述べるように、本研究ではあるループのソースコードの行をキーとしてラインナンバーテーブルを検索し、そのループに対応するインストラクションを特定できる事を利用して実装を行なっている。

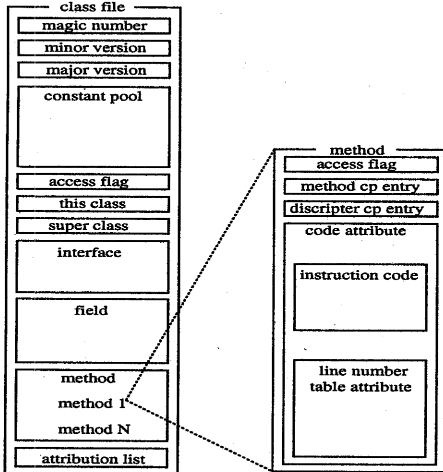

図5 クラスファイルの構成の概略図

2.4 ループアンローリング

アンローリングを適用するシステムを Java と C で作製した。このシステムは入力として次の 3 つを読み込み、アンロールされた新しいクラスファイルを出力する。

- ループを含むクラスファイル
- アンロール対象のループのオリジナルソースファイルでの行番号
- アンロール対象のループのアンロールする段数

実行時にループアンローリングするソースプログラムでは、まず元のクラスファイル名とループアンローリングを適用したいループの場所を指定する。ループの場所は元のソースファイルのループのかかれている行番号で指定する。バイトコードを使いループアンローリングを行なう様子を図 6に示す。なお、図中の演算部はループ内部の計算部分を示す。

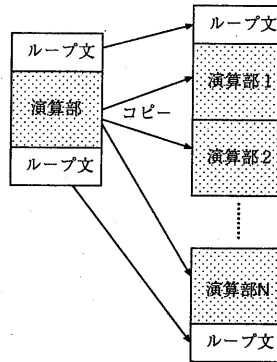

図6 バイトコードを用いたループアンローリング

2.5 ループの除去

ループの除去を適用するシステムを C で作製し、入力として次の 3 つを用いた。

- ループを含むクラスファイル
- ループ除去対象のループのオリジナルソースファイルでの行番号
- ループ除去対象のループのアンロールする段数

以上の3つを入力とし、ループが除去された新しいクラスファイルを出力する。実行時にループを完全に展開して除去するソースプログラムではまず元のクラスファイル名、ループを完全に展開したい場所の行番号、ループの回数を指定する。ループの場所は行番号で指定し、ループ回数は適用を受けるプログラムの中から渡すようにする。

2.6 ブロック化

ブロック化は多重ループに適用する最適化技法である。ブロックサイズは 1 次キャッシュに収まるサイズが理想である。1 次データキャッシュのサイズを $pri.cache$、ブロックサイズを b、行列で用いる変数のサイズを $sizeof(double)$ とするとその関係は次式の様になる。

$$3 \times b^2 \times sizeof(double) = pri.cache \quad (1)$$

これよりブロックサイズ b は次のようになる。

$$b = \left\lfloor \sqrt{\frac{\text{pri.cache}}{\text{sizeof(double)} \times 3}} \right\rfloor \quad (2)$$

ブロックサイズがこの値以下であれば、1 次データキャッシュの容量に収まる。上記で求めたサイズを元にブロック化を行なっていくが、実行する計算機が異なる場合ブロックサイズも変わる可能性があるため、実行時にキャッシュのサイズを調べ、ブロックサイズを求めることで可搬性を持たせる事ができる。

2.6.1 実行時のプログラムの流れ

実行時のプログラムの流れは図7の様になる。図7のClass0,Class1 が元のクラスファイルで Class1_x が実行時に生成されたクラスファイルである。Class1 は最適化を適用するループで構成されており、Class0 は最適パラメータの検索ルーチンとその他の部分で構成されている。また、これらのクラスは事前に手動で作成しておく。

(1) アンローリングの段数の検索範囲を決定する。ここでは start,end,step とし、start から end まで step 刻で検索を行なう事とする。
(2) まず start 段のアンローリングを行いその実行時間を計る。
(3) 次にアンローリング段数を step 分だけ増やし実行時間を計る。
(4) アンローリング段数が end になるまで 3 を繰り返す。
(5) すべての時間を計り終えたらその中で実行時間の一番速いアンローリング段数を選ぶ
(6) まだ演算を終えていない場合は上記で求めた段数でアンローリングしたクラスを使い最後まで計算を行う。

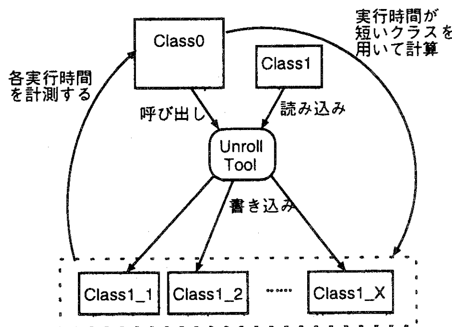

図7 実行時ループ最適化の流れ

3. 評　　価

本章では実装したプログラムの評価を行なう。始めに実行環境について述べ、次に行列積への適用、そしてSciMark2.0 への適用と結果について述べ、最後に考察を述べる。

