計算機アーキテクチャ　107－13
（１９９４．　７．　２１）

投機的実行を行なうマシンにおける例外処理方式

安藤秀樹　中西知嘉子　原哲也　町田浩久　中屋雅夫

三菱電機（株）
システムＬＳＩ開発研究所

並列度の低い非数値計算応用において、命令の並列実行により大きな性能向上を達成するにためには、投機的実行は必須の技術である。静的な命令スケジューリングによる投機的実行は、効率良く命令の並列性を引き出すことができるが、投機的命令の起こす例外の処理を適切に、また、効率良く行なうことは非常に難しい。本報告では、この問題に対してフューチャ・コンディションと呼ぶ方式を提案する。この方式では、例外をバッファリングすることによってコミット点まで処理を延期する。例外が検出された場合、投機的状態を破棄しコミット点に依存する投機的命令の再実行を行なう。この再実行は、例外を検出したコミット条件を参照するプレディケート実行によって実現され、プロセッサはマシン状態を再構築し例外命令を特定する。本方式は、単純なハードウェアで制限のない投機的実行を支援することができる方式である。

Handling Speculative Exceptions

Hideki Ando　Chikako Nakanishi　Tetsuya Hara　Hirohisa Machida　Masao Nakaya

System LSI Laboratory
Mitsubishi Electric Corporation

Speculative execution is strongly required for performance improvement in non-numerical applications which have little available instruction-level parallelism. Although static instruction scheduling efficiently exploits parallelism, it is difficult to handle speculative exceptions appropriately and efficiently. This paper introduces a new scheme called *future condition*. This scheme buffers speculative exceptions until the commit point to postpone the handling of the exception. If the postponed exception is detected at the commit point, the processor discards the speculative state and re-executes the speculative instructions depending upon the commit point. The predicated re-execution referring the commit condition rebuilds the corrupted state and identifies the excepting instruction. Our scheme supports unconstrained speculative execution through the simple hardware.

1 はじめに

投機的実行とは、分岐方向が判明する前に、その分岐に依存する命令の実行を行なう技術である。投機的実行は、分岐を越えた命令のスケジューリングを可能にし、その結果、命令の並列度を高めることができる。従って、この技術は、基本ブロック内に十分な命令の並列性が存在しない非数値計算応用において、命令の並列実行が可能なマイクロプロセッサのハードウェア資源を利用し、性能向上を達成するためには必須の技術である[1][2]。

しかしながら、投機的実行には解決しなければならない問題が2つある。1つは、プログラムの正当性維持の問題である。即ち、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。この問題をハードウェアで解決する方法では、一般に、投機的実行結果をバッファリングし、コミットされることが判明した時点で、マシンの状態を更新する方法 [3][4][5] が採られている。一方、ソフトウェアで解決する方法では、一般に、投機的命令の書き込みレジスタを空きレジスタに変更し、コミット時にその値を元のレジスタにコピーする方法が採られてる。

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、性能を大きく低下させる。なぜならば、行なった例外処理が真に必要であったかどうかは、後に決定される分岐方向に依存するからである。従って、投機的命令が起こした例外の処理は、その命令がコミットされることが判明するまで延期されるように制御される必要がある。

ハードウェアによる命令のスケジューリングで投機的実行を実現する方式においては、この例外の問題の解決は比較的容易である。即ち、書き込みを投機的命令の副作用として扱うことと同様に、例外を副作用の1つとして扱えば良い。具体的には、例外が発生した場合には、即座に例外処理を行わず、例外発生を示す情報をバッファリングする。後に、コミット点に到ったときに、バッファされた例外を検出する。例外処理が必要な場合、バッファをクリアし、例外処理を起動する信号を出す。この時点で、正確な割り込み点が確保されるため、投機的実行を行なわないマシンと同様の制御で例外からのプログラムの再開が実現できる。

一方、ソフトウェアによる命令のスケジューリングで投機的実行を実現する方式では、この例外をバッファリングするハードウエア支援だけでは、問題の正しい解決は困難である。その主な理由は、例外処理の後、プログラムの再開に先立って、例外を起こした命令の結果に直接的、あるいは、間接的に依存する命令の再実行も行なわなければならないためである。ソフトウェアのスケジューリング結果では、投機的命令はそうでない命令と区別が付かないだけでなく、命令の依存を実行時に判断するハードウェアを備えていないので、再実行する必要のある命令を特定することは困難である。これまで、この問題に対していくつかの解が報告されているが [6][7][8]、いずれも、まだ未解決の問題を含んでいる。

