ハイパフォーマンス
コンピューティング 77－5
（1999．8． 2）

実行時情報を用いた性能最適化手法

窪田昌史† 阪口陽祐† 津田孝夫†

本稿では，実行時に得られるプロファイル情報などを用いて，実行時にプログラムを書き換える性能最適化手法について述べる．本手法により，プログラムは様々な実行環境に適応して実行時に最適化される．本手法を適用して，実行環境に適応して自動的にループアンローリングを適用する行列積の Java 言語のプログラムを試作し，実行時間を計測したところ，このプログラムの実行時最適化によるオーバヘッドは小さいものであることが確認された．

Dynamic Optimization Techinique Using Run-time Profiles

ATSUSHI KUBOTA,† YOSUKE SAKAGUCHI† and TAKAO TSUDA†

This paper presents a run-time optimization technique that use run-time profiles. It enables programs to adapt dynamically to different execution environments. We have implemented a matrix multiply test program in Java that automatically apply loop unrolling to itself to adapt to execution environments. We confirmed that the overhead incurred by the run-time optimization is small.

1. はじめに

プログラムの実行性能の高速化手法としては様々なものが提案されている．それらの中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を行なわなければ適用できないものも多い．

例えば，ループ実行の最適化技法として，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など，様々な技法が知られているが，これらの適用の可否は実行される計算機の CPU 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異なる．ループアンローリングを適用する場合，最適なアンローリング段数は実行環境によって異なるため容易に求めることはできない．タイリングを適用する場合も，最適なタイルサイズを求めることは困難である．そのため，様々なアンローリング段数やタイルサイズのプログラムを記述しては実行することを繰り返し，最適な段数や最適なタイルサイズを求めるなどの手法がとられているのが現状であり，これらのパラメータをコンパイラによって自動化することは困難である．

これは，コンパイラによる最適化は，プログラムの実行前にその実行過程，つまり，命令の実行順序やメモリ上のデータのアクセス順序を予測することで行なわれていることに起因する．そこで，本稿では，プログラムの実行時に得られるプロファイル情報や入力データなどの最適化に必要な情報を得て，実行時にプログラムを書き換えて最適化を行なう手法について検討する．

本稿では以下，2 章で実行時にプログラムを書き換えて最適化を行なう手法と，それを実現するプログラム実行環境について述べる．3 章では実行環境に適応して自動的にループアンローリングを適用する行列積のプログラムを Java 言語で記述する例について述べる．4 章では，作成した行列積のプログラムの実行結果を示し，実行時のプログラム書き換えによる最適化の有効性について議論する．5 章で関連研究を紹介し，最後に 6 章で総括を行ない，今後の研究の方向について述べる．

2. 実行時プログラム最適化

2.1 実行時情報を用いた最適化

従来の単一プロセッサ向けコンパイラ，あるいは並列化コンパイラがコンパイル時に静的に最適化を行なう際には，以下のような情報が得られない，あるいは実行時にならなければ正確には求められないため，性能向上の障壁となっている．

- プログラムへの入力による挙動の変化
- 実行環境による違い
  - CPU 性能
  - 記憶装置の容量 (キャッシュ容量，メモリ容量)

† 広島市立大学 情報科学部
Faculty of Information Sciences, Hiroshima City Universty

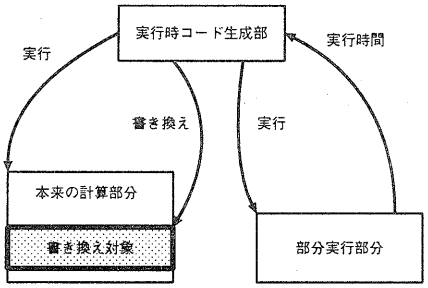

図 1　実行時に書き換えが行なわれるプログラム

- 記憶階層間のデータ転送能力，方式 (キャッシュコヒーレンスのプロトコルの違いなど)
- 並列実行環境下での，通信性能 (バンド幅，レイテンシ)
- マルチプロセス，マルチユーザ環境におけるリソース (CPU，記憶装置，ネットワーク) の取り合い

ただし，これらの情報を実行時に容易にできる場合もある．例えば，ある変数の値を実行時に検査する条件文を用意し，最適化されたコードと，最適化されていないコードへと実行を振り分けるマルチバージョンコードを生成する手法などがある．

