アルゴリズム 34－8
（1993．8．19）

クリティカルパス情報を利用したレジスタ割り付け手法

神力 哲夫† 小松 秀昭‡ 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、その性能を引き出すための最適化コンパイラが必要である。しかしながら、レジスタ割り付けに関する考察はあまり行われておらず、スカラプロセッサ用に開発された従来の手法をそのまま利用していることが多い。しかしながら、これらの手法ではプロセッサの並列性を考慮できず、いくつかの問題が発生する。本稿で述べるレジスタ割り付け手法では、プログラムのクリティカルパスを検出し、それを重視した割り付けを行なう。これによって、上記の問題を解決し、プロセッサの並列性を有効に利用したレジスタ割り付けが可能となる。

Register Allocation via Critical-Path Analysis

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

Instruction level parallel processor requires optimizing compiler in order to achieve a good performance. However, there has been few researches that take register allocation for these processors into consideration, thus, it is dominant to apply conventional techniques developed for scalar processors as it is. Since these techniques do not cover instruction parallelism, effective use of processors cannot be expected. This paper presents some methodologies which detect critical path and carry out register allocation based on it, that realizes effective utilization of processor parallelism.

1 はじめに

プロセッサ技術の進歩に伴い、比較的小規模な計算機システムにおいても最適化コンパイラの重要性が高まっている。特に、最近になって開発されている命令レベル並列プロセッサでは、最適化コンパイラなしでその性能を十分に引き出すことは難しい。

最適化コンパイラの設計・開発においては、コードスケジューリングやループ最適化などのコード最適化と、レジスタ割り付け手法が重要である。しかしながら、命令レベル並列プロセッサに向けた研究は前者に関するものがほとんどであり、プロセッサの並列性を考慮したレジスタ割り付け手法は確立されていない。そのため、スカラプロセッサ向けに開発された従来のレジスタ割り付け手法をそのまま流用していることが多かったが、このような手法ではプロセッサの並列性の有効利用は困難である。

本稿では、命令レベル並列プロセッサを対象とした場合の従来手法の問題点を明確にし、さらに、それらの問題を解決するレジスタ割り付け手法を提案する。

2 本研究の背景

2.1 従来のレジスタ割り付け手法

現在、最も一般的なレジスタ割り付け手法はグラフ彩色法を利用した手法である [1][2] レジスタ割り付けは、生存区間が重なるなどして互いに干渉し合う（つまり同じレジスタに割り付けられない）N 個の変数を、$R(<N)$ 個のレジスタに割り当てる問題と考えることができる。グラフ彩色による手法は、この問題を N 個のノードからなる無向グラフを R 色で塗り分ける問題として取り扱うものである。各ノードは、対応する変数が干渉する場合に限ってエッジで結ばれる。このようにして作成されたグラフは干渉グラフと呼ばれる。もし干渉グラフを R 色で彩色することが不可能であれば、いくつかの変数を一時的にメモリへ待避し、変数間の干渉を減少させなければならない。一般に、このような変数をスピル変数と呼び、変数をメモリへ退避するために挿入される命令をスピルコードと呼ぶ。スピル変数を決定することで変数間の干渉が緩和されるため、割り付け（彩色）に必要なレジスタ数（色数）を減少させることができる。割り付けは、彩色数が減少するようにスピル変数を決定していき、干渉グラフが R 色以下で彩色可能となったら終了する。

ある変数をスピル変数にした場合、プログラム中に挿入されるスピルコードによってプログラムの実行時間が伸びてしまう場合がある。従って、スピル変数の決定は、それに付随するプログラムの実行効率低下を予測し、それが最少になるように行なうのがよい。従来の一般的な手法では、各変数のプログラム中の使用回数などからレジスタへ割り付ける優先順位を決定し、その優先順位が低い変数からスピル変数としていく。これまで、数多くのレジスタ割り付け手法が発表されているが、それらの多くは優先順位を決定するためのヒューリスティクスを改良したものである。

