計算機アーキテクチャ　106－1
（１９９４．　６．　１３）

スーパースカラパイプラインによるブロック並列実行方式

朝生良教　柳瀬正俊　桐山佳隆　林達也

名古屋工業大学　電気情報工学科

現在シングルプロセッサにおいて、パイプラインによる命令内の基本処理の並列実行、スーパースカラパイプラインによる複数命令の並列実行が実現されている。我々はより高い並列性の追求のためシングルプロセッサにおいて、複数スーパースカラパイプラインによる基本ブロックの並列実行の研究に取り組んでいる。本報告ではこのブロック並列実行方式を提案する。この方式はアーキテクチャ上の支援によって当該実行ブロックとその分岐し得るブロックを並列に実行するもので、我々は既存のアーキテクチャにおける投機的実行を上回る性能を得ることを目標としている。

A Method of Block Parallel Execution with Superscalar Pipelines

Yoshinori Asoh　Masatoshi Yanase　Yoshitaka Kiriyama　Tatsuya Hayashi

Department of Electrical and Computer Engineering
Nagoya Institute of Technology

This paper describes a parallel execution method of basic blocks of instructions.Instruction code sequence is not necessarily scheduled statically prior to execution by the optimizing phase of compilers. Our processor has more than one superscalar pipelines.Using them, some instruction streams either in a basic block or its subsequent blocks are executed in parallel in speculative manner.Shadow buffers are provided to avoid undesirable side effcts.Improved performance of parallel execution is expected using the method described here.

1 はじめに

分岐命令に先立つ、制御フローの変化しない命令流を基本ブロックと呼び、以後はブロックと略す。ブロックの直後にある分岐命令の分岐先は成立と不成立の2つのブロックがあり得る。具体的にブロックがどのブロックへ分岐し得るかという関係は多くがコンパイラにより静的に把握可能であると考える。しかしブロックがどのブロックへ分岐するかの動的な判断には場合制御ハザードが不可避である。この制御ハザードに対してパイプラインにおいては遅延分岐が有効であったが、スーパースカラパイプラインにおいてはより有効な方法として分岐予測と投機的実行が注目され、実現されている。この投機的実行は分岐方向が未確定の、あるいは実行の必要性が未確定の時期に2つの分岐先ブロックの一方の実行を予測し開始するもので、予測が外れた場合には、実行したブロックの命令を無効化してもう一方をやり直す。予測が外れた場合のペナルティが大きいためその確度が重要となる。このペナルティを軽減する手法として、予測をせず、分岐し得る2つのブロックを2つのスーパースカラパイプラインで並列に実行し、分岐した方のみを有効とする方法が考えられる。更に当該実行ブロックと並列に実行することで分岐ペナルティの軽減と並列度の向上が期待できる。以上がブロック並列実行方式の着想の経緯であり、並列実行するブロックの組合せは当該実行ブロックとその分岐先ブロックに限られる。そして3つのスーパースカラパイプラインに各ブロックを実行させることで並列実行を行う。

スーパースカラパイプラインにおいては命令レベル並列度の向上を妨げる要因に占める制御ハザードの比率が非常に大きなものとなるため、投機的実行が注目されている。より高い命令レベル並列度を得るため、アーキテクチャ上の支援と静的スケジューリングを主体とした手法が報告されている [1, 2]。我々の目標はアーキテクチャ上の支援と動的スケジューリングを主体とした手法により単一スーパースカラパイプラインにおける投機的実行を上回る性能を得ることである。

以後は第2節でブロック並列実行の概念を説明し、第3節で必要となるアーキテクチャ上の支援を提示し、第4節で我々が検討中の実現方式について説明する。

2 ブロック並列実行

2.1 効果

図1(a)に静的な、コンパイル時に把握されるブロック関係の例を示す。図において丸はブロックを表し、破線で囲まれた3つのブロックが並列実行され得る組合せである。ここでBCDの組合せに注目すると、Bは CDの親ブロック、Cは不成立、Dは成立のそれぞれBの子ブロックと呼ぶことにする。BCDの組合せはAの不成立の組合せと呼び、Aの分岐が不成立の時に並列実行される。次に図1(b)に動的な、実行時に定まるブロック関係の例を示す。なお、これは図1 (a)の例を並列実行したものとする。ここでBCDの組合せに注目すると、Bを当該実行ブロック、Cを不成立、Dを成立の投機的実行ブロックと呼ぶことにする。また図ではDが黒いが、これはBがCに分岐したためDの実行は意味がないことを表し、これを無効ブロック、そうでないものは有効ブロックと呼ぶことにする。

