ハイパフォーマンス
コンピューティング 57−22
（1995．8．25）

パス選択によるソフトウェア・パイプライニング

中西知嘉子　安藤秀樹　原哲也　中屋雅夫

三菱電機（株）
システムＬＳＩ開発研究所
〒664　兵庫県伊丹市瑞原 4-1
e-mail: chikako@lsi.melco.co.jp

ソフトウェア・パイプライニングは、VLIW やスーパスカラ・プロセッサにおける効果的なスケジューリング技術である。従来のアルゴリズムでは、ループの全制御パスをパイプライン化するため、実行頻度の小さい制御パスによる資源占有や長いデータ依存パスによって、性能向上が制限されていた。本論文では、ループの中の実行頻度の高いパスを選択してパイプライン化するアルゴリズムを提案する。本アルゴリズムは、実行頻度の高いパスを最適にパイプライン化できるだけでなく、従来パイプライン化が困難であった最内ループでないループに対してもパイプライン化できる可能性を与える。非数値計算応用のベンチマーク・プログラムを用いて、従来のアルゴリズムとの比較評価を行なった結果、本アルゴリズムの高い有効性を確認できた。

Software Pipelining with Path Selection

Chikako Nakanishi　Hideki Ando　Tetsuya Hara　Masao Nakaya

System LSI Laboratory
Mitsubishi Electric Corporation
4-1 Mizuhara, Itami, Hyogo, 664 Japan
e-mail: chikako@lsi.melco.co.jp

Software pipelining is an effective technique for VLIW or superscalar processors. Previous algorithms for software pipelining schedule instructions of all control paths in a loop. Thus, they produce limited performance improvement due to resource conflict and long data dependence paths caused by infrequently executed paths. This paper proposes an algorithm which pipelines a loop through selecting frequently executed control paths. Our algorithm produces the best optimized pipelined code for frequently executed paths, and also provides opportunity to pipeline outer loops. The evaluation results show that our algorithm is highly effective on non-numerical applications over previous algorithms.

1 はじめに

ソフトウェア・パイプライニングは、VLIWやスーパスカラ・プロセッサにおける効果的なループ最適化技術である。一般に、ソフトウェア・パイプライニングによってスケジュールされたコードは、プロローグ、定常状態、エピローグからなる。ループ回数が多いループでは多くの実行時間が定常状態で費やされる。そのため、ソフトウェア・パイプライニングの目的は、可能なかぎりコンパクトな定常状態を得ることである。

非数値計算応用では、多くのループが条件分岐を含むため、条件分岐を含むループを最適にスケジュールするソフトウェア・パイプライニングのアルゴリズムを見つけることは極めて重要である。先行制約、及び資源制約を同時に満たしつつ、コンパクトな定常状態を得ることができるアルゴリズムの1つに、モジュロ・スケジューリング(MS) [1]がある。しかしながら、MSは、基本的には、ループ・ボディが単一の基本ブロックからなるループを最適化するアルゴリズムであるため、条件分岐命令を含むループには、適用が困難である。これまで、条件分岐を含むループにMSを適用するためには、スケジューリング前に、ループ・ボディをストレート・ライン・コードに変換することで対処する方法が提案されている（例えば、ハイアラキカル・リダクション[1]やプレディケート実行(PE)[2][3]やエンハンスト・モジュロ・スケジューリング(EMS)[4]）。

しかし、これらのアルゴリズムは、どのパスが頻繁に実行されるかを全く考慮していない。そのため、一方のパスで定常状態のサイクル数が決まってしまったり[1][4]、異なったパス上の命令間で資源競合が起こるなどの問題点がある[2]。そのため、実行頻度の低いパスで定常状態のサイクル数が決まってしまった場合のペナルティが大きくなったり、不必要な資源競合のため最適なスケジュールが得られなくなったりしていた。

本論文は、実行頻度の高いパスを選択してスケジューリングを行なう手法を提案し、他の手法との比較を行なう。本手法の利点は、実行頻度の高いパスを最適にパイプライン化できるという点である。パスの選択は、トレース・スケジューリング[5]とは異なり、複数のパスの選択も可能である。そのため、分岐方向に著しい偏りがない応用にも向く。また、パスの選択によって、従来パイプライン化が困難であった最内ループでないループに対してもパイプライン化できる可能性を与える。また、本手法がターゲットとしているアーキテクチャは、プレディケート実行をサポートする機能を持つが、マルチプル・レジスタファイル[2]等のソフトウェア・パイプライニングのための特別のハードウェアは必要ない。