我々は、最近、制限のない投機的実行のためのアーキテクチャ上の支援について提案を行なった[9]。提案の方式は、投機的実行を効率良く支援するが、例外の問題に対しては、性能やコストの点で最適な解とは言えなかった。今回の報告では、例外の問題に対して、改善された我々の新しい解を提案する。2章では、バックグランドとして我々の投機的実行方式についてレビューする。次に、3章では、投機的命令が起こす例外における問題について説明し、4章において、我々の提案する例外処理方式について述べる。5章では、これまで報告された方式との比較を行ない、最後に、本報告をまとめる。

2 投機的実行方式

最適な命令のスケジューリングを行なうことは、命令の並列実行によって大きな性能向上を達成する上で最も重要なことである。最適なスケジューリングのために必要とされることは、コンパイラの持つスケジューリング能力を最大限に生かすために、制限のない投機的実行のハードウェア・モデルをコンパイラに与えることである。性能向上のためには、さらに、制限のない投機的実行を支援するハードウェアは、高速な動作を確保するために十分に単純でなければならない。これらの要求を満足する方式では、単純で高速に動作するハードウェアによって、コンパイラにより最適にスケジュールされたコードが実行され、その結果、大きな性能向上を達成することができる。

我々の方式 [9] では、コンパイラは命令をプレディケートと呼ぶ制御依存情報でラベリングし、投機的命令移動を行なう。ハードウェアはプレディケートを利用して投機的実行の副作用を処理す

る。具体的には、ハードウェアは投機的命令の実行結果を、その命令のプレディケートと共にバッファする。バッファされた投機的結果のプレディケートは、常に分岐条件を監視し、値を評価し、その値に従って結果のコミットを制御する。この方式では、コンパイラは制御依存をプレディケート・ラベルにエンコードするだけで、その依存から開放され、データ依存と資源制約のみで定まる最適な命令スケジュールを行なう。一方、ハードウェアは制御依存情報を明示的に与えられることによって、単純な論理で効率的に副作用を処理する。

以下、より具体的に説明する。我々の方式では、全ての命令はプレディケートを持つ。即ち、命令は次のような形式を持つ。

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、図1(a)に示すプログラムにおいて、命令 i1,i2,i3 は、我々の命令では、(b)に示すようにプレディケートによってラベリングされる。

命令スケジューラは、プログラムをある大きさの部分（リージョン）に分割し、リージョン内部の命令にプレディケートを付けてスケジュールする。プレディケートに参照される分岐条件（図1における c1,c2 等）は、レジスタ CCR = {c1,c2,...} に記憶される。命令のスケジュールにおいては、制御依存の解消はハードウェアに頼ることができるため、制御依存制約はない[1]。リージョンは1つのストレート・コードであり、ジャンプなどの制御移行は他のリージョンへ移行する場合にのみ必要となる[2]。CCR の値はリージョン内部で定義され、リージョンを移動するときに未定義状態にリセットされる。従って、CCR を参照する投機的状態はリージョン内部で閉じている。即ち、あるリージョン内で作られた投機的状態はそのリージョンでコミットされるか、無効化されるかのどちらかであり、リージョンを越えて生存しない。

```
        if (c1) {
i1:       r1 = load (r2);
          if (c2)
i2:         r3 = r1 + 1;
        } else {
i3:       r4 = r1 & r2;
        }
```

(a) プレディケート・ラベリング前

```
i1:   c1        ?  r1 = load (r2)
i2:   c1 & c2   ?  r3 = r1 + 1
i3:   !c1       ?  r4 = r1 & r2
```

(b) プレディケート・ラベリング後

図1　プレディケートによる命令ラベリング

マシンは投機的実行を支援するために、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、制御依存が解消している命令の実行結果によって作られる逐次的な命令の実行状態である。もう1つの状態は**投機的状態**である。この状態は、制御依存が解消していない命令の実行結果によって作られる投機的な命令の実行状態である。