しかしながら、従来の手法ではプロセッサの並列性を考慮しておらず、命令レベル並列プロセッサに適用した場合に、以下に述べるようないくつかの問題が発生する。

2.2 従来手法の問題点

スピルコードの意味 レジスタ割り付けの本来の目的は、実行時のメモリ参照による効率低下をできる限り抑えることにある。スカラプロセッサのような1サイクルに1命令しか実行できないプロセッサの場合、プログラム中にスピルコードを挿入することは、必ず実行効率の低下につながるため、前述の目的のためにはプログラム中に挿入されるスピルコードの最少化を考えれば良かった。しかしながら、プロセッサが命令レベルの並列性を持つような場合、図1の (a) のように、並列性を利用することでスピルコードを埋め込んでしまうことができる。従って、従来手法のようにスピルコードの最少化のみを追求しても意味がない。

```
t1:=a*b   load t6,l          t1:=a*b   t5:=l*m
t2:=t1*c  nop                t2:=t1*c
t3:=t2*d  t5:=t6*m           load t6,d
t4:=t3*e                     nop
x:=t4*t5                     t4:=t2*t6
                             x:=t4*t5
                             x:=t4*t5

(a)                          (b)
```

図 1: スピルコードの埋め込み

図1からもわかるように、プログラムにはスピルコードが挿入されても実行効率に何ら影響を与えない部分（a の場合）と、逆に、そのようなコードの挿入が実行効率に対して大きく影響してしまう部分（b の場合）とが存在する。後者のような部分にスピルコードが挿入されることは、実行速度の点から好ましくない。従って、そのような部分が優先的にレジスタを利用できるような手法が必要である。

レジスタ数の最少化 命令レベル並列プロセッサに非常に有効なループ最適化手法として、ループアンローリングと呼ばれる手法がある。これは、プログラム中のループを展開する手法であるが、ループにまたがって潜んでいる並列性を抽出し、プログラムの並列度を飛躍的に向上させることができる。しかしながら、この手法は使用する変数の個数を増大し、レジスタリソースを圧迫する。展開によって変数を大幅に増加させ、スピルコードを大量に発生させてしまった場合、逆に実行効率を落してしまうことも考えられるため、ループアンローリングを効果的に行なうためには、レジスタリソースの考慮が不可欠である。このループアンローリングはレジスタ割り付け終了後に行なうのがよい。なぜなら、レジスタ割り付け前に行なった場合、使用可能なレジスタ数の判定が難しく、効果的なループ展開数を見積もることが困難となるためである。

従来の手法はレジスタ割り付けをコンパイラの最終フェーズとして実現されているため、R(レジスタ数)色

以下の彩色方法が見つかれば、さらに少ない色数での彩色が可能であるような場合でも、その時点で割り付けを終了してしまうものが多い。しかしながら、レジスタ割り付け後にループアンローリングを行なうために、できる限り多くのレジスタを残しておきたい。すなわち、上記の状態で彩色を停止させず、さらに少ない色数での彩色を試みることが必要となる。

プログラムの構造的な性質の反映の必要性 プログラムには、局所性と呼ばれる重要な性質がある。即ち、プログラムはその実行時間の大部分を、最内ループのようなプログラム中の一部の部分で費やす。レジスタ割り付けにおいて、そのような実行頻度の高い部分にスピルコードが挿入されることはプログラムの実行効率を大きく落してしまう。逆に、実行される頻度の低い部分へのスピルコードの挿入は、プログラムの実行効率への影響が小さくなる。

実行頻度の差は、プログラムのループ構造と条件分岐構造によって決定される。内側のループは外側のループよりも、分岐確率の高い部分は低い部分よりもプログラムの実行効率にとって重要である。当然、同一変数の生存区間の中にも、実行頻度の高い部分とそうでない部分とが存在する。例えば、複数のループにまたがる生存区間を持つ変数の場合、外側のループに含まれる部分と内側のループに含まれる部分とでは実行効率に対する重要度が違ってくる。従って、それらの割り付けはそれぞれ独立に行なうのがよい。従来手法では、これらをひとつの生存区間として取り扱うため、例えば、実行頻度の高い内側ループ内ではレジスタが余っているにもかかわらず、外側ループ内で利用可能なレジスタがないために、内側ループ内に不要なスピルコードが挿入されてしまう場合がある。このような問題を避けるためには、プログラムをループ構造や条件分岐構造などから、実行頻度に応じて分割し、実行頻度の高い部分から局所的にレジスタ割り付けを行なっていくような手法が必要である。この問題はプロセッサの並列性に依存するものではなく、スカラプロセッサ向けのレジスタ割り付けにおいても考慮されるべきである。

