計算機アーキテクチャ　139－6
（2000．8．3）

シリーズパラレル型レジスタ生存グラフを用いたレジスタ割付けへの動的計画法の適用

浅原英雄　†　近藤伸宏　†††　古関聰　††
小松秀昭　††　深澤良彰　†

本稿では、命令レベル並列プロセッサ向けの新しいレジスタ割付け手法を提案する。命令レベル並列性(ILP)を抽出する手法として、我々はシリーズパラレル型レジスタ生存グラフを用いた手法を提案してきた。しかし、今までの手法はワンパスのヒューリスティクスを用いた手法であったため、プログラム構造によってはオプティマムな解を得ることができないという欠点があった。そこで、解空間を広く探索し、かつ、動的計画法を用いて計算量が増加するのを抑制できるアルゴリズムを開発した。本稿では、まずレジスタ割付けに関する問題を整理し、本手法のアルゴリズムと適用例を示し、その評価を行う。

Applying Dynamic Programming Technique to Register Allocation Based on Series-parallelized Register Existence Graph

Hideo Asahara ,† Nobuhiro Kondoh ,††† Akira Koseki ,††
Hideaki Komatsu †† and Yoshiaki Fukazawa †

We introduce a new register allocation algorithm for instruction-level parallelism processors. We have suggested a method using Series-Parallelized Register Existence Graph in order to extract ILP in a program. However, this method does not always give the optimum result because it only uses one-pass heuristics. Then, we developed a new algorithm which can search the broader possibility of solutions and uses dynamic programming in order to reduce the cost of computation. In this paper, we clarify some problems on register allocation techniques, and give our algorithm with some examples, and its evaluation.

1. はじめに

一般的に、スピルコードが頻繁に発行されるとプロセッサの性能を損なう。メモリにアクセスするコストが、レジスタにアクセスするコストよりもはるかに高いためである。このため、従来の手法ではスピルコードがなるべく少なくなるように、レジスタ割付け及びコードスケジューリングが行われてきた[1)~5)]。さらに、命令レベル並列プロセッサに対しては、レジスタ割付けの段階で余分な出力依存を発生させて並列性を低下させないように考慮されてきた[2)~5)]。これらを実現するレジスタ割付けを行うには、以下の2点が問題となる。

(1) コードスケジューリングとの相互作用

コードスケジューリングを先に行うと、レジスタアロケータがスケジューリング結果を壊してしまい、並列性が失われる。逆にレジスタ割付けを先に実行すると、余計な出力依存を生成してしまい、余分なスピルが挿入されてしまう。どちらを先に行っても、その相互作用がコードの並列性を壊してしまう。

(2) 中間コードにおけるコードの並び順

中間コードにおけるコードの並び順が仮想レジスタの生存区間の解析結果を変化させてしまうため、その並び順がコードの並列性を落とす原因となることがある。

レジスタ割付けをする際には、この2つの問題を解決する必要がある。

2. 研究の背景

並列性を考慮したレジスタ割付け技法として並列化レジスタ干渉グラフを用いたレジスタ彩色法がある[2)3)]。この手法は、レジスタ干渉グラフに並列性を表すエッジを加えた並列化レジスタ干渉グラフに対し、グラフ彩色法のアルゴリズムで彩色していくことで、並列性を考慮

† 早稲田大学理工学部
School of Science and Engineering, Waseda Univ.
†† 日本IBM(株)東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.
††† (株)東芝研究開発センター
Tohshiba Corp. Corporate Research and Development Center

したレジスタ割付けを行う。しかし、この手法では並列実行可能な命令が使用するレジスタの生存区間について安全な見積りを行っているため、依存関係が複雑になってしまうという欠点があった。

実際に、サンプルコードと並列化レジスタ干渉グラフの例を図1、図2に示す。図2で明らかなように、並列化レジスタ干渉グラフが完全グラフに近い形になっている。このグラフに彩色を行なうと、不必要なスピルコードの挿入が頻発してしまう。

```
1                    cc1=r1&1
2            (!cc1)  r4=Load(0+r2)
3            (!cc1)  r5=r4+1
4             (cc1)  r6=Load(0+r2)
5             (cc1)  r7=r6+r2
6             (cc1)  r8=Load(r3+0)
7             (cc1)  cc2=r3 cmp r8
8   (cc1 & !cc2)  r9=r7 << 2
9   (cc1 & !cc2)  r10=10
10  (cc1 & !cc2)  r11=r9-r10
11  (cc1 & !cc2)  r12=r7 << 4
12  (cc1 & !cc2)  r13=100
13  (cc1 & !cc2)  r14=r12-r13
14  (cc1 & !cc2)  Store(r14,r3+4)
```