しかし，最適化の中には，以下のようにパラメータを変化させなければ最適化できないものがある．

- ループアンローリングにおける，アンローリング段数
- 多重ループのタイリングにおけるタイルサイズ
- 並列化コードのプロセッサ間通信の一括化における，メッセージサイズ

これらは，あらゆるパラメータについてのコードを生成しておくことが不可能であるため，マルチバージョンコードの生成では対処できない．

そこで，次節では，実行時に必要に応じてコードを生成して最適化を行なう手法を検討する．

2.2　プログラムの実行時書き換えによる最適化

我々は，実行中にプログラムを書き換えて最適化を行なうプログラムの実行を支援するプログラム実行環境（コンパイル，実行）の構築を目指している．

図 1 にこの実行環境のもとで実行されるプログラムの各部分の概念図を示す．元のソースプログラムは，実行前の静的コンパイル時に，本来の計算部分と，部分実行部，およびそれらを統括して実行時にコードを書き換える実行時コード生成部へと変換される．このプログラムの実行を開始すると，まず，実行時コード生成部が部分実行部を実行し，その実行時間などの情報を得る．この情報をもとに本来の計算部分を書き換え，書き換えられた計算部分を実行する．必要があれば，部分実行とコードの書き換えが繰り返されることになる．

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

図 2　行列積

このようなプログラム実行環境を実現するためには，以下の点を明らかにする必要がある．

- どのようなアプリケーションのどの部分を実行すれば，最適に近い性能を出すパラメータを検出できるか．
- 最適に近い性能を出すパラメータの検索時間と，得られる性能とのトレードオフはどこにあるのか．

現時点では，多重ループへのループアンローリングやタイリングの適用に絞り，どのループについて，最適なアンローリング段数やタイルサイズなどのパラメータをどのような範囲で，どのくらいの時間検索するかをディレクティブの形式で与えて，図 1 に示すようなプログラムを生成するシステムの構築を目指している．

さて，このような実行時書き換えを行なうプログラムを実行するために，実行時にプログラムを書き換え，書き換えられたプログラムへ実行の制御を移すプログラム実行環境は容易には実現できない．

そこで，我々はプログラムの実行環境として Java 言語とその実行環境を採用した．Java 言語のリフレクション機能の動的メソッド呼び出し (invoke) を使うと，実行前のコンパイル時に存在していないコードへの実行の移行が容易に実現できる．また，JIT(Just In Time) コンパイラにより，Java 言語のソースプログラムから変換された Java のバイトコードは，それぞれの実行環境の機械命令であるネイティブコードに変換されて高速に実行される．

3.　行列積へのアンローリングの適用

行列積の計算を行なう図 2 の 3 重ループの最内側のループを u 段アンローリングすると，図 3 のプログラムが得られる．アンローリングを適用すると，ループの反復回数が減少するため，分岐命令の実行回数も減少し，ループ全体の実行が高速化されることが知られている．

本章では，3.1 節で行列積のプログラムの最内側ループに様々な段数のアンローリングを適用した結果を示す．その結果をもとに，3.2 節で，行列積にアンローリングを適用する例をとりあげ，2.2 節で述べた実行時に書き換えが行なわれる Java プログラムの構成法を示す．

表 1　アンローリング段数による行列積の実行時間 (秒) の変化

アンローリング段数	1	2	3	4	5	6	7	8	9	10
PentiumII	11.52	9.98	9.47	9.95	9.53	9.50	9.52	9.84	9.59	9.64
UltraII	29.31	28.95	28.82	28.78	28.74	28.72	28.58	28.70	28.68	28.90

