ハイパフォーマンスコンピューティング 52−12
（1994. 7. 23）

命令レベル並列プロセッサ向けレジスタ割り付け手法とその評価

神力 哲夫[†]　小松 秀昭[‡]　古関 聰[†]　深澤 良彰[†]
[†] 早稲田大学理工学部　[‡] 日本IBM(株) 東京基礎研究所

命令レベル並列プロセッサにおいては、レジスタ割り付けに際して、1) 割り付けにより発生したリソース依存がプログラムの並列性を落さないようにすること、2) 挿入されたスピルコードができるだけ他の命令と並列に実行されるようにすること、の二点を考慮することが必要である。これらは並列性を持たないプロセッサでは問題にならないため、従来のレジスタ割り付け手法では考慮されていない。そのため、これまでの手法を命令レベル並列プロセッサに適用した場合、レジスタ割り付けによってプログラムの持つ並列性を大きく失ってしまうことがある。本稿では、前述の二点を解決するレジスタ割り付け手法を提案し、その性能を評価する。

A Register Allocator for Instruction-Level Parallel Processors and Its Evaluation

Tetsuo Shinriki[†]　Hideaki Komatsu[‡]　Akira Koseki[†]　Yoshiaki Fukazawa[†]
[†] School of Science & Engineering, Waseda University
[‡] Tokyo Research Laboratory, IBM Japan, Ltd.

In a register allocator for instruction-level parallel processors, following two points are essential: 1) not to decrease instruction-level parallelism in program by the resource dependency caused by the register allocator. 2) to execute spill codes parallelly with other instruction. These may not play an important role in scalar processor systems, then existing register allocators do not take these into considerations. Therefore, when a conventional register allocator is applied to a instruction-level parallel processors, parallelism in a program can not be reflected to the object program. This paper describes and estimates our new allocator in which instruction-level parallelism in program is maintained.

1 はじめに

最適化コンパイラにおいて重要なフェーズにレジスタ割り付けがある。特に、現在の主流であるRISCチップでは、メモリアクセスのコストが大きいため変数の値をできる限りレジスタ上に置いておくことが望ましい。このことは、近年、盛んに発表されている命令レベル並列プロセッサにおいても同様である。しかしながら、従来用いられてきたレジスタ割り付け手法はプログラムの並列性を考慮しておらず、その結果、レジスタ割り付けによってプログラムの並列性を大きく失ってしまう場合がある。

我々の手法では、プログラムをGPDGと呼ばれる命令間の依存関係を表すグラフ構造で表現する。このことにより、プログラムの並列性に留意しながらレジスタ割り付けを進めることが可能となる。

2 本研究の背景

2.1 レジスタ割り付けとリソース依存

図1(a)のプログラムのレジスタ割り付けを考える。同図(b)はプログラムを命令間の依存関係を表すグラフ構造で表現したものであり、この図からプログラムの実行には3サイクル必要であることがわかる。

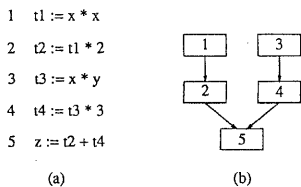


図 1: 並列プログラムの例

図2は図1に対してレジスタ割り付けを行なった結果の一つである。変数t1と変数t3を同じレジスタr4に割り付けているため、図の点線のようなリソース依存が発生し、命令1と命令3を同時に実行することができなくなっている。そのため、必要なサイクル数も増加している。これに対して、図3の割り付けではこのような問題は発生しない。

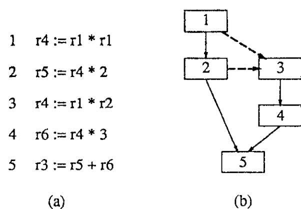


図 2: 並列性を失っている割り付け

命令レベル並列プロセッサでは後者の割り付けが望ましいことは明らかであるが、従来手法ではそのような割り付けが行なわれる保証はない。その原因は生存区間の解析にある。図1(a)で見ると変数t1と変数t3の生存区間は重ならない。しかし、同図(b)で見れば明らかなように、プログラムの並列性まで考慮すれば、二変数は生存区間を重ねており、同一のレジスタに割り付けることはできない。