命令の発行点では、まず、プレディケートが評価される。その結果、値が真であれば、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様に、実行を行ない逐次的状態を更新する。もしも、プレディケートの値が偽であれば、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。これらのいずれでもない場合、即ち、プレディケート評価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は、この命令の実行結果が有効となるかどうかが未定義の分岐条件に依存している投機的な場合である。投機的命令発行では、命令の実行を行ない投機的状態を更新する。更新の際、後のコミット制御のためにプレディケートも書き込む。

投機的状態は、レジスタとメモリに対して存在する。レジスタにおける投機的状態は、シャドウ・レジスタに記憶する。即ち、レジスタ・ファイルの各エントリは、逐次的状態を保持する通常のレジスタ（**逐次的レジスタ**）の他に、投機的状態を保持するためにシャドウ・レジスタを持つ。各エントリは、さらに、投機的状態のコミット条件であるプレディケートを保持し、状態の遷移を制御する。

メモリにおける投機的状態は、データ・キャッシュの前に置かれるストア・バッファに記憶する。レジスタ・ファイルと異なりストア・バッ

1: ベクタ・マシンにおけるプレディケート実行では、制御依存はデータ依存に変換され残っている。本方式では、投機的実行を支援するハードウェアによって解消される。
2: リージョン内のどの点からも脱出可

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し、状態遷移を制御する。ストア・バッファはFIFOで構成し、先頭のデータが有効な逐次的データである場合、データ・キャッシュ・アクセスに競合がなければ、データ・キャッシュにそのデータが書き込まれる。

以上のように、我々のマシンでは、投機的結果をプレディケートによってラベルリングし、シャドウ・レジスタとストア・バッファにバッファする。これらバッファの各エントリは、プレディケートの記憶とその値を計算するハードウェアを持ち、投機的実行の副作用を処理する。

3 投機的例外における問題

投機的命令の起こす例外を投機的例外と呼ぶ。例外処理には、一般に多くのサイクルを消費するので、投機的例外の処理はその命令がコミットされる時まで延期されるべきである。また、投機的実行は、プログラマの意図しない命令実行であるために、例外発生の頻度は通常の実行の場合よりも大幅に高まる上、回復不可能な致命的なエラーを起こす可能性がある。例えば、リンク・リストをたどる処理において、次のイタレーションの要素をロードする命令を投機的に実行するスケジュールを考える。この時、最後のイタレーションでは、NULLポインタでメモリ参照を行なうこととなり、致命的エラーを起こす。このスケジュールは、投機的実行では極めて一般的なスケジュールであり、特殊ではない。以上のように、投機的実行において例外処理の延期は非常に重要な問題である。

静的な投機的命令移動を行なわない命令スケジュールでは、リオーダ・バッファ[3]等を使用し例外の副作用をバッファリングすることによって投機的例外の問題は解決できる。しかしながら、静的な命令スケジュールを行なう場合では、例外の問題は解決が非常に難しい。一般に、静的スケジューリングでは、投機的例外処理に関して、次の3点を考慮する必要がある。

1. 投機的例外処理のコミット点までの延期
2. 再実行すべき命令の選択
3. 再実行される命令のオペランドの保持

これらの要求を次の例を用いて説明する。次のような逐次的なコードを考える。

```
i1:   r1 = load (r1 + 4)
i2:   branch LAB if r1 == 0
i3:   r2 = load (r3)
i4:   r4 = r5 + r2
i5:   r6 = r4 + r3
i6:   r3 = r3 + 1
i7:   r5 = r3 <<1
```

このコードで、命令 i3 - i7 を分岐命令 i2 の上方に移動する。移動後のコードを以下に示す。命令番号に ' が付いている命令は投機的移動された命令である。

```
i3':  r2 = load (r3)
i1:   r1 = load (r1 + 4)
i4':  r4 = r5 + r2
i5':  r6 = r4 + r3
i6':  r3 = r3 + 1
i7':  r5 = r3 << 1
i2:   branch LAB if r1 == 0
```

このコードで、命令 i3' が例外を起こした場合を考える。まず第1に、命令 i3' の例外処理は命令 i2 が実行され、コミットされることが判明するまで延期される必要がある。