3.1 実行環境

3 章で実装したプログラムを以下の環境で実行させた。

- Sun Workstation
  - CPU: UltraSparcII 360MHz
  - Cache: 内部1次データキャッシュ16KB、内部1次命令キャッシュ16KB、内部2次キャッシュ4096KB
  - Memory: 512MB
  - OS: Solaris2.6
  - Java: Sun-JDK1.2.1
- PC/AT 互換機
  - CPU: PentiumIII 850MHz
  - Cache: 内部1次データキャッシュ16KB、内部1次命令キャッシュ16KB, 内部2次キャッシュ256KB
  - Memory: 128MB
  - OS: Linux
  - Java: Sun-JDK1.2.1,IBM-JDK1.1.8

3.2 行　列　積

本節では行列積への実行時ループアンローリングを適用した時の手順を図を用いて述べ、その後実行時ループアンローリング、ブロック化の結果について述べる。なお、測定に用いた行列積の大きさは 512×512 である。

3.2.1 実行時ループアンローリング

行列積のプログラムに実行時のアンローリングを適用した時の実行手順を述べる。

オリジナルのクラスは検索時に作成したクラスを再利用するため、Class0 と Class1 に分けられており、Class1 が行列積の 3 重ループの内側 2 重ループであり、Class0 が外側ループの部分である。Class1_5 から Class1_50 は実行中にクラス変換プログラムによりClass1 の内側ループにアンローリングを適用し新たに作成したクラスである。

行列積のループの最外側の部分は Class0 に含まれており、時間の計測は最外側ループの 1 回で行なう。そのため、時間計測のために行なった演算は行列積の演算の一部となるため、余分な計算は行なわれていない。

行列積への実行時のループアンローリングを適用した時の結果を表 1、表 2に示す。表 1が C でアンローリング適用システムを作成した時のものであり、表 2が Java で作成したときのものである。表中の A,B,D はそれぞれ、プログラムに変更を加えていない行列積、実行時ループ最適化を行った場合、D は B で得られた結果と同じ最適化を初めから適用した場合の性能を MFlops で示し、C は実行中に新たにクラスファイルを作り出すのにかかった時間を示している。また、表中の (Sun) は Sun の JIT コンパイラを使用していることを示し、(IBM) は IBM のコンパイラを使用していることを示す。

表1 行列積への適用: ループアンローリング (C)

	A	B	C	D
SunWorkstation	4.72	5.98	0.20	6.04
PC/AT 互換機 (Sun)	10.30	11.90	0.074	11.91
PC/AT 互換機 (IBM)	12.30	12.82	0.33	12.46

行列積への実行時ブロック化を適用した時の結果を表3に示す。

表 2　行列積への適用: ループアンローリング (Java)

	A	B	C	D
SunWorkstation	4.72	5.63	2.32	6.04
PC/AT 互換機 (Sun)	10.30	11.09	0.96	11.91

表 3　行列積への適用: ブロック化

	A	B	C	D
SunWorkstation	4.72	4.69	0.043	4.73
PC/AT 互換機 (Sun)	10.30	14.9	0.01	14.97
PC/AT 互換機 (IBM)	12.30	31.44	0.03	31.60

3.3 SciMark2.0

SciMark2.0[4] ベンチマークプログラムへの実行時ループ最適化を適用した時の手順と結果について述べる。SciMark2.0 には5つのプログラムがあり、今回適用したプログラムは SparseCompRow,SOR の2つである。

3.3.1 SparseCompRow

SparseCompRow で行なわれている演算は疎行列とベクトルの積である。このプログラムには実行時のループの除去を適用した時の結果を表4に示す。

表 4　SciMark2.0 への適用: SparseCompRow

	A	B	C	D
SunWorkstation	16.15	25.34	0.028	25.50
PC/AT 互換機 (Sun)	29.8	55.90	0.009	55.91
PC/AT 互換機 (IBM)	115.6	105.4	0.029	117.6

3.3.2 SOR

SOR で行なわれている演算に実行時ループアンローリングを適用した結果を表5に示す。

表 5　SciMark2.0 への適用: SOR

	A	B	C	D
SunWorkstation	40.76	43.29	0.191	43.40
PC/AT 互換機 (Sun)	85.9	89.4	0.139	89.54
PC/AT 互換機 (IBM)	221.7	205.5	0.334	255.97