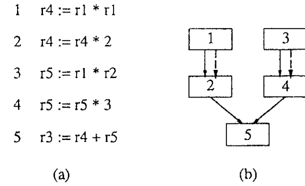


図 3: 並列性を保持している割り付け

従って、上記の問題を解決するためには、プログラムの命令レベルの並列性が明らかになるように、図1(b)のような命令間の依存関係によるグラフ構造で表現し、そのグラフ上で生存区間の解析・レジスタの割り付けを行なうことが必要である。

2.2 スピルコードの挿入

プログラム中で使用される変数が多い場合、全ての変数をそのままレジスタに直接割り付けることはできない。この場合、一つのレジスタを複数の変数で共有する必要がある。そのためには、計算された変数の値をメモリ上に書き出したり、逆に、書き出しておいた値をレジスタへ読み戻したりすればよい。このような命令を一般にスピルコードと呼ぶ。

従来のレジスタ割り付けでは、このスピルコードの個数を最少化することが命題とされた。なぜなら、並列性を持たないプロセッサでは、余計なコードの挿入は常にプログラムの実行時間を延ばしてしまうためである。しかしながら、命令レベル並列プロセッサにおいては複数の命令を同時に実行できるため、挿入されたスピルコードを他の命令と並列に実行することも可能である。そのため、スピルコードの最少化だけではなく、他の命令との並列実行をも考慮する必要がある。

3 本手法の特徴

3.1 GPDG

我々の手法では、プログラムをGPDGと呼ぶグラフ構造で表わし、全ての処理はその上で行なう。GPDG(Guarded Program Dependence Graph)は、PDG[4]を改良したもので、条件分岐構造の表現に特徴がある。

GPDGは、プログラムのループを単位として作成される。そのため、ループの先頭を表す`START`ノードと、再後尾に対応する`LOOP`ノードを持つ有効非サイクルグラフとして表現できる(図4参照)。図4中の`cc1`,`cc2`といった変数は条件変数と呼ばれ、条件分岐の分岐結果を表す。各ノードには、この条件変数の真偽の組合せによって表された実行条件が付加される。例えば、命令Bは、`a<t4`が満たされた場合にのみ実行される。

このような構造により、条件分岐を含むプログラムを簡単に表現でき、基本ブロックを越えた大域的なコード最適化を行なうことが可能となる。

以下に、本稿で必要なGPDG上のいくつかの定義を行

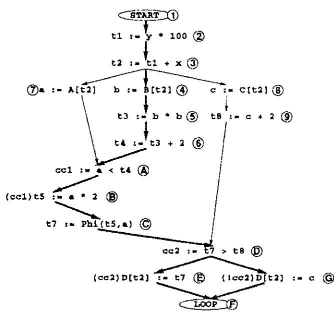


図 4: GPDG の例

なう。

パスの長さ　経路 (パス)P の "長さ"$Length(P)$ を次のように定義する。

$$Length(P) = \text{パス}P\text{に含まれるノードの個数}$$

ノードの自由度　ノード X の "自由度"$Free(X)$ を次のように定義する。

$$Free(X) = Depth(\texttt{LOOP}\text{ノード}) \\ -(Height(X) + Depth(X)) + 1$$

但し、ノード間の "距離" $Distance$、ノード "深さ"$Depth$、ノードの "高さ" $Height$ を、それぞれ以下のように定義する。

$$Distance(A, B) = Max(Length(A\text{から}B\text{へのパス}))$$

$$Depth(X) = Distance(\texttt{START}\text{ノード}, X)$$

$$Height(X) = Distance(X, \texttt{LOOP}\text{ノード})$$

分岐点　ノードが "分岐点" であるとは、そのノードが複数の子ノードを持つことをいう。図4では、{3,D} が "分岐点" である。

合流点　ノードが "合流点" であるとは、そのノードが複数の親ノードを持つことをいう。図4では、{A,D,F} が "合流点" である。

クリティカルパス　START ノードから LOOP ノードへ、自由度が 0 のノードのみを通って至るパス (複数存在) を、クリティカルパスと呼ぶ。図4で、太線で表されたパスがクリティカルパスである。GPDG で表現されたプログラムを、並列度が十分なプロセッサで実行する場合、最低でもこのクリティカルパスの長さ分だけのサイクル数が必要である。従って、クリティカルパスの長さをプログラムの実行時間の目安と捉えることができる。

