計算機アーキテクチャ　91－4
（１９９１．１１．２１）

条件実行アーキテクチャＧＩＦＴの最適化コンパイラ

中谷信太郎*1　鈴木英俊*1　小松秀昭*2　深澤良彰*1　門倉敏夫*1

*1:早稲田大学理工学部
*2:日本アイビーエム(株)東京基礎研究所

プログラム中には種々の依存関係が存在し、それにより最適化コンパイラはプログラムの並列化を妨げられている。より高い並列性を持つコードを生成するためにはこの依存関係を削減しなければならず、そのためにはハードウェア側からのサポートが必要不可欠となる。本稿では、拡張ＶＬＩＷアーキテクチャＧＩＦＴを対象に、ＳＳＡ変換の導入、拡張ＰＤＧの利用により個々の依存関係を排除し並列化をおこなう手法について述べる。また、その並列化のためにＧＩＦＴが与えている条件実行、先行実行機構等のアーキテクチャサポートについて述べる。

An Optimizing Compiler for GIFT

Shintaro Nakatani*1　Eishun Suzuki*1　Hideaki Komatsu*2　Yoshiaki Fukazawa*1　Toshio Kadokura*1

*1:School of Science & Engineering, Waseda University
*2:IBM Japan,Ltd. Tokyo Research Laboratory

A program contains various kinds of dependences, which prevent parallelization of an optimizing compiler. In order to increase parallelism of a program, it is necessary to decrease these dependences. For that purpose, some hardware supports such as conditional execution, speculative execution, dynamic memory arbitration are indispensable. In this paper, we describe parallelization techniques based on SSA(Static Single Assignment) transformation and extended PDG(Program Dependence Graph), and hardware supports by the extended VLIW architecture GIFT(Guarded Instruction architecture for Fine-grain Technique).

1．はじめに

命令レベルの並列処理により高速化を目指すアーキテクチャとして、ＶＬＩＷやスーパースカラなどが提案され、注目を集めている[1]-[3]。

スーパースカラ方式は、実行時に命令レベルの並列性を抽出し、並列実行をおこなう。これは、実行時にハードウェアによりデータ依存解析やリソース競合解析をおこなうことによって実現されている。このため、異なる並列度をもつマシン間で、プログラムのバイナリレベルでの互換性があり、拡張性に優れている。また、書かれたままのプログラムをそのまま実行しても並列処理が可能であり、コンパイラの質の低さをハードウェアにより補うことができる。そのため、コンパイラの開発も容易で、機能拡張もしやすい。しかし、実行時の命令供給の際に多くの作業をおこなうので回路が複雑になり、オーバーヘッドが大きくなってしまう。それゆえ、スーパースカラ方式ではマシンの並列度が上げにくい。

ＶＬＩＷ方式は、マイクロプログラムのように多数のフィールドに分けられた非常に長い命令語を持ち、その各フィールドで対応する演算器などの制御をおこなう。命令レベルの並列性の抽出及びスケジューリングはコンパイル時におこなう。実行時の最適化をおこなわないため、マシンの並列度を上げてもオーバーヘッドにつながらず、クロックとマシンの並列度の積で決まるマシンの実行性能が上がりやすいという可能性を持っている。しかし、静的に高い並列性が抽出できない場合、この利点は大きな意味を持たなくなってしまう。しかも、スケジューリングがマシンの命令フォーマットに合せて静的におこなわれるため、バイナリレベルでの互換性はなく、拡張性に乏しい。

書かれたままのプログラムから得られる並列性は、２ないし３命令程度といわれている[4][5]。この程度の並列性しか得られないのであれば、スーパースカラ方式で十分である。

並列性抽出を妨げている原因は、プログラム中に存在する種々の依存関係である。それらは大きく次の３つに分類できる。

(1)正依存性

データの生産者と消費者の関係にある命令間の依存関係。ある命令が生産するデータを別の命令で使う場合、その命令はデータが生産されるのを待たなくてはならない。

(2)反依存性

使用するリソースの衝突により生じる依存関係。出力依存や逆依存（3.1節参照）、同一バンクアクセス時のバストラフィックによる依存、またレジスタ割付けの際に生じる依存も含む。