図1 サンプルコード

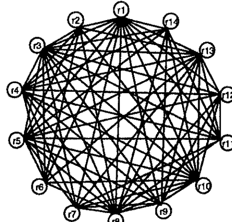

図2 並列化レジスタ干渉グラフ

我々は、レジスタ生存グラフを利用するアルゴリズムを提案し、問題を解決してきた[4)5)]。レジスタ生存グラフとは仮想レジスタをノードとし、その値の生産と使用をエッジで表したグラフである。中間コードにおける命令順序はレジスタ生存グラフに影響しないため、前述の問題(2)を解決できる。また、このグラフの特長は、ある時刻に必要になる実レジスタ数が明確になるという点である。このグラフを用いることで前もって必要な実レジスタ数を減少させ、コードスケジューリングの際に実レジスタが不足しないための保証を与えることができる。これにより、コードスケジューリング実行時には、レジスタ資源問題を考えずにスケジューリングをすることができ、問題(1)を改善できる。

我々は、まず、ワンパスのヒューリスティクスを用いてレジスタ生存グラフのトポロジを変形し、最大干渉度を低減する手法を提案した[4)]。この流れを図3に示す。

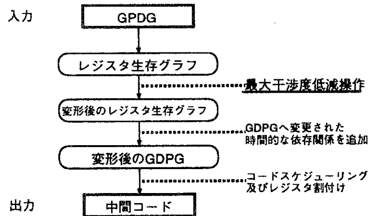

図3 レジスタ生存グラフを用いた手法のアルゴリズム

レジスタ生存グラフにおいて、同時刻に生存するレジスタ数を干渉度、その最大値を最大干渉度と呼ぶ。これを実レジスタ数以下に低減することでコードスケジューラに実レジスタが足りることを保証できる。この操作を、最大干渉度低減操作と呼ぶ。しかし、この手法で用いているヒューリスティクスは、レジスタプレッシャーが高い場合に良い結果を出すことができなかった。

この欠点を補うため、我々はレジスタ生存グラフをシリーズパラレル化することで、スピルするノードを決定するヒューリスティクスを改良した[5)]。この手法ではコード中の並列度の高さを数値化し、その数値を元に、優先してマッピングされるべき仮想レジスタを決定するヒューリスティクスを提案している。これにより、レジスタプレッシャーが高い場合にも対応できるアルゴリズムとなった。しかし、バックトラックを行なわずにヒューリスティクスにより解を一意に決めてしまう手法であったため、プログラム構造によっては質の悪い解しか得られないという欠点があった。

そこで、我々はこの欠点を補うために、動的計画法を用いて計算量を抑えながら、解空間をある程度探索し、それによってより最適解に近い値を得ることができるアルゴリズムを提案する。

3. 準　　備

GPDG(Guarded Program Dependence Graph)[6)]とは、PDG[7)]を拡張したグラフであり、プログラム中の最内ループを表現するのに用いられる。GPDGは、最内ループの入口と出口を示すSTARTノードとENDノードを持つ。GPDGのノードは命令を表し、ガードと呼ばれる実行条件が付加される。また、ガードとデータの生産と使用に関する依存関係を、有向エッジを用いて表現する。これにより、GPDG上では、制御依存をデータ依存と同等に捉えることができる。

レジスタ生存グラフ[4)]は、GPDGから生成することができる。グラフ中のノードは仮想レジスタとし、ノード間には、その仮想レジスタ値の生産と使用を示す有向エッジが張られる。また、生存時間の概念があり、同時刻に生存している仮想レジスタ同士は**等時刻線**と呼ばれる一本の線で貫かれている。