第2に、命令 i3' の起こした例外はその時点で処理されないので、書き込みレジスタ r2 には誤った値が書き込まれる。従って、コミット点まで例外が延期され、その後、命令 i3' の例外処理が行なわれた後は、レジスタ r2 を利用した命令を再実行し、マシン状態を正しくする必要がある。この例では、命令 i4' は命令 i3' の実行結果 r2 を使用するので再実行する必要がある。さらに、命令 i4' の実行結果も誤っているので、それを利用する命令 i5' も再実行されなければならない（上の例で、ボールドで書かれたレジスタは、誤った値を保持している）。これとは逆に、命令 i1, i6', i7' は、すでに正しい実行結果を得ているので、再実行の必要はない。より正確には、命令 i1, i6' に関しては、参照レジスタが破壊されオペランドが失われているので再実行してはならない。命令 i7' は、オペランドがまだ存在するので再実行しても良い。

命令 i6' の実行は、この例において、例外からの適切な復帰を不可能にしている。命令 i3' のロード・アドレスを保持しているレジスタ r3 は、命令 i6' で破壊される。このため、命令 i3' の再実行に必要なオペランドは、再実行時には存在しない。さらに、先に述べたように、例外を起こした命令の結果より生じた誤ったマシン状態を正しく

再生させるためには、命令 i4',i5' の再実行も必要であるが、それらの命令のオペランドも保持されていなければならない。この例では、命令 i6',i7' は、命令 i4',i5' を再実行するために必要なオペランドを破壊しており、これらの命令の再実行を不可能にしている。

一般に、破壊されたレジスタの内容を元に戻すことは困難であるので、投機的移動命令の中に例外を起こす可能性のある命令がある場合、その投機的命令とコミット点の間に存在し、再実行の必要のある全ての命令のオペランドは、コミット点において利用可能なようにスケジューリングを制御する必要がある。具体的には、必要なオペランドを保持しているレジスタを破壊しないように、レジスタの割当を行なうか、あるいは、例外を起こす可能性のある命令とコミット点の間に、オペランドを破壊する命令をスケジューリングしないようにするかのどちらかである。前者の場合必要なレジスタ数が大幅に増加する。後者の場合、不要な逆依存を生じさせ性能が低下する。このような考慮は、再実行される可能性のある全ての命令のオペランドに対して払う必要があり非常に大きな負担となる。

4 投機的例外処理方式

我々の提案する投機的例外処理方式では、投機的例外はプレディケートでラベルリングされ、コミット点までバッファリングされる。これによって、コミット点まで処理を延期する。例外がコミットされた場合、誤った結果を含む投機的状態を全て破棄し、その後、ハードウェアは投機的状態を再構築するためにプログラムを巻き戻し、実行を再開する。再実行において、例外命令が特定され処理される。例外命令の特定は、過去に巻き戻された実行がいずれは到る地点の分岐条件（フューチャ・コンディション）を、現在の分岐条件に先立って知ることによって行なわれる。4.2節で本方式を詳しく説明し、4.3節で例を示す。

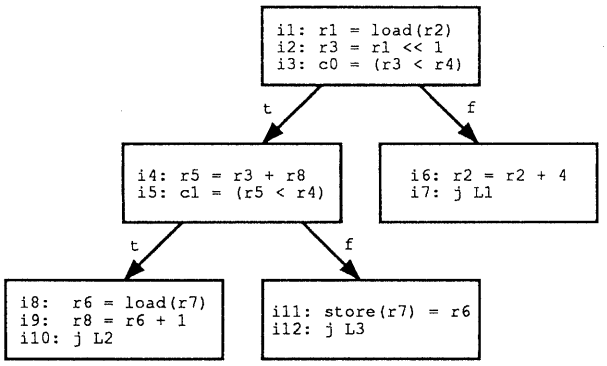
4.1 フューチャ・コンディション方式

投機的例外の延期のために、レジスタ・ファイル及びストア・バッファの各エントリに、そのエントリに書き込みを行なった命令が投機的例外を起こしたことを示すフラグ（**投機的例外フラグ**）を設ける。投機的命令が例外を起こした場合、その命令が書き込みを行なう際に、例外を起こさなかった命令と同様に、結果とプレディケートを書き込みエントリに書き込む他、そのエントリの投機的例外フラグをセットする。その後、条件の更新によってプレディケートが真になり、そのエントリの内容がコミットされる場合、結果のコミットは行なわず、代わりに、投機的例外を検出する。逆に、プレディケートが偽になったときは、そのエントリの投機的結果を破棄することと同様に、投機的例外が起こったことを記憶している投機的例外フラグもリセットする。従って、先に起こった投機的例外は検出されない。少なくとも1つの投機的例外がコミットされた場合、例外検出信号を発生する。

