計算機アーキテクチャ 119－12
（1996．8．27）

拡張VLIWプロセッサGIFTにおける投機的実行支援機構

作家 昭夫† 古関 聰† 小松 秀昭‡ 深澤 良彰†
†早稲田大学理工学部 ‡日本IBM(株) 東京基礎研究所

計算機の高速化の重要な技術に、命令レベルでの並列処理が挙げられる。VLIWは主に数値計算を対象として研究開発されてきたアーキテクチャであるが、最近では、スーパースカラの次の世代を担う汎用プロセッサとして注目されている。汎用プロセッサとしてのVLIWの性能を引き出すためには、命令の投機的移動を主体とした大域的コードスケジュリング技法が重要である。しかしながら、命令の投機的移動は単純なVLIWでは制限があるため、ハードウェアによる支援が不可欠である。本論文では、発生した例外を表すコードと例外が発生した命令の命令コードをレジスタに書き込むことによる、新しい投機的移動支援機構を提案する。本方式によって、従来の方式に比べて1.3倍の速度向上が得られることがわかった。

Mechanisms for Speculative Execution in the Extended VLIW Processor GIFT

Akio Sakka† Akira Koseki† Hideaki Komatsu‡ Yoshiaki Fukazawa†
†School of Science & Engineering, Waseda University
‡Tokyo Research Laboratory, IBM Japan, Ltd.

Instruction-level parallel processing is one of the most important techniques to achieve a good performance on computers. VLIW has been developed and researched for numerical programs, but it is nowadays focused as the main general purpose processor following superscalar processors. To achieve a good performance on VLIW as a general processor, code scheduling mainly consisting of speculative moves of instructions, is important. However, using speculative moves is limited on a simple VLIW, so some hardware support is needed. In this paper, we propose new mechanisms, in which the code of the exception that a certain instruction caused and the code of the instruction are put into a register. Using our method, 1.3 times speedup is achieved in comparison with existing methods.

1 はじめに

アプリケーションの大規模化にともない、計算機の高速化が求められている。高速化を実現する手段の一つとして、並列処理の研究が、分散処理レベルから命令レベルまで、さまざまな粒度において行なわれている。その中で我々は、命令レベル並列処理アーキテクチャであるVLIWに注目し、研究の対象としている。

VLIWは、主に数値計算を対象として研究開発されてきた経緯があるが、最近では、スーパースカラの次の世代を担う汎用プロセッサとして注目されている。汎用プロセッサとしてのVLIWを考えた場合の問題点は、非数値計算をどのように並列処理するかにある。一般に、非数値計算においては、プログラムの中に存在する並列性は2～3と言われており[1]、この問題を解決するために様々な命令スケジューリング法[2, 3, 4, 5, 6, 7]が提案されている。このようなスケジューリングによる主な並列化技術は、「命令の投機的移動」である。命令の投機的移動は、単純なVLIWでは制限があるため、ハードウェアによる支援が不可欠である。

本論文では、我々が提案してきた拡張VLIWプロセッサGIFT[8]における投機的移動支援方式として、いわゆるノンエクセプション方式[9]による投機的移動支援機構の改良を提案する。ノンエクセプション方式では、投機的に移動された命令が生成するデータを参照するような命令の投機的移動が行なえなかった。本方式では、発生した例外を表すコードと例外が発生した命令の命令コードをレジスタに書き込むことによって、これを解決する。

2 本研究の背景

VLIWの性能を引き出すためには、コードスケジューリングによるプログラムの並列化が重要である。特に

プログラムのベーシックブロック間で命令の移動を行なう大域的コードスケジューリングとして、トレーススケジューリング[2]、パーコレーションスケジューリング[3]をはじめとした様々な方法[4, 5, 6, 7]が提案されている。このような大域的コードスケジューリングの主要な技術に、命令の投機的な移動がある。

一般に、命令の投機的実行とは、本来は条件分岐が行なわれた後に実行される命令を、実行条件が決定される前に実行してしまうことであり、プログラムに存在する制御依存を緩和するための有効な方法である。大域的コードスケジューラは、コントロールフローグラフにおいて、分岐先のブロックから分岐元のブロックに命令を移動することで命令の投機的実行を実現する。これを投機的移動と呼ぶ。この投機的移動を行なうことにより、並列実行可能な命令の数を2～8倍に増やすことが可能なことが報告されている[7]。

投機的移動は並列性の抽出に大きな効果があるが、投機的実行のためのハードウェア支援がない場合は、移動可能な命令が非常に制限されてしまう。例えば、文献[10]では、投機的移動は図1のように移動の正当さと安全さによって分類されている。