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++) {
    jj = n%u;
    for (j=0; j<jj; j++)
      c[i][j] += a[i][k] * b[k][j];
    for (j=jj; j<n; j+=u) {
      c[i][j] += a[i][k] * b[k][j];
      c[i][j+1] += a[i][k] * b[k][j+1];
      ...
      c[i][j+u-1] += a[i][k] * b[k][j+u-1];
    }
  }
```

図 3　アンローリングが適用された行列積

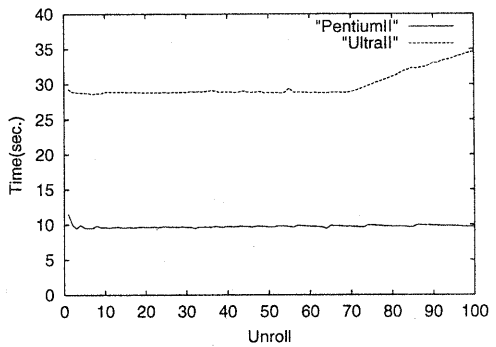

図 4　アンローリング段数による行列積の実行時間の変化

3.1　アンローリングによる効果

図 4 に 512×512 の行列積のプログラムに対し，アンローリング段数を 1(アンロリーングなし), 2 から 100 まで変化させた Java プログラムの実行時間を示す．また，10 段までの実行時間を表 1 に示す．使用した計算機の仕様と，OS, Java(JDK) のバージョンは表 2 に示す通りである (本稿では以下，それぞれの計算機を PentiumII, UltraII と表記する)．PentiumII では，64 段，9.44 秒が最適となる．また，UltraII では，7 段，28.58 秒が最適である．

3.2　実行時アンローリングを行なう行列積プログラム

実行時にアンローリングを行なう行列積のプログラムを構成するにあたり，以下のようなコードの特性を利用する．

(1)　2 から 10 程度の少数のアンローリング段数の実行時間を求めれば，かなり最適に近い実行時間を与える段数が求められる．

(2)　最内側ループの最適な段数を検索するには，プログラムのすべての部分を実行する必要はなく，最内側ループのみを実行すれば，ほぼ最適な段数を求める求めることができる．

(3)　最内側ループは，最適な段数を求めるための部分実行部分であるだけでなく，本来の計算部分にも含まれている．

ここで，(1) は，図 4 と表 1 から，アンローリングの段数を 2 から 5 くらいまで増加させると，実行時間が減少するが，それ以上段数を増加させても大幅な実行時間の減少は見られないことが根拠となる．

さて，これらの (1) から (3) のコードの特性を利用して，実行時にアンローリングを適用するプログラムを構成する．外側ループの最初の数回のイタレーションを，最内側ループのアンローリング段数を変化させながら実行すれば，最適なアンローリング段数の検索と，本来の計算とを同時に行なうことが可能である．最適なアンローリング段数が求められたら，その段数にプログラムを書き換えて，外側ループの残りのイタレーションを実行すればよい．

図 5 に，実行時書き換えが行なわれる行列積プログラムの概略を示す．部分実行部分，かつ，書き換え対象となるのは，最内側の 2 重ループにあたるクラス (Inner2) である．この 2 重ループのうち，最内側のループのアンローリング段数を変化させたコードが実行時に生成される．

ここで，検索用部分実行プログラムを，最内側のループだけでなく，内側の 2 重ループとしたのは，異なる

表 2　使用する計算機の仕様と OS, JDK

PentiumII	
CPU	Pentium II 400MHz
命令キャッシュ	16KB
1 次データキャッシュ	16KB
2 次キャッシュ	512KB
主記憶	512MB
OS	WindowsNT 4.0 SP3
Java	JDK 1.2
UltraII	
CPU	UltraSPARC-II 360MHz
命令キャッシュ	16KB
1 次データキャッシュ	16KB
2 次キャッシュ	4MB
主記憶	512MB
OS	Solaris 2.6
Java	JDK 1.2.1

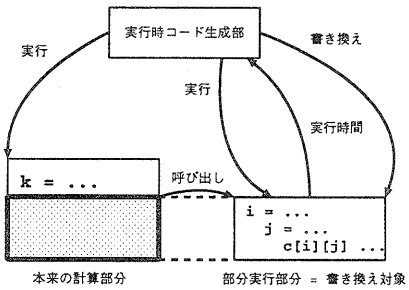

図 5　実行時書き換えが行なわれる行列積プログラム

アンローリング段数の検索用部分実行プログラム間の実行時間の差を容易に計測できるようにするためである．実際には，currentTimeMillis() メソッドを使用して実行時間の計測を行なっているため，ミリ秒単位の実行時間の差が出るようにしなければならない．

3.2.1　実行時アンローリングを行なう行列積のプログラムの流れ

作成するプログラムの実行は以下のようになる．

(1)　最外側ループのステップを k=0 とする．
(2)　アンローリングされていない内側 2 重ループを計算する Inner2 クラスを作成する．
(3)　Inner2 クラスを実行し，その実行時間を time-prev に保存しておく．
(4)　最外側ループのステップをインクリメント (k++) する．