3 本手法の特徴

3.1 クリティカルパス解析によるレジスタ割り付け

命令レベル並列プロセッサ向けのレジスタ割り付けでは、スピルコードの最少化のみを考えても意味がないことを述べた。本手法では、この問題に対して、次のような解決法を考える。まず、プログラムを命令間の依存関係で表現したデータフローグラフで表現する。このデータフローグラフにおいて、プログラムの開始を表すノードから終了を表すノードへ至る経路は複数存在する。それらの中で、最も長いものをクリティカルパスと定義する。プログラムを十分な並列度を持ったプロセッサで実行する場合、クリティカルパスの長さを実行時間の指標として考えることができる。そこで、本手法ではこのクリティカルパスを延ばさないことをレジスタ割り付けの目標とする。

具体的には、データフローグラフからクリティカルパスへの影響が強い順番にパスを抜きだし、レジスタ割り付けを行なっていく。このようにすることで、クリティカルパスの長さ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できる。

3.2 埋め込み法によるレジスタ割り付け

ループアンローリングを効果的に実行するために、レジスタ割り付けにおいてはできるだけ少ない個数のレジスタへ割り付けることを考えなければならないことを述べた。本手法でのレジスタ割り付けは、グラフ彩色法によるものとは幾分異なる。すなわち、レジスタの使用状況を表にしておき、各変数の生存区間を照らし合わせて、その生存区間を埋めこむことが可能なレジスタへ割り付けていく。この手法ならば、最終的に割り付けられるレジスタの個数は、必然的に最少なものになる。また、後に述べるSSA変換によって、変数間の無駄な干渉を取り除くことが可能になるため、従来の手法に比べて少ないレジスタ数でのレジスタ割り付けが可能となる。

3.3 クラスタ分割

文献[3]では、プログラムをループ構造を反映した階層的な構造に分割し、内側のループから順番にレジスタを割り付けていく手法が提案されている。このようにすることで、実行頻度の高い内側のループほど優先的にレジスタを利用できるようになる。また、各変数はその生存区間を分割されて割り付けられることになるため、実行頻度の高い内側のループ内ではレジスタに割り付けられ、外側のループ内ではスピル変数になるといった割り付けも可能である。

本手法でも、この手法同様にプログラムをループを単位とした構造に分割する。本手法と、[3]の手法との違いは、各ループ内部のレジスタ割り付けにある。[3]の手法では、各ループ内のレジスタ割り付けは従来のグラフ彩色法を利用していた。本手法では、各ループにおいて3.1に述べたような割り付けを行い、命令レベル並列プロセッサに対応する。

3.4 SSA変換の導入

本手法の重要な特徴の一つにSSA変換[6]の導入がある。SSA変換とは各変数の名称を代入が新たに行なわれる度に変更することにより、各変数への代入命令を生存区間の開始点の一回限りにする手法である。例えば、図2の(a)のプログラムをSSA変換すると同図の(b)のようになる。

SSA変換を導入することにより、変数間の無意味な干渉をなくし、割り付けに必要なレジスタ数を減少させることが可能である。変数間の無意味な干渉とは、次のようなものである。例えば、図2において、(a)の状態ではxとpは互いに生存区間が重なっているため、同一のレジスタへ割り付けることができない。しかしながら、xの生存区間の内、y = x * 2からx = b + 2までの間は、xの値を保存しておく必要はない。なぜなら、後者の命令によって、xの値は書き換えられてしまうためである。従って、変数xと変数pは実際は干渉しておら

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

(a)SSA 変換前

```
x1 = a1 + 2;
y1 = x1 * 2;
...
p1 = ...;
...
x2 = b1 + 2;
...
l1 = ...;
if (...)
    l2 = l1 + 2;
else
    l3 = l1 - 2;
l4 = Φ (l2,l3)
...
l5 = l4*l4;
```