本論文は、まず、2章で本アルゴリズムを適用するアーキテクチャについて説明する。3章では、ソフトウェア・パイプライニングのアルゴリズムについて述べる。そして、4章で性能評価を行ない、他のアルゴリズムの性能と比較し、5章でまとめる。

2 プレディケーティング

本章ではまず、本アルゴリズムがターゲットにしているアーキテクチャ（プレディケーティング[7]と呼ぶ）について簡単に述べる（詳しくは、文献[7]を参照）。

プレディケーティングでは、命令は次の形式を持つ。

プレディケート ? オペレーション

命令の実行結果は、命令に示されているプレディケートが真であるときのみ有効となる。実行条件は、CCR（コンディション・コード・レジスタ）に保持される。各命令のプレディケートはこのCCRの値によって評価される。例えば、

```
c1&c2 ? lw $9, 4($10)
```

は、CCRの第1エントリ(c1)と第2エントリ(c2)が共に真であるときに、lw $9...の実行結果が有効化されることを示す。CCRの内容にかかわらず、常に、有効化されることを示すプレディケートは、alwとする。

命令の発行時点では、まず、プレディケートが評価される。その結果、値が真であれば、この命令の実行結果は有効であるので、通常のマシンの命令と同様に実行を行なう。もしも、プレディケートの値が偽であれば、既に、命令の実行結果が無効となることがわかっているので、単純に無効化する。これらのいずれでもない場合、即ち、プレディケートの値が未定値の場合は、投機的に命令の実行を行なう。この場合、その実行結果は、その実行結果が有効となる条件であるプレディケートと共に、シャドウ・レジスタに格納する。シャドウ・レジスタに格納されたデータは、プレディケートの値が真、または、偽に定義された時点で、ハードウェアによって自動的に有効化、または、無効化する。

3 ソフトウェア・パイプライニング

本アルゴリズムは、以下に示す5ステップよりなる。

1. パスを選択する。
2. プレディケート・コードを作成する。
3. モジュロ・スケジューリングを行なう。
4. 命令コードを作成する。

本章では、各々のステップについて例を用いて説明する。

3.1 パスの選択

本アルゴリズムでは、コンパイラは、ループを構成する制御フロー・グラフ(CFG)の中から、いくつかの基本ブロックを選択し、その集合に対してスケジューリングを行なう。この集合を**領域**(*region*)[7]と呼ぶ。領域は、ループの入り口をヘッダ・ブロックとし、1個以上の出口を持つ。また、ヘッダ・ブロックは、領域内の全ての他のブロックを支配(*domi-*

nate)[7] する。以下に、領域選択のアルゴリズムを示す。

(1) ループの入り口の基本ブロックをヘッダとし、領域とする。

(2) 領域から外に向かって CFG のエッジをたどり、ヘッダから到達する確率の高いブロックを領域に加える。ただし、次のブロックは加えない。
- スケジュール済みのブロック
- 自分の領域内のブロック
- プロッシージャ・コールを含むブロック

(3) 領域に含まれる分岐命令の数が CCR のエントリ数に達するか[1]、または、領域から外に向かうすべてのエッジの分岐確率が基準として定めた値（**最小分岐確率**）以下になるまで、領域を拡張する。

(4) 領域の決定後、領域内にループが存在する場合は、領域のヘッダが、領域内で、領域内の全てのブロックを支配できるよう、必要ならば、ブロックを複製する。ループが存在しない場合は、パイプライン化しない。

例を使って説明する。図 1(a) は、領域選択前の CFG である。各ノードは、基本ブロックを表し、エッジに付加した数字は、分岐確率を表す。CCR のエントリ数を 4、最小分岐確率を 0.2 とする。塗りつぶしたブロックは、既にスケジュール済みの領域とする。領域選択のアルゴリズムに従うと、まずノード A、次にノード D、そしてノード E、最後にノード F が領域に加えられる。ノード B は、既にスケジュール済みのブロックなので加えない。ここで、ノード A、D、E、F が領域となるのだが、ノード F は、領域外のノード C からのパスがあるため、ノード A は領域内で、ノード F を支配できない。そこで、ノード F の複製ノード F' を作成し、ノード A が、領域内で、領域内のすべてのノードを支配できるようにする。領域選択後の CFG グラフを図 1(b) に示す。ハッチングした部分が、本アルゴリズムを適用する領域である。