ブロック並列実行の効果は、当該実行ブロックと投機的実行ブロックの大きい方の実行時間にもう一方の実行時間が吸収されることと、2回の分岐ペナルティが1回分の分岐ペナルティとなり得ることである。よって高い効果を望むには、ブロックの大きさが等しく、絶え間なく並列実行を行う必要がある。

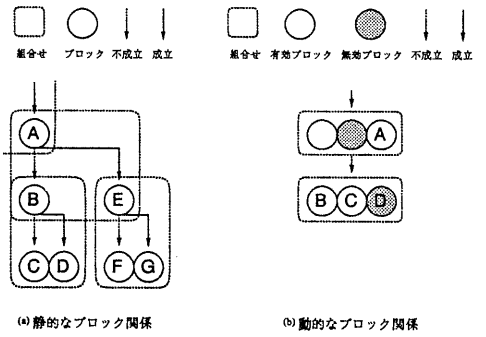

図 1: ブロック関係の例

2.2 留意点

2.2.1 構造

実装に関しては、複数のスーパースカラパイプライン、充分な機能ユニット、多重化されたバス、等のように膨大な資源を必要とすることは明らかである。また複雑な制御機構を必要とし、命令発行時のスケジューリングの処理が大きいと予想されるためスーパースカラパイプラインのサイクルタイムの増大も考えられる。また正確な割り込みの保証も複雑になると考えられる。

特殊な機構としては、投機的実行ブロックの実行結果が、その有効の確定以前に状態変更をしてしまわないように何らかのバッファリング機構を必要とする。そしてそれは有効の確定後は速やかに書き戻しが行われるものでなければならない。

2.2.2 データハザード

本来逐次実行されるブロックを並列に実行する場合、投機的実行ブロックで必要とされるソースオペランドが当該実行ブロックでまだ生成されていないというブロック間データハザードが頻発すると考えられる。また投機的実行ブロックの命令の多くはデータハザードを起こす命令の生成するデータに依存すると考えられるので、その都度インターロックしていてはブロック並列実行の効果は無きに等しい。よってスーパースカラパイプラインが極力ストールすることなく、データハザードを解消することが望ましい。つまり命令は滞ることなく発行され、それらは待機してデータハザードが解消され次第、実行を開始されるべきである。

2.2.3 情報ハザード

投機的実行ブロックは本来、当該実行ブロックの直後に実行されるべきものなので、並列に実行される場合には当該実行ブロックとの依存関係を全く知りようがないという問題が考えられる。逐次実行においてデータハザードの解消機構を持つプロセッサでは、解消のために必ずフラグなど明示的な情報を持ち、命令の発行時に利用する。しかし明示的な情報を利用できるのは、それが必ず以前に発行された命令により変更されるという暗黙的な情報が存在するからである。以上の2つの情報は分岐の前後においても保持される。しかしブロック並列実行では、投機的実行ブロックの命令の発行に当りこれらの情報が欠落しまた不十分である。これは命令の発行に関するハザードであり、情報ハザードと呼ぶことにする。具体的には投機的実行ブロックにおいて、ソースオペランドが当該実行ブロックに依存しているかということ (データ依存の有無) と、それがいつ生成されるかということ (データハザードの有無) が分からないという状態が起こる。情報ハザードの発生はレジスタ演算命令のみならずロード／ストア命令や分岐命令も例外ではなく、投機的実行ブロックの全ての命令の発行を困難にする。よってハードウェアのみでは容易に解消できない問題と考える。なお情報ハザードが完全に解消されるのは、当該実行ブロックの全命令の発行が完了する時である。