投機的例外が検出された場合、まず、投機的状態を破棄する。これによって、再実行すべき命令は、3章で述べたような例外を起こした命令とその結果に依存する命令ではなく、単純に、コミット点に依存する全投機的命令となる。また、再実行時の例外命令の特定のために、CCR の他に分岐条件を記憶するレジスタ（**FutureCCR**）を用意する。例外検出時には更新された CCR を FutureCCR にコピーし、CCR は更新される以前の状態に戻す。このために CCR をスタックで構成する[3]。この時点で、CCR にはコミット点の命令を実行する直前の分岐条件が記憶されており、FutureCCR には、コミット点の命令が実行された直後の分岐条件が記憶されている。プログラムは次に巻き戻されて再実行されるが、この再実行ではいずれはこのコミット点に達し、分岐条件は FutureCCR の持つ値となる。

プロセッサは、再実行のために PC に現リージョンの先頭アドレスをセットし、投機的命令のみを発行する。即ち、命令発行時にプレディケートを評価し、値が定義されない命令だけを発行する。また、リージョンの先頭アドレスを記憶するために、プロセッサはリージョン移行が生じる毎に無条件に PC をあるレジスタ（RPC）に記憶する。再実行中、先に例外を起こした命令は、例外がまだ処理されていないため再び例外を生じる。この時、プロセッサは FutureCCR を参照してプレディケートを評価することにより、この例外を処理すべきかどうかを決定する。もしも、プレディケートの値が真であれば、例外を処理する。そうでなければ、単に無視する。再実行は、投機的例外を検出したコミット点に到達した時終了し、通常の

3: 我々の現在の実現では、CCR の更新値はパイプラインの書き込みステージに到る以前に求まるので、投機的状態の生存時間を短くするため、それをパイプライン内でフォワードする。投機的例外検出時には、CCR のフォワード値は FutureCCR に書き込まれ、CCR は更新しないように制御する。この実現ではスタック構造は必要ない。



(a) リージョン

```
#1
i1:  alw      ?  r1 = load(r2)
i8:  c0&c1    ?  r6.s = load(r7)
#2
i6:  !c0      ?  r2.s = r2 + 4
i11: c0&!c1   ?  store(r7) = r6
#3
i2:  alw      ?  r3 = r1 << 1
i9:  c0&c1    ?  r8.s = r6.s + 1
#4
i4:  c0       ?  r5.s = r3 + r8
i3:  alw      ?  c0 = (r3 < r4)
#5
i5:  alw      ?  c1 = (r5 < r4)
i7:  !c0      ?  j L1
#6
i10: c0&c1    ?  j L2
i12: c0&!c1   ?  j L3
```

(b) スケジュール・コード（2命令発行）

図2　コード例

実行に戻る。

再実行時には、投機的命令のみを発行し、例外処理を行なうかどうかをFutureCCRを参照して決定するため、通常の動作と異なる。これを区別するために、再実行の期間を、リカバリ・モードと呼ぶプロセッサの実行モードとして、通常の実行状態と区別する。リカバリ・モードの開始は、投機的例外が検出された時であり、終了は、実行が例外検出点に達したときである[4]。

この方式は、投機的例外検出時に投機的状態を破棄することによって、正確な割り込み点を確保する。また、投機的状態の破棄によって、再実行を必要とする命令の決定を簡単にしている。再実行を行なう命令は、3章で述べた再実行を真に必要とする命令に正確に一致するのではなく、それを含む集合となるため多少の無駄がある。しかしながら、この方式はハードウェアを大幅に単純化することができ、大きな利益がある。リカバリ・モードでは、投機的命令だけが発行されるように制御されなければならないが、この制御は通常の実行において使用されるプレディケート評価の論理をわずかに変更するだけで実現できる。また、リカバリ・モードにおける例外命令の特定に必要なハードウェアは、FutureCCRとそれを参照してプレディケートを評価するハードウェアだけであり、非常に小さい。

