Java仮想マシンのベクトル化と動的最適化

鈴木信一郎† 梅谷征雄††

Java仮想マシンの実行速度を上げるために、本研究では静的ベクトル最適化と動的ループ最適化を検討した。静的ベクトル最適化では、Java仮想マシンにベクトル処理機能を実装し、手動でJavaバイトコードを書き換えて実行した。動的ループ最適化では、動的にループに対して命令数の少ない疑似命令を作成し、ループを疑似命令で実行することにより読み込み回数を減らした。KaffeVMに各最適化を実装しsimark2ベンチマークで測定した結果、オリジナルな実行に対してそれぞれで1.9倍、5.7倍の高速化を達成した。

Vector Execution and Dynamic Optimization of Java Virtual Machine

SHINICHIRO SUZUKI† and YUKIO UMETANI††

We propose static vector optimization and dynamic loop optimization for speedups of Java Virtual Machine (JVM). We designed a vector execution method to JVM and rewrite the JavaByteCode for testing static vector optimization. Also, we designed a dynamic loop optimization method that reduce the number of instructions in loop by substitution for JavaByteCode. As a result, we achieved 1.2 to 5.7 times speedup to the original execution.

1. はじめに

オブジェクト指向プログラミング言語であるJava言語は、ネットワークコンピューティングを設計思想の中心に置きており、現在インターネット上のアプリケーション配信言語として急速に普及しつつある。Javaの実行環境は図1に示すように、ソースコードをコンパイルして得られたJavaバイトコードをJava Virtual Machine (JVM)と呼ばれる仮想マシンで解釈実行する。このような実行環境は、ユーザに以下の特徴を与える。

- Javaバイトコードのサイズは他言語から得られる実行形式ファイルより小さく、プログラム転送が容易である。
- Javaバイトコードを安全に実行することができる。
- Javaの実行はプラットフォームに依存せず、移植性に優れる。

しかしながら実行速度がFortranやCなどのコンパイラを用いるプログラム実行より遅い欠点がある。

通常の解釈実行方式は、インタプリタ方式またはJITコンパイラ方式である。インタプリタ方式はJavaバイトコードを逐次解釈実行する方式であり、実行速度は遅いが実行に必要なメモリ量は少ない。JITコンパイル方式はJavaバイトコードを必要に応じて部分的にネイティブコードへ動的変換する方式であり、コンパイラのロードやネイティブコードの格納などのため十分なメモリがある場合には、高い実行速度を得ることができる。

そこでインタプリタ方式の実行速度の改善をはかることとする。特に数値計算プログラムには繰り返し回数の多いループが存在し、プログラム全体の実行時間の大半を占めている。このループに最適化技術を適用することにより、実行時間の高速化をはかることができる。

最適化技術にはループ交換、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などがある。このような最適化技術の多くはソースコードまたは Java バイトコードの書き換えによって適用されることが多い。ソースコードや Java バイトコードを書き換えることを、上記で挙げた利点を損なうことになりかねない。

本研究では、必要なメモリ量が少ないインタプリタ方式において実行速度の高速化を試みる。特に Java バイトコードを内部的に書き換えてベクトル化を適用してみる。そしてその効果をもとに、Java 言語の利点を損なうことなく実行できる動的最適化を試みる。最終的に静的最適化と動的最適化の比較と評価を行い、これらの最適化手法についてまとめる。最適化を適用する JVM には KaffeVM1.0.6(Java VM) を用いる。ベクトル化には、数学的ベクトル化として広く使われている scimark2 ベンチマークの scimark2 ベンチマークは以下に示す 8 つの数学的演算から構成される。

- FFT 変数 Fourier 変換に関する変換を高速に実行するアルゴリズム、高速 Fourier 変換を行う
- SOR Gauss-Seidel 法において過補正を施し将来の補正を先取りする、逐次過緩和法 (Successive over-relaxation) を行なう
- Monte Carlo 数学的問題に乱数を用いた無作為抽出を利用するモンテカルロ法を行なう
- SparseMatmul 残行列とベクトルの積を実行する
- LU 1 つの行列を下三角行列と上三角行列に分解する LU 分解を行なう

2. ベクトル最適化

ベクトル化は繰り返し演算を構成する複数の命令を融合して 1 つの命令にまとめる操作であり、元来はスーパコンピュータの機構を用いたものである。例えば Java 言語での以下のように演算を行うと、

```java
for (i = 0; i < 100; i++)
    c[i] = a + b[i];
```

演算部分のバイトコードを 100 回読み込んで実行することになる。これは Java バイトコードの読み出しと解釈にほとんどどの時間を費やすインタプリタにおいて、多くの時間を浪費することになる。そこでベクトル演算を行うバイトコードを内部的に用意すると

```java
\overline{c} = a + \overline{b}
```