2.2.4 制御ハザード

パイプラインやスーパースカラパイプラインにおいて不可避の制御ハザードは、基本的には分岐条件の判断と分岐アドレス生成の遅延により分岐先ブロックの命令のフェッチができないことに起因する。ブロック並列実行では、その分岐命令の実行以前にその分岐先ブロックの命令のフェッチは行われているので、一見制御ハザードとは無縁に思われるが、次の2点において問題がある。第1に当該実行ブロックはその子ブロックである投機的実行ブロックの有効／無効を決定するため、可及的早期に分岐方向を判断する必要があること。第2に投機的実行ブロックは新たな並列実行をさせるため、可及的早期に分岐方向を決定しそのブロックの組合せを指定する必要があること。以上より制御ハザードは厳然と存在し、いずれも解消されなければ性能低下は免れ得ない。

3 アーキテクチャ上の支援

ブロック並列実行を行うため、あるいは性能を向上させるために必要なアーキテクチャ上の支援について説明する。

3.1 並列実行開始方法

投機的実行ブロックは2つであり、開始の指定に際しては2つのアドレスを必要とするため措置が必要である。開始指定方法は2つ考えられる。1つは分岐命令で指定する方法である。これは当該実行ブロックを含めた3つのブロックを完全に同時に開始できる

反面、分岐命令の命令語に求められるオペランド数とビットスペースに問題がある。例えば通常オペランド数は、分岐条件レジスタ番号とベースレジスタ番号及びそのオフセットの3つであるが、この方法では分岐条件レジスタ番号とベースレジスタ番号3つ及びそのオフセット3つで計7つである。オペランド数の増加は命令フォーマットを複雑にする。また必然的にビットスペースも大きなものとなる。このため命令長が大きくなり不統一となる。もう1つは当該実行ブロックの先頭で指定する方法である。これはブロックに2命令増えるだけだが、制御ハザードと同様の理由により投機的実行ブロックの開始が遅れる。スーパースカラにおいてはこの遅延が並列度低下の非常に大きな要因になると考える。よって以後は前者を前提とする。

並列実行の組合せは成立と不成立の2つであり、2つの分岐命令を必要とする。不成立の組合せを実行する分岐命令は無条件分岐命令で、成立の分岐命令の直後に位置する必要がある。また分岐命令を並列に実行可能でなければならない。

3.2 情報ハザードへの対処

投機的実行ブロックの命令の発行に際し、当該実行ブロックの全命令の発行終了時の情報が得られないために発生する情報ハザードは、動的なブロック間の命令のデータ依存の有無とデータハザードの有無が未知であるということであり、ハードウェアのみでは解消が困難である。よってソフトウェアによる支援を行い、不十分な情報を補う。

まず動的なブロック間の命令のデータ依存の有無の未知を補うための情報を考える。静的なブロック間の命令のデータ依存関係はコンパイル時に大部分が分かると考えられるが問題がある。子ブロックに対して親ブロックは複数存在し得るため、親ブロックの命令に対するデータ依存関係は分岐する親ブロックによって変化する。つまり親ブロックの命令に依存する可能性のある命令が実際には依存しないという場合が起こり得る。このため子ブロックで全ての親ブロックの場合に対処しようとするのは難しい。よって親ブロックが子ブロックに対して情報を提供するようにする。具体的にどのような情報を提供すべきかを考えると、親ブロックが子ブロックに依存され得る全てのデスティネーションオペランドを特定できれば良いことが分かる。よってこの特定にはレジスタ演算についてはレジスタ数分のビット列を各レジスタに対するフラグとして用意すれば良い。しかしメモリ操作については特定は不可能である。なぜなら変更するメモリをビット列で示すことはできず、また変更する命令の分だけアドレスを用意することも現実的ではないからである。よってレジスタ演算の場合にのみ、情報の提供が可能で、以後はこれを変更レジスタビット列と呼び、分岐命令の命令語に付加する。ただし情報のサイズが非常に大きいことに注意を必要とする。親ブロックの子ブロックへの対処は完全ではないので子ブロックの親ブロックへの対処も行なう。子ブロックが親ブロックに依存し得る全てのソースオペランドを特定できるようにする必要がある。よってこの特定には、ソースオペランド毎に1ビットのフラグを設定すれば良い。ソースオペランドは2つ以下であるから2ビットを必要とし、これを全命令の命令語に埋め込む。これを依存ビットと呼ぶことにする。