また、本アルゴリズムは、領域内にループが存在すれば、最内ループ [7]、最内ループでないループ（これを、以下、**外側ループ**と呼ぶ）の区別なくスケジューリングが可能である。図 1(b) において、ループ ADEF は、元の CFG では外側ループであるが、領域内では最内ループと見なせるため、パイプライン化が可能となる。そのため、A、D、E、F のパスが頻繁に実行される場合、性能が飛躍的に向上する。

3.2 プレディケート・コード

MS を行なう前に、条件分岐と制御依存制約を取り除き、ストレート・ライン・コード（プレディケート・コード）を作成する。プレディケーティングにより、制御依存は、ハードウェアによって解消できるため、領域内における制御依存制約はなくなる。また、分岐命令は、領域の外に出る分岐命令、もしくは、ループ・バックの場合のみ必要となる。

図 2(a) は、領域選択後の CFG の例であり、ハッチングした部分が領域である。図 2(b) は、図 2(a) の領域内のコードをプレディケート・コードに変換したコードである。プレディケーティングにより、基本ブロック A、C、D を一つの基本ブロックの様に扱うことが出来る。そのため、MS の適用が可能となる。

3.3 モジュロ・スケジューリング

ソフトウェア・パイプライニングの基本的なアルゴリズムとして MS を用いる。以下に、本手法におけるスケジューリング・アルゴリズムの概要を示す。

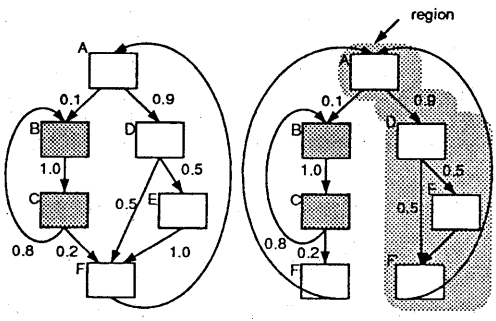
(1) 開始間隔の下限の決定

開始間隔 (*II*:*Initiation Interval*) の下限 S_{min} を、1 イタレーションの実行に最大限必要な資源数より決定される最小開始間隔 (S_R) と、イタレーション間の依存関係より決定される最小開始間隔 (S_D) の最大値とする。本アルゴリズムでは、S_R は領域内のすべての命令を考慮し、S_D は、異なるイタレーションに属する同一命令間の満たすべき条件より求めるとする。

(2) 命令のスケジューリング

本手法では、領域に対してリスト・スケジューリングを行ない、得られたサイクル数を *II* の上限 S_{max} とした。即ち、S_{max} は、パイプライン化しない場合の最小の *II* である。*II* の初期値を $s = S_{min}$ として、$s \leq S_{max}$ の範囲で s を 1 ずつ増やし、スケジュールが成功するまで、MS を行なう。

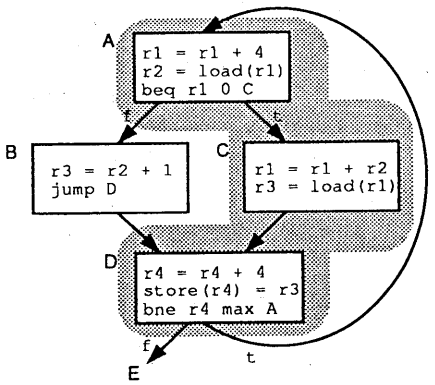
命令のスケジューリングは、依存制約を満たす、最も高い優先度を持つ命令から行なう。本アルゴリズムでは、各命令の優先度は、データ依存グラフの高さと、命令のスケジュール可能な範囲内の命令が利用できる資源の空き状況によって決める。命令が利用できる資源の空き状況を優先度に取り入れたのは、本アルゴリズムは、命令によって利用できる資源が異なるマシン（非均質マシン）を仮定している



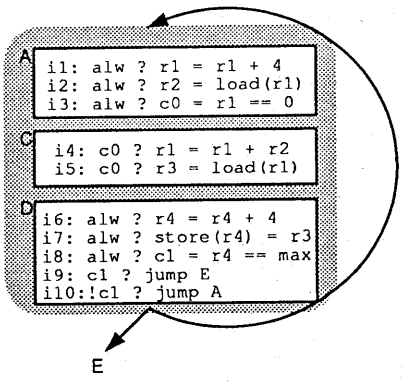
(a) 領域選択前　　(b) 領域選択後

図 1　領域

1. プレディケーティングでは CCR のエントリ数と等しい数の分岐を越えて命令の移動が可能。



(a) 制御フロー・グラフ



(b) プレディケート・コード

cycle	slot1 (load/store)	slot2 (load)	slot3	slot4
1	i4	i3		
2	i6	i5	i1 (next)	
3	i8	i2 (next)		
4	i7	i9	i10	

