計算機アーキテクチャ 105－5
ハイパフォーマンス
コンピューティング 50－5
（1994. 3. 10）

投機的実行のためのアーキテクチャ上の支援

安藤秀樹、町田浩久、中西知嘉子、原哲也、中屋雅夫

三菱電機（株）
システムＬＳＩ開発研究所

命令レベル並列性を利用してマイクロプロセッサの性能向上を図るためには、命令の投機的実行と最適なスケジューリングを、同時に実現する必要がある。本報告では、複数の制御フローにおける投機的な命令移動を可能にするアーキテクチャ上の支援を提案する。このアーキテクチャ上の支援は、コンパイラに広い命令移動のウィンドウと制限のない投機的命令移動の自由を与え、最適なスケジューリングを可能にする。本方式では、命令に制御依存のある分岐条件をプレディケートとして持たせ、投機的実行を制御する。投機的実行は、状態の書き換えと例外発生と言う２つの副作用を引き起こすが、本方式では、副作用をプレディケートと共にバッファすることにより、複数の制御フローにおける投機的状態の正しいコミットと投機的例外の適切な延期を実現する。

An Architectural Support for Speculative Execution

Hideki Ando, Hirohisa Machida, Chikako Nakanishi, Tetsuya Hara, Masao Nakaya

System LSI Laboratory
Mitsubishi Electric Corporation

Both speculative execution and a good instruction schedule are required for microprocessor performance improvement through the effective exploitation of instruction-level parallelism. This report proposes an architectural support for speculative code motion in multiple control flows. Our architectural support allows the compiler to best schedule through a big scheduling window and unconstrained speculative code motion. Our architecture predicates an instruction with control-dependent branch conditions, and the predicate controls speculative execution. Speculative execution may cause two side effects: a machine state write and an exception. Our mechanism buffers the side effects along with their predicates. The bufferred predicate correctly commits the speculative state and appropriately postpones the speculative exception.

1 はじめに

マイクロプロセッサの性能を向上させるためには、命令の並列実行は必須である。しかしながら、非数値計算プログラムは、データ間の依存関係が多く存在し、しかも、分岐の頻度が高いため、命令の並列度は低い。そのため、単純に機能ユニットを増加させる等の既存の技術だけでは、性能を大幅に向上させることは困難である。アーキテクチャとコンパイラに種々の仮定を行ない、利用可能な命令レベルの並列度の限界を調べた研究[1][2]によれば、限界を上げるためには、投機的実行と最適な命令のスケジューリングが必要であると報告している。ここで、投機的実行とは、実行が必要であるかどうか明らかになる以前に命令の実行を行なうことである。

これまで、投機的実行に関して多くの研究がなされてきた。ハードウェア主体で実現する方法[3][4]では、分岐予測に従って命令をフェッチし、ハードウェアが発行可能な命令を発見しスケジュールする。この方法は、ハードウェアがマシン状態の適切な更新を完全に行なうため、命令のスケジューリングに制限がないという点で優れている。しかし、発行可能な命令発見のウィンドウがマシンによってフェッチされた命令の範囲に限定されるために、最適な命令のスケジューリングを行なうことが難しい。さらに、複雑なハードウェアを必要とするために、マシンを高速動作させることが難しいという欠点がある。

ソフトウェア主体で実現する方法[5]では、命令のスケジューリングはコンパイラが行なう。この方法は、命令発見のウィンドウが大きいため、最適な命令スケジューリングを行なうことが可能である上、マシンを単純にできるため高速動作を実現することが容易である。しかし、コンパイラには、投機的実行によって生じる問題に対する解決能力に限界があるため、スケジューリングに大きな制限を与え、コンパイラの能力を最大限に生かすことが困難である。この欠点を補うためにアーキテクチャ上の支援を加えた研究報告[6][7][8]もあるが、依然としてスケジューリングに対する制限が大きい。