(3)制御依存性

条件分岐により切り換えられる各コンテクスト間に存在する依存関係。

これらのうち、(1)はプログラム中に含まれる本質的な依存関係であり、これによる依存関係鎖（dependence chain）はそれ以上縮めることができない。しかし、(1)のみを考えた場合に得られる並列性は少なくはない。この並列性を落としているのが(2)および(3)の依存関係である。よってコンパイラはこれらを考慮して、(1)で得られる並列性に近いものを得ることが目標となる。そのため、近年トレーススケジューリング法[6]やパーコレーションスケジューリング法[7]などが考案され成果を上げているが、十分であるとはいえない。コンパイラのみでは得られる並列性に限界があるからである。ゆえに、より高度な並列性を得るためには、アーキテクチャからのサポートが必要不可欠である。

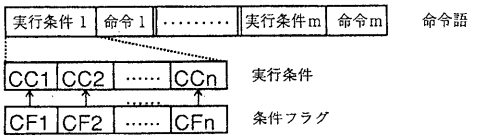
2．ＧＩＦＴの概要

我々はＶＬＩＷ方式の拡張アーキテクチャＧＩＦＴ（Guarded Instruction Architecture for Fine-grain Technique）を提案してきた[8]-[10]。ＧＩＦＴは、ＶＬＩＷ方式の高速化、高並列化の可能性に注目し、条件実行や先行実行によって並列性抽出度を向上させ、さらにスーパースカラの持つような拡張性を持たせることにより、ＶＬＩＷ方式を拡張させたアーキテクチャである。

ＧＩＦＴではそれぞれの命令に、実行する際に満足していなければならない「実行条件」を付加している。実行条件は図１のように複数の条件から構成される。各条件はマシンの持つ個々の条件フラグと対になっており、対応する条件フラグを実行条件にどのように関与させるかを記述している。そして、実行時にマシンの条件フラグの状態を満たす実行条件を持つ命令のみが実行される。これが、ＧＩＦＴの条件実行機構である。これにより、たとえば図２のようなフローグラフを考えた場合、各命令間に正依存性や反依存性が存在しなければ、全体を１命令語として実行可能である。すなわち、条件実行により、制御依存性の削減が可能になり、並列性抽出度が増すことになる。

また、ＧＩＦＴはＶＬＩＷでありながら、拡張性（スケーラビリティ）を持っている。ＧＩＦＴでは、並列性の抽出はコンパイラが無限のリソースを仮定して静的におこない、マシンのリソースに依存したスケジューリングは動的におこなうという二段階スケジューリング方式をとっている。これにより、オブジェクトコードの互換性が保たれ、スケーラビリティを持つことになる。ここで、静的にスケジュールされた命令語を「論理命令語」、リソース構成に従い動的にスケジュールされた命令語を

