プレディケーティング：VLIWマシンにおける投機的実行のためのアーキテクチャ上の支援

安藤秀樹[†] 中西知嘉子[†]
原哲也[†] 中屋雅夫[†]

VLIWマシンは，並列性を高める最適な命令スケジューリングにより，高い性能を達成する可能性を持っている．しかし，並列性を高めるために必要な投機的実行がコンパイラだけでは実現することが難しいという問題がある．本論文では，コンパイラによる投機的実行に対する制限を取り除くプレディケーティングと呼ぶ機構を提案する．プレディケーティングでは，投機的に行った命令の実行結果にタグとして，その命令の制御依存情報であるプレディケートを付加し一時的に格納するハードウェアを用意する．格納した投機的実行結果は，このプレディケートを用いて，マシン状態の更新と実行結果の無効化を効率良く行う．本機構を用いれば，コンパイラは複数の制御パスから複数の基本ブロックを越えて命令を移動することが可能となり，その結果，最適に命令をスケジューリングできる．評価の結果，従来の機構を搭載した場合に比べて大幅に性能を改善できることを確認した．

Predicating: An Architectural Support for Speculative Execution in a VLIW Machine

HIDEKI ANDO,[†] CHIKAKO NAKANISHI,[†] TETSUYA HARA[†] and MASAO NAKAYA[†]

A VLIW machine has a potential to achieve good performance since the compiler schedules instructions best to exploit instruction-level parallelism (ILP). It is, however, difficult to improve performance due to a limited ability to speculative execution for exploiting ILP. This paper proposes a new mechanism, called *predicating*, which removes restrictions that are imposed on speculative execution through the compiler. Predicating buffers the result of speculative execution with the predicate of the instruction as a tag; the predicate is control dependence information. The buffered result is efficiently updated to the machine state or squashed by referring to the predicate. Predicating allows the compiler to move instructions from multiple paths beyond multiple basic blocks. As a result, the compiler can optimize the instruction schedule. Evaluation results show that our mechanism significantly improves performance.

1. はじめに

プログラムの中に存在する命令レベルの並列度を調べた研究報告[1),2)]によれば，非数値計算応用のプログラムでは，基本ブロック内に存在する命令レベルの並列度は非常に小さい．そのため，単に資源を増加させるだけではマイクロプロセッサの性能を大きく向上させることはできない．さらにこれらの研究報告は，もしも制御依存を取り除くことができるならば，命令レベルの並列度は飛躍的に増加するとしており，制御依存の除去が性能向上の鍵となっていると結論している．

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う技術である．投機的実行を行えば制御依存を取り除くことができ，並列度を高めることができる．したがって，大幅な性能改善のためには必須の技術である．

最新のマイクロプロセッサは，種々のスーパスカラ技術[3)~5)]を用いて，投機的実行を行い，高い性能を実現している．しかし，複雑な論理を必要とするためハードウェア量が多く，また，動作速度向上が難しいという欠点がある．

これに対して，VLIWマシンでは命令のスケジューリングをコンパイラが行うため，本質的には高い並列度が得られるうえ，ハードウェアを単純化できるという長所があり，スーパスカラ・プロセッサより高い性

† 三菱電機株式会社システムLSI開発研究所
System LSI Laboratory, Mitsubishi Electric Corporation

能を達成する可能性を持っている．半面，投機的実行を行うことが難しいため，実際には単純なVLIWマシンで高い性能を達成することは困難である．投機的実行が困難な理由は，投機的実行により発生する種々の問題をコンパイラだけで完全に解決できないためである．このため，コンパイラは，投機的実行を行うための命令移動（**投機的命令移動**と呼ぶ）に関して，制限を受けていた．

これまで，投機的実行により生じる問題を解決し，コンパイラによる命令移動に対する制限を取り除くためのハードウェア機構が数多く提案されている（たとえば，*guarding*[6]）．これらの機構を用いれば，投機的命令移動に課せられていた制限を緩和することはできるが，依然としてコンパイラの自由な命令移動は制限されていた．

本論文では，**プレディケーティング**と呼ぶハードウェア機構を提案する．この機構は，コンパイラに課せられていた投機的命令移動に対する制限を取り除くことができる．本論文では，以下，まず2章で投機的実行における問題について説明する．3章では，我々の提案するプレディケーティングについて議論する．4章では性能向上を評価した結果について述べる．5章で結論を述べる．

2. 投機的実行における問題

投機的実行には，一般的に，次の2点に関して問題がある．

- プログラムの意味の維持
- 投機的に移動された命令（**投機的命令**と呼ぶ）の起こした**例外の処理**

以下，この2つの問題に関して説明する．

2.1 意味の維持

コンパイラは投機的命令移動を行う際，移動の前後でプログラムの意味が変わらないようにしなければならない．具体的には，分岐を越えて命令移動を行う場合，その命令の書込みが，他の分岐方向で使用される値を破壊してはならない．次に示す例を考える．

```
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場合を考える．この命令i2の実行により，レジスタr2の値が破壊されてしまうので，ELSE部に制御が移行した際に，命令i3の実行結果は正しくない．つまり，プログラムの意味を保っていない．このように命令の移動がプログラムの意味を変えてしまう場合，この命令移動は**不正**（*illegal*）であるという．

不正な命令移動に関しては，コンパイラは**レジスタ・リネーミング**によって**回避**することができる．ある分岐を越えて命令移動を行う場合，レジスタ・リネーミングでは，まず，その命令の書込みレジスタを，他の分岐方向でliveでない値を持つレジスタ（空のレジスタと呼ぶ）に変更する．その後，その変更した書込みレジスタから元のレジスタに値をコピーする命令を挿入する．先にあげた例では，レジスタ・リネーミングを用いて，次のように命令i2を分岐命令i1の上に移動することができる．

```
i2': r5 = load r3; # 書込みレジスタの変更
i1 : if (r1)
i4 :    r2 = r5;     # 元のレジスタに値を
     else                              コピー
i3 :    r4 = r2 + r3;
```

ここで，レジスタr5は，ELSE部へのパス上で空とする．命令i2の書込みレジスタをr5に変更（命令i2'）し，さらにr5の値を元のレジスタr2にコピーする命令i4を挿入する．このようにして，不正な命令移動を解決できるが，コピー命令の挿入により資源を消費する．

不正な命令移動は，メモリ上の値に関しても同様に存在するが，レジスタ・リネーミングは適用できない．

2.2 例外の処理

投機的実行のもう1つの問題は，例外処理に関するものである．投機的に実行された命令が起こした例外を，**投機的例外**と呼ぶ．投機的例外が生じた場合，通常のプロセッサと同様に，即座に例外処理を行うと，不正にプログラムを終了させてしまうか，あるいは，性能を大幅に低下させる．なぜならば，例外処理を行う時点では，投機的命令の実行結果が必要かどうか不明なためである．したがって，投機的命令が例外を起こした場合，次にあげる2点が要求される．

- 投機的例外の処理は，例外を発生した命令の実行結果が必要となる（あるいは，必要であることが分かる）ときまで延期されなければならない．
- 延期されている例外処理は，例外を発生した命令の結果が不要と分かった時点で，サイクルを消費することなく，無効化されなければならない．

延期されている投機的例外の処理が必要であると判断することを，「投機的例外を検出する」という．また，投機的に移動した命令が例外を起こす可能性がある場合，この命令移動は**危険**（*unsafe*）であるという．

投機的例外に関しては，例外処理の延期以外に，実

行再開に関する問題がある．もし例外が致命的でないならば（たとえば，ページ・フォールト），例外を処理した後，実行を再開しなければならない．通常の実行（非投機的な実行，あるいは，逐次的実行と呼ぶ）において生じた例外は，実行の再開が必要な場合，例外を起こした命令から実行を再開すればよい．すなわち，例外を起こした命令を再実行し，それに続く命令より実行を新たに開始すればよい．投機的例外の場合，次の2つの問題により実行再開は簡単ではない．

- 再実行を行わなければならない命令の選択
- 再実行を行わなければならない命令のオペランドの保持

投機的例外処理からの実行再開では，再実行すべき命令は，例外を起こした命令だけではない．なぜなら，投機的命令が例外を起こした時点で例外は処理されず．その投機的命令の誤った実行結果は，レジスタやメモリに書き込まれる．さらに，その誤った結果を参照して実行される命令があれば，その実行結果も誤っており，それも書き込まれる．さらに，これを参照する命令があれば，誤った実行結果の書込みは続く．これら一連の誤った実行結果の書込みは，最初に起きた投機的例外が検出されるまで続く．

このため，例外からの実行再開では，例外を起こした命令の再実行だけではなく，例外を起こした命令に依存している命令も再実行しなければならない．投機的例外処理の延期により不正となったマシン状態を，命令の再実行によって正しいマシン状態にすることを，**マシン状態の再構築**と呼ぶ．ここで，マシン状態とは，命令の実行結果により構成される状態をいう．

再実行によってプログラムの意味が変わってはならないから，再実行される可能性のある命令のオペランドは，そのときまで保持されなければならない．すなわち，再実行される可能性のある命令の参照するレジスタは，投機的例外が検出される時点まで，上書きしてはならない．

以下，投機的例外処理の延期とマシン状態の再構築について，次のコード例を用いて説明する．

