計算機アーキテクチャ　147－19
ハイパフォーマンス　89－19
コンピューティング
（2002．3．8）

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構を実装し，手動でJavaバイトコードを書き換えて測定した．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scimark2ベンチマークで測定した結果，オリジナルな実行に対しそれぞれ最大で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据えており，現在インターネット上のアプリケーション記述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[1)]．

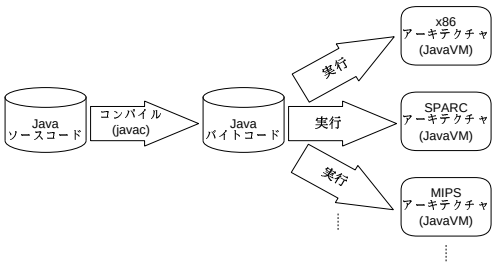

図1　Java実行形態

- 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に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法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．

最適化技法にはループ交換，ループアンローリング，

† 静岡大学大学院 情報学研究科
Graduate School of Information, Shizuoka University
†† 静岡大学 情報学部 情報科学科
Department of Information Science, Faculty of Information, Shizuoka University

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法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ものが多い[2]. ソースコードや Java バイトコードを書き換えることは, 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. 始めに 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. そしてその評価をもとにして 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. 最後に静的最適化と動的最適化の比較と評価を行ない, これらの最適化方法についてまとめる. 最適化を適用する JVM には KaffeVM1.0.6[3] (KaffeVM) を用いる. ベンチマークには, 数学的ベンチマークとして広く使われている scimark2 ベンチマーク[4] を対象とする. scimark2 ベンチマークは以下に示す 5 つの数学的演算から構成される.

FFT 離散 Fourier 変換に関連する変換を高速に実行するアルゴリズム, 高速 Fourier 変換を行なう

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, 遂次過緩和法 (Successive over-relaxation) を行なう

Monte Carlo 数学的問題に乱数など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

SparseMatmult 疎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へ分解する 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, 元来はスーパーコンピュータの機構を使うための技術である. 例えば Java 言語で以下の繰り返し演算を行なうと,

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

演算部分のバイトコードを 100 回読み込んでは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読にほとん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

$$\vec{c} = a * \vec{b}$$

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 Java バイトコードの命令で使用されていない番号 (0xe5) 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別は下記の通りである.

- ベクトルとベクトルまたはスカラとベクトルによる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総和演算
- 配列の間接参照を直接参照に変換するベクトルムーブ (ex. a[b[i]] → c[i])

ベクトル実行を行な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.

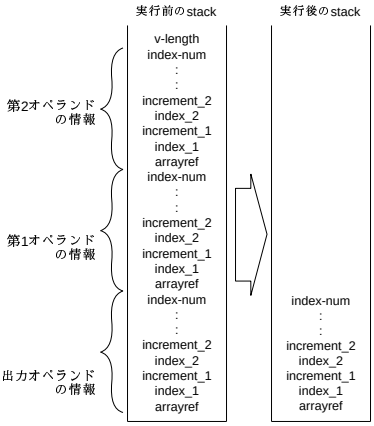

図 2 ベクトル実行によるスタック変化

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ル情報を読み取って実行する. 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バイト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[5] を用いてアセンブ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, 手動でアセンブリコードを書き換える. その後 Jas-

min[6] を用いて, 書き換えた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. ベクトル命令は 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.

もしベクトル演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を必要とするので, ローカル変数の最大値を増やし, 中間ベクトル用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 つの数学的演算のうち, 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に分け, 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.

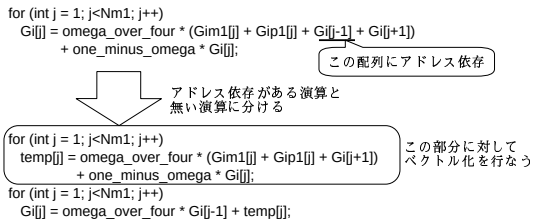

図 3 SOR のアドレス依存

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.vect/intr は vector/intrp 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)

	intrp	vector	vect/intr
FFT	1.0603	1.3381	1.2620
SOR	3.9051	4.5371	1.1618
Sparse	1.9460	3.0449	1.5647
LU	2.4319	4.8462	1.9928

LU が一番効果が現れ,1.99 倍もの速度向上がみられた. 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 倍の速度向上がみられた.