「物理命令語」と呼ぶことにする。また、この二段階スケジューリングは、コンパイラに対して論理最大並列を求めることを許している。



CCi($1 \leqq i \leqq n$) ：条件コード
CFi($1 \leqq i \leqq n$) ：条件フラグ

n=4 とし

CC1: And ; CC2: Not And ; CC3: Don't Care ; CC4: And

のとき

実行条件 = $(CC1\&\overline{CC2}\&CC4)$

図1　ＧＩＦＴの条件実行機構

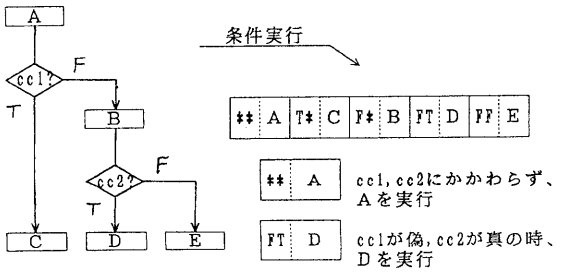


図2　条件実行の例

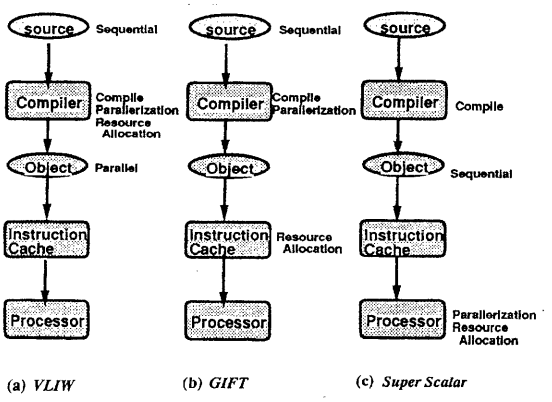


図3　二段階スケジューリング

3．最適化ＧＩＦＴコンパイラ

ＧＩＦＴコンパイラでは、論理最大並列に近い、高い並列性を得るために、反依存性及び制御依存性を可能な限り抑えることを目標とする。本章では各々の依存性を排除するためにＧＩＦＴコンパイラが用いる手法と、その際にＧＩＦＴが与えるアーキテクチャサポートについて述べる。

3．1 反依存性の排除

3.1.1 ＳＳＡ変換による逆依存、出力依存の排除

図４において、命令②は本来命令①で定義されたａの値を使用するため、命令③は命令②より前に実行することはできない。これを逆依存という。また、命令④は本来命令③で定義されたａの値を使用するため、命令③は命令①より前に実行することはできない。これを出力依存という。この逆依存及び出力依存は本質的な依存関係ではなく、並列化を妨げるものである。そこでＳＳＡ(Static Single Assignment)変換を導入し、これらを排除する。

ＳＳＡ変換はコントロールフローグラフに対しておこなわれ、それにより、プログラムは次の２つの性質をもつ形式に変換される。

(1)プログラム中の個々の変数への代入は多くて１回。
(2)Φファンクションの挿入により元のプログラムの意味を静的に保持。

逆依存や出力依存は同じ変数を定義する命令が複数ある場合、それぞれの定義が論理的に正しく使用されることを保証するために生じる依存関係である。しかし、性質(1)によりある変数の定義は一意に限られる。よって、逆依存及び出力依存は排除される(図４)。

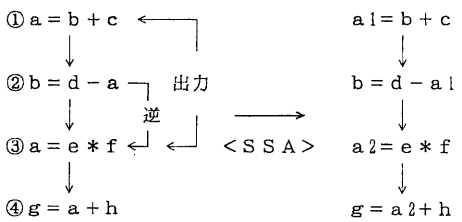


図4　逆依存、出力依存の排除

また、ＳＳＡ変換の際、性質(2)によりプログラム中にΦ-functionが挿入される。これは図５(a)のように、プログラム中の制御の流れの合流点において、到達する各変数の値を静的に切り換えるものである。これによりプログラムの単一代入性が保証される。このΦ-func-

tionは、最終的には図５(b)のように２命令に展開され、プログラム中の別々の流れに挿入される。その結果、この２命令の間に制御依存性が生じるため、並列には実行できなくなる。これらは展開される前は同じ場所に位置しており、本質的に同時実行可能であったため、この展開によって並列性が損ねられてしまうことになる。これに対し、ＧＩＦＴでは、条件実行によるサポートをおこなう。すなわち、条件実行によりΦファンクションは図６のように、それぞれの値を取る実行条件を付加し展開することにより命令として実現される。よってΦ-functionの展開による並列性の損失を避けることができる。