これらの方式に対して、両方の長所を取り入れた方式がある[9]。この方式では、コンパイラが命令のスケジューリングを最適化する。この時、ソフトウェア主体の方法でスケジューリングに課せられていた制限を取り除くために、投機的実行により生じる問題はハードウェアが解決する。この方式では、コンパイラは分岐予測に従ってスケジューリングのウィンドウを形成し、その中で最適のスケジューリングを行なう。一方、ハードウェアはスケジューリングの結果に従って、命令を順に発行するが、投機的実行結果はバッファされ、分岐予測の正誤に従って副作用を解決する。この方式は、スケジューリングへの制限を大幅に解消できる点で非常に優れた方式である。しかしながら、これまで報告されている方式では、バッファのコミット制御は、分岐予測の正誤によって行なわれる。このため、コンパイラによるスケジューリングも分岐予測によって制限されており、依然として、分岐方向の予測が困難な非数値計算応用に対しては制限が大きい。

本報告では、命令のスケジューリングに与えられる制限を極めて少なくした投機的実行のアーキテクチャを提案する。我々の提案するアーキテクチャでは、プレディケート実行を用いて、複数の制御フロー上の命令を同時に実行できる。投機的実行により生じる問題を解決するために、投機的実行の結果はプレディケートとともにバッファされる。バッファ内のプレディケートは、実際にとられた制御フローを表す分岐条件を参照し、真であれば対応するデータはコミットされ、偽であれば無効化される。これによって、プログラムの意味上の正当性を保証する。バッファ内のプレディケートは、さらに、投機的な実行の最中に起きた例外の副作用の解決に対する支援も行なう。これにより、適切で効率的な投機的例外処理を実現することができる。

本報告では、まず、2章と3章で、投機的実行とプレディケート実行に関して、これまで行なわれた研究について説明する。4章では、我々の提案する投機的実行モデルと、その実現方法について議論し、最後に、本報告をまとめる。

2 投機的実行

通常の実行では、分岐に依存した命令は分岐方向が決定する前に実行することはできない。この依存のことを制御依存と呼ぶ。これに対して、投機

的実行とは、分岐方向が判明する前に分岐に依存した命令の実行を行なう。投機的実行を可能にすれば、制御依存が解消でき、スケジューリングにより多くの自由を与えることができる。

しかしながら、投機的実行には2つの問題がある。1つはプログラムの正当性維持の問題である。分岐を越えて命令移動を行なう場合、その命令の実行結果が他の分岐方向で使用されるデータを破壊しないようにしなければならない。次のような例を考える。

```
i1:   if (r1)
i2:     r2 = load r3;
      else
i3:     r4 = r2 + r3;
```

命令 i2 を分岐命令 i1 より先に投機的に実行するとする。この実行では r2 の値を破壊してしまうために、else 部に制御が移行した際に、命令 i3 は r2 にあったデータを使用することができない。この場合、プログラムの正当性が失われる。

投機的実行のもう1つの問題は、例外処理である。投機的に実行された命令が例外を引き起こした場合、通常のプロセッサで行なわれている場合と同様に即座に例外処理を行なうことは、性能を大幅に低下させる。なぜならば、例外処理を行なう時点では、投機的命令の結果が必要かどうか不明であるからである。従って、もしも、投機的例外を起こした命令の実行が無効化される場合、単に、この例外を無視し、コミットされることが判明した時点で例外処理が起動されるように延期される必要がある。上の例では、命令 i2 はロード命令なので、例えば、ページ・フォールトを起こす可能性がある。もしも、投機的実行時に例外を起こした場合、分岐命令 i1 の実行によって、命令 i2 の結果がコミットされるかどうか判明するまで例外処理を延期する必要がある。

ハードウェアでこれらの問題を解決する方法では[3]、分岐予測に従って投機的実行を行ない、実行結果をバッファに一旦蓄える。バッファに蓄えられたデータは、分岐予測が正しければマシン状態を更新し、正しくなければ無効化する。また、もしも、分岐が正しく予測され、かつ、投機的実行が例外を起こしていれば、例外を処理する。このように、ハードウェアは投機的実行の2つの問題を解決することができる。