(5)　アンローリング段数を (k+1) として，内側 2 重ループを計算する Inner2 クラスを作成する．
(6)　Inner2 クラスを実行し，その実行時間を time-cur に保存しておく．
(7)　time-prev と time-cur を比較し，time-prev の方が大きい，つまり前回実行した Inner2 より今回実行した Inner2 の方が実行時間が短かった場合，time-prev に time-cur を代入して，(4) へ戻る．
(8)　最適なアンローリング段数を k であると判定し，残りの最外側ループをアンローリング段数 k の Inner2 を呼び出して実行する．

3.2.2　部分実行部分の実行時生成法

最適なアンローリング段数を検索するための Inner2 クラスの生成法，つまり Inner2 クラスのバイトコードを計算中に求める方法として以下の 3 つの方法を採用した．

(1)　bytecode
アンローリング段数を与えると，アンローリングされた Inner2 クラスのバイトコードを生成する．
(2)　javac
アンローリング段数を与えると，アンローリングされた Inner2 クラスのソースコードを生成し，さらに javac でコンパイルしてバイトコードを得る．
(3)　jikes
(2) と同じであるが，javac のかわりに IBM が研究用に開発している jikes[1] を用いてバイトコードを得る (PentiumII のみ)．

(2) および (3) の場合は，ソースコードを出力するだけでよいため，生成ルーチンは簡単になるが，javac や jikes のコンパイルにかかるオーバヘッドが大きくなる．それに比べ，(1) の場合は，バイトコードを直接出力するため，コンパイルにかかるオーバヘッドはなくなるが，バイトコードの生成ルーチンがやや複雑になる．

Inner2 クラスは，もとのプログラムのコンパイル時には存在しないクラスである．そのため，Inner2 クラス内の内側 2 重ループを実行するメソッドの呼び出しには，Java 言語のリフレクション機能で一つである動的メソッド呼び出しを行なう invoke メソッドを用いる．

(1) から (3) のいずれの場合も，生成されたバイトコードは，さらに JIT コンパイラによって実行される計算機のネイティブコードに変換されるため，高速に実行される．

3.2.3　最適なアンローリング段数の検索法

実行中に最適なアンローリング段数を検索する方式として，以下の 2 通りの方式を採用した．

検索法 1　3.2.1 節で述べたように，アンローリングなし，アンローリング段数 2, 3... とアンローリング段数を増加させ，内側の 2 重ループ Inner2 の実行時間が減少から増加へ転じたところで，その 1 つ前の段数を最適な段数と判定する．

検索法 2　アンローリングなし，アンローリング段数 2 から 10 までの内側 2 重ループ Inner2 の実行時間をすべて計測し，実行時間が最小のものを最適な段数と判定する．

後者の検索法 2 はアンローリング段数 10 段までの中では最適な段数を求めることができるが，最適なアンローリング段数を検索する時間が長くなる可能性がある．なお，検索するアンローリング段数を 10 段までに限ったのは，3.1 節で述べたように，10 までのアンローリング段数で，最適な段数が得られるか，あるいは，かなり最適な段数に近い効果が挙げられ，それ以上の検索は，検索時間を長くするのみであまり効果が得られないためである．

それに対して前者の検索法 1 は，最適なアンローリング段数を検索する時間が少なくてすみ，また，検索する段数の範囲をあらかじめ指定する必要もない．ただし，局所解におちいり，最適な段数とは大きく違う段数を最適と判定する可能性がある．

表 3　実行時アンローリングを行なった行列積の実行時間

(a) PentiumII

		検索法 1		検索法 2	
		段数	時間 (秒)	段数	時間 (秒)
bytecode	best	7	10.20	7	10.24
	worst	3	10.83	2	11.67
jikes	best	6	10.83	7	11.03
	worst	3	11.22	1	14.02
javac	best	2	15.70	7	24.25
	worst	6	19.92	2	25.74

(b) UltraII

		検索法 1		検索法 2	
		段数	時間 (秒)	段数	時間 (秒)
bytecode	best	4	29.34	8	29.50
	worst	3	29.39	4	29.65
javac	best	2	33.87	7	44.42
	worst	5	38.53	5	44.63

4. 性能評価

本章では，前章で述べた実行時アンローリングを適用した行列積のプログラムの実行結果を示し，その評価を行なう．なお，使用した計算機は前章と同様に 2 で示した PentiumII と UltraII である．また，使用した行列のサイズはすべて 512×512 である．