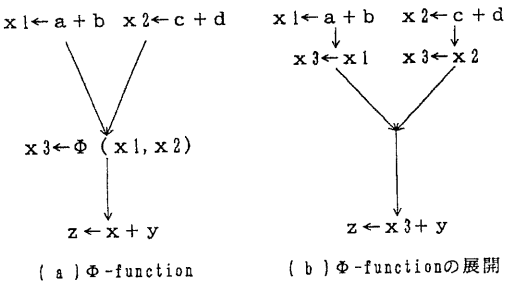


図５　Φ-function

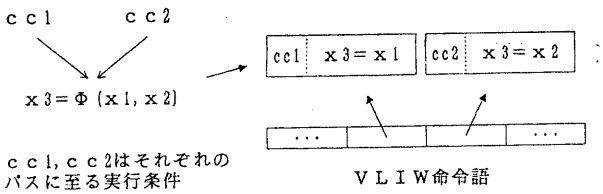


図６　条件実行によるΦ-functionの実現

3.1.2 メモリ参照における並列性の制限の排除

メモリの同一アドレスにアクセスする命令間においては、本来持っていた実行の前後関係が守られなくてはならない。メモリにアクセスする２命令の関係として、アクセスするアドレスが(1)同一である場合、(2)異なる場合、(3)分らない場合、の３通りがある。このうち(1)の場合、この２命令を同時実行できないのはもちろんであるが、(3)の場合においてもアクセスアドレスが同一である可能性があるため、それらが本来持っている実行の前後関係を守るためにはこれらを同時に実行することはできない。すなわち、これらの間には依存関係が存在する。これも反依存性の一種と考えられる。コンパイラはメモリアクセス命令における並列性を得るために、(1)(2)(3)のどの状態であるかをチェックしなければならない。しかし、通常ではこれを完全におこなうことは困難であり[12]、同一性の分らない(3)の場合が多い。この場合、従来、コンパイラは参照アドレスが同一であることを仮定して、本来の実行順序が守られるようにスケジュールする。しかし、実行時には異なるアドレスを参照する可能性があり、その場合、実行時の並列性が失われてしまう。

ＧＩＦＴではバンク化されたキャッシュメモリを持っており、バンクへのアクセスを動的に管理し、バンクが衝突した命令間に対しては、各ロード・ストアユニットに対してあらかじめ定めている優先順位に基づいて実行する機構をそなえている。そのため、メモリアクセス命令間の参照アドレスの同一性が分らない場合、ＧＩＦＴではそれらを本来の実行順に、優先順位の高いユニットから割り付けていくことにより、同時に投入することが可能である。しかも、それらの参照アドレスが実行中に衝突してしまった場合においても、図７のように衝突した命令間の実行サイクルを一部オーバーラップさせて実行するため、シーケンシャルに実行する場合よりもよい結果を出すことができる。よって、ＧＩＦＴコンパイラはこのメモリ参照に関する反依存性を考慮することなくコンパイルすることが可能であり、得られる並列性が向上する。

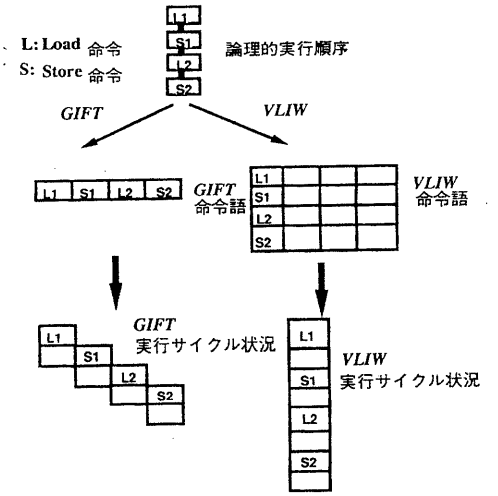


図７　メモリ参照命令のオーバーラップ

３．２ 制御依存性の排除

3.2.1 制御依存性

制御依存性とは、条件文の後に続く命令の実行が分岐条件の真偽のために左右されることを意味する。制御依存性には次の二つのタイプがあると考えられる。一つは、

条件を設定する条件文を実行してからでなければ、その後に続く命令を実行することができないという場合である（図８において、op2及びop3はset cc1よりも前に実行することはできない）。これを確率的制御依存と呼ぶことにする。もう一つは、分岐条件が確定するまでは、それに続くどちらの命令を実行すべきかが分らないため、同じ条件文に実行を左右される命令どうしも並列実行することはできないという場合である（図８において，op2とop3は同時に実行できない）。これを排他的制御依存と呼ぶことにする。