```
i1': r1 = load r2
i2 : r3 = r3 + 1
i3': r4 = r1 + r5
i4': r6 = r4 & 1
i5 : branch LAB if r3
```

ここで，命令 i1', i3', i4' は，分岐命令 i5 の分岐先（LAB）から i5 の上方へ移動した投機的命令，命令 i2 は通常の（逐次的）命令である．

投機的命令 i1' が例外を起こしたと仮定する．まず，この例外は分岐命令 i5 が実行されるまで，処理が延期される．そして，分岐命令 i5 が実行されて分岐することが分かった場合，この例外を検出し処理を行う．分岐しない場合，例外を無効にし，分岐命令 i5 に続く命令の実行を続行する．

例外を検出した場合，例外処理の後，例外を発生した命令 i1' を再実行する．その後，不正な値を保持していた r1 を参照した命令 i3' と，i3' の不正な実行結果を参照した命令 i4' を再実行し，マシン状態を再構築する．逆に，命令 i2 は再実行してはならない．また，コンパイラは，コード生成の際，レジスタ r2 と r5 を例外が検出される命令 i5 の実行の時点まで，再利用してはならない．これらのレジスタが保持している値は，再実行時に使用されるからである．

コンパイラだけでは，投機的例外の処理の延期，再実行によるマシン状態の再構築はできないので，ハードウェアによる支援が必要である．

3. プレディケーティング

命令の並列実行による性能向上を達成するうえで，最適な命令のスケジューリングは最も重要である．そのためには，投機的命令移動に関して制限のないハードウェア・モデルをコンパイラに対して与えることが必要である．性能向上のためにはさらに，そのような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．

本章では，コンパイラによる投機的命令移動に関する制限を除く機構であるプレディケーティング[7),8)]について提案を行う．まず最初に，プレディケーティングにおける 実行モデルを述べ，次にそれを実現するアーキテクチャについて説明する．次にコード例を用いて，詳細な動作を説明する．そして，投機的例外の処理方式について議論する．最後に，既存の方式との比較を行う．

3.1 実行モデル

プレディケーティングでは，すべての命令はプレディケートを持ち，次のような形式である．

プレディケート ? 操作

プレディケートは，その命令が依存する分岐条件の論理式である．命令の意味は，「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」である．

プレディケートが参照する分岐条件は，汎用レジスタではなく，CCR（Condition Code Register）と呼ぶレジスタに記憶する．CCR は複数のエントリを持ち，各エントリは異なる分岐条件の値を記憶する．

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

(a) (b)

図1 プレディケーティングにおけるコード例
Fig. 1 Code example with predicating.

図1にプレディケーティングにおけるコードの例を示す．図1(a)が通常のマシンに対するコードで，図1(b)がプレディケーティングにおける対応するコードである☆．命令i1とi1'，i2とi2'，i3とi3'が対応している．cnはCCRの第nエントリを示す．

マシンは投機的実行を行うために，**逐次的状態**と**投機的状態**の2つのマシン状態を持つ．逐次的状態は，制御依存が解消している命令の実行結果により構成されるマシン状態である．投機的状態は，制御依存が解消していない命令の実行結果により構成されるマシン状態である．命令の実行方式を説明する．

命令の発行時，プレディケートが評価される．その結果，値が真であれば，通常のマシンの命令と同様に，実行を行い逐次的状態を更新する．これは逐次的実行である．もしも，プレディケートの値が偽であれば，発行時点で命令は無効化される．

プレディケーティングでは，命令の発行時点において，プレディケートが参照する分岐条件の値がまだ定義されておらず，その結果，プレディケートの値の真偽が不明な場合でも命令を実行する．これは依存している分岐条件の値が定まる前の実行（制御依存が解消してない命令実行）であるから，この命令の実行結果は投機的状態に書き込む．このとき，逐次的状態への書込みと異なり，書込みを行う命令のプレディケートを実行結果にタグとして付ける（実現方法については，3.2節で述べる）．

このように，マシン内部ではプレディケートの評価値として，真と偽という値以外に，「値がまだ定まっていない」ということを意味する「値」を持つ．以下，この「値」を**未定値**と呼ぶ．すなわち，プレディケートの値は，マシン内部では，**真**，**偽**，未定値のいずれかをとる．

投機的状態にある実行結果にタグとして付けられたプレディケートは，CCRを参照し毎サイクル評価される．評価値が未定値の間は，対応する実行結果はそのまま保持される．あるサイクルにおいて分岐条件の値が定義され，プレディケートの値が真あるいは偽に定まったとする．プレディケートの値が真になった場合は，その実行結果を逐次的状態に移行させる．すなわち，その実行結果で逐次的状態を更新し，その実行結果を投機的状態から消去（無効化）する．偽になった場合は，その実行結果は無効化される．

☆ このコードはスケジューリングによって並べ変えられるので，図1(a)のコードが一意に図1(b)のコードになるわけではない．

```
i1:  alw   ? c1 = r1 < 0
i2:  c1    ? r2 = r3 + 1
```

(a)