一方、ソフトウェアで解決する方法では、正当性維持の問題に対してはレジスタ・リネーミングによって対応する。しかし、この方法はリネームされたレジスタを元のレジスタに戻す操作が必要であり、性能を低下させる。さらに、メモリ操作に関してはリネームすることはできない。例外に対しては、ソフトウェアだけでは解決は極めて難しい。そのため、次のようなハードウェア支援を加えることによって対処する方式が提案されている[6]。通常の命令と同一の操作を行なうが、即座には例外処理を行なわない命令（*non-excepting* 命令）を用意する。例外を起こす可能性のある命令を投機的に実行する場合は、代わりに non-excepting 命令をスケジュールする。もしも、non-excepting 命令が例外を起こした場合は、例外処理を行なわず、単に、結果を書き込む。その際、例外を起こした命令の実行結果であることを示すマークを付け、同時に例外を起こしたプログラム・カウンタを記憶しておく。後に、通常の命令がマークされたデータを参照した際に、例外処理を行なう。この方式では例外の延期は実現されているが、例外からの復帰が困難である。例えば、再実行されなければならない命令の決定が困難であるし、また、再実行命令のオペランドが再実行時にも破壊されずに残っているという保証がない。保証するためには、オペランドに対する逆依存としてスケジューリングに制限を与える必要がある。

3 プレディケート実行

我々の提案する実行モデルは、プレディケート実行方式を採っている。プレディケート実行方式は、複数の制御フロー上の命令を同時に実行することに利用できるため、分岐方向の予測が困難な非数値計算プログラムの実行の高速化には興味ある方式である。プレディケート実行とは、命令の実行の際に実行の条件を参照し、真であるときのみ実行を有効化する実行方式である。この方式では、プログラムの中の分岐を除くことができるため、複数の制御フローを単一のフローにすることができる。この方式は、これまでベクタ・マシンにおいて実現され、分岐をループ・ボディに持つループをベクタ化するために利用されている[10]。例えば、次のようなループを考える。

```
DO I = 1,64
    IF (A[I] .NE. 0) A[I]=A[I]+B[I]
CONTINUE
```

このループは分岐がループ・ボディにあるためベクタ化できないが、プレディケート実行を用いれば、次のようにベクタ化できる。

```
i1:    V1 = load A
i2:    V2 = load B
i3:    F0 = (V1 != 0)
i4:    V1 = V1 + V2   if F0
i5:    A = V1
```

命令 i4 はプレディケート実行である。即ち、命令 i4 はベクタ・マスク・レジスタ F0 が 1 の時加算を行ない、0 の時 NOP となる。

近年、プレディケート実行を非数値計算プログラムの実行の高速化に適用した研究が発表されている [11][12][13]。これらの方法は、これまでのプレディケート実行から、非数値計算プログラムに適合するように拡張されているが、投機的実行への支援が不十分で、コンパイラの命令スケジューリングに大きな制限を与えている。これらの方式では、投機的実行結果の無効化をパイプラインの無効化制御に頼っている。即ち、パイプライン内部で状態の更新が生じる前に、プレディケートの真偽が判明し、それに従って、命令のコミット、あるいは、無効化を行なう。このため、制御依存はプレディケートのオペランドのデータ依存として存在し、実質的に制御依存が解消されていない。これらの方式は、投機的実行を支援するハードウェアとしては非常に小さいが、スケジューリングに大きな制限を与えるという欠点がある。

4 アーキテクチャ

我々の提案するアーキテクチャは、コンパイラが最大の能力を発揮できるための最小のハードウェアを供給する。コンパイラのスケジューリング能力を最大限に生かすためには、スケジューリングのウィンドウに存在した制限を除き、かつ、コンパイラには困難である投機的実行により生じる問題の解決から、コンパイラを開放することが必要である。さらに、この要求を満足するハードウェア支援は、高速動作が可能なように、十分に単純である必要がある。

我々の提案するアーキテクチャは、プレディケート実行を支援することにより、複数の制御フローをウィンドウに加える。さらに、投機的実行結果をプレディケートとともにバッファし、複数の制御フローにおける投機的なマシン状態を完全に、かつ、効率的に取り扱う。ハードウェアは、コンパイラが最適にスケジュールしたコードを単純に発行し、コードに含まれたプレディケート情報をヒントにしてプレディケート実行と投機的実行を制御する。本章では、最初に、我々の提案するアーキテクチャにおける投機的実行モデルについて述べる。その後、それを支援するハードウェア構成について説明し、最後に投機的な例外の延期方法と実行の再開方法について述べる。