これらの制御依存性がプログラムの並列実行を妨げる大きな要因になっている。

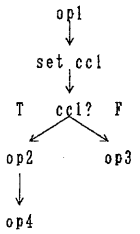


図８　制御依存性

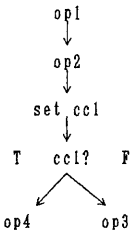


図９　先行実行

3.2.2 先行実行とそのアーキテクチャサポート

確率的制御依存を取り除くためには、条件分岐を越えた先行実行を行う必要がある（図９）。これにより得られる並列性は飛躍的に上昇する。しかし、条件分岐を越えて命令を移動した場合、本来なら実行されるべきでない命令が実行されてしまうことにより、プログラムの実行に支障をきたす可能性がある。すなわち、条件分岐を越えて先行実行された命令が、計算値を生成する際に、キャッシュミスやページフォールト、演算エラー等を発生させ、かつ、その計算値が後の処理で使用されなかった場合、本来ならばおこなわれないはずであるそれらのための復旧処理により、プログラムの実行は大きな遅延を被る。そのため、ＧＩＦＴでは命令に通常命令と先行実行命令の二種類を用意し、先行実行命令がこれらの事態を発生させたときには、その時点では割込みを発生させず、その計算値を後の処理で通常命令が使用する時点で割込みをかけ、復旧処理をおこなうことにより、無用な遅延を抑えている。

これにより、ＧＩＦＴコンパイラは積極的に先行実行をおこなうことが可能になり、高い並列性を得ることが可能となる。

3.2.3 条件実行とそのアーキテクチャサポート

排他的制御依存に対しては、条件実行により排除できる。すなわち、条件によってどの命令を実行するか分らない場合、図２のように、それらに実行条件を付加することにより同時投入することが可能になる。条件実行のためのアーキテクチャサポートは、前章で図１を用いて述べた通りである。

3.2.4 ＰＤＧの拡張と投機的移動

正依存性，制御依存性の２つを扱うのに都合のよいデータ構造としてＰＤＧ(Program Dependence Graph) [13]がある。我々はこのＰＤＧを拡張したものに対して投機的移動処理をほどこすことによって、先行実行をおこなう。

ＰＤＧは、プログラム中の各命令をノードとし、正依存性、制御依存性を有向辺とするグラフである。正依存と制御依存の両依存性が一つのデータ構造として表現されているため、両者とも考慮する必要のある場合に都合がよい。しかし、それぞれの依存性を表すエッジは、その性質が異なっているため同等には扱えない。そのためＰＤＧにより論理最大並列を得るのは困難である。そこで、我々はＰＤＧの各ノードに実行条件を付加し、その命令の実行条件を構成する各条件の生産者からその命令に対して制御依存エッジを張った形式にＰＤＧを拡張する。これにより、制御依存性を正依存性と同様に扱うことが可能となり、ＰＤＧはプログラムの本来持っている並列性を表す形になる。このＰＤＧの拡張によって、より高い並列性を得ることが可能となる。ＧＩＦＴはこの拡張を前述の条件実行機構によりサポートしている。また、我々は制御構造、特にループを基盤とした単位で処理をおこなうため、startノードとloopノードの２つの特別なノードを含める。拡張ＰＤＧの例を図１０に示す。

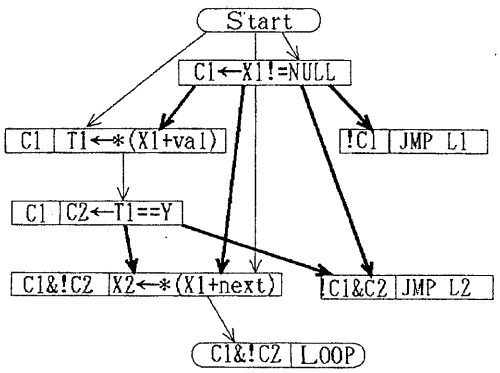


図１０　拡張ＰＤＧ

拡張ＰＤＧは、ＳＳＡ変換をおこなったコントロールフローグラフをもとに作成される。この拡張ＰＤＧはプログラムの持つ本質的な並列性を表しているため、ＧＩＦＴの条件実行機能により単純なレベリングアルゴリズ