```
i2:  c1    ? r2 = r3 + 1
i1:  alw   ? c1 = r1 < 0
```

(b)

図2 逐次的実効と投機的実効（プレディケートalwは恒真を意味する）
Fig. 2 Sequential execution and speculative execution (predicate alw means "always true").

プレディケーティングにおける命令実行を，**図2**に示す例で説明する．図2(a)のコードでは，命令i1によって分岐条件c1の値が定義される．命令i2の実行では，プレディケートが参照する分岐条件c1の値はすでに定義されている．プレディケートの評価値が真ならば実行が行われ逐次的状態を更新，偽ならば無効化される．

図2(b)のコードでは，命令i1の前にi2が実行される．命令i2の実行では，プレディケートが参照する分岐条件c1の値は定義されていないので，プレディケートの評価値は未定値である．命令i2の実行結果は，いったん投機的状態に書き込まれる．その際，その実行結果にタグとしてプレディケートc1を付加する．命令i1が実行され，c1が真に定義されたならば，実行結果に付加されたプレディケートの評価値も真となり，実行結果は逐次的状態に移動される．c1が偽に定義されたならば，実行結果に付加されたプレディケートの評価値も偽となり，実行結果は無効化される．

投機的状態にある実行結果で逐次的状態を更新することを，投機的実行結果の**コミット**という．またこのとき，この実行結果を生成した投機的命令はコミットされたという．さらに，ある投機的実行結果のコミットが行われた命令列中の点を，その実行結果の，あるいは，その実行結果を生成した命令の**コミット点**という．

3.2 アーキテクチャ

前節で説明した実行モデルを実現するアーキテクチャについて説明する．**図3**にハードウェア構成を示

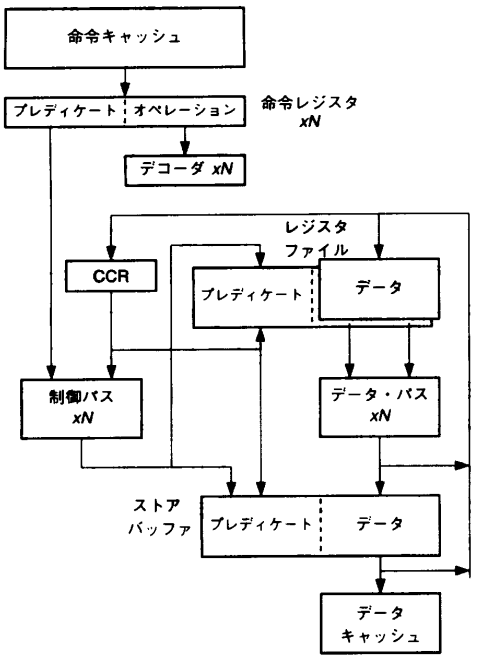


図3 ハードウェア構成（N 命令発行）
Fig. 3 Hardware organization (N-instruction issue).

す．以下，従来の単純な VLIW マシンと異なる点について説明を行う．

3.2.1 CCR

CCR は，先に述べたように複数の分岐条件の値を記憶するレジスタである．CCR の各エントリは各分岐条件に対応し，以下の 3 値のいずれかの値を記憶する．

- 真：対応する分岐条件が真に定義されている．
- 偽：対応する分岐条件が偽に定義されている．
- 未定義：対応する分岐条件が未定義である．

CCR のエントリは，条件の評価結果を書き込む命令（**分岐条件設定命令**）によって真か偽に定義される．分岐条件設定命令は，2 つのレジスタ，あるいは，1 つのレジスタと即値の大小一致比較を行い，CCR の指定されたエントリに書込みを行う．

また，CCR の全エントリは，制御移行を行うジャンプ命令の実行によって，「未定義」値に設定される．CCR の全エントリの値を「未定義」にすることを，「CCR をリセットする」という．ジャンプ命令実行時に CCR をリセットする理由は，コンパイラの命令スケジューリングと深くかかわっているので，付録 A で説明する．

ハードウェアによってリセットされるまで，コンパイラは CCR のエントリの再割当てを行わない．また，分岐条件設定命令は例外を起こすことはない．このため，分岐条件の値を定義する命令のプレディケートは，その制御依存にかかわらず `alw`（恒真を意味する）である．

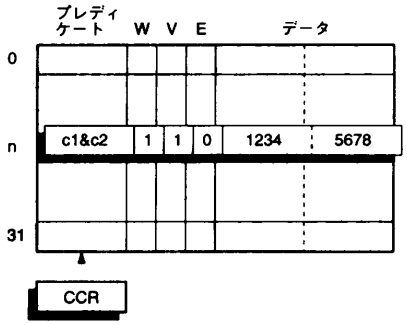


図4 プレディケート付きレジスタ・ファイル
Fig. 4 Predicated register file.

3.2.2 制御パス

制御パスは，実行中の命令のプレディケートを，CCR に記憶された分岐条件の値を参照して評価する．プレディケートが真と評価された場合，データパスで実行中の命令の実行結果を逐次的状態に書き込むように制御する．プレディケートが偽と評価された場合，実行結果を無効化する．プレディケートが未定値と評価された場合，実行結果を投機的状態に書き込むように制御する．

3.2.3 レジスタ・ファイル

レジスタ・ファイルの各エントリは，**逐次的レジスタ**と**投機的レジスタ**と**プレディケート記憶部**を持つ．逐次的レジスタは，そのエントリに対応するレジスタ値に関する逐次的状態を保持し，投機的レジスタは，そのエントリに対応するレジスタ値に関する投機的状態を保持する．

制御パスにおいて命令のプレディケートが真と評価された場合，実行結果は逐次的レジスタに書き込まれる（偽であれば，書込み前に無効化される）．未定値と評価された場合，実行結果は投機的レジスタに書き込まれる．同時にその命令のプレディケートも，対応するエントリのプレディケート記憶部に書き込まれる．各エントリのプレディケート記憶部は CCR を参照し，プレディケートの値を毎サイクル評価する専用のハードウェアを持つ．プレディケートが真と評価された場合は，投機的レジスタに格納された値は逐次的レジスタにコミットされ，偽と評価された場合は，投機的レジスタに格納された値は無効化される．

図 4 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 つのデータ記憶部，1 つのプレディケート記憶部，3 つのフラグ（W，V，E）

を持つ．2つデータ記憶部のどちらか一方が，逐次的レジスタであり，他方が投機的レジスタである．フラグWは，どちらのデータ記憶部が投機的レジスタであるかを示す．フラグVは，フラグWで指定された投機的レジスタの保持する値が有効であることを示す．フラグEは，そのエントリに書込みを行った命令が実行中に例外を起こしており，かつ，その処理が延期されていることを示す．

投機的実行結果は，フラグWで指定されたデータ記憶部に書き込まれる．同時に，フラグVがセットされ，プレディケートが書き込まれる．コミット動作は，フラグWとVを操作することによって行われる．すなわち，プレディケートが真と評価された場合，フラグWを反転させ，フラグVをリセットする．これは投機的実行結果のコミットに相当する．また，プレディケートが偽と評価された場合，フラグVをリセットする．これは投機的実行結果の無効化に相当する．

投機的命令が実行中に例外を起こした場合，フラグVをセットしプレディケートを書き込むほか，フラグEをセットする．後に，フラグVとフラグEの両方がセットされているエントリのプレディケートが真と評価された場合，レジスタ・ファイルは投機的例外を検出したことを知らせる信号を出す．この信号によって，マシンは投機的例外の処理に関する動作を行う．この動作については3.4節で述べる．

論理的には，各逐次的レジスタには異なるプレディケートを持つ複数の投機的状態が存在しうるので，複数の投機的レジスタが必要である．しかし，我々はハードウェア量を抑えるために各逐次的レジスタにはただ1つの投機的レジスタを与えることとした．この制限によって，異なるプレディケートを持つ実行結果の書込みの間で競合が生じる可能性があるが，実際には，この競合はほとんど生じることはなく，性能に与える影響は小さい．無限の投機的レジスタを持つマシンに対する性能低下を評価したところ，性能低下はわずか1%以下であった．

3.2.4 ストア・バッファ

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．ストア・バッファはFIFOで構成し，ストア命令の実行が，逐次的か投機的にかかわらず，ストア値はストア・バッファにいったん書き込まれる．FIFOの先頭の値が有効な逐次的値であれば，その値はデータ・キャッシュに書き込まれる．レジスタに関する投機的状態と同様に，1つのメモリ・ロケーションに対して，1つの投機的状態しか許さない．これは，後続のロード命令へのデータのフォワーディングを単純にするためである（付録B参照）．

ストア・バッファの各エントリは，1つのデータ記憶部のほかに，レジスタ・ファイルと同様にプレディケート記憶部と3つのフラグ（W, V, E）を持つ．ただし，フラグWは，そのエントリのデータ記憶部が保持する値が，投機的実行結果であることを示す．

ストア命令は，ストア・バッファの最終エントリに書込みを行う際，フラグVをセットする．もしも，ストア命令の実行が投機的であれば，フラグWをセットする．ストア・バッファの先頭のエントリのフラグVがセットされており，かつ，フラグWがセットされていなければ，そのエントリに格納されている値はデータ・キャッシュに書き込まれる．

3.2.5 命　　令

命令の操作部，プレディケート部について順に説明を行う．

(1) 操作部

操作部は，通常のマシンの命令と基本的に同一である．ただし，読出しを行うレジスタの指定においては，逐次的レジスタと投機的レジスタのどちらを参照するかを命令の中で明示する．たとえば，次の命令では，レジスタ・ファイルの第2エントリの投機的レジスタ（r2.s）と，第3エントリの逐次的レジスタ（r3）を参照することを示す．

```
c1 ?  r1 = r2.s + r3
```

読出しレジスタ番号に付けた .s は，投機的レジスタを参照することを意味し，**投機的レジスタ指定子**と呼ぶ．

一方，書込みレジスタに関する逐次的レジスタか投機的レジスタかの指定は，命令コードにエンコードしない．制御パスが実行時に指定するからである．

制御移行を明示的に行う命令（**制御移行命令**：通常のマシンでは，分岐命令とジャンプ命令☆）は，プレディケーティングでは，ジャンプ命令のみである．条件分岐は，プレディケート付きジャンプ命令で実現する．たとえば，アドレスLABへ制御移行を行う分岐命令，

```
branch LAB if r1 < 0
```

はプレディケーティングでは，

```
alw   ? c1 = r1 < 0
c1    ? jump LAB
```

とする．

☆ 条件ジャンプ命令を分岐命令，無条件ジャンプ命令を単にジャンプ命令と呼ぶ．

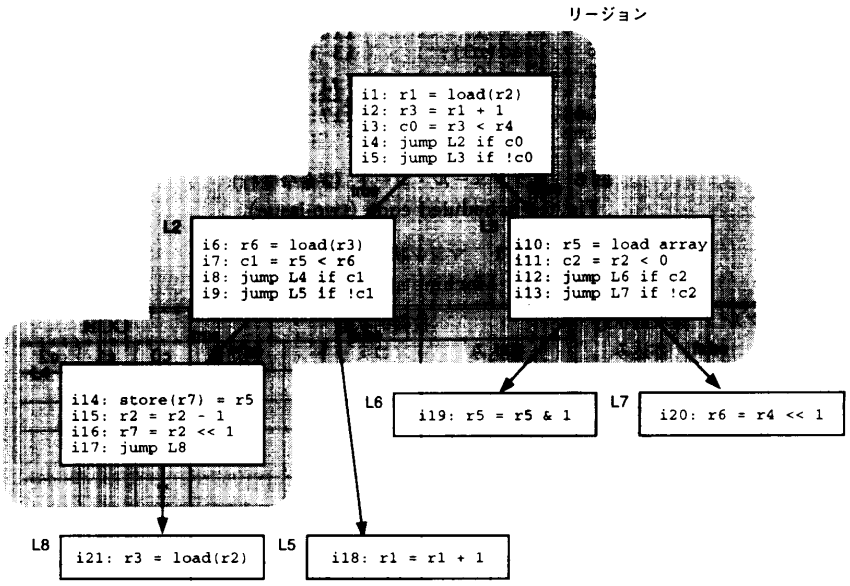


図5 スケジューリング前のコード
Fig. 5 Code before scheduling.

(2) プレディケート部

プレディケートは，一般的には，いかなる論理式でもよい．しかし，複雑なプレディケートは，その評価に必要なハードウェア量と評価論理における信号遅延時間の観点から，現実的でない．そこで，プレディケートに許される論理式として，分岐条件およびその否定の論理積に限定した．たとえば，プレディケート c1&c2 や c1&!c2 は許すが，c1|c2 は許さない．

さらに，このプレディケートを，真（1）か偽（0）か“don't care”（X）のいずれかの値を持つ要素の列（ベクタ）でエンコードする．プレディケートの論理式が cn（の否定でない）項を持つとき，ベクタの第 n 要素を 1，cn の否定項 !cn を持つとき，ベクタの第 n 要素を 0 とする．プレディケートの論理式が cn，!cn のどちらの項も持たないとき，ベクタの第 n 要素を X とする．たとえば，CCR が 3 つの分岐条件 c1,c2,c3 の値を保持する 3 つのエントリを持つ場合，プレディケート c1&!c2&c3 は {1,0,1}，プレディケート c1&c3 は {1,X,1} とエンコードする．

以上のようにすることによって，プレディケートの評価はきわめて単純となる．基本的には，プレディケートのベクタと CCR の内容が一致しているかどうかを検査することによって，プレディケートの評価を行うことができる．

ただし，次の比較結果はマスクする．

- プレディケートのベクタのある要素が X であるとき，その要素に関する比較結果
- CCR のエントリの値が「未定義」であるとき，そのエントリに関する比較結果

この「マスク付き一致検査」の結果，不一致ならば，プレディケートの値は偽である．マスク付き一致検査の結果が一致で，かつ，X でないプレディケートのベクタの要素に対応する CCR のすべてのエントリが定義されている場合，プレディケートの値は真である．そうでない場合，すなわち，マスク付き一致検査の結果が一致で，かつ，X でないプレディケートのベクタの要素に対応する CCR のエントリの少なくとも 1 つが未定義である場合，プレディケートの値は未定値である．

以上のように，プレディケートの評価は基本的にマスク付き一致検査で実現でき，単純である．

3.3 動 作 例

例を用いてマシンの動作を説明する．**図 5** は，スケジューリング前のコードである．説明のために，分岐の両方向にジャンプ命令を挿入している．命令スケジューラは，このコードよりスケジューリングの対象とするコード（リージョン：付録 A 参照）を選択し，スケジューリングを行う．同図において，ハッチングした部分をリージョンとし，2 命令発行のマシンに対してスケジューリングを行うとする．**図 6** にスケジュール・コード示す．このコードでは，データの依存関係が理解しやすいように説明上，書込みレジスタにも投機的レジスタ指定子 .s を付加している．実際の命令コードには 3.2.5 項で述べたようにエンコードされな