次に動的なブロック間の命令のデータハザードの有無の未知を補うための情報を考える。子ブロックの命令のソースオペランドが親ブロックの命令に依存することを判断できたとしても問題がある。親ブロックにおいては同じデスティネーションオペランドを変更する命令が複数存在し得るため、子ブロックの命令の発行時に、親ブロックではどの命令までが発行されたかを判断できない。子ブロックの命令が依存するソースオペランドは必ず親ブロックの命令で最後に変更するデスティネーションオペランドである。よって提供すべき情報は、親ブロックにおいて変更する、全ての命令に対し、最後に変更するものを特定する必要があり、デスティネーションオペランドに対し1ビットのフラグを設定し命令語に埋め込む。これを提供ビットと呼ぶことにする。

以上よりソフトウェアとしては全ての命令について命令語の中から計3ビットを割いて埋め込み、また分岐命令にはレジスタ数分のビット列を加えることが情報ハザード解消のために必要である。

3.3 制御ハザードへの対処

当該実行ブロックの分岐命令の分岐方向によって投機的実行ブロックの有効／無効が判明した後、投機的実行ブロックの分岐命令により、新たな並列実行の組合せを開始する。つまりどちらの分岐命令も早期に実行可能とならなければスーパースカラパイプライン

のストールは増大する。よって我々は可能な限り早期に分岐を実行可能とする方法としてカウント分岐を考えた。

カウント分岐はいわば遅延命令数が可変の遅延分岐である。本来ブロックの最後に位置する分岐命令とその分岐条件生成命令をコンパイラが前に移動させ、かつその移動数をカウントする。それをハードウェアが利用して分岐を実行することにより、分岐ペナルティを軽減し、また分岐結果を早く知るというものである。このためにはコンパイラが計算したカウントを分岐命令に埋め込む必要がある。よって理由は後述するが2ビットを分岐命令の命令語に埋め込む。

4 実現方式

前説までは具体的な実現方式については触れなかった。本節では現在我々が検討している、アーキテクチャ上の支援に基づいた実現方式について基本部分に関して説明する。それはデータハザードの解消機構である拡張Tomasuloアルゴリズムと、制御ハザードによる分岐ペナルティの軽減手法であるカウント分岐の2つである。なお以後は全てレジスタ演算を前提としている。

4.1 拡張 Tomasulo アルゴリズム

まず各ブロックを実行するスーパースカラパイプラインは4命令をin-orderに発行するものとし、これにはデータハザードを解消しout-of-order実行を許す機構であるTomasuloアルゴリズムを適用する。なぜなら前述のようにブロック並列実行ではブロック間のデータハザードが頻発することが予想され、あるデータを当該実行ブロックの命令と投機的実行ブロックの命令が同時に必要としているという状況は多分に起こり得るからである。このためTomasuloアルゴリズムの、命令発行を滞らせずに待機させ、ブロードキャストにより同一データを待つ命令を一斉に実行開始させられるという特徴は非常に有用である。よって充分な資源が与えられれば純粋にデータハザードで実行不可能な命令のみが待機させられることになる。Tomasuloアルゴリズムの基本構成は、レジスタファイル(RF:resister file)に対応する管理情報を記憶するレジスタ状態(RC:resiter condition)、発行された命令を待機させるリザベーションステーション(RS:reservation station)とロードバッファ(LB:load buffer)、ストアバッファ(SB:store buffer)、およびブロードキャストのための共通データバス(CDB:common data bus)から成る。表1にそれぞれの1エントリを示す[5]。ブロック並列実行のために3つのスーパースカラパイプラインを使用するが、その構成はこれらを3つ用意し共通データバスを結合したものとなる。図2に構成を示す。なお制御機構と局所バスは省略してある。各スーパースカラパイプラインは並列実行の組合せに固定的に対応し、当該実行ブロックを実行するものを当該実行スーパースカラパイプライン(図2中CON)、投機的実行ブロックを実行するものを投機的実行スーパースカラパイプライン(図2中SP1、SP2)と呼ぶことにする。また投機的実行スーパースカラパイプラインの持つレジスタファイル(以後は便宜的に裏を付けて区別する)が投機的実行結果のバッファリングを行い、並列実行の完了後に裏レジスタ状態とともに速やかにレジスタファイルとレジスタ状態に書き戻されるものとする。よって投機的実行ブロックの命令は必要なソースオペランドをレジスタからフェッチした後は、裏レジスタに格納し参照する。バッファリングによってブロック間の命令のWAW(write after write)、WAR(write after read)データハザードが解消される。