(b)SSA 変換後

図 2: SSA 変換の例

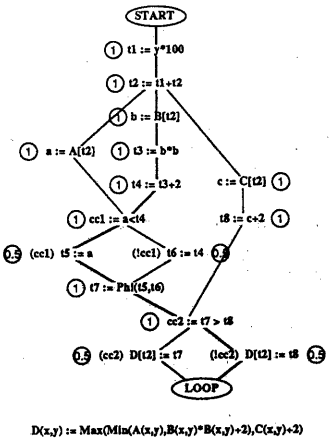

図 3: GPDG の例

ず、同一のレジスタへ割り付けることができるはずである。このように、SSA 変換される前のプログラムでは、実際には存在していない干渉が含まれている。SSA 変換後の (b) においては、x は x1 と x2 の二つの変数に分割され、上記のような問題は解決される。

4 本手法の概要

本手法は、大きく二つのフェーズから構成される。一つは、[3] の手法と同様に、プログラムのループ構造を反映した階層的な構造に分割するクラスタ分割フェーズ、もう一つは、それぞれのクラスタ内部に対して、クリティカルパス情報を利用してレジスタ割り付けを行なうフェーズである。また、プログラムは GPDG と呼ばれるデータ構造で表現されている。それぞれの特徴を以下に述べる。

4.1 GPDG とクリティカルパス

GPDG(Guarded Program Dependence Graph) は、プログラム中の各命令に対応するノードとその間のデータの生産・消費関係を表すエッジからなるデータフローグラフである。このような構造を表現するものとして PDG(Program Dependence Graph) が有名であるが、条件分岐を含むプログラムを表現するためには、コントロールフローグラフなどと組み合わる必要がある。従って、複数の基本ブロックにまたがるような最適化は非常に複雑なものとなってしまう。GPDG はこの点を改良し、条件分岐を含むプログラムを、データフローグラフのみで表現可能にしたものである。この GPDG を利用することにより、複数の基本ブロックにまたがる大域的な最適化が容易に行える。

条件分岐の概念をデータフローグラフ中に取り入れるため、各命令に対して、それが実行時に満たすべき実行条件を付加する。実行条件は条件変数と呼ばれる条件分岐の結果を表す仮想的な変数の真偽の組み合わせによって表されており、プログラム中の条件分岐命令はこれらの条件変数への代入命令で表される。図 3においては、cc1 および cc2 が条件変数を表している。これによって、プログラム中の制御依存を他のデータ依存関係と同様にデータ（条件変数）の生産・消費関係のみで表せるようになる。

この GPDG は、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、終端を表す LOOP ノードが含まれ、また、バックエッジは LOOP ノードから START ノードへ至る一本に限定される。

GPDG の場合、条件分岐の概念を含んでいるため、それぞれの命令の実行される確率は実行条件によって変化する。従って、クリティカルパスの定義を次のようにする。まず、各ノードに対して実行確率による重み付けを行ない、START ノードから LOOP ノードへ至る経路中で、その重みの和が最大となるような経路をクリティカルパスと定義する。図 3において、条件分岐の分岐確率は 50% ずつであるとすると、各ノードの重みは○中の数字のようになる。この値を用いてクリティカルパスを求めると、太線のパスが選択される。

4.2 クラスタ分割フェーズ

与えられたプログラムのループ構造を解析し、ループを単位とした階層的な構造に分割する。分割された各部分をクラスタと呼ぶ。例えば、図 4の (a) のプログラムは、(b) の様にクラスタ分割される。

我々がプログラムをクラスタに分割する理由は二つある。一つは、3.3でも述べたように、プログラムの構造的な性質をレジスタ割り付けに反映させるためである。こうすることにより、プログラムの実行頻度の高い部分にスピルコードが挿入されることをできる限り抑える。もう一つは、レジスタ割り付けにかかる時間の削減である。従来のグラフ彩色法を用いたレジスタ割り付けでは、プログラム全体に対して大きな干渉グラフを作成していた。プログラムは巨大化する一方であり、彩色に利用される干渉グラフも非常に大きなものとなっている。そのため、レジスタ割り付けフェーズはコンパイラの処