```
(1)   i1 :   alw      ?  r1 = load(r2);          i15:  c0&c1   ?  r2.s = r2 - 1;
(2)   i10:   !c0      ?  r5.s = load array;      i14:  c0&c1   ?  store(r7) = r5;
(3)   i2 :   alw      ?  r3 = r1 + 1;            i16:  c0&c1   ?  r7.s = r2.s << 1;
(4)   i6 :   c0       ?  r6 = load(r3);          i3 :  alw     ?  c0 = r3 < r4;
(5)   i11:   alw      ?  c2 = r2 < 0;            -  :  alw     ?  nop;
(6)   i7 :   alw      ?  c1 = r5 < r6;           i12:  !c0&c2  ?  jump  L6;
(7)   i9 :   c0&!c1   ?  jump L5;                i17:  c0&c1   ?  jump  L8;
(8)   i13:   !c0&!c2  ?  jump L7;                -  :  alw     ?  nop;
```

図6 スケジュール・コード (2命令発行)
Fig. 6 Scheduled code (two-issue).

表1 マシン状態の遷移
Table 1 Machine state transition.

サイクル	逐次的状態	投機的状態				CCR		
	書込み	書込み		コミット	無効化	c0	c1	c2
1		c0&c1	r2					
2	r1	c0&c1	sb1					
3	r3	!c0 c0&c1	r5 r7					
4						T		
5	r6				r5			F
6							T	
7				r2, r7, sb1				

い．命令のレイテンシは，ロード命令のみ2サイクル，その他は1サイクルである．

表1は，図6に示したスケジュール・コードを実行した際のマシン状態のサイクルごとの遷移を示している．逐次的状態，あるいは，投機的状態への書込み，コミット，無効化の各欄では，そのサイクルにおいて，書込み，コミット，無効化されたレジスタ（たとえば，r1）あるいはストア・バッファのエントリ（たとえば，sb1）を表している．特に，投機的状態への書込みの欄では，実行結果とともに書き込まれたプレディケートを示している．CCRの欄は，CCRの各エントリ（c0, c1, c2）が，更新された際の値を示している．このコードの実行が開始される前のCCRの初期値はすべて「未定義」である．

第1サイクルでは，i1とi15が実行される．i1のプレディケートはalwなので，実行は逐次的であり，r1の逐次的レジスタにロード・データが書き込まれる．ロード命令のレイテンシは2サイクルなので，第2サイクルにおいてレジスタr1の値が更新される．i15の実行は，これとは逆に，プレディケートc0&c1が参照するどちらの分岐条件も値がまだ定義されていないので投機的となる．レジスタr2の投機的状態を更新する．この際，プレディケートも同時に書込みエントリのプレディケート記憶部に書き込む．

第2サイクルでは，命令i10とi14が実行される．命令i10のプレディケート!c0の参照する分岐条件c0の値はまだ定義されていないので，実行は投機的であり，レジスタr5の投機的状態を更新する（ロード命令のレイテンシは2サイクルなので，第3サイクルに書込みが起こる）．ストア命令i14の実行も投機的なので，ストア・データはストア・バッファ（sb1）に書き込む．

第3サイクルでは，i2，i16が実行され，レジスタr3の逐次的状態とレジスタr7の投機的状態が更新される．

第4サイクルでは，命令i3によって分岐条件c0が真に定義される．これにより，次のサイクルで，r5の投機的状態のプレディケート!c0が偽と評価され，その結果，r5の投機的状態は無効化される．また，i6の実行は，実行開始時点では，そのプレディケートc0が未定値であるので投機的であるが，分岐条件c0がこのサイクルで真に定義されることによって，次の第5サイクルでは，実行は非投機的となる（ロード命令のレイテンシは2サイクルであることに注意）．したがって，実行結果は，レジスタr6の逐次的状態に書き込まれる．

第5サイクルでは，命令i11が実行され，分岐条件c2を偽に定義する．

第6サイクルでは，命令i7が実行され，分岐条件c1を真に定義する．これにより，次のサイクルで，r2，r7とsb1の投機的状態のプレディケートc0&c1が真と評価され，これらの値はコミットされる．また，ジャンプ命令i12は，プレディケートが偽と評価されるので実行されない．

第7サイクルでは，命令i17のプレディケートが真となり，ジャンプが実行され，次のリージョン（L8）に制御が移行する．

3.4 投機的例外処理

本節では，フューチャ・コンディション方式[8]と呼ぶ投機的例外処理方式を提案する．本方式による投機的例外処理は，次の3段階よりなる．

(1) 投機的例外処理の延期
(2) 再実行の準備
(3) 再実行

以下，順に説明を行う．

(1) 投機的例外処理の延期

投機的例外処理の延期を実現するために，レジスタ・ファイルおよびストア・バッファの各エントリに，書込みを行った命令が投機的例外を起こしたことを示すフラグを設ける．これは先に3.2.3，3.2.4項で述べたフラグEである．投機的命令が実行中に例外を起こした場合，実行結果の書込みの際にフラグEをセットする．

その後，プレディケートの値が真になった場合，投機的実行結果のコミットは行わず，投機的例外の検出信号を発生させる．

逆に，プレディケートが偽になったときは，そのエントリのフラグEをリセットする．したがって，先に起こった投機的例外は検出されることはなく，何のペナルティもなく無効化される．

(2) 再実行の準備

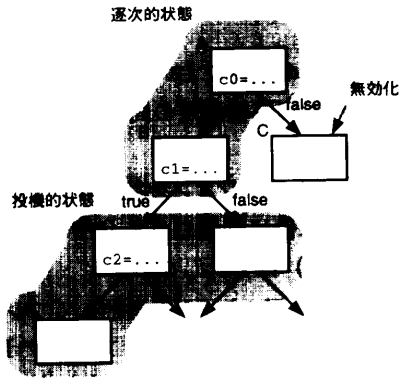
マシン状態再構築のための再実行の前に，準備を行う．以下，**図7**を用いて説明する．図7(a)に示すプログラムをスケジューリングした結果を図7(b)とする．図7(b)で，命令a～fはそれぞれ，図7(a)のプログラムの基本ブロックA～Fに属していた命令である．たとえば，命令aはブロックAに属していた命令である．

今，最初のサイクルでc0が真に定義され，その後命令dが実行中に例外を起こしたとする．命令dのプレディケートはc0&c1であり，c1はまだ定義されていないから，命令dが起こした例外は投機的例外である．その後，c1が真に定義されたとき，この例外は検出される．例外を検出した際に実行された命令の命令列における位置を，「投機的例外の検出点」と呼ぶ．

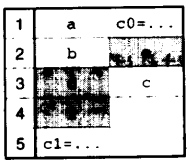
投機的例外が検出されたとき，命令a～fの実行結果は次の状態にある．

- 命令a，bの実行結果は逐次的状態にある．
- 命令cの実行結果は無効化されている．
- 命令d，e，fの実行結果は投機的状態にある☆．

☆ 投機的例外が検出された場合は，投機的状態はコミットされないことに注意．



(a)



投機的実行

(b)

図7 正確な割込み
Fig. 7 Precise interruption.

投機的例外が検出されたとき，まず，すべての投機的状態を破棄する．すなわち，命令d，e，fの実行結果を無効化する．これによって，**正確な割込み**[9]が実現される．

投機的状態をすべて破棄したので，再実行すべき命令は，例外の検出点で定義された分岐条件に依存するすべての投機的命令である．図7の例では，c1に依存する命令d，e，fである．これらの命令は，現在実行中のリージョンの先頭から投機的例外検出点の間に存在する．なぜならば，命令移動は，リージョン内部だけで行い，リージョンの外からも外へも移動しない(付録A参照)．

この範囲の中から再実行すべき命令を選択し，かつ，例外を発生した命令を特定するために，**カレント・コンディション**と**フューチャ・コンディション**という2つの分岐条件の値の組を記憶する．カレント・コンディションは，投機的例外検出の直前の分岐条件の値の組，フューチャ・コンディションは，投機的例外検出の直後の分岐条件の値の組である．図7の例では，カレント・コンディションは{1,U,U}（各要素はそれぞれc0，c1，c2に対応する．Uは「未定義」を表す）である．これは，制御が元のプログラム（図7(a)）において，ブロックBまで到達したことを意味している．一方，フューチャ・コンディションは{1,1,U}で

ある．これは，制御がさらにDに到達することを意味している．

投機的例外検出の直後において，現在のリージョンの先頭から投機的例外検出点の間にある命令は，次の2つに大別できる．

- カレント・コンディションを参照してプレディケートが真または偽に評価される命令（ブロックA，B，Cの命令）は実行しない．