本稿で使用する**シリーズパラレル型レジスタ生存グラフ**は、レジスタ生存グラフを拡張したグラフである。各ノードは生存時間をノードの長さで表し、その生存時間の分だけ等時刻線と交わる。また、ガードが生成されたサイクルの直後に**条件境界**を持ち、Topノードの直後とBottomノードの直前にもそれぞれ**Topノード境界**、**Bottomノード境界**を持つ。境界はノード、境界とは交わらず、エッジのみと交わる。またグラフ中において、**条件境界領域**と**分割領域**が定義される。条件境界領域は、条件境界に上下を挟まれたノードの集合である。分割領域は、条件境界領域内のノード中で、データ依存関係を持つノードの集合である。条件境界領域中には、

1つ以上の分割領域がある。

4. 本手法の詳細

全体の流れを、図4に示す。

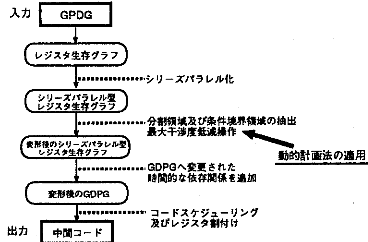

図4　本手法の流れ

SSA変換[8] を施されたGPDGを入力とし、そこからレジスタ生存グラフを得る。このグラフをシリーズパラレル化し、分割領域を抽出することで、最大干渉度低減操作を動的計画法で実行することができる。これらの作業で得られた時間的制約を満たすためのエッジをGPDGに追加し、コードスケジューリング及びレジスタ割付けを行う。

本手法において最も特徴的な点が、最大干渉度低減操作である。本手法では解空間を探索することで最適解を得るが、そのためのコストを動的計画法を用いて抑えている。

動的計画法とは、目的関数に含まれる変数をある制約の下で操作し、目的関数の最適解とその変数の値の組合せを探索する数学的計画法の一つである。目的の解を得るために何度も同じ小問題を解かなければならない場合に、その解を再利用することで全体の計算量を抑えるという特徴を持つ。本手法で作成する動的計画法の表を、図5の(a)に示す。

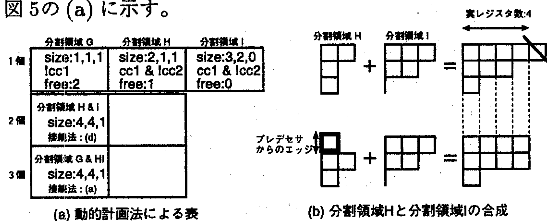

図5　最大干渉度低減操作

同一の条件領域内に生存する全ての分割領域を最少のグラフとし、グラフを結合していく。結合の際、最大干渉度が実レジスタ数を越えない範囲で、クリティカルパス長がもっとも短くなる結合方法を選択する。結合法は、基本的にはグラフを同時刻に生存させるパラレル接続か、同時刻に生存しない様に片方のグラフの生存時刻を遅らせて結合する、シリーズ接続の2種類である。この結合結果を比較し、最も良い解を合成していくことで、全体の最適解を探索する。この作業を各条件境界領域に対して行い、最終的に全ての分割領域を一つのグラフに結合し、最大干渉度を低減されたグラフを得る。

ここで、図4の流れに従い、図1のサンプルコードに対して本手法を適用した結果を述べる。まず、GPDGが入力となる。図1に対するGPDGを作成したのが、図6である。このGPDGから図7のレジスタ生存グラフを導き、図8(a)のシリーズパラレル型レジスタ生存グラフを得る。この変形を行う理由を、以下に示す。