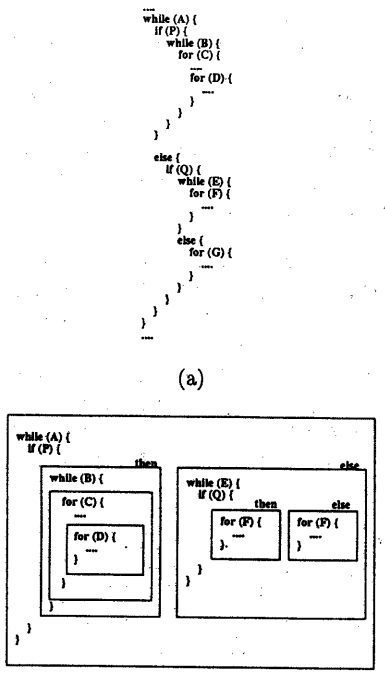

図 4: クラスタ分割の例

理の中でも最も時間のかかる部分となっていた。最適化コンパイラにおいては、作成された実行コードの質を向上させることも重要であるが、コンパイル時間を短くすることも同様に重要である。コンパイルに膨大な時間のかかるコンパイラは、作業効率の悪化をもたらす。本手法では、プログラムを分割し、それぞれに対して局所的なレジスタ割り付けを行っている。即ち、分割統治によりレジスタ割り付けに必要な計算量を減少させていることになる。

4.3 クラスタ内部のレジスタ割り付け

クラスタ内部のレジスタ割り付けは、GPDG においてクリティカルパスをできる限り延ばさないことを目標として進められる。具体的には、GPDG からクリティカルパスへの影響が大きい順にパスを抜きだし、割り付けを行なっていく。このパスの選択は、途中に含まれるノードの重みの和が大きい順に選ばれるが、分岐確率の高いパスほど優先順位は高くなる。そのため、[3] の手法では考慮されていなかった、プログラムの分岐構造による実行頻度の違いを考慮することが可能となる。また、各ループ内部においても変数の生存区間は分割されて割り付けられ、クリティカルパスに影響を与えやすい部分ではレジスタへ割り付けられ、そうでない部分ではスピル変数となるような割り付けも可能となる。

5 クラスタ分割

我々はプログラムの制御構造を反映し、プログラムの重要な部分ほどレジスタを優先的に利用可能にする。そのために、プログラムをクラスタと呼ばれるループを単位とした階層的な構造に分割する（図4参照）。

いくつかのクラスタにまたがるような長い生存区間を持つ変数は、その生存区間をいくつかに分割され、それぞれ独立して割り付けられることになる。そのため、内側のクラスタでの割り付け結果と、外側のクラスタでの割り付け結果とが異なる可能性がある。このような場合、どちらかのクラスタ内でレジスタの入れ換えを行なうか、内側クラスタへの入口に適当なデータ転送命令を挿入するなどして、クラスタ間の整合性をとる必要がある。そのため、各クラスタの入口と出口とに図5のようなインターフェースノードを挿入し、適当なデータ転送命令を挿入する。

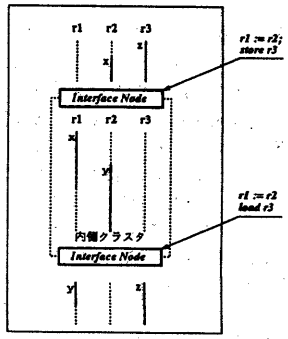

図 5: インターフェースノードの例

5.1 インターフェースノード

インターフェースノードは、クラスタへの入口とクラスタからの出口とに挿入される。全てのクラスタでの局所的なレジスタ割り付けが終了してから、このインターフェースノードへの命令の挿入が、内側のクラスタから外側のクラスタへと行われる。

インターフェースノードにデータ転送命令を必要とするのは、生存区間が複数のクラスタにまたがっているような変数である。そのような変数には、

1. 注目クラスタ内で参照されているもの
2. 注目クラスタ内で一度も参照されていないもの

とがある。1. の場合、内側クラスタと外側クラスタでの割り付け状況が異なれば、適当なデータ転送命令が必要になる。具体的には、どちらか一方のクラスタでレジスタに割り付けられ、他方でメモリへ割り付けられているような場合に、メモリ・レジスタ間の転送命令が、双方のクラスタでレジスタに割り付けられているにもかかわらず、レジスタ番号が異なるのであれば、レジスタ間の転送命令が、適当なインターフェースノードに挿入される。