レジスタ状態	
名称	内容
Busy	データ待ちフラグ
Q_i	機能ユニット番号

リザベーションステーション	
名称	内容
Busy	使用中フラグ
Op	演算の種類
Q_j	機能ユニット番号
Q_k	機能ユニット番号
V_j	データ
V_k	データ

ロードバッファ	
名称	内容
Busy	待機中フラグ
Address	アドレス

ストアバッファ	
名称	内容
Busy	待機中フラグ
Q_i	機能ユニット番号
V_i	データ
Address	アドレス

表1: Tomasuloアルゴリズムの各管理テーブルの1エントリ

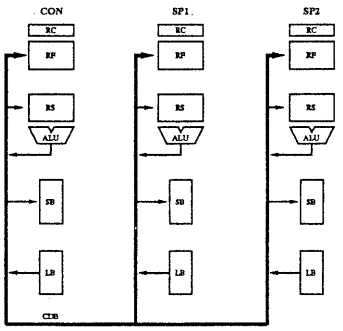

図 2: 基本構成

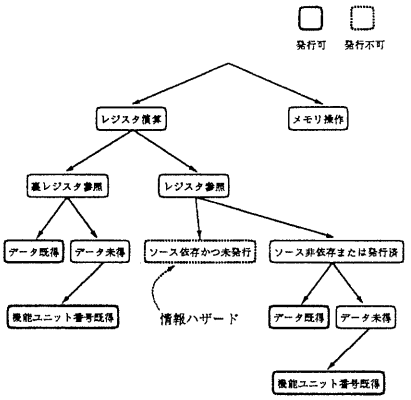

図 3: 投機的実行ブロックの命令の発行手順

書き戻しが行われていれば、当該実行ブロックの命令の発行には何ら問題はない。よって投機的実行ブロックの命令の発行手順について考えるが、その様子を図3に示す。命令がレジスタ演算であるとすると、ソースオペランドをフェッチするに当り、まず裏レジスタを参照するかどうかを判断する必要がある。そのため各裏レジスタにフラグを必要とし、それは各裏レジスタに最初に格納する命令の発行時にそのデスティネーションレジスタに対応するものがセットされる。以後これを Ready フラグと呼び、裏レジスタ状態に持たせる。これがセットの時、ソースオペランドのフェッチは裏レジスタを参照すれば良い。クリアの時はレジスタを参照する必要があるが、この場合ソースオペランドが当該実行ブロックの命令に依存するかどうかを判断する必要がある。この判断は依存ビットを検査すれば可能であるが誤る可能性があるので、当該実行ブロックで変更されるレジスタをセットとするレジスタ変更ビット列に委ねる。レジスタ変更ビット列は分岐命令に含まれ、分岐の実行時にセットする領域を必要とする。よってレジスタファイルの各レジスタに Ready フラグを用意し、対応するビットが反転してセットされるものとする。その後は、当該実行ブロックの命令で提供ビットがセットであるものの発行時にセットされる。これがセットの時投機的実行ブロックの命令のソースオペランドは非依存か、あるいはそれを変更する当該実行ブロックの命令が発行済なので問題はない。クリアの時は依存があり、かつ未発行である。命令語に情報を埋め込むというアーキテクチャ上の支援によりこの状態は特定できるが問題がある。なぜなら Tomasulo アルゴリズムではデータの生成の遅延というデータハザードを、データのタグとして機能ユニット番号 (FUN:functional unit number) を利用して発行することで回避するが、この場合は依存するソースオペランドを変更する命令が未発行なため、機能ユニット番号を利用できない。つまり機能ユニット番号についての情報ハザードである。