4.2 例

図2 (a) に示すリージョン（スケジュールの単位）をスケジュールしたコード (b) を用いて、フューチャ・コンディションによる例外処理方式を説明する。スケジュール・コードにおいて、レジスタ番号に付けられた指定子 `.s` は、シャドウ・レジスタを指定することを表す。例えば、`r6.s` とは、レジスタ `r6` のシャドウ・レジスタを表す。また、プレディケートが `alw` とは、その操作が分岐条件に関わらず常に実行されることを示す。

まず、このリージョンに移行する際にリージョンの先頭アドレスが、RPCに記憶される。これは、投機的例外が検出された際の再実行開始アドレスを与えるためである。最初のサイクルでは、命令 `i1,i8` が実行される。この時、命令 `i8` が例外を起こしたとする。この例外は投機的であるから、レジスタ `r6.s` の投機的例外フラグがセットされるだけで、例外処理は行なわれない。さらに、次のサイクルにおいても、命令 `i11` が例外を起こしたとする。この例外も投機的なので、単に、ストア・バッファの書き込みエントリの投機的例外フラグがセットされるだけである。実行が進み、命令 `i3` により分岐条件 `c0` が真に定義され、さらに、命令 `i5` により分岐条件 `c1` が真に定義された

4: 投機的例外の検出は例外原因の1つであり、通常の例外と同様に、例外アドレスはそれを記憶する特別なレジスタ（EPC）に記憶される。従って、リカバリ・モードの終了は、EPCとPCが一致したときである。

とする。この場合、命令 i8 の書き込みレジスタ r6.s のプレディケートは c0&c1 なので、投機的例外がコミットされる。このため、投機的例外が検出されたことを示す信号が発せられる。この信号に反応して、投機的状態は破棄される。即ち、命令 i8,i9,i11 の結果は破棄される。さらに、現在の CCR を FutureCCR にポップする。これにより、この時点で、CCR = {1,U}, FutureCCR = {1,1} となる。ここで、CCR, FutureCCR の各要素は、それぞれ、分岐条件の c0,c1 に対応する。値 U は未定義を意味する。

さらに、マシンはリカバリ・モードに遷移し、プログラムは、RPC を使ってこのリージョンの先頭に巻き戻され、投機的状態の再構築を開始する。リカバリ・モードでは、CCR を参照して投機的命令のみが発行される。例外が起こった時は、Future CCR を参照して例外処理を行なうかどうかを決定する。リカバリ・モードの最初のサイクルでは、命令 i1 は投機的命令でないので発行されない。命令 i8 は投機的命令なので発行され、再び例外を起こす。この場合、命令 i8 のプレディケートは FutureCCR を参照して評価される。FutureCCR は {1,1} であるから、命令 i8 のプレディケート c0&c1 は真になる。従って、この例外は処理される。例外処理は、通常の非投機的実行マシンの例外処理と同様であり、処理後、命令ブロック#1 の実行に戻る（依然としてリカバリ・モードは続く）。次のサイクルでは、命令 i6 は投機的でないので発行されない（CCR は {1,U} であることに注意）。命令 i11 は投機的なので発行され、再び例外を起こす。しかしながら、この命令のプレディケートは FutureCCR を参照すれば偽であるので、例外処理は行なわれない。次のサイクルでは命令 i9 が実行される。その次のサイクルでは、ブロック#4 の命令はどちらも投機的でないので発行されない。ブロック#5 は、初めの投機的例外を検出した地点であり、PC と EPC が一致することによって検出される。これにより、リカバリ・モードを終了し通常の実行モードに戻る。

5 他の研究との比較

我々の投機的例外の処理方式は、投機的例外処理における問題を全て解決しているだけでなく、これまで報告されている方式よりも効率の点で優れている。これまで報告されている方式で、ハードウェアの支援なしに問題を解決できる方式はない。少なくとも、例外を起こした命令が投機的命令であり、その命令の起こした例外を延期させる必要があることをハードウェアに知らせる手段が必要である。この最小のハードウェア支援を持った方式が、**non-excepting** 命令方式 [6] である。この方式では、通常の命令と同一の操作を行なうが、例外処理を起動しない命令（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命令移動する場合は、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、書き込み先に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る。後に、他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。この方式は、例外の延期を非常に単純なハードウェアで実現しているが、プログラムの再開が困難である。即ち、例外発生点が不明であり、マシン状態を正しくするために再実行されなければならない命令の決定が困難である。さらに、再実行命令のオペランドの保持に関しても解決されていない。