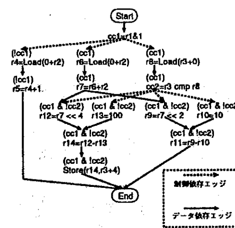

図6　GPDGの例

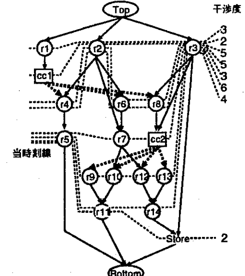

図7　レジスタ生存グラフの例

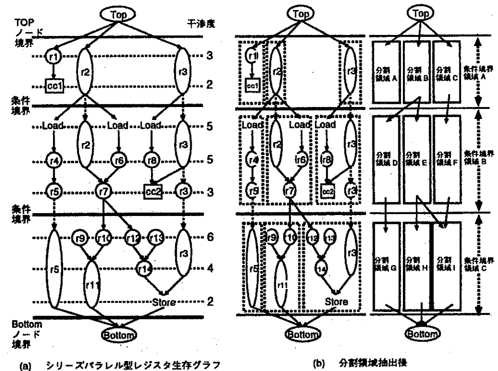

図8　シリーズパラレル型レジスタ生存グラフと分割領域の抽出

本手法は、最大干渉度低減操作を行う際に論理排他性を利用する。ここで、ある命令A,Bが論理排他性をもつとは、両者が2つとも実行される実行条件がないことを保証することである。この場合、命令A,Bの目的オペランドの生存範囲が重なる場合、必要となるレジスタが1つになる。この論理排他性を考慮することで、実レジスタをより効率よく使用できる。

最大干渉度を低減するには、グラフトポロジを変化させる必要があるが、グラフトポロジを無制約に再構築すると、論理排他性を持つ命令同士の実行サイクルがずれてしまい、プログラムの持つ論理排他性を活用できなくなる恐れがある。そこで、本手法ではグラフトポロジの再構築を条件境界領域内に留め、論理排他性を最大限に利用する。そのため、シリーズパラレル型レジスタ生存

グラフを利用している。こうして得られたシリーズパラレル型レジスタ生存グラフから、分割領域を抽出する。

分割領域を抽出するのも、論理排他性を考慮するためである。条件境界領域を分割領域に分割すると、分割領域には同じガードを持ったノードが集まる。この分割領域を動的計画法に用いる最小のグラフとすることで、論理排他性を十分に生かすことができる。また、計算量も減少させることができる。

分割領域を抽出した結果を、図 8(b) に示す。

分割領域を抽出後、最大干渉度低減操作に入る。図 5(a) に示した表を埋めていき、最終的に最大干渉度が実レジスタ以下に低減された１つの領域を得る (図 9)。

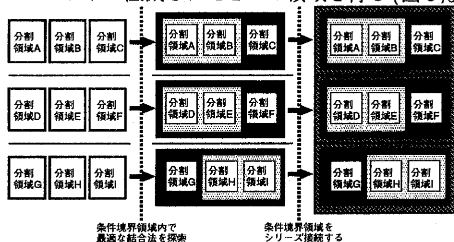

図 9　分割領域の統合

5. アルゴリズム

5.1 シリーズパラレル化

シリーズパラレル化の手順を示す。

１：ノードの生存時間を、ノードの長さで表す。

２：違う時刻にある複数のエッジのプレデセサになるノードは、エッジがある部分でノードを分割し、ノードをコピーする。また、ノード間に同じ値を保持することを示すエッジを張る。

３：Top ノード境界・条件境界・Bottom ノード境界を引く。境界を越えて生存するノードは、境界の上下で分割し、同じ値を保持することを示すエッジを張る。

こうしてシリーズパラレル型レジスタ生存グラフが得られる。例を、図 8(a) に示す。

5.2 分割領域の抽出

次の作業を行うことで分割領域を抽出する。

１：仮想レジスタの生成条件ごとに取り出した条件境界領域内のノードを分割する。

２：全体がパラレル接続になっている部分があれば、更にそれぞれを分割する。

３：分割されたもので、それ自身の最大干渉度が実レジスタ数を超えているものは、その最大干渉度を低減する。操作は次の通りに行う。

(a)：２つ以上の仮想レジスタのプレデセサになっているノードを、コピーする。

(b)：スイートを抽出し、高さ・自由度を計算する。ここでスイートとは、パラレル、シリーズ接続をしているノードの集合を指す、また、高さとはパラレル接続のネストの数を、自由度は、その命令の実行をどれだけ遅らせても影響が出ないかという指標である。

(c)：高さを降順に、自由度を昇順にソートし、その順番に新たなグラフ中に追加していく。この際に、干渉度が実レジスタ数を越えてしまうような場合には、そのプレデセサをスピルさせ、干渉度に余裕がある時刻まで遅らせる。