- カレント・コンディションを参照してプレディケートの値が未定値である命令（ブロックD，E，Fの命令）は投機的実行を行う．もしも，実行中に例外を発生した場合，次の3つの場合がある．
  1. 処理しなければならない（ブロックD）
  2. 無効化しなければならない（ブロックE）
  3. 処理を延期しなければならない（ブロックF）

  フューチャ・コンディションを参照してプレディケートが真に評価される場合が，1に相当し，この場合のみ例外を処理する．

カレント・コンディションはCCRに保持させ，フューチャ・コンディションは，Future CCRと呼ぶレジスタを新たに設け，保持させる．Future CCRは，CCRと同じ構造である．

投機的例外を検出したとき，そのときに定義される分岐条件の値をCCRに書き込まないことによって，CCRにカレント・コンディションを保持させる．図7の例では，投機的例外が検出されたとき，分岐条件c1に対応するCCRのエントリの更新を行わない．一方，Future CCRは，投機的例外検出点で定義される分岐条件の値を反映した最新の分岐条件の値の組を保持させる．以上述べた投機的例外検出時におけるCCR，Future CCRの内容の設定は，ハードウェアで自動的に行う．

(3) 再実行

次に，再実行を行いマシン状態の再構築を行う．再実行の開始点はリージョンの先頭である．このアドレスを供給するために，通常の実行において，制御が現在実行中のリージョンを抜け，次のリージョンに移動するたびに，ハードウェアは自動的に飛び先の命令アドレスを**リージョン・プログラム・カウンタ**（RPC: Region Program Counter）と呼ぶ特別のレジスタに格納する．ハードウェアはRPCの値をPCにセットし，再実行を開始する．

再実行においては，先に述べたように，CCRに格納されたカレント・コンディションを参照して，プレディケートが未定値に評価される命令（図7のブロックD，E，Fの命令）だけを実行する．実行結果は投機的状態に書き込まれる．

再実行中に生じた例外を，Future CCRに格納されているフューチャ・コンディションを参照しプレディケートを評価することにより，処理すべきか，無効化すべきか，処理を延期すべきかを判断する．プレディケートの値が真であれば（図7のブロックDの命令），例外を処理する．偽であれば（ブロックEの命令），例外処理は行わず実行を終了する．このとき，書込みエントリのフラグEがセットされるが，後に，投機的例外検出点の分岐条件設定命令（c1=...）の実行によって無効化される．プレディケートの値が未定値であれば（ブロックFの命令），書込みエントリのフラグEをセットして実行を終了する．

以上述べた実行制御は，通常時のものとは異なる．通常の実行と区別するために，**リカバリ・モード**と呼ぶ実行モードを設ける．リカバリ・モードは再実行を開始したときに始まり，制御が投機的例外を検出した点に到達したとき（正確には，投機的例外を検出した命令の直前の命令の実行終了まで）終了し，通常の実行に戻る．

なお，投機的例外の検出点は，次のようにして認識する．

- 投機的例外の検出を，例外原因のひとつとする．したがって，投機的例外を検出した際，通常の他の例外の場合と同様に，例外発生アドレスを**例外プログラム・カウンタ**（EPC: Exception Program Counter）に保持する．すなわち，投機的例外が検出されたアドレスがEPCに保持される．
- リカバリ・モード時には，PCとEPCを比較する．一致したとき，制御が投機的例外の検出点に達したと認識する．

3.5 他の研究との比較

guarding[6]，ハイパブロック・スケジューリング[10]，*GIFT*[11]は，プレディケートを命令が持ち実行がそれにより制御される点で，プレディケーティングと同じである．しかし，これらの方式では，不要な実行をパイプライン内で無効化するために，実行終了前にプレディケートの値が真か偽に定まらなければならない．このため命令移動に制限が生じる．これに対して，プレディケーティングでは，プレディケートの値が定まらなくても，実行を開始し終了できる．したがって，これらの方式のような命令移動の制限はない．

boosting[12]は，レジスタ・ファイルとストア・バッファに投機的状態を記憶する構造を持つ点で，プレディケーティングと同じである．しかし，単一の制御パスでのみ投機的命令移動を可能にしただけであり，プレ

ディケーティングのように複数の制御パスで投機的命令移動を可能にしたわけではない.

non-excepting 命令[11),13)]による例外処理方式は，投機的例外が生じたことをレジスタ・ファイルに記録することで処理を延期する点で，プレディケーティングと同じである．しかし，マシン状態再構築の機構がないので，例外を起こす可能性のある命令に依存する命令は投機的移動できない.

4. 評価結果

表2 に示すベンチマーク・プログラムを用い，実行サイクル数によって性能を評価した．メモリ・システムは理想的なものと仮定している．基本マシンを，スカラ・マシンである MIPS R3000[14)] とし，R3000 に対する性能比で評価した.

以下の評価結果は，特に断らなければ，4 つの ALU，4 つの分岐ユニット，2 つのロード・ユニット，1 つのストア・ユニット，4 つの CCR のエントリを資源として持ち，4 命令を同時に実行できるマシンを仮定する．命令のレイテンシは R3000 と等しい．ただし，分岐のペナルティは分岐先バッファ（BTB）を用いて減少できると仮定しており，遅延分岐方式をとっていない．プレディケーティングでは多くの分岐命令が取り除かれるので，それにより通常の実行より分岐ペナルティは減少する．我々は，BTB により予測可能な分岐にペナルティはなく，そうでない分岐（たとえば，レジスタ間接ジャンプ）は，1 サイクルのペナルティが存在するとした．この仮定はやや楽観的ではあるが，BTB を実現した我々の cycle-by-cycle シミュレータで測定したところ，実際との誤差は数%以下と非常に小さく問題ないと考えられる.

本章では，まず初めに，投機的実行結果を一時的に格納する機構を持たないモデルについて評価する．その後，プレディケーティングを導入したアーキテクチャに対する評価を行う．我々の機構は追加のハードウェアが必要なので，そのハードウェア量についても評価する.

表2 ベンチマーク・プログラム
Table 2 Benchmark Program.

プログラム	行数	R3000 サイクル	機能
comp	1,557	21.3 M	データの圧縮
eqn	3,441	1,351.6 M	真理値表の生成
esp	13,511	1,119.9 M	PLA の最適化
grep	430	15.8 M	文字列の検索
li	7,429	1,245.5 M	Lisp 翻訳
nroff	7,276	56.0 M	文書の清書

4.1 投機的実行結果を一時的に格納する機構を持たないモデル

次の 4 つのモデルの評価を行った.

- グローバル・スケジューリング・モデル
- パイプライン無効化モデル
- トレース・スケジューリング・モデル
- リージョン・スケジューリング・モデル

図8 に評価結果を示す.

(1) グローバル・スケジューリング・モデル（GS モデル）

GS モデルでは，投機的実行への支援ハードウェアをまったく持たない．命令スケジューリング手法としては，ほぼパーコレーション・スケジューリング[15)]と同様の手法を用いた．すなわち，すべての基本ブロックに対して，隣り合う基本ブロック間での変換を性能の改善がなくなるまで繰り返した．パーコレーション・スケジューリングで行われているように，空のブロックの削除，制御が合流するブロックから上方への命令移動におけるブロック複写を行った.

投機的命令移動に関しては，2.1 節で説明したレジスタ・リネーミングを用いた．レジスタ・リネーミングにより生成されるコピー命令による性能低下を抑えるために，**コピー伝搬**[16)]，**無用コード削除**[16)]もあわせて行った．危険な投機的命令移動は，投機的実行のためのハードウェア支援がないので行うことはできない.

このモデルは，コンパイラだけによる投機的実行による性能向上を示す．投機的命令移動が大きく制限されているので，図 8 に示すように，スカラ・マシンに対する性能向上は幾何平均でわずかに 1.27 倍であった.

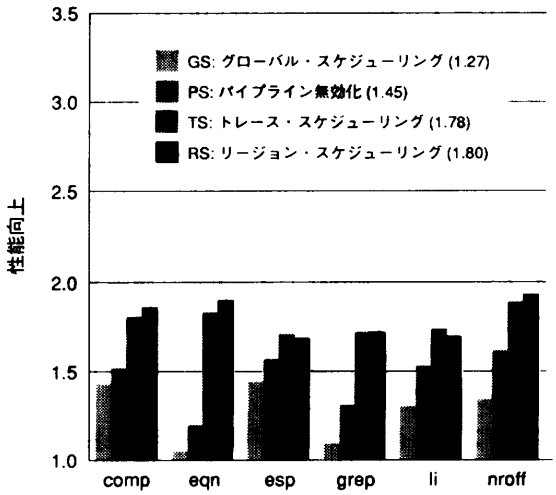


図8 投機的実効結果をバッファリングしないモデルでの性能向上
Fig. 8 Performance of models without buffering speculative execution results.

(2) パイプライン無効化モデル（PS モデル）