と表され、1 度の読み込みで演算を実行することができる。Java バイトコードの読み込み回数を減らすことにより実行速度の高速化が期待できる。

2.1 静的ベクトル最適化の実装

静的ベクトル最適化を行なえるように、KaffeVM に対しても Java バイトコードの命令で使用されていない番号 (0x55) にベクトル命令を登録する。ベクトル命令は 2 バイト命令で、後半 1 バイトでベクトル操作を示す。用意したベクトル操作の種類は下記の通りである。

- ベクトルとベクトルまたはスカラーとベクトルによると四則演算を各データ型同士行う基本ベクトル演算
- 各データ型それぞれのベクトル演算と
- 配列の間接参照を直接参照に変換するベクトル演算 (ex. a[i][j] → c[i])

ベクトル演算を行うのに必要なデータは、例えば基本ベクトル演算の場合には図 2 のようにスタックへ積む。

![図 2 ベクトル実行によるスタック変化](image-url)

トップにベクトル長 (v-length)、そして第 2 オペランドと第 1 オペランド、出力オペランド各々のベクトール情報を積む。各オペランドに積む情報は、インデックス数 (index-num)、各インデックスの始めの値 (index) と増加量 (increment)、最後に配列参照 (arrayref) となる。ベクトル演算は、スタックに積まれたベクトール情報を読み取りて実行する。演算実行後は、出力オペランドに対する情報はスタックに積み残し、必要ない場合にはポップ命令で取り除く。

繰り返し演算のベクトル化にバイコードへの書き換えはインタプリタ内で自動的に行うのが望ましいが、ベクトル化判定と書き換えのコストが発生する。そこで評価の第 1 ステップとして、人手による書き換えを行った。

Java クラスファイルへの書き換えは、まず Djava を用いてアセンプリコードをファイルへ出力する、出力したファイルに対してベクトル演算を行なえるように、手動でアセンプリコードを書き換える。その後 Jas-
min(6)を用いて、書き換えアセンブリコードのファイルをJavaクラスファイルに戻す。ベクトル化はJasminを用いてJavaクラスファイルに戻すことができないので、代わりに適当な命令を2バイト分記述しておき、バイナリエディタを使用してベクトル命令に書き換える。

もしベクトル計算でオペランド数が3つ以上存在する場合には中間ベクトルが必要とるので、ローカル変数の数を増やし、中間ベクトルの変数をベクトル演算前に定義しておく。

2.2 実行結果

scimark2ベンチマークを構成する5つの数値演算のうち、ベクトル演算が存在しないMonte Carloを除く4つの数学的演算に対してベクトル最適化を施した。ただしSORに対しては配列にアドレス依存が存在するため、図3のようにアドレス依存が存在する演算と存在しない演算を分けて、アドレス依存が存在しない演算に対してのみベクトル演算を適用した。

for (int j = 1; j < Nm1; j++)
+ one_minus_omega * Gi[j];
+ one_minus_omega * Gi[j];

Gi[j] = omega_over_four * (Gim1[j] + Gip1[j] + Gi[j-1] + Gi[j+1])

図3 SORのアドレス依存

静的ベクトル最適化の実行結果を表1に示す。表中のintrpはKaffeVMのオリジナルの実行時間を演算数で計って算出したMFLOPS値を、vectorはKaffeVMに静的ベクトル最適化を実装した場合のMFLOPS値を示す。vec/intrはvector_intrの比である。実行環境はCPU: P-III 800MHz,Memory:256M,OS:Linux2.2.19である。

表1 静的ベクトル最適化の実行性能（単位 MFLOPS）

<table>
<thead>
<tr>
<th></th>
<th>intrp</th>
<th>vector</th>
<th>vec/intr</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>1.0663</td>
<td>1.3381</td>
<td>1.2620</td>
</tr>
<tr>
<td>SOR</td>
<td>3.9051</td>
<td>4.5371</td>
<td>1.1618</td>
</tr>
<tr>
<td>Sparse</td>
<td>1.9460</td>
<td>3.0449</td>
<td>1.5647</td>
</tr>
<tr>
<td>LU</td>
<td>2.4319</td>
<td>4.8462</td>
<td>1.9928</td>
</tr>
</tbody>
</table>

LUが一か所へ効果が現れ、1.99倍もの速度向上が見られた。あまり大きな効果が得られなかったのはFFTとSORであるが、それでも1.26倍と1.16倍の速度向上が見られた。

2.3 考察と動的最適化方法の検討

大きな効果を示すことができたLUとSparse matrixの数学的演算は、負荷の大きい部分が2重ループになっており、ベクトル最適化を施した最内ループに対する負荷を軽減することができたことによると考えられる。