2.3 考察と動的最適化方法の検討

大きな効果を得ることができた LU と Sparse matmult 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, これは思うように実行速度が上がらなかった. これは 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で, 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なる. これはベクトル情報をスタックへ積む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多くの演算がある場合や 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, それに伴いスタックへ値を積むロード命令を数多く必要とする. サイズの肥大の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, 最内ループに対して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にアドレス依存が存在する場合, 手動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機構の実現が難しく, 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式として, 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 バイトコードを内部的に命令数の少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 バ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依存を気にする必要が無く, ベクトル長を求める必要もない.

JVM は実行時間のほとんどを Java バイトコードの読み出しと解読に費やすため, 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 バイトコード命令で, 一番多く存在する命令は Load/Store 命令である[7]. 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] * B[i]

のような 2 オペランドの配列演算を実行する際には, 以下のような Java バ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) スタックに積まれた 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) スタックに積まれた 2 つの値を取り除き, 配列 B[i] の値をスタックに積む.
(9) スタックに積まれた 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 命令中 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なうことにより, 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 バイトコード群を疑似命令群にすると以下のようになる.

(I) 配列 A[i] と配列 B[i] の乗算結果をスタックトップに積む.
(II) 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 バイトコードから疑似命令を作りだすときに, 配列に対する情報は別途取りだしておくようにした. 配列へのオペランド参照は, 別途取り出してお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 (次元数分存在する)

この疑似命令仕様に従って上記に示した (1)〜(10) の Java バ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

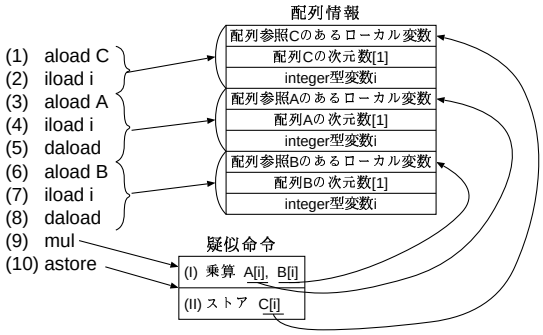

図 4 疑似命令への変換

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 の配列情報を作成する.(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に値を取り出してくる. 直接取り出してきた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 バイトコードへコンパイルすると必ず出現する組み合せが図 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なえるかを判断するフラグ

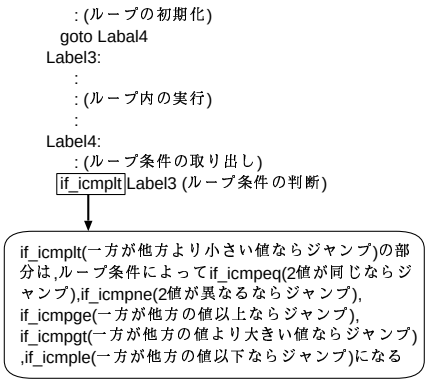

図 5　for ループの Java バイトコード

を作成する. そしてループ条件の判断を行なう if_icmplt 命令等に動的ループ最適化を行なう機構を実装する.

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た後, 最内ループ内のトップから Java バイトコードを読み込んで疑似命令に置き換える. そして残りのループ回数分は疑似命令を実行することになる. もし最適化のためにループ内の Java バイトコードを読み込んでいる最中に, メソッド呼び出しや条件分岐命令など数値操作命令以外が出現したときには最適化を行なえないと判断し, 疑似命令作成を中断して通常のループ実行を行なう. そして再び同じループ実行へ入ったとき, 再度疑似命令を作成しないようにフラグを立てる.

また数値演算プログラムには多重ループが多く存在する. そこで最内ループに対して動的ループ最適化を行なうことができたとき, 再び同じループに入ったときに疑似命令の作成を行なわずに以前の疑似命令を利用する.

3.3 実行結果

動的ループ最適化の実行性能を表 2 に示す. 表中の loop は KaffeVM に動的ループ最適化を実装した場合の性能である.loop/intr は loop/intrp の比である.

表 2　動的ループ最適化の実行性能 (単位 MFLOPS)

	intrp	loop	loop/intr
FFT	1.0603	6.2394	5.8846
SOR	3.9051	10.0228	2.5666
Sparse	1.9460	2.9776	1.5301
LU	2.4319	3.8466	1.5817

一番効果が現れたのは FFT と SOR で, それぞれ 5.88 倍と 2.57 倍もの速度向上を得ることができた.Sparse matmult と LU も 1.53 倍,1.58 倍と期待通りの効果があった.