(next)は、次のイタレーション

(c) リソース予約テーブル

cycle	code
1	i4: c0 ? r1 = r1 + r2; i3 :alw ? c0 = r1 == 0;

2	i6: alw ? r4 = r4 + 4;
	i5: c0 ? r3 = load(r1);
	i1: c1 ? r1 = r1 + 4;

3	i8: alw ? c1 = r4 == max; i2: c1 ? r2 = load(r1);
4	i7: alw ? store(r4) = r3; i9: c1 ? jump E; i10:!c1 ? jump A;

(d) 定常状態のコード

図2 スケジュール例

ので、スケジュール可能な範囲が同じであっても、利用できる資源数が命令によって異なるからである。この優先度により、資源制約の厳しい命令を優先的にスケジュールし、最適なスケジュール結果が得られる可能性を高くしている。

命令 u が利用できる資源の空き数は、命令 u のスケジュール可能な範囲と既にスケジュールされている命令が占有する資源数より求める。

スケジュール可能な範囲の計算は、基本的には、Lam の *connected algorithm*[1] を使用している。以下に簡単に説明する。まだスケジュールされていない命令 u の開始間隔 s の時のスケジュール可能な範囲の上限 $Ub(u)$、下限 $Lb(u)$ は、命令 v がサイクル $cycle(v)$ にスケジュールされた場合、以下の式によって計算される。

$$Ub(u) = \min_{u \in P2}(Ub(u), cycle(v) + d(u, v) - s * p(u, v)) \quad (1)$$

$$Lb(u) = \max_{u \in P1}(Lb(u), cycle(v) + d(v, u) - s * p(v, u)) \quad (2)$$

ただし、$P1$ は、データ依存グラフにおいて v からのパスを持つ命令の集合、$P2$ は、v へのパスを持つ命令の集合とし、$d(u, v)$、$d(v, u)$ をそれぞれ、u から v、v から u への依存関係を満たすために必要なサイクル数、$p(u, v)$、$p(v, u)$ をそれぞれ、u から v、v から u への依存関係が生じるまでのイタレーション回数とする。

MS では、開始間隔 s における、ある命令 v が占有する資源 k の数は、以下の式で表される。

$$\rho_v^s(i, k) = \sum_{j \in Z} \rho_v(i + js, k) \quad (Z：整数) \quad (3)$$

ここで、$\rho_v(i, k)$ は、命令 v が発行されてから i サイクル後に命令 v が要求する資源 k の数とする。

(1)、(2)、(3) 式より、命令 u の利用できる資源 k の空き数 $e_k(u)$ は、以下の式で得られる。

$$e_k(u) = \sum_{Lb(u) \le i \le Ub(u)} (R_k - \sum_{v \in Sched} \rho_v^s(i - cycle(v) \bmod s, k))$$

ただし、$Sched$ は、スケジュール済みの命令の集合、R_k は、資源数とする。

ひとたび、命令がスケジュール可能となると、その命令は、資源競合がないかぎり、スケジュールできる最も遅いサイクルにスケジュールする。本アルゴリズムでは、スケジューリングをボトムアップで行なう。即ち、命令は、その命令に依存するすべての

命令がスケジュールされた後、スケジュール可能な最も遅いサイクルにスケジュールする。資源競合が生じた場合は、その次の最も遅いサイクルにスケジュールする。

図2(b)のコードをスケジュールした時の資源予約テーブルを図2(c)に示す。この例は、4つのALU(slot1-4)、4つの分岐ユニット(slot1-4)、2つのロード・ユニット(slot1-2)、1つのストア・ユニット(slot)のマシンを仮定している。

3.4 命令コード生成

MSが終了したら、命令コードの生成を行なう。図2(d)は、図2(c)で得られたスケジュール結果から、コードを生成したものである。次イタレーションの命令には、ループ条件が成立したとき実行が有効化されるプレディケートを付加する。このプレディケートは、分岐条件が不成立の場合、これらの命令の実行結果を無効化にする。そのため、エピローグ部は不要となる。