2. のような変数は、内側のクラスタでのレジスタ割り付け時には全く考慮されていない。従って、内側クラスタではレジスタにもメモリにも割り付けられてはいない。そのため、内側のクラスタ中でこの変数の値を保存するために、入口のインターフェースノードにメモリへの書き込み命令を、出口へのインターフェースノードにメモリからの読み込み命令を挿入する。もちろん、外側のクラスタにおいてメモリに割り付けられているような場合には、このような処理は必要ない。

もし、内側のクラスタ内でレジスタが余っているのであれば、そのような変数をレジスタへ割り付けることもできる。しかし、我々はレジスタ割り付けの後のループアンローリングのために内側クラスタのレジスタはできるだけ空けておくようにしている。もし、ループアンローリングによる最適化効果よりも、そのような変数をレジスタに割り付けておく効果の方が大きいと判断できるならば、レジスタへ割り付けてしまうこともできるが、ループアンローリングの効果を事前に予測することは難しい。

実際には、個々のクラスタのレジスタ割り付けが全て終了した後に、インターフェースノード中に必要な命令を挿入していく。また、内外のクラスタどちらか一方において、レジスタの交換が可能な場合、変数が連続して同一のレジスタへ割り付けられるようにレジスタの適当な交換を行なう。例えば、ある変数が内側のループにおいてレジスタ $R1$ に割り付けられ、外側のループにおいてレジスタ $R2$ に割り付けられており、かつ、内側ループにおいて $R1$ と $R2$ とが交換可能なのであれば、交換を行いレジスタを連続させる。

6 クラスタ内部の割り付け

各クラスタ内部のレジスタ割り付けは、作成されたGPDGからクリティカルパスへの影響が強い方から順番にパスを抜きだし、それぞれレジスタ割り付けを行なっていく。このとき、各変数の生存区間はいくつかに分割されて、それぞれ部分的に割り付けられていくことになる。このため、変数が連続して同一のレジスタに割り付けられずに、いくつかのレジスタに跨って不連続な割り付けをされてしまうことが考えられる。そこで、本手法ではレジスタスワップと呼ぶフェーズを用意し、そこで変数のレジスタへの割り付けができるだけ連続するように調整を行なう。

6.1 割り付け

パスの選択 割り付けを行なうパスは、クリティカルパスから始めて、クリティカルパスへの影響の強い方から選びだされる。具体的には、GPDGのSTARTノードからEXITノードへ至る全経路を、途中に含まれるノードの重みの和が大きい順に並べ、順番に選択する（ノードの重みは、前述のように各ノードの実行確率から計算される）。これは、重みの和が大きなパスの方が、スピルコードが挿入された場合に、そのパスが新たなクリティカルパスとなってしまう可能性が大きいからである。

生存区間表の作成 抜き出されたパスに対して、その中で使用されている変数の生存区間を解析することが必要となる。生存区間は、GPDG中の命令のSTARTノードからの距離を利用して定義する。例えば、図3において、変数t4を最初に使用している命令t4:=t3+2のSTARTノードからの距離が5、t4を最後に使用している命令t6:=t4の距離が7であることから、t4の生存区間は5～7であるとする。ここで、GPDG中のノード n のSTARTノードからの距離とは、STARTノードからノード n へ至る経路中で、途中に含まれるノード数が最大になる場合のノード数である。ここではノードの重みは考慮しない。

あるパスが与えられた場合に、その上の変数の生存区間は以下のようにして決定する。もし、それ以前の割り付けにおいて、その変数の割り付けが一度も行なわれていないのであれば、変数の生存区間はそのパス中でその変数を使用している最初の命令から最後の命令までとなる。他方、その変数の割り付けが以前のパスにおいて既に部分的に行なわれているような場合には、図6の変数cのように、以前の割り付け部分を含むように前後に生存区間を延ばす。