3.4 考　　察

全体として期待通りの結果を得ることができた. 特に多重ループがあり演算のオペランド数も多く存在する FFT と SOR で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, 多重ループ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行なえるようにできなかった.LU と Sparse matmult 共に演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を得ることができた.

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揃えて表 3 に示す.

表 3　両最適化方法の結果

	intrp	vector	loop
FFT	1.0603	1.3381	6.2394
SOR	3.9051	4.5371	10.0228
Sparse	1.9460	3.0449	2.9776
LU	2.4319	4.8462	3.8466

静的ベクトル最適化では効果を得ることのできなかった 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は 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もベクトル情報をスタックに積み直してベクトル実行をしているため, 大きく差をつけられている. 特に FFT では外側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, 動的最適化のためによるコストの差が現われ, 静的ベクトル最適化の方が高い性能を示した.

総合的に判断して, 両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法で, ソースファイルや実行形式ファイルを書き換えて最適化を施す研究は今までに数多く存在する. しかしながら動的に最適化を施

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からである.

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出してそれを元に Java クラスファイルを直接書き換える研究がある[2].

Java バイトコードを実行するときに最適化を行なうものとしては, 実行方式がハードウェア直接実行方式によるものも研究されている.Java プロセッサを仮定して 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でている[8]. また, 命令畳み込みの適用によりスタックレジスタが 8 本, ローカルレジスタが 16 本という比較的小さい構成でも平均 30.5%の動的ステップ数を削減できた研究もある[9]. ただし, データ投機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段数やタイリングの最適なタイルサイズを求める研究として C 版で存在する ATLAS を Java 処理系で動かす研究も行なわれている[10].

本研究ではそれらとは異なり, 繰り返し演算に対するバイトコード形式の内部的な変換により性能向上させるものである.

6. おわりに

本研究では, ループ実行時に Java バイトコードの読み出しと解読に対する時間を減らして実行速度の高速化をはかる手法として, 静的ベクトル最適化と動的ループ最適化を提案した. 静的ベクトル最適化では, 実行時間の多くを占める最内ループに対する Java バイトコードを書き換えてベクトル化し, 最内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なってしまうが, 最適化を施したベンチマーク全てで実行速度が速くなった. 動的ループ最適化による手法では,Java バイトコード実行時に最内ループを命令数の少ない疑似命令を作成して実行し, また多重ループ時は作成した最内ループの疑似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られることが判明した.

今後は, 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, 遺伝的アルゴリズムなどといった他の最適化技法を組み合わせた静的最適化を作成する予定である. 動的ループ最適化においては, 更なる効果を得ることができる疑似命令仕様や最適化技法を検討したい. また, インタプリタ方式だけでなく JIT コンパイラ方式を両最適化技法適用の対象としていく.

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秡川友宏助手に感謝致します.

参考文献

1) F.Boisvert, R., Moreira, J., Philippsen, M. and Pozo, R.: "JAVA AND NUMERICAL COMPUTING", *COMPUTING IN SCIENCE & ENGINEERING*, Vol. 3, No. 2, pp. 18–24 (2001).
2) 山崎泰伯, 窪田昌史, 津田孝夫: "Java クラスファイルの実行時ループ最適化", 情報処理学会 HPC 研究会報告, Vol. 82, No. 21 (2000).
3) Kaffe.org: Welcome to Kaffe. http://www.kaffe.org/
4) Pozo, R. and Miller, B.: "SciMark2.0", National Institute of Standards and Technology. http://math.nist.gov/scimark2/
5) Pozo, R. and Miller, B.: "Djava", National Institute of Standards and Technology. http://mrl.nyu.edu/meyer/jvm/djava/
6) Pozo, R. and Miller, B.: "Jasmin", National Technology. http://mrl.nyu.edu/meyer/jvm/jasmin.html
7) 渡辺健司, 金田正一, 大津山公平: "Java Virtual Machine の静的・動的解析", 情報処理学会 ARC 研究会報告, Vol. 125, No. 13 (1997).
8) 山田克樹, 中島康彦, 冨田眞治: " 投機的手法を用いたデータ再利用による Java 仮想マシンの高速化", 情報処理学会 ARC 研究会報告, Vol. 139, No. 29 (2000).
9) 重田大助, 小川洋平, 山田克樹, 中島康彦, 冨田治: " 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, Vol. 41, No. SIG 5 (2000).
10) 長谷川広和, 松岡聡, 伊藤茂雄: "Java による並列 LU の性能評価", 情報処理学会 ARC 研究会報告, Vol. 137, No. 15 (2000).