ムによって、プログラムに対する最大並列を容易に得ることができる。ここで、さらにＧＩＦＴの先行実行機能により、図１０のように命令を条件文を越えて移動させる。これを投機的移動と呼ぶことにする。このとき、ストア命令のように副作用を起こす可能性のある命令や、ジャンプ命令のようにプログラムのコンテクストを変えてしまう命令は投機的に移動してはならない。また、ＳＳＡの形をしているため、命令が条件を越える際に生じる、反対側の流れに対する出力依存は考慮する必要がない。操作としては、ある命令に向かっている制御依存性エッジをその下の命令に差し替えるという簡単な処理により実現できる。これにより制御依存による制約がはずれ、並列性が飛躍的に増すものと考えられる。

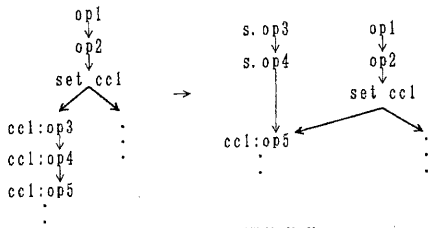


図１１　投機的移動

移動可能なもの全てを投機的に移動させれば論理最大並列を得ることが可能である。しかし、その移動がプログラムの最小実行時間を保証するクリティカルパスの短縮につながらないものであった場合、単にプロセッサ資源の浪費をするに過ぎない。そこで、クリティカルパス上の命令から順に投機的移動をおこない、クリティカルパスが縮小しなくなるまで処理を続ける。

ＰＤＧにはデータ依存と制御依存が合わせて表現されているため、プログラムのもつ依存構造、並列性が容易に把握できる。すなわち、ＰＤＧを用いると、バーコレーションスケジューリング法における各命令の移動処理や、その際の各ノードに上がれる命令の再計算等の複雑な処理が必要なくなる。

３．３ レジスタ割付けの最適化

スケジューリングをおこなう前にレジスタ割付けをおこなった場合、割付け結果により新たに生じる反依存性により、スケジューリングにおける並列性が大幅に減少する。逆に、スケジューリングの後にレジスタ割付けをおこなう場合においても、実レジスタ数の制限上反依存性が生じるが、スケジューリングがすでに決定しているため、レジスタの値をメモリに退避したり、再ロードする命令（スピルコード）が挿入される。このように、レジスタ割付けによって生じる反依存性により、プログラムの並列性が低下するのを抑える手法を提案する。

3.3.1 プログラムの性質を反映したレジスタ割付け手法

プログラムには、その実行時間の多くを局所的な部分で費やすという性質がある。この性質を考慮しないでレジスタ割付けをおこなった場合、プログラム中における実行回数の多い部分にスピルコードが挿入され、その部分の並列性を低下させてしまうことがある。よって、効果的にレジスタを割り付けるためには、実行回数の多い部分にレジスタを割り当てるようにするのが自然である。しかし、コンパイラはこの実行の局所性を静的に知ることができないため、通常、実行の要所をループの部分に想定する。

大域的にレジスタ割付けをおこなう手法においては、ループ、特に内側のループほど重要であるとして重み付けをし、その重み順にレジスタを優先して割り付けるというアプローチをとっている。しかし、実行の要所が同程度である部分が複数存在する場合、互いが干渉し合うため、そのどこかにスピルコードが生じてしまう可能性がある。よって、この干渉を断つために、プログラムの制御構造を考慮して局所的な部分にプログラムを分割し、各部分で個々にレジスタを割り付ける手法が提案され、成果をあげている[13][14]。この局所単位を我々はクラスタと呼ぶ。我々は、スケジューリングをクラスタ単位でおこなうため、クラスタの選択方法として、ストリップマイニング法等のループ最適化をおこなったものを選び、スケジューリングにおける高並列化につなげる。

