特別研究報告書

ロード ・ストア命令の分離と投機の評価

指導教官　　富田 眞治 教授

京都大学工学部情報学科

小出 義和

平成 14 年 2 月 3 日

ロード ・ストア命令の分離と投機の評価

小出 義和

内容梗概

スーパスケーラ ・プロセッサの性能向上を妨げる要因に、メモリアクセス命令における参照アドレスの曖昧性が挙げられる。命令スケジューリングは命令間のデータ依存関係を追跡しこれを満たすように行われる必要がある。データ依存関係には二つのタイプが存在し、一つはレジスタを介した依存関係、もう一つはメモリを介した依存関係である。

このうちレジスタを介した依存関係は、デコード時に命令内のオペランドフィールドを評価することで検出することが可能である。一方、メモリを介した依存関係は、同一アドレスを参照するメモリアクセス命令間の依存となるが、参照アドレスが実行時にならないと決定されないため、スケジューリング時に検出することはできない。したがってデータ依存関係を満たすことを保証するには、あるストア命令に続くロード命令は、先行するストア命令が完了しなければ実行することができない、という制約が必要である。

これは異なるアドレスに対する後続ロード命令でさえも、先行ストア命令に依存することになる。これが曖昧なメモリ参照の問題であり、命令レベル並列性 (ILP) を引き出す際の大きな妨げとなっている。

この問題に対して、メモリアクセス命令を、参照アドレス計算とメモリアクセスそのものの二つに分離することが考えられる。

参照アドレスは、メモリポートやストア命令における値のようなメモリアクセスに必要な資源の利用可能性とは関係なく求めることができる。ロードアクセスはその参照アドレスが先行ストア命令の参照アドレスと一致していなければ、そのストアアクセスに先んじて実行することが可能となる。分離を行うことで先行ストア命令の参照アドレスが早期に求められるため、それにより曖昧性の解消を早めることができる。

本稿では、曖昧なメモリ参照の問題を解決する方法の一つとしてこの分離方式を評価した。

またメモリ参照の曖昧性の問題に対して、ロードの投機実行が提案されている。これは先行ストア命令と後続ロード命令のアドレス一致/不一致を予測し、

不一致と予測されたロード命令を投機的に実行する方式である。

予測方式と投機ミス時の回復方法について幾つか提案されており、本稿では予測方式に StoreSet を利用した方法を用いた。StoreSet とは、メモリアクセス命令間の依存関係をテーブルを利用して保持する機構のことである。アドレス予測時には、このテーブルを参照してロード命令のストア命令に対する依存を検出することで参照アドレスを予測する。投機ミス時の回復方法としては, 命令無効化処理を選択した。

命令無効化処理には三種類の方法を用いて、それぞれを比較した。その三種類の方法とは、ミスをしたロード命令に依存するものを選んで無効化させる選択的手法、wakeup logic を用いて再発行する命令を選択する手法、ミスをしたロード命令よりも後にフェッチされた命令を全て無効化する非選択的手法、の三種類の手法を選択し、実装 ・評価した。

その結果、参照アドレスの曖昧性により生じる問題に対して、分離する方法と投機的実行が有効であることが確認された。また、各無効化処理の評価も概ね予想通りのものであり、参照アドレスの予測を行わない場合には選択的方法および wakeup logic を用いた方法の効果が高く、非選択的方法ではやや性能が落ちることがわかった。

その理由として、無効化命令数の違いがあげられる。アドレス不一致予測の手法である StoreSet の投機ミスに対する効果も十分であり、これにより投機ミスの割合は総ロード命令発行回数に対して 1% 以下となった。その場合は三種類の無効化処理の方法に性能の差はほとんどみられなかった。これは命令▾ 無効化の数が著しく減少したために再発行処理によるパフォーマンスの低下による影響が抑えられた事を示している。

Evaluation of splitting and speculation of load/store instructions

Yoshikazu Koide

Abstract

One of the factors that prevent the performance of supar scalar processor is the problem of ambiguous memory dependence. Instructions scheduling grasps and satisfies the data dependence among some instructions. There are two types on the data dependence.

They are the dependence through the register and the memory. The dependence through the former can be resolved. But, The latter that is of the dependence among the memory access instructions can be not resolved, because the referenced address is not determined until the instruction is executed. Therefore, the limitation that load instructions following a store instruction can not be executed until the store instruction is completed is needed to assure that the deta dependence is satisfied.

This makes the load instructions following a store instruction depend on the preceded store instruction,even if the addresses of the load instructions are different from that of the store instruction. This is the problem of ambiguous memory dependence and makes prevent ILP to be extracted.

For this problem,there is one of the solutions that memory access instructions are split and the instructions calculating the referenced address and just memory accessinstructions are made.

Store instructions need the resources like a value to be executed but calculating the referenced address instructions are executed without them. Therefore, the referenced address is determined rapidly. If the referenced address of load instructions do not coincide with that of store instructions preceded ,then load instructions can be executed before the store instructions is done. Splitting the memory access instructions, referenced address of prior store instruction is executed rapidly, so the problem of ambiguous memory dependence is solved quickly.