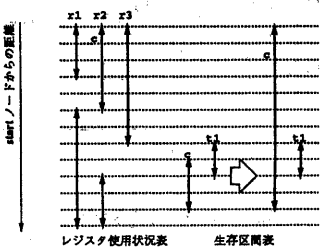

図 6: 生存区間の決定

パス上の変数の割り付け まず、次のような規則に従って、各変数の割り付けを行なう優先順位を決定する。

1. パス上での利用回数の多い変数ほど優先順位を高くする。
2. 利用回数が同じである場合には、生存区間が短いものほど優先順位を高くする。

変数の優先順位が決定したら、その優先順位の高い方から順番に割り付けを行なっていく。具体的には、変数の生存区間とレジスタ使用状況表とを比較し、空いているレジスタを割り付けていく。このとき、次の3通りが考えられる。

1. “変数の生存区間全体に渡って利用可能なレジスタ”が存在する（図7変数a）
2. “変数の生存区間全体に渡って利用可能なレジスタ”は存在しないが、生存区間全体に渡って、いずれかのレジスタが利用可能である（同b）
3. “変数の生存区間全体に渡って利用可能なレジスタ”は存在せず、かつ、生存区間の一部において、全てのレジスタが利用不可能である（同c）

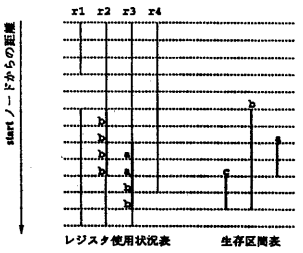

図 7: 生存区間表とレジスタ使用状況表との比較

1. の場合は、利用可能なレジスタに変数の全生存区間にわたって割り付ける。2. の場合は、複数のレジスタへ生存区間を分割して割り付ける。このとき分断されて割り付けられた変数は、後のレジスタスワップフェーズにおいて、連続して同一のレジスタへ割り付けられることが保証されている。3. の場合、ある時点において必要なレジスタ数がプロセッサのレジスタ数を超過していることになるため、このままではレジスタ割り付けを行なうことができない。次に述べるような処理を行ない、いずれかの変数がその区間を使用しないようにする必要がある。

利用可能レジスタがない場合の処理 GPDG 中の各命令はそれぞれスケジューリングされうる範囲を持っている。例えば、図 3において命令t8:=c+2は、START ノードからの距離が 4 の位置にある。しかしながら、この命令の結果を利用する最初の命令 cc2 := t7>t8 は、START ノードから 9 の位置にある。従って、最初の命令は 4 ～ 8 のどこかで実行されればよいことになる。この各ノードのスケジューリングの余裕の幅をマージンと呼ぶことにする。従って、先頭命令のマージンが n であるような生存区間は、その生存区間の開始を n だけ後方にずらすことができる。そこで、割り付けるべきレジスタがない場合には、次のような順で問題の解決をはかる。

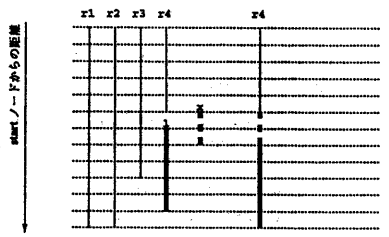

図 8: 生存区間の後方移動

1. 重なっている生存区間のうち、いずれかの生存区間の開始を後方に移動する。
2. 割り付けを行なおうとしている変数をスピル変数とする。

1. の場合にはクリティカルパスが延ばされる可能性はない。しかしながら、2. の場合には挿入されたスピルコードによって、クリティカルパスを延ばしてしまうことが考えられる。そのような場合には、次のようなことも考慮する。

命令レベル並列プロセッサの場合、変数の生存区間の始めから終りまで、終始レジスタに割り付けられている必要はない。例えば、図 9の場合、(a) のように割り当てられたレジスタ R1 の一部を、(b) に示すようメモリにスピルすることができる。この場合、プログラムの実行時間は伸びていない。一般的にいうと、レジスタ → メモリの転送に N クロック、メモリ → レジスタの転送に M クロックかかるプロセッサにおいて、変数の生産命令の後ろ N クロックと、消費命令の前 M クロックはそのレジスタ上になければならないが、それ以外の部分ではメモリへスピルすることができる。図 9の場合、生存区間の中の太線で表されている部分（強い生存と呼ぶ）では変数がレジスタ上になければならないが、細線の部分（弱い生存と呼ぶ）はそうでなくとも良い。即ち、この細線の部分は他の変数の割り付けに利用することが可能である。