同じようにFFTもベクトル最適化を施した部分が2重ループになっているが、これは数値演算の慣性が上がらないため、これ2重ループの外側のループの方が最内ループのベクトル長より長いためと考えられる。またSORも大きな効果を得ることができなかったが、ベクトル化を行なえるようにするために演算を2つに分けたことが原因と考えられる。ベクトル化を行なえなかった部分は通常のループ実行を行なっているのと、ベクトル化の恩恵を受けない。

また全体としてJavaバイトコードが大きく、これはベクトル情報を使うと加算のためのロード命令を多く必要とするためである。FFTのように最内ループでの計算がある場合やSORのようにオペランド数が多く中間ベクトルを定義しなければならない場合、それに伴うスタックヘッダ領ロード命令を数多く必要とする。サイズの増大が多くを占めるスタックへのロード命令による時間の浪費が実行速度の高速化の妨げになる。

しかしながらベクトル最適化を施すことのできないMonte Carlo以外はそれなりの効果が現れた。このベクトル最適化を動的に行なう方法として、1度バイトコードレベルでループを実行することにより最内ループを見つけ出し、最内ループに対するベクトル演算を行う方法を考えた。しかしながらこの方法の実装には以下に示す課題が存在した。

- 演算のオペランド数が3つ以上存在するときには中間ベクトルを必要とするため、ローカル変数の最大値を増やす必要がある。
- 中間ベクトルの存在が実行速度向上の妨げになる。
- SORのように配列のインデックスを使ってアドレス依存がある場合、手続きで行なったようなアドレス依存が無い部分のみベクトル最適化を行なうといった実装の実現が難しく、また判定に時間を要する。
- for文のループ条件が複雑な場合、ベクトル長を求めることが難しい。

これらの課題を解決する最適化方法として、ループ最適化を検討する。
3. ループ最適化

ループ最適化では、ループ内の Java パイトコードを内部的に命令数のない疑似命令に置き換え、疑似命令を実行することにより実行速度の高速化をはかる。疑似命令は Java パイトコードを実行した場合と同じ動作をするため、ベクトル最適化のように配列のインデックスにあるアドレス値を気にする必要が無く、ベクトルを求める必要もない。

JVM は実行時間のほとんどを Java パイトコードの読み出しと解釈に費やすため、疑似命令数をできる限り少なくすることが実行速度の高速化へつながる。

3.1 疑似命令仕様

多くのプログラムに対する Java パイトコード命令と、一番多く存在する命令は Load/Store 命令である。特に数値計算プログラムでは Load 命令が多く存在する。例えば

\[ C[i] = A[i] \times B[i] \]

のような 2 つオペランドの配列演算を実行する際には、以下のよう Java パイトコード群になる。

(1) ローカル変数から配列 C の参照をスタックに積む。

(2) ローカル変数から配列 C のインデックス値 i をスタックに積む。

(3) ローカル変数から配列 A の参照をスタックに積む。

(4) ローカル変数から配列 A のインデックス値 i をスタックに積む。

(5) スタックに積まった 2 つの値を取り除き、配列 A[i] の値をスタックに積む。

(6) ローカル変数から配列 B の参照をスタックに積む。

(7) ローカル変数から配列 B のインデックス値 i をスタックに積む。

(8) スタックに積まった 2 つの値を取り除き、配列 B[i] の値をスタックに積む。

(9) スタックに積まった 2 値を取り除き、乗算結果をスタックトップに積む。

(10) スタックトップの値を配列 C[i] へ書き込む。

この場合には 10 命中 8 命令がロード命令となっている。

疑似命令を作る際は、ローカル変数からスタックに値を積む工程を無くし直接ローカル変数の値を用いて演算を行うことにより、大幅な疑似命令数削除が可能になる。またスタックに値を積む工程が無くなることは実行速度の高速化にもなる。上記で示した Java パイトコード群を疑似命令群にすると以下のようになる。

(1) 配列 A[i] と配列 B[i] の乘算結果をスタックトップに積む。

(2) スタックトップの値を配列 C[i] に書き込む。

疑似命令は 16 ビット×2 命令で、始めの 16 ビットでどのような疑似命令かを示し、後半 16 ビットは 2 つに分けて 2 つのオペランド参照を示す。上記で示した疑似命令群のように Load 命令を削除するために、Java パイトコードから疑似命令を作り出すときに、配列に対する情報は別途取っておくようにした。配列へのオペランド参照は、別途取って出ていた配列情報への参照に用いる。配列情報は以下の構成になっている。

- 配列参照のあるローカル変数
- 配列の次元数
- 各配列インデックスのデータ (次元数を含む) など

この疑似命令仕様に従って上記に示した (1)〜(10) の Java パイトコードを疑似命令 (I)(II) へ変換すると、図 4 で示す構成になる。 (1)(2) から配列 C の配列情報を作成し、(3)(4)(5) から配列 A の配列情報、(6)(7)(8) から配列 B の配列情報を作成し、(9) から (I) の疑似命令を作成し、(10) から (II) の疑似命令を作成する。