レジスタ状態 (CON,SP1,SP2)	
名称	内容
Ready	参照可能フラグ

リザベーションステーション (SP1,SP2)	
名称	内容
$Temp_j$	タグ記憶フラグ
$Temp_k$	タグ記憶フラグ
T_j	タグ (レジスタ番号)
T_k	タグ (レジスタ番号)

ストアバッファ (SP1,SP2)	
$Temp_i$	タグ記憶フラグ
T_i	タグ (レジスタ番号)

表 2: 各管理テーブルの 1 エントリ (追加分)

この情報ハザードの要因は機能ユニット番号の決定遅延であるので、Tomasulo アルゴリズムと同様の解消法が適用できる。つまり機能ユニット番号のタグを利用して発行し、機能ユニット番号が決定され次第、専用のバスによるブロードキャストを行い機能ユニット番号を得る。なお機能ユニット番号の決定は必ずデータの生成よりも早いため、ブロードキャストが遅らされなければ、機能ユニット番号の獲得が遅れてデータのブロードキャストを取りこぼすことはない。なおタ

グは依存するソースオペランドのレジスタ番号である。またバスは共通機能ユニット番号バス (CFB:common FUN bus) と呼び、レジスタ番号と機能ユニット番号を同時にブロードキャストできるだけのバス幅を持つ。またこの方法の実現にともない、投機的実行スーパースカラパイプラインのリザベーションステーションとストアバッファにはタグの記憶を表す Temp フラグとタグの記憶領域を必要とする。以上によりブロック間の RAW(read after write) データハザードが解消される。図 4に共通機能ユニット番号バスを加えた構成を示す。また表 2にレジスタ状態、リザベーションステーション、ストアバッファの 1 エントリについて追加されたものを示す。

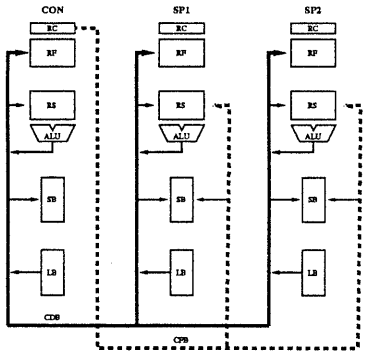

図 4: 共通機能ユニット番号バスを加えた基本構成

4.2 カウント分岐

この節では便宜上分岐命令はブロックに 1 命令として考える。スーパースカラパイプラインにおいては分岐命令のスロットにおける位置によって分岐ペナルティが異なる。最悪の場合は図 5 (a) 、最良の場合は図 5(b) となる。図 5(a) に注目すると、無効命令は分岐命令の存在するスロットと次のスロットである無効スロットからなる。ここで分岐命令がもう 1 命令前にあれば、無効スロットはなくなる。この、分岐命令を存在するスロットより 1 スロット前に移動させ、早く分岐方向を判断し、無効スロットをなくすことがカウント分岐の目的であり、そのためにコンパイラにより分岐命令のスケジューリングを行なう。つまり分岐命令を可能であれば前に移動させ、その移動数をカウントとして数え、分岐命令の命令語に埋め込む。分岐命令の実行時にはそのスロット内の位置とカウントに対するルールに従い後続の命令を適宜無効化する。このカウントの情報量は前述のように 2 ビットで良い。なぜなら仮に図 5(a) において 5 命令前に移動させるとすると、分岐命令は本来の位置より 2 スロット前に移動することになる。この場合、本来存在したスロットから次のブロックの命令をフェッチできるが、ブロック並列実行においては問題がある。それは書き戻しのタイミングに関連する。当該実行スーパースカラパイプラインの同スロット内に、前の当該実行ブロックと次の当該実行ブロックの命令が存在する場合、それらは同時に発行されるが次の当該実行ブロックの命令は書き戻し後に発行されなければならない。この実装は難しいと思われるので、分岐命令は最大 4 命令、つまり 1 スロットだけ前に移動させれば良く、そのカウントの表現には 2 ビットあれば十分である。