4.1 投機的実行モデル

プレディケート実行のために、すべての命令はプレディケートを持つ。命令の形式を次のように定める。

プレディケート ? 操作

命令のプレディケートは、ブール値を持つ分岐条件の論理式である。プレディケートが真であるときのみ操作部で示された操作の結果が有効となる。例えば、次に示す命令では、分岐条件 c1 が真で、かつ、分岐条件 c2 が偽の時、プレディケートは真となり、その時に限って、r2 と r3 の加算結果は r1 に書き込まれる。

```
c1&!c2 ? r1 = r2 + r3
```

マシンは 2 つの実行状態を持つ。1 つは、逐次的状態である。この状態は、データ依存と制御依存が共に解消した時にのみ命令の実行が行なわれる、逐次的な実行の状態であり、投機的実行の結果を含まない。もう 1 つの状態は投機的状態である。投機的状態は投機的実行結果のみからなり、それぞれの結果が逐次的状態にコミットされる条件であるプレディケートを同時に持つ。命令のオペランドがどちらの状態にあるかは、コンパイラが明示的に、参照レジスタ番号に .s と言うサフィックスを付けて表わす。例えば、次に示す命令では、加算の最初のオペランドは逐次的状態の値であり、2 番目のオペランドは投機的状態の値である。

```
c1&!c2 ? r1 = r2 + r3.s
```

命令は発行される際、そのプレディケートが計算される。プレディケートが真である場合、この時点で、この命令の実行結果は有効であることが判明しているので、通常のマシンの命令と同じように、実行を行ない逐次的状態を更新する。もしも、プレディケートが偽であれば、この時点で、この命令の実行結果は無効であることが判明しているので、単純に無効化する。これらのいずれの場合でもない場合、即ち、プレディケート計算に必要な分岐条件の少なくとも一つが未定義であるために、プレディケートの値が定義できない場合、この命令の実行結果が有効となるかどうかは、未定義の分岐条件に依存している。この命令発行は、分岐に依存した命令の、依存分岐条件が定まる前の命令発行であるから、投機的である。投機的発行では、命令の実行を行ない投機的状態を更新する。更新の際には、投機的状態の後の制御のためにプレディケートも書き込む。

投機的状態にあるデータのプレディケートは、必要な分岐条件を常に参照し、値が計算される。値が定義されない間は、対応する実行結果は保持される。必要な分岐条件が揃い、プレディケートが真になった場合、対応する実行結果を逐次状態に移行させる。偽になった場合は、値を無効化する。

4.2 ハードウェア

図 1 に、ハードウェア構成（*N* 命令発行）を示す。マシンは、通常のマシンと同様にデータを命令に従って処理するデータ・パスの他に、プレディケートの計算を行なう制御パスを持つ。制御パスは、命令のプレディケート部を受け、分岐条件を参照して、実行中の命令のプレディケートを計算する。

レジスタ・ファイルは、逐次的状態を保持する通常のレジスタ・ファイル（逐次的レジスタ・ファイル）の他に、投機的状態を保持するためにシャドウ・レジスタ・ファイルを持つ。シャドウ・レジスタは、論理的には、可能な投機的状態の数と等しい数だけ必要であるが、適度なハードウェア・コストにするために、各逐次的レジスタに対してシャドウ・レジスタは 1 つだけある。これによって、レジスタのあるエントリに対して複数の