PS モデルでは，ハードウェアはパイプライン無効化を用いて投機的実行を支援する．命令スケジューリング手法は，GS モデルと同じである．ただし，危険な投機的命令移動に対しては，パイプライン無効化制御を用い，パイプライン内で無効化できる範囲に限定して移動が可能である．すなわち，投機的命令は実行を終了する前に，依存している分岐条件の値が定まり，実行が不要であると判明した場合，書込み前に無効化される．PS モデルは，投機的実行に対しては非常に小さな支援しか必要としないが，性能向上も小さく，スカラ・マシンに対する性能向上は 1.45 倍であった．

(3) トレース・スケジューリング・モデル（TS モデル）

TS モデルは，PS モデルのハードウェアにおいて，命令スケジューリング手法を変えたモデルである．TS モデルで用いた命令スケジューリング手法は，我々のスケジューリング手法（付録 A）において，リージョンをトレースに限定することにより実現した．この手法は，最も頻繁に実行されるパスを最適にスケジューリングできる点で，GS, PS モデルでのスケジューリング手法より優れている．図 8 に示すように，TS モデルの性能は，スケジューリング手法の改善によって PS モデルでの性能を上回り，スカラ・マシンの 1.78 倍となった．

(4) リージョン・スケジューリング・モデル（RS モデル）

RS モデルでは，命令にプレディケートを付加し，ハードウェアは実行時にそれを評価し，不要な投機的実行をパイプラインの中で無効化するハードウェアを持つ．我々のスケジューリング手法を用いた．命令移動はトレースに制限されていないので，分岐の予測が困難なプログラムに対して有利であると考えられる．このモデルでは，頻繁に実行される複数のパスを最適にスケジューリングできる点で，TS モデルより優れている．そのため TS モデルに対する性能向上が期待されたが，実際には，図 8 に示すように性能向上はほとんどなかった．これは，危険な投機的移動の範囲が，パイプライン内で無効化できる範囲に限定されているため，TS モデルに対して新たに加えられた命令スケジューリングの能力（複数のパスからの命令移動）が発揮されていないためと考えられる．

4.2 プレディケーティング機構を用いた投機的実行

本節では，我々の提案したプレディケーティング機構を採用した際の性能の評価結果を述べる．リージョン・プレディケーティングとトレース・プレディケーティングという 2 つの実現オプションをあげ，それらのハードウェア量について議論する．次に，それらを用いた場合の性能向上について議論する．最後に，資源の量と性能向上の関係についての評価結果を示す．

4.2.1 実現のオプション

(1) リージョン・プレディケーティング（RP モデル）

RP モデルは，これまで説明したプレディケーティングの機構を完全に実現したモデルである．このモデルでは，不正な投機的命令移動も危険な投機的命令移動も複数のパスに沿って自由に行うことができる．

プレディケーティング実現において最も複雑なのは，プレディケートの評価を行うハードウェアである．3.2.5 項で述べたように，プレディケートの評価の論理は，プレディケートと CCR の間の「マスク付き一致検査」である．これは，対応するエントリを比較する XOR ゲートと，マスクを行う OR ゲートと，最終的な一致を検出するための AND ゲートからなる．現在の我々のレジスタ・ファイルの実現においては，プレディケートの評価とそれによるフラグの更新を，半サイクルで行えばよく，サイクル時間に影響しない（付録 C 参照）．

プレディケート付きレジスタ・ファイルを実現するには，通常のレジスタ・ファイルに追加のトランジスタが必要である．8 つの読出しと 4 つの書込みポートを持つ 32 エントリのレジスタ・ファイルについて，トランジスタ数を調べた．従来の VLIW マシンのレジスタ・ファイルに必要なトランジスタ数は約 31 K であった．これに対して，4 つの CCR エントリを持つプレディケーティングのレジスタ・ファイルに必要なトランジスタ数は約 66 K であった．すなわち，プレディケーティングに必要なレジスタ・ファイルは，通常のレジスタ・ファイルの約倍のトランジスタを必要とする．このトランジスタ増加量は，最近のハイエンドのマイクロプロセッサを実現するために必要なトランジスタ（数 M トランジスタ）[17]のわずか数%であり，許容できると考えられる．

さらに，命令語に追加のビットが必要である．まず，プレディケートのエンコードにビットが必要である．プレディケートは真，偽，“don't care” の 3 値のいずれかをとるので，各値を表すには 2 ビットが必要である．したがって，CCR のエントリ数を K とすると，$2 \times K$ ビットが必要である☆．さらに，1 つのソース・

☆ 各分岐条件に対応して 3 値をとるので，プレディケートのエンコードに必要なビット数は，$\lceil \log_2 3^K \rceil$ ビットにすることができる．この場合，プレディケートの評価論理を単純にするために，実行時に $2 \times K$ ビットのコードに変換する複雑なハードウェアが必要である．

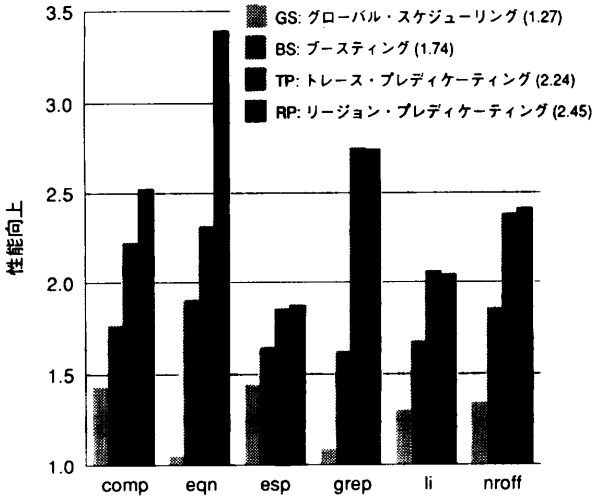


図9 プレディケーティングによる性能向上
Fig. 9 Performance with predicating.

レジスタの投機的レジスタ指定子に1ビット必要である．4.2.3項（図10）で述べるように，対コスト性能比を考えると，K は3〜4を必要とするので，命令のエンコードには，約1バイト追加する必要がある．

(2) トレース・プレディケーティング（TPモデル）

プレディケーティングの別の実現方法として，TPモデルを提案する．プレディケートのエンコードに必要なビット数を削減することが目的である．このモデルでは，命令移動をトレース内に制限する．ただし，トレース内で設定される分岐条件の値が真のときは，制御がトレースから外へ出ることはなく，トレース内で移行するようにコードを変換する．このように変換すれば，プレディケートは命令が依存する分岐条件の数でエンコードできるので，$\lceil \log_2 K \rceil$ ビット（K はCCRのエントリ数）でエンコードできる．エンコードされたプレディケートは，実行時にハードウェアによりRPモデルにおけるプレディケートのエンコード形式に変換され，その後の実行はRPモデルと同じである☆．

4.2.2 プレディケーティングによる性能向上

図9に，これまで述べたプレディケーティングの2つの実現モデルと，2つの既存のモデル（GSモデルとboostingモデル〔BSモデル〕）との性能比較結果を示す．

(1) BSモデル

BSモデルは文献12)に書かれたboostingの実現モデルとほぼ等しい．ハードウェアは投機的実行結果を一時的に格納する構造を持ち，トレースの中で自由に投機的命令移動を行うことができる．投機的実行結果は，依存している分岐の数を保持するカウンタをタグとして持つ．命令スケジューリングには，GSモデルでのスケジューリング手法を用い，予測されたパスに沿った投機的命令移動には制限がないとした．投機的実行結果は，依存している分岐の数の情報を持ってはいるが，どの分岐に依存しているかの情報を持っていないので，分岐命令は元のプログラムの順に実行されなければならないという制限がある．この制限のため，図9に示すようにスカラ・マシンに対する性能向上は1.74倍となり，性能を大きく改善することはできなかった．

表3 連続する分岐の予測精度
Table 3 Prediction accuracy of successive branches.

分岐数	1	2	3	4	5	6	7	8
comp	.88	.76	.66	.56	.46	.36	.27	.22
eqn	.87	.77	.68	.61	.56	.53	.50	.49
esp	.85	.72	.62	.54	.47	.41	.36	.33
grep	.97	.95	.93	.90	.88	.86	.85	.83
li	.88	.77	.68	.61	.55	.49	.43	.38
nroff	.98	.96	.94	.93	.91	.89	.88	.86

(2) TPモデル

TPモデルの性能向上は2.24倍であり，TSモデルやBSモデル等の既存のモデルの性能を大幅に上回った．これは，TPモデルがTSモデルに比べると，投機的命令移動の自由度において優れており，また，BSモデルに比べると分岐命令の削除によりパス長が削減されたためである．

(3) リージョン・プレディケーティング・モデル（RPモデル）

RPモデルは，TPモデルの性能をさらに上回り，2.45倍の性能向上を得た．しかし，TPモデルに対する性能向上はベンチマーク・プログラムで大きく異なる．この第1の原因は，静的分岐予測の精度の違いにある．TPモデルにおけるトレース内でのみ命令移動が可能という制限は，分岐予測精度がきわめて高い場合，性能にほとんど影響を与えない．**表3**に連続する複数の分岐の予測精度を表す[18]．同表に示すように，grepとnroffはきわめて予測精度が高く，その他のプログラムはきわめて高いわけではない．分岐予測精度がきわめて高いgrepとnroffに関しては，RPモデルは，TPモデルに対して性能向上がほとんどない．一方，分岐予測精度がきわめて高いわけではないeqnとcompに関しては，RPモデルは，TPモデルに対して，大きく性能を改善している．これに対して，esp