3.3.2 クリティカルパスを重視した漸次的なレジスタ割付け手法

制御構造、すなわちループ構造を反映させて構成されたクラスタ単位でレジスタを割り付けることにより、局所的にスピルコードを極小化することができる。ここで、我々はさらにこの局所部分においてクリティカルパスにスピルコードが入ることによる並列性の低下を防ぐレジスタ割付け手法を考える。

スカラプロセッサでは、スピルコードにより実行効率が損なわれてしまうため、レジスタ割付けはスピルコードを極小化することを目標とする。それに対して、ＶＬＩＷでは命令の空きスロットを有効に利用することにより、スピルコードによる損失を隠すことが可能である。

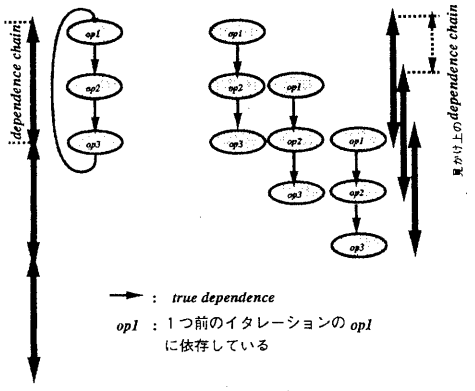
我々は、このことに注目し、レジスタ割付け時にスピルコードがクリティカルパスに挿入されるのを防ぐことを目標とする。そのために、レジスタの割付けはクリティカルパスから順に漸次的におこない、クリティカルパスにレジスタを優先的に与える。しかし、クリティカルでないパスに対してはスピルコードの挿入が必要となってくる。このスピルコードの挿入によるクリティカルパ

スの延長も考えられるため、各パスに対してレジスタを割り付ける度にそのパスに対するスケジューリングを考慮する必要がある。したがって、レジスタ割付けとスケジューリングを漸次的に交互におこなうことにより、スピルコードによるクリティカルパスの延長を防ぐ。

本手法をクラスタリングによる手法と組み合わせることにより、プログラムにおける重要な部分で局所的にスピルコードが極小化され、さらに、その各部分において、クリティカルパスが抑えられた形でのレジスタ割付けが可能である。

4．正依存性の縮小

また、正依存性を排除することはできないが、これによる依存関係鎖を見かけ上短縮する方法として、図12のようにループをオーバーラップさせる手法がある。



(a) ループをオーバーラップさせない場合　(b) ループをオーバーラップさせた場合

図12　ループのオーバーラップによる正依存性の見かけ上の短縮

この考えに基づく効果的な手法として、ソフトウェアパイプライン技法があり、パーフェクトパイプライン[16]等が知られている。我々は他に、アルゴリズムの停止性を保証するために、1サイクルずつ次のループイタレーションから取り込み徐々にパイプライン化していく方法を検討中である。このソフトウェアパイプライン技法はＶＬＩＷにとって強力なものであるが、非常に多くのレジスタを使用することになる。ＧＩＦＴではレジスタを６４個用意しており、ソフトウェアパイプラインの機会を与えている。

4．評価

ＧＩＦＴのアーキテクチャサポートがコンパイラに対して与える効果を評価するために、シミュレーションをおこなった結果を表１に示す。ＶＬＩＷが不得意とする一般的に並列性の少ないプログラムに対して、ＧＩＦＴの条件実行、先行実行、およびバンクの動的調停による効果をみるため、対象プログラムとして並列性の低いソートを選んだ。シミュレーションをおこなう際、ＶＬＩＷのユニット構成は、固定小数点命令、浮動小数点命令、ジャンプ命令、ロード／ストア命令ユニットをそれぞれ４個ずつ持つものとし、命令、データ両キャッシュにおけるミスヒットはなく、メモリアクセスには３サイクル、他の命令には１サイクルの実行時間を要すると仮定している。ＳＳＡ変換導入のもとでＶＬＩＷ、およびＧＩＦＴに対してコンパイルをおこない、スカラプロセッサに対する速度向上率を調べた。