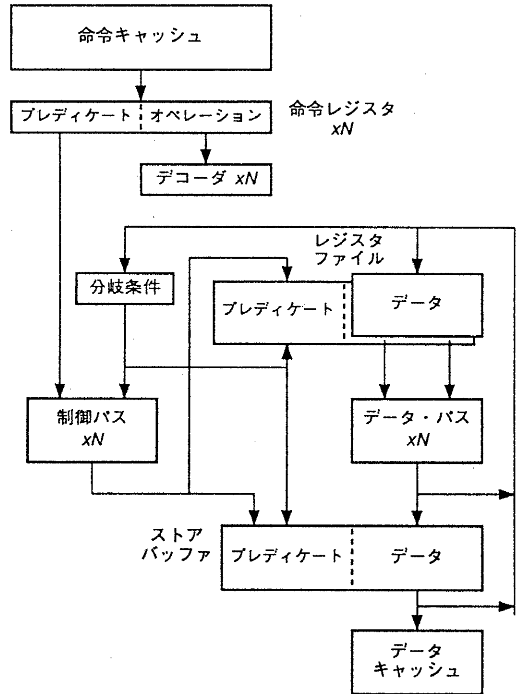

図 1　ハードウェア構成（N 命令発行）

投機的状態の必要性が生じた場合に、シャドウ・レジスタ上でコンフリクトが起こる。しかしながら、このシャドウ・レジスタでの出力依存が生じる頻度は低いと推測できるので、問題は少ないと考えられる。また、レジスタの各エントリは、投機的状態を保持するシャドウ・レジスタの他に、投機的状態のプレディケートを保持する。

メモリ・データに関しては、データ・キャッシュの前にストア・バッファを設ける。レジスタ・ファイルと異なりストア・バッファには分離したシャドウ構造はない。即ち、ストア操作が逐次的か投機的かにかかわらず、データはストア・バッファに書き込まれる。ストア・バッファの各エントリは保持しているデータが投機的である場合、そのプレディケートを保持する。ストア・バッファは FIFO で構成され、先頭のデータが有効な逐次的データであり、データ・キャッシュにロード命令によるアクセスがないサイクルに、データ・キャッシュに書き込まれる。

レジスタに書き込みを行なう命令の場合、制御パ

スでの計算結果であるプレディケートの値に従って、レジスタが更新される。即ち、プレディケートが真の場合は、逐次的レジスタが更新され、偽の場合は書き込みが無効化され、未定義の場合は、プレディケートと共にシャドウ・レジスタに実行結果が書き込まれる。ストア命令の場合、ストア・バッファはシャドウ構造のない FIFO であるので、プレディケートが偽で書き込みが無効化されなければ、ストア・データはストア・バッファに書き込まれる。ただし、プレディケートが未定義であれば、プレディケートも共に書き込まれ、書き込まれたデータが投機的であることを示すフラグをセットする。

図2にレジスタ・ファイルの構成を示す。レジスタ・ファイルの各エントリは、2つのデータと1つのプレディケートと3つのフラグの記憶を行なうフィールドよりなる。2つのデータ・フィールドは、逐次的レジスタとシャドウ・レジスタであり、それぞれ、逐次的状態と投機的状態を保持する。フラグ W は、対応するエントリに投機的状態が存在するならば、どちらのデータ・フィールドがシャドウ・レジスタであるかを示す。フラグ S は、フラグ W によって示されたシャドウ・レジスタが有効な値を保持していることを示す。さらに、フラグ E は、そのエントリに書き込みを行なった投機的命令が、実行中に例外を起こしたことを示す（例外については、3.3 節で述べる）。

各エントリは、さらに、プレディケートを計算するハードウェアを持ち、対応するエントリのプレディケート・フィールドにあるプレディケートと分岐条件を参照して計算を行なう。レジスタ・ファイルの投機的状態のコミット、あるいは、無効化は、このプレディケートの計算結果に従って、フラグ S と W を変更することによって行なう。投機的実行結果は、フラグ W で示されるシャドウ・レジスタに書き込まれ、フラグ S がセットされる。この時、同時にプレディケートが書き込まれる。この時点では、プレディケートの値は未定義の状態である。プレディケートが未定義の状態を続けている間は、そのエントリの内容は保持される。プレディケートのオペランドが揃い、値が真になった場合、フラグ W を反転させ、フラグ S をリセットする。こうすることによって、シャドウ・レジスタにあるデータを逐次的状態する。プレディケートが偽になった場合、単純に、フラグ S をリセットし、投機的状態を無効化する。