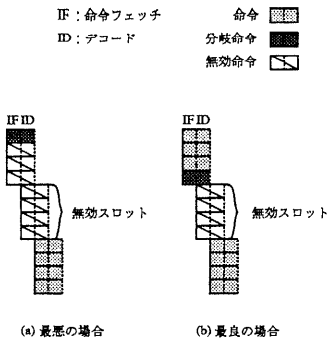

図 5: スーパースカラパイプラインにおける分岐ペナルティの例

カウント分岐の実現に必要な資源は、正確な割り込みの保証のためにカウントを記憶するレジスタである。分岐命令が 1 スロット前に移動されているとして、何らかの割り込みによってその直後のスロットから再開する必要があり、そのスロットには無効命令があるとする。この場合スロットの無効命令を決定するカウントがないという問題が発生する。よって分岐命令の実行に伴いカウントを記憶する必要がある。また遅延分岐の問題と同じく 2 つのアドレスを退避する必要もあるが、遅延分岐とは以下の点が異なる。

- 遅延スロットを持たずカウントによって命令の

無効化を行なうため、NOP(何もしない命令)で埋める必要がない。

- カウントは普遍的な情報であるため、スーパースカラパイプラインの発行数が増えても再コンパイルしなくても良い。
- 制御機構が複雑となる。

分岐命令は最大4命令を前に移動させれば良いが、分岐条件生成命令はより移動させなければカウント分岐の効果は望めない。なぜなら分岐の実行時には分岐条件が生成済でなければならないからである。またブロック並列実行では投機的実行ブロックの命令は多くが待機させられるが、分岐条件生成命令が待機させられることは極力避けなければならない。よってこのためにはコンパイラによるコードスケジューリングが必要である。

5 おわりに

本論文においてブロック並列実行方式の提案を、アーキテクチャ上の支援と実現方式について行なった。アーキテクチャ上の支援はコンパイラによって情報を与えることであり、その情報は普遍的なものである。実現方式は可能となり次第実行するという点に重点をおいたものである。

実現方式において、メモリ操作に関しては触れなかった。メモリについては管理情報がレジスタのように固定的かつ恒常的には用意されないため寿命があること、変更レジスタビット列と同様に情報を与えることが不可能なため依存ビットを利用するが誤りがあり得ること、裏レジスタと同様に裏キャッシュを用意するのは困難なことなど、色々と問題が生じるため現在検討中である。

今後はメモリ操作も含めた実現方式を定めた後、シミュレーションによる性能評価を行いたいと考えている。その結果によってはより現実的な実現方式として、並列実行するブロックの組合せを2つに減らして不成立の投機的実行ブロックを実行しないことも考えられる。なぜなら本来不成立の分岐先ブロックの命令は分岐ペナルティーなしにフェッチ可能であり、ペナルティーの大きい成立の分岐先ブロックのみ並列実行しておけば、分岐ペナルティーの軽減という一点においては効果は下がらない。また分岐命令の命令語が小さくなること、分岐命令が1つで済むことにより、制御機構が単純となることや、2つのスーパースカラパイプラインで理論的には2倍の性能が得られるため効率が良いことなどが考えられるためである。

謝辞

本研究を進めるに当り富士通研究所の安里 彰氏には、数度に渡る質問に答えていただき、この場を借りて御礼申し上げます。

参考文献

[1] 原 哲也, 安藤 秀樹, 中西 知嘉子, 町田 浩久, 中屋 雅夫, “スーパースカラ・プロセッサにおける分岐命令の並列実行”, 情報処理学会研究報告 93-ARC-101-9,Oct 1993

[2] 安藤 秀樹, 町田 浩久, 中西 知嘉子, 原 哲也, 中屋 雅夫, “投機的実行のためのアーキテクチャ上の支援”, 情報処理学会研究報告 94-ARC-105-5,Mar 1994.

[3] 安里 彰, 志村 浩也, “スーパスカラプロセッサにおけるリカバリー方式”, 情報処理学会研究報告 93-ARC-101-10,Oct 1993

[4] 西本 晴子, 志村 浩也, 木村 康則, “paratool によるスーパスカラプロセッサの評価”, 情報処理学会研究報告 94-ARC-105-6,Mar 1994.

[5] John L.Hennessy,David A.Patterson, (富田眞治, 村上和彰, 新實治男 訳), “Computer Architecture:A Quantitative Approach”, 1990