コンパイラは、このクリティカルパスをできるだけ短くするように処理を行なう必要がある。逆に、このクリティカルパスがコンパイラの最適化の指標となる。レジスタ割り付けにおいても、このクリティカルパスをできるだけ延ばさないような割り付けを行なうことが必要とされる。

3.2 GPDG を利用したレジスタ割り付け

既に述べたように、命令レベル並列プロセッサを対象とするレジスタ割り付けでは、

1. 割り付けによって発生するリソース依存
2. スピルコードと他の命令の並列実行

に留意する必要がある。

GPDG 上で考えた場合、自由度の低いノードの前にスピルコードが挿入されることは望ましくない。なぜなら、新たなノードの挿入によってそのノードの深さを下げた場合、クリティカルパスを大きく延ばしてしまう可能性が高いためである。逆に、自由度の高いノードの前であれば、スピルコードと他の命令の並列実行の可能性も高く、プログラム全体の実行時間を延ばす可能性は低い。

そこで、我々の手法では、自由度の低いノードから順にレジスタ割り付けを進めていく。このようにすることで、スピルコードやリソース依存は、自由度の高いノードの前に発生することになり、プログラム全体の実行時間が延びることを抑制する。

3.3 SSA 変換

```
x = a + 2;
y = x * 2;
...
p = ...;
...
x = b + 2;
...
k = ...;
if (...)
    k = k + 2;
else
    k = k - 2;
...
k = k*k;
```

(a)SSA 変換前

```
x1 = a1 + 2;
y1 = x1 * 2;
...
p1 = ...;
...
x2 = b1 + 2;
...
k1 = ...;
if (...)
    k2 = k1 + 2;
else
    k3 = k1 - 2;
k4 = Φ (k2,k3)
...
k5 = k4*k4;
```

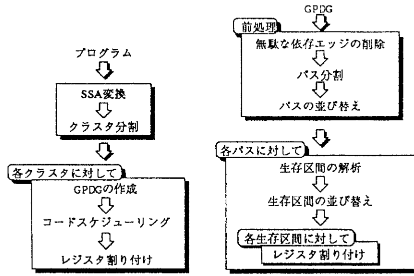
(b)SSA 変換後

図 5: SSA 変換の例

一般的なプログラミング言語では、各変数への代入は繰り返し行なうことができる。例えば、図 5(a) の場合、変数 x および変数 k は、複数回の代入が行なわれている。

SSA 変換 [5] とは、各変数の名前を代入が新たに行なわれる毎に変更し、各変数の単一代入を保証する手法である（図 5）。

SSA 変換により、変数間の無駄な干渉を取り除き、必要なレジスタ数の増加を防ぐ。図 5の場合、変数 x と変数 k は生存区間が重なっているが、x=b+2 において x の値は更新されてしまうため、この間変数の値を保存をしておく必要はない。すなわち、この区間は他の変数が利用することができるはずである。これに対して、SSA 変換後の図 5(b) では、x は二つの変数に分割され、上記のような無駄もなくなっている。



(a) コンパイラの構成　　(b) 割り付けの流れ

図 6: レジスタ割り付けの流れ

3.4　クラスタ分割

プログラムは、その実行時間の多くを最内ループで費すため、内側のループを優先的に最適化することは重要である。レジスタ割り付けにおいても、プログラムをループを単位とした階層的な構造に分割し、内側のループから割り付けを行なうことで、内側のループほど優先的にレジスタが利用できるようになる [2][3]。このような割り付けにより、外側ループ内の変数がレジスタを占有してしまったために、内側のループ内に無駄なスピルコードが挿入されるようなことがなくなる。