(d)：分割領域のクリティカルパスが伸びてしまった場合には、同一条件境界領域内の全ての別の分割領域の生存時間を同じだけ延ばす。

４：元のグラフにおけるノードへのエッジを、そのノードが属する分割領域へのエッジに追加する。またノードの生成条件を、そのノードが属する分割領域の生成条件にする。

サンプルコードから分割領域を抽出した結果を、図 8(b) に示す。

5.3 最大干渉度低減操作

分割領域が抽出されたシリーズパラレル型レジスタ生存グラフに対し、最大干渉度低減操作を行う。

接続された後のグラフを G、接続するグラフを G'、G'' とすると、接続式は基本的には次のようになる。

$$G_{width[i]} = G'_{width[i]} + G''_{width[i]}$$
$$G_{depth} = G'_{depth} = G''_{depth}$$

式中で、$G_{width[i]}$ は分割領域の各サイクルの干渉度で、G_{depth} は分割領域のクリティカルパス、G_{free} は、分割領域の自由度を表す。最大干渉度低減操作の手順を以下に示す。

１：同じプレデセサを持つ分割領域の組を探す。

２：動的計画法の表を作る。図 8(b) の条件境界領域 C に対する表を、図 5(a) に示す。すなわち、１で得た分割領域の組を優先的に接続していき、最終的に全体を接続した場合の接続法、接続結果を得る。

ここで、各接続法は次のようにして決定する。以下の優先順位で、最大干渉度が実レジスタ数以下になるものを選択する。

(a)：パラレル接続する。

$$G_{width[i]} = G'_{width[i]} + G''_{width[i]}$$
$$G_{depth} = G'_{depth} = G''_{depth}$$

となる。論理排他性がある場合は、$G_{width[i]}$ の計算を次のようにする。

$$G_{width[i]} = \mathrm{Max}[G'_{width[i]}, G''_{width[i]}]$$

(b)：最大干渉度が実レジスタ数を越えている部分にスピルコードを入れ、パラレル接続する。ただしスピル

コードは分割領域単位となるため、$G''_{width[i]}$ が全てのiに対して1であるような分割領域のみを対象とする。

$G_{width[i]} = G'_{width[i]} + G''_{width[i]}$

$G_{depth} = G'_{depth} = G''_{depth}$

ただし、スピルさせた時刻iのwidth[i]は1減らす。

(c)：分割領域の自由度が大きい方を、それが同じならその分割領域のプレデサセからのエッジが少ない方をできるだけ早い時刻でスピルさせる。G'' がスピルし、スピルさせたクロックを d, エッジの本数を α とすると、次のようになる。

$0 < i \leq d$ ならば $G_{width[i]} = G'_{width[i]} + \alpha$

ただし、スピルインの位置はレジスタが許す限り早い位置に移動する。

$d + 1 \leq i \leq G'_{depth}$ ならば

$G_{width[i]} = G'_{width[i]} + G''_{width[i+d]}$

$G'_{depth} < i \leq G'_{depth} + G''_{depth} - d - G''_{free}$ ならば

$G_{width[i]} = G''_{width[i+d]}$

かつ $G_{depth} = G'_{depth} + G''_{depth} - d - G''_{free}$

(d)：分割領域の自由度が大きい方を、それが同じなら分割領域のプレデセサからのエッジが少ない方をスピルさせることで、シリーズ接続にする。

$0 < i \leq G'_{depth}$ ならば　$G_{width[i]} = G'_{width[i]} + \alpha$

ただし、スピルインの位置は、レジスタが許す限り早い位置に移動する。

$G'_{depth} \leq i < G'_{depth} + G''_{depth}$ ならば

$G_{width[i]} = G''_{width[i]}$

$G_{depth} = G'_{depth} + G''_{depth}$

3：スピルイン命令をレジスタ制約が許す限り早い位置に移動、無駄なスピルアウト命令を消去する。

以上で最大干渉度低減操作における、動的計画法の表の作成法を述べた。サンプルコードの条件境界領域 C に対してこの手順を適用した際の様子を図5に示す。

図5(a)は動的計画法による全体の表で、(b)が分割領域HとIを合成した時の流れである。ここで手順(1)の段階で、分割領域HとIは同一の分割領域からエッジがあるとわかる。つまり、HとIを先に接続すれば良いので、(a)の表では2段目にHとIの接続のみを試し、4つある接続法を順に試す。最初のパラレル接続は、1サイクル目の干渉度が実レジスタ数を越えるため、接続できない。そこで、自由度の高い分割領域Hをスピルさせる方法を試す。分割領域の干渉度から、1サイクル分スピルさせることで干渉度の制約を満たすことが分かる。ここで、プレデセサからのエッジが一つあるため、スピルさせるには、そのロード命令を可能な限り早い位置に挿入する必要がある。ここでは、ロード命令が元のノードと重なるため、ロード命令を入れずにプレデセサのノードの生存を延ばす。また、分割領域Hは自由度が1であるので、クリティカルパスを伸ばさずに接続できる。この様にして、最終的に(a)のグラフを得る。