プログラム	速度向上率	
	VLIW	GIFT
選択ソート	１．３８	３．３５
バブルソート	１．４５	４．４６
シェルソート	１．３７	３．２１

表１　ＧＩＦＴコンパイラによる速度向上率

ＧＩＦＴの方がＶＬＩＷに比べ２～３倍の速度向上が得られた。このように比較的よい結果が得られた要因としては、ソートのような、与えるデータにより制御依存性が動的に変ってしまうプログラムに対して、条件実行及び先行実行が効果的であったこと、及び、メモリ参照のチェックをおこなう必要のないことによる並列性の向上によるところが大きい。

5．おわりに

コンパイラがより高度な並列性を得るために必要不可欠なアーキテクチャサポートをＧＩＦＴが与えていること、及び、そのサポートをいかし効果的かつ効率的にスケジューリングを行う最適化コンパイラについて述べた。

現在、ハードウェアの方はＶＨＤＬ上で論理設計中である。同時にコンパイラシステムのインプリメントも進めている。

今後は、最適化技法に関してさらに考察を進め、本コンパイラによるコードをもとに、ＧＩＦＴアーキテクチャの評価をおこなう予定である。

6．参考文献

[1] J. A. Fisher,
"Very Long Instruction Word Architectures and the ELI-512", Proc. of 10th Int. Symp. on Computer Architecture, pp. 140-150 (1983)

[2] R. P. Colwell, et al.
"A VLIW Architecture for a Trace Scheduling Compiler", 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 180-192 (1987)
[3] K. Murakami et al.
"SIMP (Single Instruction stream / Multiple instruction Pipelining) : A Novel High-Speed Single-Processor Architecture", Proc. 16th ISCA, pp. 78-85 (1989)
[4] William, J. M.,
"Super-Scalar Processor Design", Technical Report No. CSL. TR. 89-383, Computer Systems Laboratory, Stanford University (1989)
[5] Sohi, G. S., et al.
"Instruction Issue Logic for High Performance Interruptable Pipelined Processors", The 14th Annual Symposium on Computer Architecture, IEEE Computer Society Press, pp. 27-34 (1987)
[6] John R. Ellis,
"Bulldog: A Compiler for VLIW Architectures", The MIT Press, ACM Doctoral Dissertation Award (1985)
[7] Kemal Ebcioglu and Alexandru Nicolau,
"A Global Resource-constrained Parallelization Tequnique", ACM SigARC International Conference on Supercomputing (1989)
[8] 鈴木, 小松, 深澤, 門倉
"条件実行アーキテクチャＧＩＦＴのメモリインタフェース", SWoPP'91, 91-ARC-89, 89-14, pp. 95-102
[9] 鈴木, 小松, 深澤, 門倉
"条件分岐の効率的実行を可能とする細粒度アーキテクチャ", SWoPP'90, CPSY90-53, pp. 91-96
[10] 小松, 鈴木, 深澤, 門倉
"条件分岐の効率的実行を可能とする細粒度並列アーキテクチャ", 情報処理学会第４１回全国大会, 3P-8 (1990)
[11] R. Cytron, J. Ferrante, et al,
"An Efficient Method of Computing Static Single Assignment Form", ACM POPL16 (1989)
[12] J. Ferrante, K. J. Ottenstein and J. D. Warren,
"The Program Dependence Graph and Its Use in Optimization", ACM Transactions on Programing Languages and Systems, Vol. 9, No. 3 (1987)
[13] 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, D-80, p. 6-80, (1991)
[14] David Callahan and Brian Koblenz,
"Register Allocation via Hierarchical Graph Coloring", Proc. of the ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation (1991)
[15] Alexander Aiken and Alexandru Nicolau,
"Perfect Pipelining: A New Loop Parallelization Technique", Research Report, 87_873, Dept. of Computer Science, Cornell Univ. (1987)