4 評価結果

EMS、PEを使ったスケジューリングと本アルゴリズムの比較評価を行なった。

4.1 ベンチマーク

ベンチマークに用いたプログラムは、espresso、eqntott、li、compressの4個のSPECベンチマーク、awk、grep、nroffの3個のUNIXユーティリティ、スタンフォード・ベンチマークである。これらの中のすべてのループに対してアルゴリズムを適用した。

4.2 マシン・モデル

評価を行なったマシン・モデルは、最大命令発行数が2、4、8の3つである。最大命令発行数が8のマシンのすべてのユニットは、すべての命令が実行できるとし、2、4のマシンは、すべての命令が実行できるユニット数が1、ロード命令と演算命令のみが1、残りは、演算命令のみとする。命令のレイテンシは、ロード命令が2サイクルであり、その他の命令はすべて1サイクルである。実行条件を保持するCCRは、4個のエントリを持ち、最大4分岐を越える投機的実行が可能である。また、投機的実行結果をバッファリングするため、1組のシャドウ・レジスタ・ファイルを持つ。しかし、動的リネーミングのためのマルチプル・レジスタファイルなどのソフトウェア・パイプライニングのための特別なハードウェアは持たないとする。

EMS、PEを使ったスケジューリングも、このマシン・モデルに対して適用した。そのため、EMSのアルゴリズムは文献[4]と同じであるが、ハードウェア・サポートがあるため、ループ回数が実行時までわからない場合やループの途中からループを抜ける場合に対する制限がない。また、コード生成を実際に行なっていないため、分岐命令を入れていないという違いがある。そのため、文献[4]よりは高い性能を示していると考えられる。

4.3 評価方法

本手法によりパイプライン化されたループ[2]を評価の対象とし、各ループの1イタレーションの実行に必要な平均サイクル数の総和 cyc_{loop} により、各アルゴリズムを評価した。評価にループの実行回数の項目を含めなかったのは、ループによって繰り返し回数に偏りがあるため、一部の繰り返し回数の多いループの特性によって評価結果が左右されるのを避けるためである。

EMS、PEは最内ループのみに適用可能なため、cyc_{loop} は、各最内ループに対して、パイプライン化されたループの開始間隔の和と、各外側ループに対して、パイプライン化されていないループの1イタレーションの実行に必要なサイクル数の和である。即ち、

$$cyc_{loop} = \sum_{l \in L_{inner}} II_l + \sum_{l \in L - L_{inner}} cyc_l$$

である。ここで、Lは、評価の対象となったループの集合、L_{inner}は、L内の最内ループの集合、II_lは、パイプライン化されたループの開始間隔、cyc_lは、パイプライン化されていないループを実行するのに必要なサイクル数とする。

一方、本手法では、分岐確率によってパスを選択して、パイプライン化するため、cyc_{loop}は、パイプライン化されたパスのループの開始間隔とパイプライン化されなかったパスのループの1イタレーションの実行に必要なサイクル数の分岐確率による相加平均である。即ち、

$$cyc_{loop} = \sum_{l \in L} (II_l * p_s + cyc_{out} * (1 - p_s))$$

である。ここで、p_sは、パイプライン化されたパスへの分岐確率、cyc_{out}は、パイプライン化されなかったパスを実行するのに必要なサイクル数とする。

4.4 評価結果

図3に、最大命令発行数2、4、8の各マシンにおける性能向上率を示す。性能向上率は、評価対象のループの1回の実行に必要なサイクル数の総和を4.3の方法で得られた総和で割った値とした。凡例の( )内は、性能向上率の算術平均である。

最大命令発行数2のマシンの場合、grep、nroffにおいてEMSの性能がよい。これは、分岐確率に片寄りのないループのスケジューリングにおいて、他のアルゴリズムは複数のパス上の命令間で資源競合が生じ、IIが大きくなってしまったためである。bench、eqntott、espressoで、本手法の性能がよいのは、パスの選択により、実行頻度の高いパスが最適にスケ

2. 本手法では、ループの一部のパスを選択してパイプライン化するので、「パイプライン化されたループ」とは、ループの中の少なくとも1つのパスがパイプライン化されているループのことを言う。

ジュールできたことによる効果が大きいためである。しかし、資源数が少ないために外側ループをパイプライン化できる場合が少なく、その効果は少ない。

最大命令発行数4のマシンの場合、すべてのプログラムにおいて本手法の性能がよい。理由は、資源数が増えたことにより、分岐確率に偏りのないループにおける資源競合が緩和され、性能低下が少なくなった一方、豊富な資源を利用し、外側ループがパイプライン化できたことによる効果が現われたためである。