投機的移動の正当さとは、移動された命令がプログラムの意味を変えないことをいう。プログラムの意味を変える命令の移動は不当な移動と呼ばれる。不当な移動が行なわれると、移動された命令の後に実行される命令が読み込むべきデータが失われてしまう。また、投機的移動の安全さとは、移動された命令が例外を起こさないことをいう。例外を起こす可能性のある命令の移動は危険な移動と呼ばれる。危険な移動が行なわれると、本来起こるはずのないプログラム停止が起こったり、本来起こるはずのない例外の復旧処理によって、プログラムの実行が大きく遅延される可能性がある。図1の中で、スケジューラが単純に投機的移動を行なうことができるのは、安全かつ正当な移動だけである。

命令の投機的移動の制限を緩和するためには、なんらかのハードウェア支援が必要である。我々は、投機的実行を支援するハードウェア機構を以下の二つに分類している。

- データ駆動方式
- コントロール駆動方式

前者は、簡単に述べると、ノンエクセプション方式[9]やGIFTにおける先行実行機構[8]にみられるように、投機的に移動されていない命令（以下、非投機的命令）にデータが参照されるまで、投機的に移動された命令（以下、投機的命令）が発生させた例外による停止または復旧処理（以下、例外処理）を延期する方法である。また、後者は、簡単に述べると、TORCHにおけるブースティング[11]やSPEVにおけるプレディケーティング[12]にみられるように、投機的に移動された命令が本来実行されるべき実行条件が決定したときに例外処理を行なう方法である。各方式について、以下に詳しく説明する。

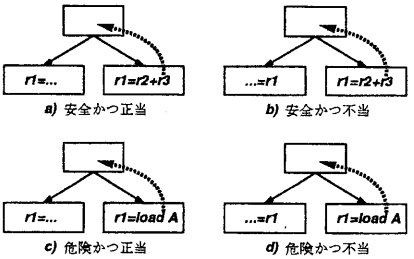

図1: 投機的移動の分類

データ駆動方式

データ駆動方式による支援機構は、危険な投機的移動に対する支援を提供する。危険な投機的移動の問題点は、投機的命令が、本来起こるはずのないプログラム停止や例外の復旧処理を発生させてしまうことである。これを解決するため、データ駆動方式では、投機的命令が例外を起こした場合、その命令のロード、ストアあるいは演算結果の書き込み先に、例外が発生したことを示すマークと、例外処理を再開するための情報（投機的命令のアドレス、または例外コード等）を書き込み、例外処理の延期を行なう。次に、その後に実行された非投機的命令のロード、ストアあるいは演算の読み込み元に例外発生のマークがついていた場合、実際の例外処理が行なわれる。

コントロール駆動方式

コントロール駆動方式による支援機構は、危険及び不当な投機的移動に対する支援を提供する。不当な投機的移動の問題点は、移動された命令の後に実行される命令が読み込むべきデータが失われてしまうことである。これを解決するため、コントロール駆動方式では、投機的命令の命令のロード、ストアあるいは演算結果を通常のメモリやレジスタではなく、シャドウレジスタファイルやシャドウストアバッファ(総称してシャドウリソース)に書き込む。さらに、各投機的命令を、その命令が本来実行されるべき分岐条件を保持

させておくようにし、マシンは、実行された投機的命令が保持していた実行条件を記憶しておく。実際の分岐が行なわれたときに、マシンに記憶された実行条件と分岐結果を比較し、シャドウリソースのデータのコミット（通常のメモリまたはレジスタにコピーすること）または無効化が行なわれる。

また、危険な投機的移動の問題点は、シャドウリソースに例外が発生したことを示すマークと、例外処理を再開するための情報（投機的命令のアドレス、または例外コード等）を書き込み、例外処理の延期を行なうことで解決する。例外処理は、マシンに記憶された投機的実行の本来の実行条件と分岐結果を比較することで行なわれる。

各方式には、それぞれ、長所と欠点が存在する。次章において各方式を比較し、我々の方針について述べる。

3 支援方式の比較と我々の基本方針

本論文では、データ駆動方式の長所を活かし、欠点を回避した投機的移動支援機構を提案する。

データ駆動方式には、コントロール駆動方式に比べ以下の長所がある。

- シャドウリソースを必要としないのでチップ面積を圧迫しない。
- 投機的移動によって越える分岐の数に制限がない。
- 投機的移動をよって越える分岐の方向に制限がない。