疑似命令の演算実行時に配列から値を取り出すときは、配列情報をもとにスタック操作を行わずで値を取り出すことができる値に対して内部スタックを利用して演算を実行する。ループ条件も疑似命令にして実行することにより、配列インデックスを更新していく。

3.2 動的ループ最適化の実装

for ループを Java パイトコードへコンパイルすると必ず出されようなる組み合わせが図 5 である。KaffeVM の各命令に対する動作を記述するファイルに対して，goto 命令に動的ループ最適化を行うか否かを判断するフラグ
3.4 考察
全体として期待通りの結果を取ることができたが、特に多層ルーブがあり計算のオペランド数も多く存在するFFTとSORでは効果が大きかったのは、Javaバイトコードの命令数よりも疑似命令の方が命令数を大幅に削減できたことによると考えられる。また、多層ルーブ時の疑似命令を再利用することによる効果も大きい。
LUにも2重ループが存在するが、外側ループにある条件分岐により最内ループが変わるため疑似命令の再利用を実行することができなかった。LUとSparse matmultに計算のオペランド数が少ないが、それでも疑似命令化によるコード削減の効果が得ることができた。

4. 比較評価と考察
静的ベクトル最適化と動的ループ最適化の実行性能を図3に示す。

<table>
<thead>
<tr>
<th>表3 近似量化方法の結果</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
</tr>
<tr>
<td>SOR</td>
</tr>
<tr>
<td>Sparse matmult</td>
</tr>
<tr>
<td>LU</td>
</tr>
</tbody>
</table>

静的ベクトル最適化では効果を得ることができなかったFFTとSORが、動的ループ最適化では大きな効果を得ることができた。これは動的ループ最適化で2重ループを実行する際に1度最適化を行なったループは以前作成した疑似命令列をそのまま実行するのに対し、静的ベクトル最適化では再度ループを実行するときにベクトル情報やスパン数を削減しさ棒でベクトル実行をしているため、大きな差をつけていける。特にFFTではエクシルループの繰り返し数が大きいため、著しく実行速度に差がでたと考えられる。

Sparse matmultとLUはどちらも期待通りの結果を得ることができたが、動的最適化のためのコストの差が現れ、静的ベクトル最適化の方が高い性能を示した。
総合的に判断すれば、最適化に対する判定や書き換えなどのコストを考慮するとループ最適化の方が有利と言える。

5. 関連研究
ループに対する最適化技術で、ソースファイルや実行形式ファイルを書き換えで最適化を施す研究は今までに数多く存在する。しかしながら動的最適化を施す
す研究は少ない。これは昔から広く普及している C や Fortran にとって動的に最適化を施す利点が存在しないためである。

しかしながら近年急速に普及している Java 言語は動的に最適化を施すことにより、Java 言語の利点を活かしたまま実行速度を上げることができる。実行時に得られるプロファイル情報を利用して、複数のアンローリング段数候補から最適と思われるものを選び出してそれらを元に Java クラスファイルを直接書き換える研究がある。

Java バイトコードを実行するときに最適化を行うものとしては、実行コードがハードウェアに直接実行されるようにするものがあるが、Java に基づくものです。Java バイトコード実行時にデータ再利用を適用し、実行性能を上げるために再利用表に関する調査を行なったところ高いヒット率の結果が得られている。

その他の方法は、基本的に切断計算プログラムについて最適なループアンローリング手法を検討した。ただし、データ使用についてはあまり効果が得られていない。

その他には、基本的な行列計算プログラムについて最適なループアンローリング手法の検討を行なったが、実行性能が増大する限り、ループに対する読み込みを 1 度にすることで実行速度の高速化を検討した。その結果、Java バイトコードがわずかに大きく増大してしまうが、最適化手法を適用したベンチマーク全てで実行速度が速まった。動的バイトコード最適化による方法では、Java バイトコード実行時に各ループ内コードを命令数の少ないう索引命令を生成して実行し、各ループ毎に生成されたループを命令命令を再利用することにより実行速度の高速化をはかった。これにより Java バイトコードを外部的に書き換えることに無く最適化を実装することが可能であり、Java バイトコードの読み込みを減らすことで実行速度の高速化を得ることが明示した。

以下は、静的ベクトル最適化で Java バイトコードを読み込んで自動的に書き換えるツールを作成したい。また、ベクトル化でなくループ交換、ループアンローリング、多重ループのタイリング、伝算アルゴリズムなどといった他の最適化手法を組み合わせた静的最適化を実行するものである。動的ループ最適化においては、さらなる効果を発揮できる新たな索引命令生成や最適化手法を検討した。

謝辞 研究を行なう上で、数多くの有益な助言を頂いた静岡大学情報学部情報科学科の神川友宏拝師に感謝致します。

参考文献