最大命令数8のマシンの場合は、資源競合が少なくなるため、EMSとPEを使ったスケジューリングの性能向上率は、ほぼ等しくなっている。本手法においてもパスの選択による資源競合の緩和で得られる効果は減るものの、外側ループのパイプライン化による効果で、他のアルゴリズムより良い性能向上が得られている。

以上より、本手法は、資源数が少ない場合は、実行頻度の低いパスの命令の影響を受けずにスケジュールでき、資源数が多い場合は、分岐方向に偏りのないループにおいても良い性能が得られるということがわかる。また、外側ループも最内ループと同様に扱え、パイプライン化できることが、性能に寄与していることがわかる。

5 まとめ

本論文では、従来のアルゴリズムであるEMS、PEを使ったスケジューリングの欠点を緩和するソフトウェア・パイプライニングのためのアルゴリズムを提案した。本手法は、ループの中の実行頻度の高いパスを選択してスケジュールを行なうことにより、実行頻度の低いパスによる性能低下を防ぎ、ループの実行速度を改善する。さらに、従来パイプライン化が困難であった外側ループに対しても、パイプライン化できる可能性を与えることができる。

今後の課題としては、逆依存関係を解決するためのレジスタ・リネーミングや、ループ・アンローリングを行なうことにより、より性能を向上する工夫を行なっていく予定である。また、非数値計算応用のプログラムにおいて、ループの性能向上を得るためには、何が必要であるかを検討していきたいと思っている。

参考文献

[1] M.S.Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," In *Proc. SIGPRAN '88 Conf. Program. Lang. Des. Implement.*, pp.318-328, June 1988.

[2] J.C.Dehnert, P.Y.Hsu, and J.P.Bratt, "Overlapped loop support in the Cydra 5," In *Proc. 3rd Inter. Conf. Architectural Support for Programming Languages and Operating Systems*, pp.26-38, Apr. 1989.

[3] T.Nakatani and K.Ebcioglu, '"Combining" as a Compilation Techinique for VLIW Architectures,' In *MICRO-22*, pp.43-55, 1989.

[4] N.J.Warter, G.E.Haab and J.W.Bockhaus, "Enhanced Modulo Scheduling for Loops with Conditional Branches," In *MICRO-25*, pp.170-179, Dec. 1992.

[5] J.A.Fisher, "Trace Scheduling: A Technique for Globalmicrocode Compaction," *IEEE Transactions on Computer*, C-30(7), pp.478-490, Jul.1981.

[6] H. Ando, C. Nakanishi, T. Hara, and M. Nakaya, "Unconstrained Speculative Execution with Predicated State Buffering," In *Proc. the 22nd Annu. Inter. Symp. on Computer Architecture*, pp.126-137, June 1995.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Weslay Publishing Company, Reading, Massachusetts, 1986.

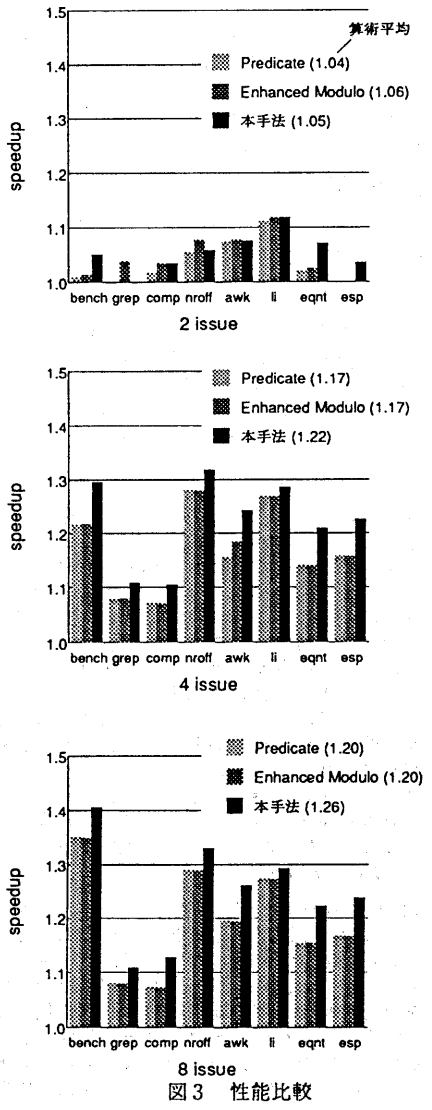


図3 性能比較