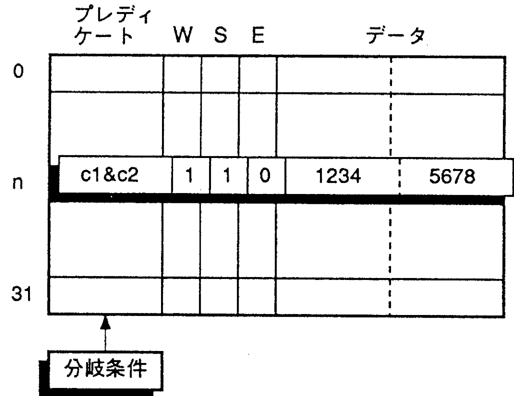

図2　レジスタ・ファイル

プレディケートは、一般的には、いかなる論理演算式でも許されるが、マシン・サイクルに影響を及ぼさない高速なプレディケート計算を、適度なハードウェア量で実現するために、現在のインプリメンテーションでは、論理積に限定している。また、命令のプレディケート部は、マシン資源上利用可能な全ての分岐条件に対応して、プレディケートを真にするために要求されるブール値でエンコードする。例えば、3つの分岐条件を記憶するレジスタ（c1, c2, c3）を持つマシンでは、命令のプレディケート部は3つのフィールドを持ち、プレディケート c1&!c2 は、"10X" とエンコードされる。即ち、参照されない分岐条件に対応するフィールドには、値 x を与える。このエンコーディングによって、プレディケートの計算は、x の値を持つフィールドに対応した分岐条件をマスクした、分岐条件とプレディケート部のビット比較演算となり、簡単なハードウェアで高速な計算が実現できる。

従来のマシンから、本アーキテクチャに追加されるハードウェアは、命令のプレディケート部を記憶する命令キャッシュのデータ・アレイ、シャドウ・レジスタ・ファイル、ストア・バッファのプレディケート記憶部、制御パス部である。制御パス部は前述のように簡単な数ビットの比較回路からなり、非常に小さなハードウェアである。命令キャッシュは、プレディケート部を1バイトとすると、25% のデータ・アレイの容量増加となるが

大きな問題ではない。シャドウ・レジスタ・ファイルに関しては、我々の見積では従来のレジスタ・ファイルとほぼ等しい ハードウェア量(数十Kトランジスタ)を必要とするだけである。これは、近年の数Mトランジスタを集積するマイクロプロセッサでは、わずか数%の増加でしかなく、問題はないと考えられる。ストア・バッファのプレディケート記憶部に関しても、ストア・バッファには8エントリ程度あれば良いと考えられ、ハードウェアの増加量(数Kトランジスタ)は小さい。また、新たに付加された回路は非常に単純であるので、速度に関するペナルティはほとんどないと考えられる。

4.3 投機的例外

我々のアーキテクチャでは、投機的命令による例外は、その実行結果がコミットされるまで延期され、また、その時に処理されるように制御される。この制御の中心的なハードウェアは、レジスタ・ファイルとストア・バッファであり、それらに記憶されているプレディケートが適切な例外の延期を効率的に行なう。

投機的命令が例外を起こした場合、例外処理を即座に行なわず、例外を起こさなかった投機的命令と同様に、レジスタ・ファイルに(無意味な結果ではあるが)実行結果とプレディケートを書き込む。この時、そのエントリのフラグEをセットする。もしも、プレディケートが偽になった時は、投機的状態を破棄する際、フラグEもリセットし、投機的例外は破棄される[*1]。もしも、プレディケートが真になった時は、レジスタ・ファイルは例外処理を開始することを知らせる信号を出す。同時に、レジスタ・ファイルの全てのエントリのフラグSとEをリセットし、投機的状態を無効化する。少なくとも1つのエントリの投機的例外をコミットした場合、レジスタ・ファイルは投機的例外処理開始の信号を出す。以上のようにして、適切な例外の延期と、正確な割り込み点を確保する。

例外処理が開始される時に投機的状態は破棄されるので、例外からの実行の再開には、コミット点に依存する投機的な命令を再実行する必要がある。そこで、プログラムの各コミット点に対して、投機的に移動された命令だけからなるコード(リカバリ・コード[14])をコンパイル時に用意する。例外ハンドラは、コミット点に対するリカバリ・コード を呼出し実行する。リカバリ・コードは、例外を起こした命令を含んでいるので、もう一度例外を起こす。今度は、逐次的状態であるから、例外は即座に処理される。リカバリ・コードからは、ジャンプ命令で中断したコミット点へ戻る。