3.4 考　察

本節では、以下の3つの点について比較し、実行時のオーバーヘッドの割合も含め実行時の最適化の有効性について考察を行なう。

- 最適化していないもの
- 動的最適化するもの
- 静的最適化されているもの

表6に SunWorkstation の、表7に PC/AT 互換機の実行時最適化を適用したプログラムでの適用前のプログラムの実行時間に対する適用後のそれらの割合と、最適化済に対する実行時最適化の割合、そして割合の項目に全体に対するクラス作成時間、または検索時間の割合を示す。また、表9に Java で作成したアンローリングシステムを行列積に適用した時の各計算機についての同様の内容を示す。表中の E,F は前節までで示した結果よりそれぞれ、$\frac{\text{適用後}}{\text{適用前}}$、$\frac{\text{適用後}}{\text{最適化済}}$ を示し、G は実行時最適化にかかる時間の実行時間に対する割合 (%) を示す。

表 6　項目の比較:SunWorkstation

	E	F	G
行列積: ループアンローリング	1.27	0.99	0.90
行列積: ブロック化	0.99	0.99	0.16
疎行列とベクトルの積	1.61	0.99	0.028
SOR 法	1.15	1.00	0.19

表 7　項目の比較:PC/AT 互換機 (Sun)

	E	F	G
行列積: ループアンローリング	1.15	0.99	0.93
行列積: ブロック化	1.45	0.99	0.24
疎行列とベクトルの積	1.87	0.99	0.019
SOR 法	1.04	1.00	0.12

表 8　項目の比較:PC/AT 互換機 (IBM)

	E	F	G
行列積: ループアンローリング	1.04	1.02	0.03
行列積: ブロック化	2.55	0.99	0.007
疎行列とベクトルの積	0.91	0.90	0.038
SOR 法	0.926	0.91	0.23

表 9　項目の比較: 行列積への実行時ループアンローリング適用 (Java)

	E	F	G
SunWorkstation	1.19	0.93	10.21
PC/AT 互換機	1.08	0.93	9.45

表6、7の第2列より、実行時の最適化を行なったものは最適化を行なっていないものに比べ行列積のループ最適化を行った場合、SunWorkstation と PC/AT 互換機で Sun の JDK を使用した場合に効果が現れている。SunWorkstation の行列積のブロック化では誤差程度で適用前と適用後に差が見られないが、これは2次キャッシュが 4096KB であり行列すべてが2次キャッシュ内に収まるためだと思われる。

疎行列とベクトルの積、SOR 法でも SunWorkstation と PC/AT 互換機で Sun の JDK を使用した場合効果が現れている。

Java でループアンローリング適用システムを作成した場合、表9の第2列より効果が現れているのが分かるものの、第4列を見ると実行時における最適化パラメー

タ検索時間の割合が両方の計算機で 10% 近くとかなり高くなっている。このことより Java で作成したループローリング検索システムを使用した場合はオーバーヘッドが大きく無視できない。

実行時最適化のオーバーヘッドは、表1から表5の検索時間、クラス生成時間を見ると十分小さくまた、初めから最適化されたものと比較しても表6、7より 1% 以内となっており、オーバーヘッドが無視できる大きさになったといえる。

IBM の JIT を使用した場合、性能が低下する場合があるのは、今回の手法を用いるとクラスが分割され、動的クラス呼び出しを用いたために最適化が不可能になったためと思われる。また、今回ループアンローリングとブロック化を同時に適用できなかったのはプログラムの仕様によるもである。

4. おわりに

本稿では、実行時にループ最適化を Java のクラスファイルに適用する手法を提案した。今回は最適化を適用するプログラムに行列積、SciMark2.0 ベンチマークを採用し、実行時ループ最適化のクラスファイルへの適用を行ない、その評価をした。

自動でクラスを変換する時間は極めて短くオーバーヘッドを無視できる大きさまで小さくする事ができた。また、実行時ループ最適化による効果もそれぞれの適用したプログラムにおいて確認する事ができた。今回はループアンローリング、ブロック化をそれぞれ単独で実行時に適用を行なっただけだが、この結果よりループの最適化を実行時情報を用いて行なうことの有用性を示すことができた。

今後の展望として、今回手動で行なった以下の部分の自動化が望まれる。

- 実行前のクラス作成。
- アンローリング段数の検索範囲の決定。
- ループの自動認識

謝辞　研究を行なう上で、有益な助言を頂いた広島市立大学コンピュータアーキテクチャ講座の諸氏に感謝致します

参考文献

1) R.Clint Whaley and Jack J.Dongarra: "Automatically tuned linear algebra software", IEEE Supercomputing, 1996.

2) 阪口 陽祐: "Java 言語による実行時プログラム変換を用いた高速化手法", 広島市立大学情報科学部情報工学科卒業論文, 1999.

3) 窪田 昌史、阪口 陽祐、津田 孝夫: "実行時情報を用いた性能最適化手法", 情報処理学会研究報告,99-HPC-77,P23,1999.

4) Roldan Pozo, and Bruce Miller: "SciMark 2.0", National Institute of Standards and Technology.http://math.nist.gov/scimark2/