この作業を全ての条件境界領域ごとに行い、最終的に一つに合成された分割領域を得ることができる。この様子を図9に示す。

5.4 コードスケジューリング

図10にある、最大干渉度を低減するための依存を加えたGPDGに対して、レジスタ割付け・及びコードスケジューリングを行う。

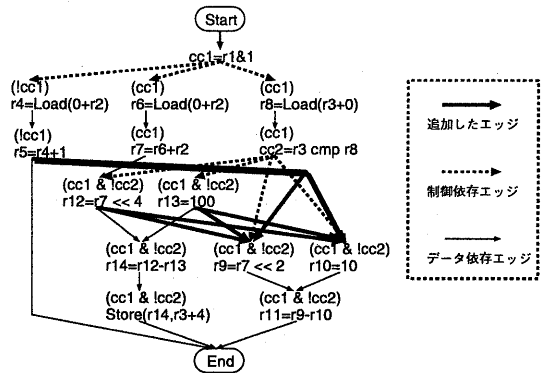

図10　変更後のGPDG

1：マシンサイクルを1とする。

2：作業用リストにGPDG上での深さが1のノードをすべて入れる。

3：GPDGのスタートノードにおいて既に生存している仮想レジスタを実レジスタにマッピングする

4：作業用リストが空でない間以下を繰り返す。

(a)：作業用リストのノードを自由度の低い順にソートする。

(b)：作業用リストの先頭からALU数を越えない数のノードに対応する命令をそのマシンサイクルの命令スロットに割り付け、それらを作業用リストから外す。

(c)：割り付けた命令が定義する仮想レジスタを実レジスタにマッピングする。

(d)：作業用リストにあるノードの自由度を1減らす。

(e)：命令スロットに割り付けられた命令の結果により依存関係が満たされたノードをリストに加える。

(f)：マシンサイクルを1増やす。

このような手順でレジスタ割付け及びコードスケジューリングを行う。サンプルコードに対して2つのALU、4つのレジスタを持つハードウェアを想定してスケジューリングした結果を、図11に示す。

clock	ALU1	ALU2
1	cc1=r1&1	
2	r6=Load(0+r2	r8=Load(r3+0
3	r6=Load(0+r2	r8=Load(r3+0
4	r4=Load(0+r2)	r7=r6+r2
5	r4=Load(0+r2	cc2=r3 cmp r8
6	r5=r4+1	r12=r7<<4
7	r13=100	r9=r7<<2
8	r14=r12-r13	r10=10
9	r11=r9-r10	Store(r14,r3+4)
10		

clock	レジスタ1	レジスタ2	レジスタ3	レジスタ4
1	r1	r2	r3	empty
2	empty	r2	r3	empty
3	(r6)	r2	r3	(r8)
4	r6	r2	r3	r8
5	(r4) or r7	r2	r3	r8
6	r4 or r7	empty	r3	empty
7	r5 or r7	r12	r3	empty
8	r5 or r9	r12	r3	r13
9	r5 or r9	r14	r3	r10
10	r5	r11	empty	empty

図 11　スケジューリング結果

6. 評　　価

本アルゴリズムを採用したレジスタアロケータの評価を行う。ハードウェアは、通常の命令を1クロック、Load 命令を2クロックかかるとした。レジスタ数が8、ALU が2・4・8・∞の各場合について評価を取った。Stanford-Integer Benchmark 中のいくつかのプログラムから最内ループを取り出し、それに対するレジスタ割付け及びコードスケジューリングを行った。比較した手法は、レジスタ彩色法 (手法 1)、並列化レジスタ干渉グラフを用いた手法 (手法 2)、シリーズパラレル型レジスタ生存グラフを用いた手法 (手法 3)、そして、本手法の4種類である。