sentinel scheduling [7] は、non-excepting 命令を拡張し、Colwel の方式での問題点をいくつか解決している。sentinel scheduling では、投機的命令は全てラベリングする。投機的命令が例外を起こした場合、結果に例外を起こしたことを示すマークを付けるだけでなく、書き込み先にその命令のアドレスを書き込む。さらに、後に、その結果を non-excepting 命令が利用する場合は、例外処理を起動するのではなく、マークと例外アドレスを受取り、それを書き込みレジスタを用いて伝達する。例外処理は、マークされた結果を通常の命令が利用する時のみ起動される。例外が検出された場合、伝達された例外アドレスから、投機的ラベルのついた命令だけを再実行しマシン状態を再構築する。sentinel scheduling では、このように例外命令の特定とマシン状態の再構築に対して一定の解を与えている。しかしながら、再実行される命令はコミット点に依存しない命令も含み非常に多く、オペランドの維持のためにスケジューリングに大きな制限を与えている。

boosting [8] は、投機的例外からのプログラムの再開の問題に対して、コンパイラによる解決を示している。boosting では、静的に最も通る確率の高いパス（トレース）を撰択し、そのトレースの中で命令スケジューリングを行なう。投機的に命令移動を行なった場合、その命令が越えた分岐の数（**boosting** レベル）で命令をラベリングする。正しく分岐が予測された時、結果がコミットされ、そうでなければ破棄される。投機的例外に対しては 1 ビットのシフタを用意し、例外における副作用をバッファする。即ち、例外が発生した時は、

boosting レベルに対応したビットをセットし、分岐予測がヒットする毎にシフトする。セットされた値がシフタより出てきた場合、投機的例外が検出される。投機的例外が検出された場合は、投機的状態を破棄すると共に、リカバリ・コードと呼ぶ再実行命令だけからなるコードを呼出し実行する。再実行中に例外が再発生し、プロセッサはこれを処理する。

リカバリ・コードは、各コミット点に対して依存する投機的命令だけからなるコードであり、コンパイラによって用意さる。この方式は、再実行の開始点の特定と再実行すべき命令の選択は、ハードウェアには困難であるが、コンパイラには容易であることを利用している。リカバリ・コード方式は、投機的例外における問題を全て解決しているが、効率の点で2つの問題を持っている。1つは空間効率の問題である。即ち、この方式は、各コミット点に対し、リカバリ・コードとコミット点からそれを呼び出すジャンプ・テーブルが必要である。これによって、全バイナリ・サイズがプログラム・コード・サイズの倍以上になる。2つ目は時間効率の問題である。即ち、ジャンプ・テーブルやリカバリ・コードの参照は、通常の実行プログラムとはアドレスが大きく異なる空間の参照であるので、実行における空間局所性が悪い。従って、参照に大きな時間を要する可能性が高い。これは、TLB ミス等の軽い例外処理では、性能を低下させる原因となる。

6 まとめ

フューチャ・コンディション方式は、プレディケートで副作用のラベリングをすることによって、適切な例外の延期を実現しているだけでなく、静的スケジューリングにおいて問題となるプログラムの再開に対しても、ハードウェアの支援によって解決している。我々の方式は、投機的例外がコミットされた時に、コミット点に依存する投機的命令を再実行することによって例外命令の特定とマシン状態の再構築を行なう。これを実現するためのキー・アイデアは、投機的例外を検出しプログラムを過去に巻き戻して実行を行なう際に、再実行命令の選択をマシンの現在の分岐条件を参照して行なう一方、例外命令の特定は、巻き戻された実行がいずれは到る将来の地点の分岐条件を参照して行なうことである。この方式は、プレディケート実行に必要な機構にわずかなハードウェアを追加するだけで実現でき、実行時にオーバヘッドがない。以上のように、本方式は、高速動作を確保できる単純なハードウェアで、制限のない投機的実行を支援することができる方式である。

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] 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.

[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] 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.

[9] 安藤秀樹、中西知嘉子、町田浩久、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" 、*第1回「ハーバフォーマンス・コンピューティングとアーキテクチャ評価」に関する北海道ワークショップ（HOKKE-1）、情報処理学会研究会報告*、94-ARC-105-5、pp. 33-40, 1994 年 3 月.