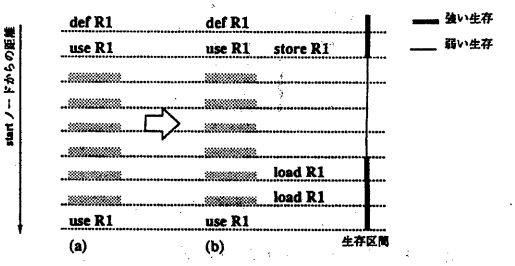

図 9: 実行時間を延ばさないスピル

そこで、重なっている生存区間の中に、弱い生存区間が含まれているのであれば、上記のようなスピルコードを挿入し、生存区間の重なりを解決する。この手法はコード量を増大させてしまうため、クリティカルパスを延ばしてしまうような状態でしか考えない。これは、一般的には、レジスタリソースに比べて ALU リソースの方が不足しがちであると考えられるためである。もし、この手法も不可能であるのならば、クリティカルパスを延ばすようなスピルコードを挿入する。

GPDG の修正 各変数のレジスタ割り付け終了後、後のレジスタ割り付けのために GPDG の修正を行なう必要がある。なぜなら、同一のレジスタに割り付けられたことにより、各ノード間にそれまでは存在しなかったレジスタリソースの衝突による逆依存の関係が発生するからである。さらに、挿入されたスピルコードなどによっても GPDG が変化する。これらの影響により、各パスのマージンも変化するため、これらの再計算を行なう必要がある。

6.2 レジスタスワップ

本手法では、クラスタ内の変数は断片的にレジスタへ割り付けられていく。そのため、一つの変数が複数のレジスタに不連続に割り付けられてしまう。そこで、レジスタを適当に入れ替えることによって、不連続な割り付けを排除していく。例えば、図10のように割り付けらているのならば、図11のようにレジスタの入れ替えを行なう。

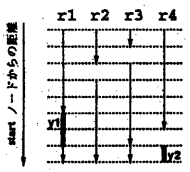

図 10: レジスタスワップ前

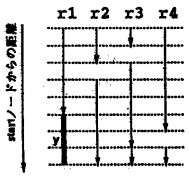

図 11: レジスタスワップ後

実際には、このレジスタスワップフェーズは、レジスタ割り付けと独立したフェーズではない。ある変数の割り付けを行なう度に、前後の割り付けと連続するように、レジスタのスワップを行っていく。

7 評価

本手法の有効性を確認するために、簡単なプログラムにおいて従来手法との比較を行なった。比較に用いたプログラムはクイックソートである。プロセッサの ALU スロットは十分にあるものとしている。

結果は図12に示す通りである。レジスタが7個の場合は、双方ともスピルコードは発生しなかった。レジスタが5個になると、双方ともスピルコードが必要となった。この場合、本手法の方が約 1.17 倍高速になっている。今回は、実際に挿入されたスピルコードの個数は両者とも 2,3 個であるから、プログラムが大きくなり大量のスピルコードが挿入された場合、差はさらに拡大すると思われる。

従来手法ではプログラムのクリティカルパスを考慮していないため、クリティカルパス上にスピルコードが挿入されてしまうことが多くみられた。逆に本手法の場合には、スピルコードの多くはクリティカルパスではない部分に挿入されたため、プログラムの実行時間をそれほど延ばしていない。

図 12: 従来手法との比較

レジスタ数		従来手法	本手法
7	使用レジスタ数	7	7
	実行時間	1	1
5	使用レジスタ数	5	5
	実行時間	1.27	1.09

8 おわりに

本稿では、プログラムのクリティカルパスを解析することで、従来の手法では困難であった、プロセッサの並列性を活かしたレジスタ割り付けを可能とする手法を提案した。本手法を、コードスケジューリングと組み合わせて実行することにより、例えば、レジスタリソースを考慮したスケジューリングが可能となり、さらに効果的なレジスタ割り付けを行なうことが可能となる。今後は、このコードスケジューリング手法との連携を考えている。

参考文献

[1] G.J.Chatin, M.A.Auslander, A.K.Chandra, J.Cocke, M.E.Hopplins and P.W.Markstein, "Register Allocation via Coloring", Computer Languages 6(1981), pp47-57.

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

[3] 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.

[4] 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, p6-80.

[5] 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.

[6] 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.