表 3 に実行時アンローリングを行なった行列積の計算の実行時間と，得られた最適なアンローリング段数を示す．なお，測定時の実行時間，求められた最適な段数にばらつきがみられたため，それぞれの計算機，最適な段数の検索法，バイトコードの生成法に対して，5 回計測を行ない，実行時間の最も短いもの (best) と最も長いもの (worst) を示した．

4.1 実行時間

PentiumII では，バイトコードを出力する方法により，10.20 秒まで実行時間を短縮することができている．これは，3.1 節で示した最適な段数 (64 段) での実行時間である 9.44 秒には及ばないものの，アンローリングを適用していない場合の 11.52 秒よりも高速となり，最適化の効果が現れていることが示された．

UltraII でも，同じくバイトコードを出力する方法により，29.34 秒となっている．これは，3.1 節で示した最適な段数 (7 段) での実行時間である 29.58 秒や，アンローリングを適用していない場合の 29.31 秒とほぼかわらない実行時間となっている．

ソースコードを出力してコンパイルする方法の場合，jikes を使ってコンパイルすればかなり高速に実行されるが，javac を使った場合は Inner2 クラスのコンパイルに 1.5 秒程度かかるためオーバヘッドが大きい．特に，10 段までアンローリングされたコードを生成してコンパイルする検索法 2 の場合，必ず 10 回はコンパイルが行なわれるため全体の実行時間が大幅に増加してしまう．

表 4　実行時最適化プログラムの実行時間の内訳
(1) は全体の実行時間 (秒)，(2) はそのうちのクラス生成にかかる時間 (秒) である．(2)-(1) は本来の計算にかかる時間 (秒)，(2)/(1) は，クラス生成が全体の実行時間に占める割合 (%) である．

(a) PentiumII

検索法	段数	(1)	(2)	(2)-(1)	(2)/(1)
bytecode					
1	4	10.69	0.05	10.64	0.5
2	5	10.36	0.03	10.33	0.3
jikes					
1	7	10.84	0.70	10.14	6.5
2	7	11.06	0.88	10.18	8.0
javac					
1	3	16.27	5.42	10.85	33.3
2	3	24.95	13.99	10.96	56.1

(b) UltraII

検索法	段数	(1)	(2)	(2)-(1)	(2)/(1)
bytecode					
1	4	29.21	0.18	29.03	0.6
2	6	29.37	0.37	29.00	1.3
javac					
1	3	35.54	6.41	29.13	18.0
2	3	44.75	15.44	29.31	34.5

4.2 最適なアンロール段数の検索

実行と同時に求められたアンローリング段数であるが，表 3 に示すとおり，最適な段数に比べてばらつきがある．特に，検索法 1 では，局所最適解におちいって小さな段数で最適であると判定される傾向にある．しかし，最適な段数の検索時間が短くなるため，全体の実行時間は検索法 1 の方が短くなっている．

ばらつきが生じるのは，Inner2 クラスの 1 回の実行では測定誤差が大きいためと思われる．特に，PentiumII 上では時間計測の最小単位が 15ms 程度とみられ，測定誤差が大きい．

4.3 実行時間の解析

表 3 のプログラムと同じ実行条件で，全体の実行時間とともに，実際の計算にかかった時間と実行時にクラスを生成するためにかかった時間を測定した．その結果を表 4 に示す．

バイトコードを直接出力する方式ではオーバヘッドが非常に小さく，全体の実行時間の 1%前後であるに対し，コンパイルによる方式，とくに javac を使う方式では実行時間の半分近く，あるいはそれ以上を javac によるコンパイラに使われている．これは，javac によるコンパイルは PentiumII と UltraII のどちらでも，1 回あたり約 1.5 秒，jikes の場合は約 0.08 秒かかるためである．

また，最適なアンローリング段数の検索方式を比較すると，バイトコードを直接出力する方式ではオーバ

表 5　内側ループの呼び出し法による実行時間の比較

	(1)no call	(2)call	(3)invoke
PentiumII	11.50	11.98	12.10
UltraII	30.16	29.46	29.45

ヘッドが非常に小さいため，どちらも実行時間にあまり差がないのに対し，コンパイルによる方式では検索法 2 の場合，必ず 10 回分のコンパイル時間がかかるために，実行時間が増大してしまっているといえる．

4.4 動的なメソッド起動のオーバヘッド