☆ 後で述べるBSモデルのように，制御依存情報をカウンタで記憶する方式では，どの分岐に依存するのか特定できないため，分岐条件設定の命令の順を入れ替えることができない．ベクタ形式ではこの制限がない．

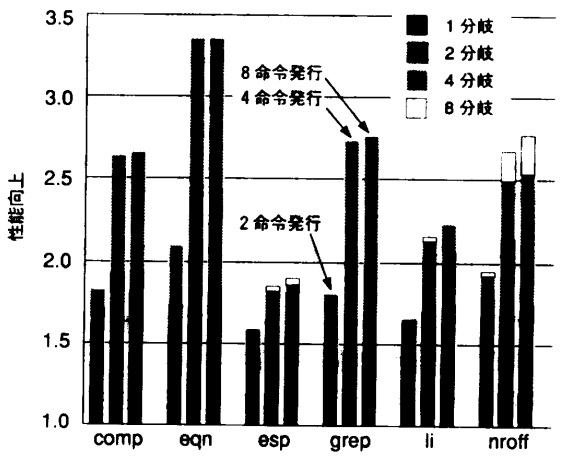


図10 資源の量と性能の関係
Fig. 10 Performance vs. resource amount.

とliに関しては，分岐予測精度がきわめて高いわけではないにもかかわらず，TPモデルに対する性能向上がほとんどない．コードを調べてみた結果，プレディケーティングを用いても，これらのプログラムには並列に実行可能な命令が少なく，さらに新たな機構が必要であると考えられる．

4.3 資源の量と性能向上の関係

図10に，資源の量と性能向上の関係を示す．マシンはこれまでの評価で仮定したマシンと異なり，機能ユニット，レジスタ・ポート，データ・キャッシュのポート等の資源を，同時命令発行数にあわせて完全に複写したマシンである．図10に示すように，2命令発行マシンでは，2つの分岐を越えた投機的命令移動で，十分な性能向上を得ることができる．4命令発行マシンの豊富な資源を有効利用し，ハードウェア・コストに見合った性能向上を得るには，4つの分岐を越えた投機的命令移動が必要である．一方，さらに8分岐までの投機的命令移動や，資源の8倍の複写を行っても，ほとんど性能向上はなかった．これらの豊富な資源を有効に利用するには，プログラムの並列性をさらに増加させる技術，たとえば，ループ・アンローリングや関数展開等が必要であると思われる．

5. まとめ

本論文では，プレディケーティングと呼ぶ投機的実行を支援するハードウェア機構を提案した．本提案のキー・アイデアは，投機的状態を制御依存情報であるプレディケートを付けて一時的に格納するハードウェアを用意し，そのプレディケートを参照したコミットと無効化の制御をハードウェアで行うことにある．本機構を用いれば，コンパイラは複数の制御パスから複数の基本ブロックを越えて命令を移動することが可能となり，その結果，最適に命令をスケジューリングすることができる．特に，複数のパスからの命令移動が可能なので，分岐予測が困難な応用に対しても，高い性能を得ることができる．

評価を行った結果，提案の機構を搭載すれば，従来の機構を搭載した場合に比べて大幅に性能を改善できることが確認された．投機的実行を行わないスカラ・マシンの2.45倍の性能向上を達成した．

謝辞 本研究に対してご支援いただいたシステムLSI開発研究所部長・角正氏に感謝いたします．また，貴重なコメントをいただいた査読者の方に感謝いたします．

参考文献

1) Wall, D.W.: Limits of Instruction-Level Parallelism, *Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.272–282 (1991).
2) Lam, M.S. and Wilson, R.P.: Limits of Control Flow on Parallelism, *Proc. 19th Int. Symp. on Computer Architecture*, pp.46–57 (1992).
3) Tomasulo, R.M.: An efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal*, Vol.11, No.1, pp.25–33 (1967).
4) Smith, J.E. and Pleszkun, A.R.: Implementation of Precise Interrupts in Pipelined Processors, *Proc. 12th Int. Symp. on Computer Architecture*, pp.36–44 (1985).
5) Murakami, K., Irie, N., Kuga, M. and Tomita, S.: SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture, *Proc. 16th Int. Symp. on Computer Architecture*, pp.78–85 (1989).
6) Hsu, P.Y.T. and Davidson, E.S.: Highly Concurrent Scalar Processing, *Proc. 13th Int. Symp. on Computer Architecture*, pp.386–395 (1986).
7) 安藤秀樹，中西知嘉子，原 哲也，中屋雅夫：プレディケート付き状態バッファリングによる投機的実行，並列処理シンポジウム *JSPP'95*, pp.107–114 (1995).
8) Ando, H., Nakanishi, C., Hara T. and Nakaya M.: Unconstrained Speculative Execution with Predicated State Buffering, *Proc. 22nd Int. Symp. on Computer Architecture*, pp.126–137 (1995).
9) Hennessy, J.L. and Patterson, D.A.: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, CA (1990).

10) Mahlke, S.A., Lin D.C., Chen, W.Y., Hank, R.E. and Bringmann, R.A.: Effective Compiler Support for Predicated Execution Using the Hyperblock, *Proc.MICRO-25*, pp.45–54 (1992).
11) 小松秀昭，古関 聰，鈴木秀俊，深澤良彰：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, pp.2599–2610 (1993).
12) Smith, M.D., Lam, M.S. and Horowitz, M.A.: Boosting Beyond Static Scheduling in a Superscalar Processor, *Proc. 17th Int. Symp. on Computer Architecture*, pp.344–355 (1990).
13) Colwel, R.P., Nix, R.P., O'Donnell, J.J., Papworth D.B. and Rodman, P.K.: A VLIW Architecture for a Trace Scheduling Compiler, *Proc. Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.180–192 (1987).
14) Kane, G.: *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ (1988).
15) Nicolau, A.: Percolation Scheduling: A Parallel Compilation Technique, Computer Sciences Technical Report 85-678, Cornel University (1985).
16) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Weslay Publishing Company, Reading, MA (1986).
17) 浅見直樹，枝 洋樹：次世代マイクロプロセサ，スーパスカラとVLIWが融合，日経エレクトロニクス，No.626, pp.67–150 (1995).
18) Ando, H., Nakanishi, C., Machida, H., Hara, T., Kishida, S. and Nakaya, M.: Speculative Execution and Reducing Branch Penalty in a Parallel Issue Machine, *Proc. Int. Conf. on Computer Design*, pp.106–113 (1993).
19) 安藤秀樹，中西知嘉子，原 哲也，中屋雅夫：非数値計算応用におけるプレディケート実行向け命令スケジューリング，並列処理シンポジウム *JSPP'96*, (1995). (掲載予定)
20) 中西知嘉子，安藤秀樹，原 哲也，中屋雅夫：パス選択によるソフトウェア・パイプライニング，情処研報，95-HPC-57, pp.127–132 (1995).