また、データ駆動方式には、コントロール駆動方式に比べ以下の欠点がある。

- 投機的命令の結果を参照する命令の投機的移動ができない。
- 不当な投機的移動の支援がない。

コントロール駆動方式に関する最近の研究成果では、投機的移動によって越える分岐の数や方向の制限の緩和が行なわれている [10, 12]。しかしながら、シャドウリソースが必要とするコストは依然として大きな問題となっている。従って、我々は、シャドウリソースを必要としないデータ駆動方式を改良し、同時にこの方式の欠点を回避するアプローチを採る。具体的なアプローチを以下に示す。

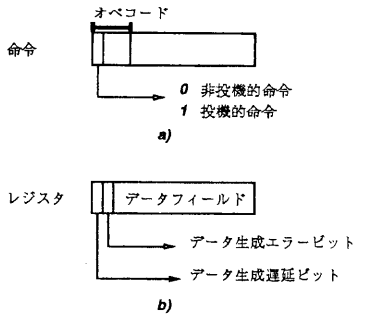

図 2: 追加ビットフィールド

- 投機的命令の結果を参照する命令の投機的移動を支援する機構を導入する。
- 不当な投機的移動は、レジスタリネーミングを主体としたコンパイル技術で回避する。

以下、4章において、この支援機構を実現する改良型先行実行機構について述べる。また、コンパイル技術については誌面の都合上省略する。

4 GIFT における改良型先行実行機構

本機構を実現するため、GIFT では投機的命令と非投機的命令の二種類を用意し、これらをオペランドに設けられたビットフィールドで区別する（図 2a)）。投機的命令が例外を起こした場合、その時点では例外処理を行なわないで、非投機的命令がその結果を参照するまで例外処理を遅延させる。この処理を実装するため、我々は、例外を以下の二つに分類し、それぞれ異なった例外の遅延と再開を行なう。

- データ生成エラー（0 除算、アドレス保護違反、浮動小数点例外）
- データ生成遅延（ページフォールト）

データ生成エラーとは、プログラムを停止させるような例外である。また、データ生成遅延は、データを生成するのに非常に時間がかかるような例外である。それぞれの例外についての処理を以下に説明する。

4.1 例外処理の遅延と例外の検出

例外処理の遅延と例外の検出は、各レジスタに図 2b) のようなビットフィールドを設けた上で、各命令を以下のように実行することで実現する。

・投機的命令の場合
　・その命令のソースレジスタにデータ生成エラー

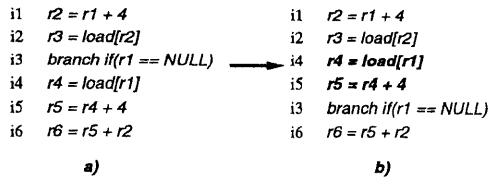

図 3: 投機的移動の例

ビットが立っていた場合

ディスティネーションレジスタにエラーコードを書き込み、データ生成エラービットを立てる。

・データ生成遅延ビットが立っていた場合

ディスティネーションレジスタにインストラクションコードを書き込み、データ生成遅延ビットを立てる。

・どちらのビットも立っていなかった場合

・データ生成エラーを起こした場合

ディスティネーションレジスタにエラーコードを書き込み、データ生成エラービットを立てる。

・データ生成遅延を起こした場合

ディスティネーションレジスタにインストラクションコードを書き込み、データ生成遅延ビットを立てる。

・何も起こさなかった場合

通常の処理を行なう。

・非投機的命令の場合

・その命令のソースレジスタにデータ生成エラービットが立っていた場合

データ生成エラーのシグナル生成

・データ生成遅延ビットが立っていた場合

データ生成遅延のシグナル生成

・どちらのビットも立っていなかった場合

・データ生成エラーを起こした場合

データ生成エラーのシグナル生成

・データ生成遅延を起こした場合

データ生成遅延のシグナル生成

・何も起こさなかった場合

通常の処理を行なう。

図 3を用い、例外処理遅延の例を示す。図 3a) に投機的移動を行なう前のコード、図 3b) に投機的移動を行なった後のコードを示す。太字で書かれた命令が投機的命令である。今、あるサイクルにおいて命令 i4 が実行され、ページフォールトを起こしたとする。この命令は、投機的命令であるので例外処理は遅延される。また、$r4$ のデータ生成遅延ビットが立てられ、$r4$ のデータフィールドには $r4 = load[r1]$ を表すインストラクションコードが書き込まれる。次のサイクルでは、命令 i5 が実行される。i5 は投機的命令であり、ソースレジスタである $r4$ にはデータ生成遅延ビットが立っているので、$r5$ のデータ生成遅延ビットが立てられ、$r5$ のデータフィールドには $r5 = r4 + 4$ を表すインストラクションコードが書き込まれる。このように、命令 i4 が起こしたページフォールトは、$r4$ または $r5$ が非投機的命令によって読み出されるまで遅延される。