本節では，Java 言語のリフレクション機能 (invoke) を用いた動的なメソッドの呼び出しのオーバヘッドを解析する．ただし，1 回あたりのメソッドの呼び出しの前後で経過時間を計測する方法では差がみられないため，ここでは，アンローリングを行なっていない 3 重の行列積のプログラムを，以下の 3 通りの方法で記述し，その実行時間を比較してオーバヘッドを求めている．

(1) no call
すべて 1 つのクラスの 1 つのメソッド内で実行する．

(2) call
内側の 2 重ループを別のクラス Inner2 のメソッドに分け，メソッドを呼び出しながら実行する．

(3) invoke
(2)call と同様であるが，メソッドの呼び出しにリフレクションを用いる．

表 5 は，上記の 3 つのプログラム (1), (2), (3) の実行時間を示したものである．表 5 より，内側の 2 重ループを別のクラスにしたこと，および，それをどのように呼び出したかによる性能低下は小さいことがわかる．これにより，動的なメソッド呼び出しによる性能低下はほとんど生じていないと考えられる．

5. 関連研究

プログラム実行時の最適化の研究は近年，活発に行なわれている．Voss と Eignmann[2]，Diniz と Rinard[3] は並列化コンパイルにおける実行時の最適化について述べている．

Java HotSpot^{TM} Performance Engine[4] は，実行時にプロファイルをとり，ガベッジコレクションの最適化，メソッドのインライン化などを行なう．

基本的な行列計算プログラム，特に行列積について，ループアンローリングの最適な段数，タイリングの最適なタイルサイズを求める研究として ATLAS[5] などがあり，ATLAS の Java 処理系で稼働するバージョン [6] も開発されている．

6. おわりに

本稿では，プログラム実行中に，プログラムの一部分の実行時間をもとに，プログラムの一部分を書き換える最適化を行なう手法について考察した．また，行列積プログラムへの実行時アンローリングを適用し，実行時最適化の有効性の評価を行なった．その結果，Java で記述されたプログラムの内側ループに実行時にアンローリングを適用しながら実行を行なうと，小さなオーバヘッドで実行時に最適化されたコードを生成することが可能であることが確かめられた．

現在，行列積のタイリングの適用において，自動的に最適なタイルサイズを求めてプログラムを書き換える手法についても検討している．

また，現状では，出力された Java のバイトコードは，JIT コンパイラによってネイティブコードに変換されて実行されている．これを，JIT へ何らかの指示を与えて，バイトコードではなく，直接ネイティブコードを出力するようにして，さらにオーバヘッドを削減する手法についても検討している．

謝　辞

研究を行なう上で，有益なご助言をいただ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いたします．

なお，本研究の一部は広島市立大学 特定研究費「データ並列言語に対する自動並列化コンパイラの研究」による．

参考文献

1) IBM. IBM Research Jikes Compiler Project. http://www.research.ibm.com/jikes/.
2) Michael Voss and Rudolf Eigenmann. Dynamically adaptive parallel programs. In *Proc. of Int'l Symposium on High Performance Computing*, pp. 109–120. Springer, May 1999. LNCS 1615.
3) Pedro Diniz and Martin Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Programming Language Design and Implementation(PLDI '97)*, pp. 71–84. ACM SIGPLAN, June 1997. Las Vegas, NV, USA.
4) Sun Microsystems. The Java HotSpot^{TM} performance engine architecture. http://java.sun.com/products/hotspot/whitepaper.html, April 1999.
5) R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proc. of SC98*, November 1998. Orlando, FL.
6) 伊藤茂雄, 松岡聡. 複数の Java 処理系における高性能計算の性能評価にむけて. 情処研報 99-HPC-75-6, 情報処理学会, March 1999.