評価値は、コードスケジューリングを行った後のクリティカルパス長を用い、手法1を1とした時の性能比を示している。この評価を図 12のグラフに示す。

全体として、命令レベルでの並列性を抽出できている。手法1は並列性を考慮しない方式であり、プロセッサの並列性が上がっても性能は期待できない。図 12では、手法1を基準とした性能比のグラフを使用している。手法2は、ある程度並列性を考慮した手法であるが、手法1と比較して、常に良い結果を出すわけではなく、劣る結果を出す場合もある。これは背景でも述べた通り、余計なスピルコードが出てしまうからである。本手法は手法3と比較すると違いが少ないが、これは手法3のヒューリスティクスが評価に使用したプログラム構造に適しているためだと考えられる。Permute のように、プログラム構造が適していない場合には、はっきりとその差が出ている。Permute 以外の評価でも、手法3の値を上回っており、コードの持つ並列性を十分に生かしたレジスタ割付けができていると言える。

7. 終わりに

本稿では、命令レベル並列プロセッサ向けにコードを最適化するレジスタ割付け手法を提案した。動的計画法を用いたアルゴリズムを用いることでコストを低減しつつ、広く解空間を探索するヒューリスティクスを考案し、よりコードの並列性を生かすスケジューリングを行った。しかし、本手法は分割領域の数が極端に少ないと、その性能を発揮できない。今後はこの欠点を解消する手法を研究していきたい。

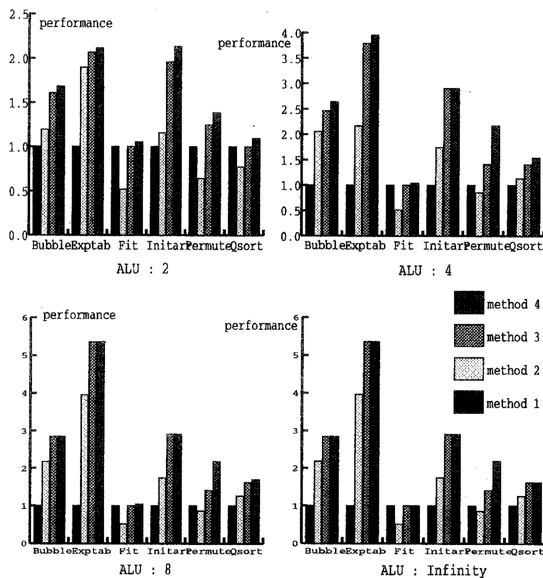

図 12　スケジューリング結果の性能比

参 考 文 献

1) G.J.Chaitin, etc.: "Register Allocation via Coloring", *Computer Languages Vol.6*,1981, pp.47-57.

2) C.Norris and L.L.Pollock: "A Scheduler-Sensitive Global Register Allocator", *Proc. of the ACM SIGPLAN '93 Conf. on Supercomputing*, 1993, pp.804-813.

3) S.S.Pinter: "Register Allocation with Instruction Scheduling: a New Approach", *Proc. of the ACM SIGPLAN '93 Conf. on Programming Languages Design and Implementation*, 1993, pp.248-257.

4) 古関聰, 小松秀昭, 百瀬浩之, 深澤良彰："命令レベル並列アーキテクチャのためのコードスケジューラ及びレジスタアロケータの協調技法", 情報処理学会論文誌, Vol.38, No.3, 1997, pp.584-594.

5) 近藤伸宏, 小松秀昭, 古関聰, 深澤良彰: "シリーズパラレル型レジスタ生存グラフを用いたレジスタ割付け技法とその評価", 並列処理シンポジウム JSPP'99, pp47-54.

6) 小松秀昭, 古関聰, 深澤良彰："命令レベル並列アーキテクチャのための大域的コードスケジューリング技法", 情報処理学会論文誌, Vol.6, 1996, pp.1149-1161

7) J.Ferrante, K.J.Ottenstein and J.D.Warren: "The Program Dependence Graph and Its Use in Optimization", *ACM Trans. Prog. Lang. Syst. Vol.9, No.3*, 1987, pp319-349.

8) R.Cyton, J.Ferrante, B.Rosen, M.Wegman and K.Zadeck: "An Efficient Method of Computing Static Single Assignment Form", *Conf. Record of the 16th ACM Symposium on the Principles of Programming Languages*, 1989, pp.25-35.