4.4 他の研究との比較

我々の提案するアーキテクチャは、命令のスケジューリングに与える制限を極めて少なくした投機的実行を可能にする。我々のアーキテクチャは、ガード命令アーキテクチャ [11] 、ハイパー・スケジューリング [12] 、GIFT [13] とプレディケート実行を行なうという点で共通しているが、これらのアーキテクチャと異なり、投機的実行に対して十分なアーキテクチャ上の支援があり、制御依存を解消できる。また、ブースティング [9] とシャドウ構造を有し、例外処理にリカバリ・コードを使用するという点で共通しているが、ブースティングと異なり、投機的実行による副作用はプレディケートを持ち、複数の制御フローにおける投機的命令移動に対する、コミット制御と例外の適切な延期を実現している。また、我々の投機的例外の延期方式は、non-excepting 命令による方式 [6] と、レジスタ・ファイルにその情報を書き込むと言う点で共通しているが、投機的状態が持つプレディケートを利用して例外をコミット点で検出することができ、正確な例外を実現している。

5 まとめ

本報告では、制限の少ない投機的実行を可能にするアーキテクチャ上の支援を提案した。投機的実行とコンパイラによる命令発行の最適なスケジューリングは、命令の並列実行により性能を向上させるために不可欠である。ここで提案したアーキテクチャ上の支援により、複数の制御フローからの投機的命令移動を制限なく行なうことができ、コンパイラはより最適化されたコードを生成することができる。分岐方向に依存せず制御

*1: フラグEは、この動作以外ではリセットされない。即ち、レジスタ・ファイルへの書き込みによって、フラグEはリセットされない。

依存を解消することができるため、分岐方向の予測が困難な非数値計算応用に対して有用である。

提案の方式のキー・アイデアは、投機的状態をプレディケートを付けてバッファすることである。この方式は、投機的状態のコミット制御と投機的例外の適切な延期を、単純なハードウェアで実現でき、マシンのサイクル時間に与える影響は少ない。以上のように、提案の方式は、高い動作速度と高い命令レベル並列を同時に実現できる効果的な方式である。

現在、我々は、アーキテクチャの有効性を評価するためにコード・スケジューラの開発と、ハードウェアの検討を行なっている。

参考文献

[1] D. W. Wall, "Limits of Instruction-Level Parallelism", In *Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.272-282, April 1991.

[2] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism", In *Proc. 19th Int. Symp. on Computer Architecture*, pp.46-57, June 1992.

[3] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors", In *Proc. 12th Int. Symp. on Computer Architecture*, pp.36-44, June 1985.

[4] K. Murakami, N. Irie, M. Kuga, and S. Tomita, "SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture", In *Proc. 16th Int. Symp. on Computer Architecture*, pp.78-85, June 1989.

[5] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Trans. on Computers*, C-30(7):478-490, July 1981.

[6] R. P. Colwel, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler", In *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.180-192, April 1987.

[7] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors", In *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.238-247, October 1992.

[8] H. Ando, C. Nakanishi, H. Machida, T. Hara, S. Kishida, M. Nakaya, "Speculative Execution and Reducing Branch Penalty in a Parallel Issue Machine", In *Proc. Int. Conf. on Computer Design*, pp.106-113, October 1993.

[9] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor", In *Proc. 17th Int. Symp. on Computer Architecture*, pp.344-355, May 1990.

[10] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence", In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pp.177-189, January 1983.

[11] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," In *Proc. 13th Int. Symp. on Computer Architecture*, pp.386-395, June 1986.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock", In *Proc. MICRO-25*, pp.45-54, December 1992.

[13] 小松、古関、鈴木、深澤、" 拡張 VLIW プロセッサ GIFT における命令レベル並列処理機構" 、*情報処理学会論文誌*、Vol. 34, No.12, 1993 年 12 月.

[14] M. D. Smith, M. A. Horowitz, and M. S. Lam, "Efficient Superscalar Performance Through Boosting", In *Proc. Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.248-259, October 1992.