4　本手法の流れ

4.1　レジスタ割り付けの前処理

無駄な依存エッジの削除　GPDG の各依存エッジは、ノードに対応する命令の実行順序を表すためにある。従って、エッジの起点ノードを S、終点ノードを E とする時、S から E へそれ以外のノードを通ってたどり着く経路が存在するようなエッジは明らかに無意味である。そこで、この条件を満たすエッジを削除する。

パス分割　パス分割フェーズでは、与えられた GPDG を次のような条件を満たすパスに分割する。

- 各パスは分岐点ノードもしくは START からはじまり、合流点ノードもしくは LOOP で終る。
- 各ノードは、それを含むパスが必ず一つ以上存在する。但し、分岐点・合流点・START および LOOP 以外のノードは、それを含むパスが複数あってはならない。
- 連続する、自由度の同じノードは、その双方を含むパスが必ず存在する。

この条件にあてはまるよう図 4を分割すると、1-2-3-4-5-6-A-B-C-D-E-F、D-G-F、3-7-A、3-8-9-D の 4 つに分割される。上の定義では分割結果は一意に決まらないが、そのどれであっても構わない。

パスの並び替え　このフェーズでは、分割されたパスを、自由度の低い順番に並び替える。自由度が同じパスに関しては、長さが長い方を先にする。前述のように、自由度の低い経路中にスピルコードが挿入されることは好ましくない。並び替えられた順番にレジスタ割り付けを行なうことにより、自由度の低い経路ほど優先的にレジスタを利用できるようにする。

4.2　各パスのレジスタ割り付け

生存区間の取り出し　切り出されたパス中には、複数の変数が含まれている。レジスタ割り付けを行うためには、それらを解析し各変数の生存区間を求める必要がある。例えば、上述の 1-2-3-4-5-6-A-B-C-D-E-F 中には 11 個の変数が含まれており、それぞれの生存区間は図 7のように解析される。

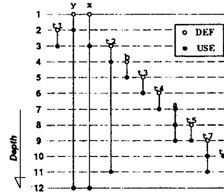


図 7: 生存区間解析の結果

生存区間の並び替え　取り出された生存区間を、次の規則に従って並び替える。

- 生存区間の短いものが優先
- 長さの同じものは、含まれるノード数の多い方を優先

これは、生存区間が長い変数のほうが、スピルコードを挿入する余裕があると考えられるためである。言い替えると、定義されてから参照されるまで時間のある変数に対して、その間ずっとレジスタを占有させることは、レジスタリソースを無駄使いしていることになる。

図 7の各生存区間を並び替えると、t8-t1-b-t3-t4-t5-a-t7-y-x の順になる。

生存区間の割り付け　取り出された生存区間に対して、並べ替えられた順番に、利用可能なレジスタを割り付けていく。図 7の各生存区間を、レジスタ数が 6 個のプロセッサに割り付けると、図 8のように割り付けられる。

割り付け可能なレジスタが存在しないならば、次章に示す方法で、割り付けが可能になるように GPDG に手を加える。

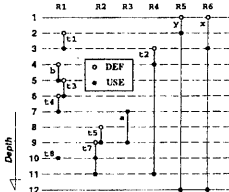


図 8: レジスタが 6 個の場合

4.3 スピル処理

生存区間に割り付けるレジスタがない場合、GPDG に対して何らかの変更を加えて割り付けが可能となるようにしなければならない。その手段として、“レジスタの交換”、“ノードの移動”、“スピルコードの挿入”の3つを考える。これらの方法は、後者にいく程、プログラムの性能を大きく落す可能性がある。従って、これらの手段は順番に行なわれ、前の処理ではレジスタを割り付け可能にすることができない場合に、後者へと進んでいく。

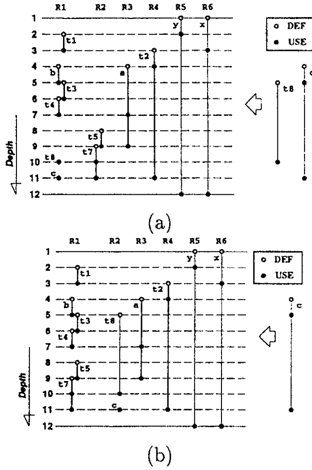


図 9: レジスタ交換の例

レジスタの交換 レジスタの交換とは、既に割り付けが終了している変数に対して、いずれかのレジスタの利用可能な区間が広がるように、割り付けられたレジスタを他の適当なレジスタに変更する事である。図8の例に対して、さらに割り付けを続けていくと、図9(a)のようになる。ここで、変数 t8 を割り付けるのであるが、t8 が利用できるレジスタがない。しかし、同図(b)のように、レジスタ R1 とレジスタ R2 の一部 (Depth=8 以降) を入れ換えることで、変数 t8 をレジスタ R2 へ割り付けることが可能となる。