4.2 例外処理の再開

例外処理の再開は、非投機的命令が、データ生成エラーあるいはデータ生成遅延ビットが立ったレジスタを読み込んだ場合に行なわれる。データ生成エラービットが立っていた場合は、レジスタにエラーコードが格納されているので、それに従って処理を行なう。データ生成遅延ビットが立っていた場合は、以下のようなアルゴリズムと等価な処理を行なうことでデータを復元できる。

- レジスタに格納されたインストラクションコードを読み込む。
- そのインストラクションコードにおけるソースレジスタを読み込む。
- ソースレジスタにデータ生成エラービットが立っていた場合、そのレジスタに格納されているエラーコードに従って処理を行なう（プログラムは停止する）。
- ソースレジスタにデータ生成遅延ビットが立っている場合は、そのレジスタに格納されているインストラクションについて再帰的に復旧処理を行なう。
- ソースレジスタにどちらのビットも立っていなかった場合は、レジスタの値を読み込み、インストラクションの実行を行なう。

図 3を用い、例外処理再開の例を示す。前節で説明したように処理が行なわれた後、命令 i6 が実行されたとする。この命令は、データ生成遅延ビットが立てられ

たレジスタを読み込んでおり、かつ、非投機的命令であるので例外処理が再開される。まず、$r6 = r5 + r2$ を計算するために、$r5$ の値が必要であり、$r5 = r4 + 5$ を計算するために、$r4$ の値が必要であることがわかるので、$r4$ のデータフィールドに格納されている $r4 = load[r1]$ を実行し、ページフォールトの復旧を行なう。$r4$ の値が生成された後は、$r5 = r4 + 5$、$r6 = r5 + r2$ が順に行なわれ、マシンの状態を元に戻す。

4.3 コンパイラサポート

本方式において例外処理の再開を正しく行なうためには、コンパイラによるサポートが必要である。

本方式では、投機的命令が読み込むレジスタは、投機的命令が移動してきた方向に分岐が行なわれ、非投機的命令が投機的命令の結果を参照するか、それとは反対の方向に分岐が行なわれるまで更新してはならない。例えば、図3では、i4、i5 が読み込む、$r1$、$r4$ は、i3 の分岐が成立するか、i6 が実行されるまで更新してはならない。

5 評価

本章では、各種アーキテクチャにおける実行性能向上の比較をおこなう。表1に、各評価モデルを示す。表1において、局所的コードスケジューリングはリストスケジューリング [13]、大域的スケジューリングは文献 [7] で提案された方式、また、データ駆動方式投機的実行支援機構は2章において述べたこれまでの方式、改良型データ駆動方式投機的実行支援機構は4章において述べた改良型先行実行機構を意味するものとする。

表1で示した各モデルについて並列度を 1,2,4,6 と変化させ、リストの最大値を求めるプログラム、及び、リストを選択ソートするプログラムについて、スカラプロセッサでの実行速度を1としたときの相対実行速度を測定した。なお、ここでいう並列度とは、同時発行可能な命令の数を表すものとし、算術演算U、ロード/ストア、ジャンプユニットを「並列度」個づつ持つものとした。

表2、3より、局所的コードスケジューリングでは、マシンの並列度を高くしても実行速度が向上していないことがわかる。これに比べ、モデル2では、並列度が高くなるにつれ実行速度は向上しており、また、モデル1と比べて若干の速度向上が見られる。これは、大域的コードスケジューリングによる、命令の投機的移動の効果によるものである。しかしながら、2章で述べた通り、この移動は非常に制限されており、危険な投機的移動を行なうことはできない。モデル3では、この制限はある程度緩和されている。これにより、並列度が6の場合に、2.2倍の速度向上が得られている。本方式 (モデル4) では、改良型先行実行機構により、投機的移動の制限を非常に少なくすることに成功している。この方式を採用することで、並列度が6の場合に、2.9倍の速度向上が達成されており、本方式がこのようなプログラムの並列実行に非常に有効であることがわかる。

表 1: 評価対象

モデル番号	評価条件
1	VLIW+ 局所的コードスケジューリング
2	VLIW+ 大域的コードスケジューリング
3	VLIW+ データ駆動方式投機的実行支援機構 + 大域的コードスケジューリング
4	VLIW + 改良型データ駆動方式投機的実行支援機構 + 大域的コードスケジューリング