21) Hara T., Ando, H., Nakanishi, C. and Nakaya M.: Performance Comparison of ILP Machines with Cycle Time Evaluation, *Proc. 23nd Int. Symp. on Computer Architecture*, pp.213–224 (1996).

付録A 命令スケジューリング手法

簡単に，命令スケジューリング手法について述べる．詳しくは，文献19)を参照してほしい．

プレディケーティング向けの我々の命令スケジューリング手法は，ハイパブロック・スケジューリング[10]を基本としている．次の手順を繰り返しスケジューリングを行う．

1. リージョンの選択
2. 末尾複写
3. スケジューリング

リージョンとは，スケジューリング対象とするプログラムの一部であり，実行頻度の高い基本ブロックをプロファイルを用いて選択する．基本的には，リージョンのヘッダとなるブロックを1つ選択し，実行確率の高い後続ブロックに向かって，リージョン内に含まれる分岐の数がCCRのエントリ数に達するまでリージョンを拡張することにより選択を行う．

次に，リージョン・ヘッダ以外にリージョン外から入るエッジがある場合，そのエッジが入るノード以降でリージョン内部にあるブロックをすべて複写する．これを末尾複写と呼ぶ．これにより，制御の合流点が除かれるので，命令の上方移動にともなう複写は必要ない．

次に，リージョン内の命令にプレディケートを付加する．これにより，リージョンより外に出るための制御移行命令を除いて，リージョン内部での制御移行命令は不要になるので削除する．この後，リージョン内の命令をスケジューリングする．プレディケーティングの機構によって，コンパイラには投機的命令移動に関して制限がない．

スケジューリングは，制約グラフを作成しリスト・スケジューリングによって行う．スケジューリングに用いる優先度は，制約グラフにおけるノードの高さ(リーフからノードへの最大パス長)とリージョンのヘッダから命令の属する基本ブロックへ制御が到達する確率との積[10),11)]とした．

命令移動はリージョン内部だけで行い，リージョンの外からも外へも命令は移動しない．また，リージョン内で参照される分岐条件の値は，すべてリージョン内で定義する．したがって，リージョンをスケジューリングした後のコードにおいて，その先頭(スケジュール・コードの最初の命令が実行される直前)では，リージョン内の命令が参照する分岐条件はすべて未定義である．リージョンへは，末尾複写によりリージョン・ヘッダからしか制御は入らない．また，リージョンを出るときのみジャンプ命令は実行されるから，ジャンプ命令が実行されたとき，CCRの全エントリを「未定義」にリセットする．これにより，リージョン・ヘッダで分岐条件の値をすべて「未定義」にすることができる．

この手法では，スケジューリングに先立って，複数の基本ブロックにまたがる制約グラフを作成するので，スケジューリングの優先順位を広い範囲で決定できる．このため，パーコレーション・スケジューリング[15]のような隣り合うブロック間で最適化を繰り返すことによりプログラム全体を最適化する手法に比べて，よりよいスケジュール・コードが得られる．また，あらかじめ実行頻度の低い基本ブロックをスケジューリング対象から除くので，実行頻度の低い基本ブロックの命令によって資源が占有され，実行頻度の高い命令のスケジューリングが妨げられることを防ぐことができる．

さらにこのほかにも，たとえば，ループに対するソフトウェア・パイプライニング[20]や，先行ブロックを複数持つブロックに対するノード分割[10]などの最適化を行っている．

付録B　オペランド・フォワーディング

オペランド・フォワーディング（あるいは，**バイパシング**）とは，パイプラインでの書込み前に，実行結果が得られた時点で，実行を開始する命令に実行結果を転送することをいう．フォワーディングを行えば，命令のレイテンシを小さくできるので，高速化には必須の技術である．3.2.3，3.2.4 項で述べたように，プレディケーティングでは，1 つのレジスタ，1 つのメモリ・ロケーションに対して，投機的状態を 1 つしか許さないので，フォワーディングは従来のマシンと同様，簡単な論理で行うことができる．具体的には，すでに実行結果を得ている命令の書込みレジスタ番号と，レジスタ・フェッチを行う命令の読出しレジスタ番号の比較のほかに，転送すべき値が投機的かどうかを検査する．すなわち，次のいずれかの場合を検出する．

- 実行結果が逐次的で，かつ，読出しレジスタの投機的レジスタ指定子がセットされていない．
- 実行結果が投機的で，かつ，読出しレジスタの投機的レジスタ指定子がセットされている．

比較すべきビット数は，従来のレジスタ比較のビット数（32 エントリのレジスタ・ファイルならば 5 ビット）に，さらに上記条件をチェックする 1 ビットが追加されるだけである．

ストア・バッファ内のデータのフォワーディングに関しても，同様の議論が成り立つ．ストア・バッファからのフォワーディングでは，通常のマシンで必要なアドレス比較以外に，フォワーディングすべき値が投機的なのかどうかを検査するだけでよい．

付録C　レジスタ・ファイルの動作速度

図 11 にレジスタ・ファイルの 1 エントリにおける制御論理を示す．2 つの値を記憶するデータ・セル A，B のどちらかを選択する選択器（**レジスタ選択器**）が，アドレス・デコーダの後方に挿入されている[21]．また，制御論理は，プレディケートを記憶するレジスタ，フラグ W，V，E を記憶するフリップ・フロップ（FF），さらに，プレディケートを評価する論理を持つ．フラグ W，V，E を記憶する FF は，プレディケートの評価結果にしたがって更新される（フラグ W の反転，フラグ V，E のリセット）．

一般に，レジスタ・ファイルは 1 サイクルの間に書込みと読出しを行わなければならない．プレディケーティングのレジスタ・ファイルでは，プレディケートの書込みの後，それを評価しフラグを更新する必要がある．したがって，レジスタ・ファイルのクリティカル・パスは，「プレディケートの書込み，プレディケートの評価，フラグの更新，レジスタの選択，データ・セルの読出し」のパスである．このパスの信号遅延は，1 サイクルの時間内でなければならない．

我々のレジスタ・ファイルの設計では，書込み時間を短くするために，書込みを行う前のサイクルで，書込みレジスタ番号のデコードと書込みワード線の選択を，読出しと並行して行っておく．これによって，クリティカル・パスに含まれるプレディケートの書込み時間は，書込みビット線の駆動時間だけとなる（したがって非常に短い）．

通常，レジスタ・ファイルの読出しには，半サイクル近くかかるので，このクリティカル・パスの遅延時間はおおまかに見積って，「プレディケートの書込み，

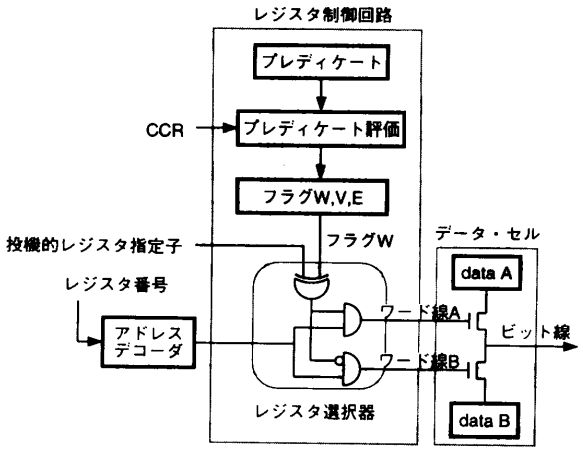


図 11　レジスタ・ファイルの制御回路
Fig. 11　Control logic in register file.

プレディケートの評価，フラグの更新」（パス1）が半サイクル程度，「レジスタの選択，データ・セルの読出し」（パス2）が半サイクル程度であればよいと考えられる．実際に測定した結果[21]，ALUの実行結果を後続の命令にフォワードする時間をサイクル時間の下限としたとき，パス1，2の遅延時間は，それぞれ，サイクル時間の下限の30%，42%であった．

(平成7年8月31日受付)
(平成8年9月12日採録)

安藤 秀樹（正会員）

1959生年．1981年大阪大学工学部電子工学科卒業．1983年同大学大学院修士課程修了．同年三菱電機(株) LSI研究所に入社し，ISDN用ディジタル信号処理LSI，第5世代コンピュータ・プロジェクトの推論マシン用プロセッサの設計に従事．1991年Stanford大学に客員研究員として留学．現在，計算機アーキテクチャ，コンパイラの研究に従事．

中西知嘉子

1988年大阪大学基礎工学部情報工学科卒業．同年三菱電機（株）LSI研究所に入社．以来，計算機用LSIの開発に従事．現在，同社システムLSI開発研究所所属．

原 哲也（正会員）

1966年生．1989年九州大学工学部情報工学科卒業．1991年同大学大学院総合理工学研究科情報システム学専攻修士課程修了．同年三菱電機（株）LSI研究所に入社し，細粒度並列処理アーキテクチャの研究に従事．1996年より信号処理プロセッサの開発に従事．

中屋 雅夫（正会員）

1951年生．1974年早稲田大学理工学部電子通信学科卒業．1976年同大学大学院修士課程修了．1988年工学博士（早大）．1976年三菱電機(株) 入社．以来，高速MOSゲートアレイ，ECLゲートアレイ，MOSA/D，D/Aコンバータ，三次元回路素子，通信用LSI，ISDN用LSIの研究開発を経て，現在，並列処理プロセッサ，ATM-LAN用LSI等のシステムLSIの研究開発に従事．1993年度電子情報通信学会論文賞受賞．現在，三菱電気(株) システムLSI開発研究所設計技術第二部第1Gグループマネジャー．IEEE，電子情報通信学会各会員．