ノードの移動 図10(a)において、太線で表されたパス上の変数 i3 にレジスタを割り付ける場合を考える。図左の線はレジスタの利用状況と変数 i3 の生存区間を示しているが、変数 i3 が利用可能なレジスタは存在しないことがわかる。そこで、次のような処理を施す。

1. レジスタ R2 に割り付けられた後半の変数を、レジスタ R3 に割り付け直す (図10(b))。図中の太い点線は、レジスタの入れ換えによって発生した逆依存を表す。

2. 矢印でしめされたノードを、レジスタ R2 が利用可能となる位置まで移動する (図10(c))。

この操作によって、変数 i3 をレジスタ R2 へ割り付ける事が可能となる。

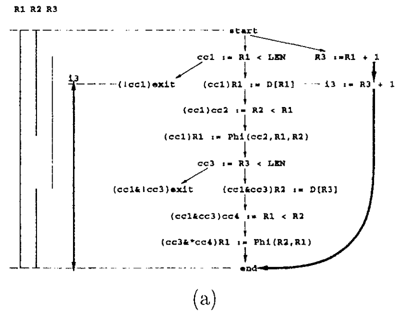
ノードの移動は、他の割り付けの終了していないノードの自由度をも圧迫する。そのため、過度のノード移動は、それ以降のレジスタ割り付けに悪影響を及ぼすことが考えられる。そこで、ノード移動に次の条件を加える。

条件

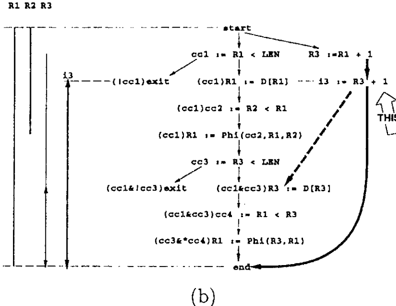
ノード移動後の GPDG において、そのノード移動によって割り付けが可能となったノードの集合を X、割り付けが終了していないノードの集合を Y とするとき、

$$n \in X, m \in Y, DEPTH(n) > DEPTH(m)$$

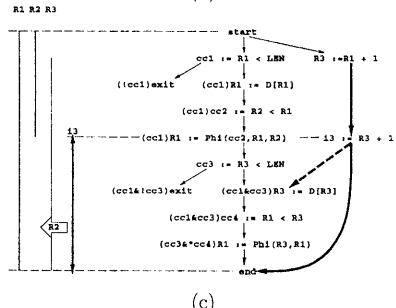
を満たす n, m が存在してはならない。



(a)



(b)



(c)

図 10: ノード移動の例

スピルコードの挿入 上記の“レジスタ交換”、“ノードの移動”のいずれの手法を用いても割り付けが可能とならない場合には、プログラム中に適当なスピルコードを挿入する必要がある。

具体的には、割り付けようとしている生存区間において、変数の値を定義する命令の直後にスピルアウト命令

を、その値を参照する命令の直前にスピルイン命令を挿入する。これにより、その生存区間は複数の小さな生存区間に分割され、割り付けが可能となる。例えば、図9(b) において、変数 c を割り付ける場合、1個のスピルアウト命令と2個のスピルイン命令が挿入され、3個の生存区間に分割される。

また、場合によっては既に割り付けの終了している生存区間に対してスピルコードを挿入することも考えられる。例えば、図9において、レジスタ R5 に割り付けられている変数 y は、スピルコードを挿入することで、レジスタの大部分を利用可能にすることができる。また、最初に値が定義されてから、最後にその値が参照されるまでに時間的な余裕があるため、例えその間にスピルコードが挿入されたとしてもクリティカルパスへの影響は少ない。即ち、挿入されたスピルコードは他の命令と並行に実行される。