表 2: Max of a List の相対実行速度比較

モデル番号	並列度			
	1	2	4	6
1	1.000	1.000	1.000	1.000
2	1.308	1.700	1.700	1.700
3	1.308	2.125	2.125	2.125
4	1.308	2.429	2.833	2.833

6 関連研究との比較

ノンエクセプション方式の改良として、センチネルスケジューリング [14] が挙げられる。この方式では、例外を起こした投機的命令のアドレス (プログラムカウンタの値) をディスティネーションレジスタに格納し、各レジスタに附属している例外発生ビットを立てる。投機的命令が、以前に実行された例外を起こした投機的命令の実行結果を参照した場合、つまり、例外ビットが立っているレジスタを読み込んだ場合、レジスタに格納されているアドレスをディスティネーションレジスタに書き込む。非投機的命令が、例外ビットが立っているレジスタを読み込んだ場合は、レジスタに格納されているアドレスから実行を再開し、例外処理を再開する。

この方式は、本方式に比べて例外処理の再開が単純であるが、投機的に移動された命令の後の全ての命令

表 3: Sort of a List の相対実行速度比較

モデル番号	並列度			
	1	2	4	6
1	1.000	1.222	1.222	1.222
2	1.222	1.833	2.200	2.200
3	1.222	2.200	2.750	2.750
4	1.222	2.200	2.200	3.667

を再実行しなければならない。そのため、それらの命令の全てのオペランドを保持しなければならないという問題点がある。本方式は、再実行をしなければならない命令を特定できるので、この方式に比べて最適化コンパイラに与える制約が少ない。

7 おわりに

本論文では、単純なハードウェアを用いた投機的実行支援機構を提案した。これによって、従来のデータ駆動式の支援機構に比べて、より多くの命令の投機的移動の制限が緩和され、プログラムの実行速度が向上することがわかった。今後、VLIW の命令供給に関してスケーラビリティを持たせるハードウェア機構とコンパイラ技術について研究を進める予定である。

参考文献

[1] Nicolau, A. and Fisher, J. A.: A Measuring the Parallelism Available for Very Long Instruction Word Architectures, *IEEE Trans. Comput.*, Vol. C-33, No. 11, pp. 968-976 (1984).

[2] Fisher, J.A.: Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. Comput.*, Vol.C-30, No. 7, pp.478-490, (1981).

[3] Aiken, A. and Nicolau, A.: A Development Environment for Horizontal Microcode, *IEEE Trans. Softw. Eng.*, Vol. 14, No. 5, pp. 584-594 (1988).

[4] Ebicioğlu, K. and Nicolau, A.: A Global Resource-Constrained Parallelization Technique, *Proceedings of the 3rd International Conference on Supercomputing*, pp.154-163 (1989).

[5] Hwu. W. W., et al.: The Superblock: An Effective Technique for VLIW and Superscalar compilation, *Journal of Supercomputing*, Vol. 7, No. 1, pp.229-248 (1993)

[6] 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, *Proceedings of the 25th Annual International Symposium on Micro Architecture*, pp.45-54 (1992)

[7] 小松, 古関, 深澤: 命令レベル並列アーキテクチャのための大域的コードスケジューリング技法, 情報処理学会論文誌, Vol. 37, No. 6, pp. 1149-1161 (1996).

[8] 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, pp2599-2610 (1993).

[9] Colwell, R. P., Nix, R. P., O'Donnel, J. J., Papworth, D. B. and Rodman, P. K.: A VLIW architecture for a Trace Scheduling Compiler, *IEEE Trans. Comput.*, Vol. C-37, No. 8, pp. 967-979 (1988).

[10] Smith, M. D., Horowitz, M. A. and Lam, M. S.: Efficient Superscalar Performance Through Boosting, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.248-259 (1992).

[11] Smith, M. D., Lam, M. S. and Horowitz, M. A.: Boosting Beyond Static Scheduring in a Superscalar Processor, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp.344-354 (1987).

[12] 安藤, 中西, 原, 中屋: 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 予稿集, pp. 107-114 (1995).

[13] Coffman, E. G., Jr.: *Computer and Job-Shop Scheduling Theory*, John Willy & Sons (1976).

[14] Mahlke, S. A., Chen, W. Y., Hwu, W. W., Rau, B. R. and Schlansker, M. S.: Sentinel Scheduling for VLIW and Superscalar Processors, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.238-247 (1992).