実際には、両者の内、実行後のクリティカルパスの延びの短い方を採用する。この例では、前者の方が選択され、最終的な割り付け結果は図 11のようになる。

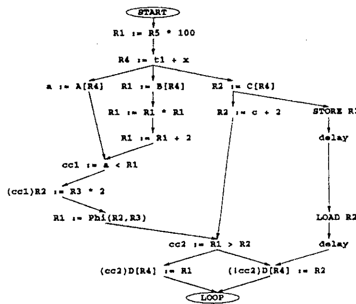


図 11: GPDG 例の最終割り付け結果

5 評価

図 12に、従来の彩色法による割り付けとの比較結果を示す。評価は、“フィボナッチ数列の計算” および “Knuth の乱数発生法” のプログラムの最内ループに対して、割り付け後のプログラムのクリティカルパスの長さで評価した。従って、数値が小さいほど高速にプログラムを実行できることになる。上記の結果では、本手法を用いた場合の方が最大2倍程度高速に実行できることがわかる。他のプログラムにおいても、ほぼ同様の結果が得られた。

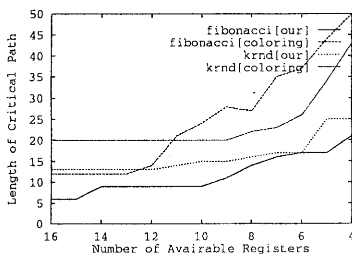


図 12: 従来手法との比較

従来のレジスタ割り付け手法においては、レジスタ数が減少するに従って、プログラムのクリティカルパスが大きく延びている。これは、スピルコードがクリティカルパス上に挿入され、プログラムの並列性が大きく失われてしまうためである。しかし、本手法の場合には、スピルコードができるだけ他の命令と並列に実行できるように割り付けを行なうため、プログラムの並列性を大きく落すようなことはない。

また、プログラム中の変数の大部分は、計算の途中で必要となる一時的な変数である。そのような変数の生存区間は一般に非常に短かく、GPDG 上で移動した場合のコストが小さいため、上記の “ノード移動” の手法が有効である場合が多い。そのため、本手法の方が、挿入されたスピルコード数が少なくなる場合も見られた。

また、レジスタが十分にある場合にも、従来手法では割り付けによって発生したリソース依存によりプログラムの並列性を落してしまっている。そのため、本手法と比べると使用しているレジスタ数は少ないが、プログラムのクリティカルパスは2倍程度に延びてしまっている。

6 おわりに

本稿では、GPDG を利用しプログラムのクリティカルパスを解析することにより、命令レベル並列プロセッサにおいて最適なレジスタ割り付けを行なう手法を提案し、従来手法との評価を行なった。

評価結果から、並列性を持つプロセッサにおいては本手法が有効であることがわかる。逆に、従来の割り付け手法をそのまま利用した場合には、プログラムの並列性が大きく損なわれてしまう。従って、命令レベル並列プロセッサにおいては、本手法のような新しいレジスタ割り付け手法が必要とされる。

参考文献

[1] G.J.Chaitin, “Register Allocation & Spilling via Graph Coloring”, Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction(June 1982), pp98-105.

[2] D.Callahan and B.Koblenz, “Register Allocation via Hierarchical Graph Coloring”, Prceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation(June,1991), pp192-203.

[3] 丹羽, 小松, 新井, 深澤, 門倉, “クラスタリングによるレジスタ割付け”, 電子情報通信学会春期全国大会 (1991), D-80, p6-8.

[4] J.Ferrante, K.J.Ottenstein and J.D.Warren, “The Program Dependence Graph and Its Use in Optimization”, ACM Transactions on Programming Languages and Systems(July 1987), Vol.9, No.3, pp319-349.

[5] R.Cytron, J.Ferrante, B.K.Rosen, M.N.Wegman and F.K.Zadeck, “An Efficient Method of Computing Static Single Assignment Form”, Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages(January 1989), pp25-35.