This paper sais that the method of splitting the memory access instructions

solves the problem of ambiguous memory dependence.

And the load instructions are executed speculatively to solve the problem of ambiguous memory dependence. Whether the address of store instructions coincide with that of load instrutions following store instructions, or not is predicted and if the prediction is that the address of the load instructions does not coincide with another, then the load instructions are executed speculatively. Some methods about the prediction and recovery have already proposed.

This paper sais that we evaluate performance of super scalar processor with StoreSet as the method of prediction and *reissue* mechanism as that of recovery from speculative mistakes. StoreSet is the system that maintains the dependence amaong memory access instructions using unique table. Referencing this table to detect the dependence of a load instruction for store instructions, referenced address is predicted. And that we evaluate about three *reissue* mechanisms for speculative mistakes, they are *non selective nullification*, *selective nullification*, and the way of using *wakeup logic*.

First is the method that all the instructinos fetched after miss-load instruction was done are *reissued*. Second is the method that only the instructions depending on miss-load instructon are *reissued*.Third is the method of using *wakeup logic*.

ロード ・ストア命令の分離と投機の評価

目次

第 1 章　は じ め に　1

第 2 章　分離方式　2

2.1　参照アドレスの曖昧性 2

2.2　メモリアクセス命令の分離 3

第 3 章　ロード命令の投機的実行　5

3.1　ロード命令の投機的実行 5

3.2　予測方式 6

3.3　投機ミス時の無効化処理 9

3.3.1　issue queue 9

3.3.2　非選択的無効化方式 12

3.3.3　選択的一括無効化方式 13

3.3.4　*wakeup logic* を用いた無効化方式 13

3.3.5　RUU を用いた命令スケジューリング 14

3.3.6　命令の無効化処理 15

第 4 章　性能評価　18

4.1　評価環境 18

4.2　評価結果 18

第 5 章　お わ り に　23

謝辞　24

参考文献　24

第1章　はじめに

スーパスケーラ・プロセッサの性能を向上させるために、命令レベル並列性 (Instruction level parallelism : ILP) の向上が考えられる。ILP の向上を制限する制約の一つに、データ依存がある。データ依存関係には二つのタイプが存在し、一つはレジスタを介した依存関係、もう一つはメモリを介した依存関係である。

レジスタを介した依存のうち、WAW ハザード (write after write hazard) と WAR ハザード (write after read hazard) はレジスタ・リネーミングにより解消可能であるが、RAW ハザード (read after write hazard) は解消する方法がない。一方、メモリを介した依存関係は、同一アドレスを参照するメモリアクセス命令間の依存となるが、参照アドレスが実行時にならないと決定されないため、スケジューリング時に検出することはできない。したがって、あるストア命令の実行中には、そのデータ依存関係が不明であるため、そのストア命令に続くロード・ストア命令を実行できない。これは、アドレスが異なるメモリアクセス命令でも、先行するストア命令に依存している事を表している。これが曖昧なメモリ参照による依存 (ambiguous memory dependence) であり、ILP を引き出す際の大きな妨げとなっている。

本稿では、曖昧なメモリ参照による依存を解消する方法として、以下のような手法を評価した。

まず一つに、メモリアクセス命令を、参照アドレス計算とメモリアクセスそのものの二つに分離することが考えられる。参照アドレスは、メモリポートやストア命令における値のようなメモリアクセスに必要な資源の利用可能性とは関係なく求めることができる。参照アドレスが求まった時点で、メモリ参照の曖昧性は解消されるため、ロードアクセスはその参照アドレスが先行ストア命令の参照アドレスと一致していなければ、そのストアアクセスに先んじて実行することが可能となる。

またメモリ参照の曖昧性の問題に対して、ロードの投機実行が提案されている。これは先行ストア命令と後続ロード命令のアドレス一致/不一致を予測し、不一致と予測されたロード命令を投機的に実行する方式である。それに伴い、ロード命令に依存する命令も投機的に実行する対象とする。データ依存の予測には、StoreSet を用いた方法がある。また、データ依存の予測を実現するため

には、予測を失敗した場合にプロセッサを投機状態から正しい状態に回復させるための機構も考慮する必要がある。この方法には、命令をフェッチからやり直す方法などがあるが、本稿では無効化処理を用いた。無効化処理には、issue queue を使った方法である、非選択的無効化処理、及び選択的一括無効化処理、$wakeup\ logic$ を用いた方法の三種類について評価した。この内、非選択的無効化処理は分岐予測の失敗時の回復と同様に、投機ミス後の命令全てを破棄する事でプロセッサの状態回復を実現するものであるが、投機ミスによるミスペナルティは分岐予測のそれと比べて非常に大きいため、これは最適ではない。そこで、投機ミスをした命令と依存関係にある命令を選択し、無効化する手法である選択的一括無効化方式と、$wakeup\ logic$ を用いた無効化方式も考えられる。

本報告では、SPEC95 ベンチマークプログラムに対し、上記で述べた参照アドレスの曖昧性の問題を解決する方法として、分離方法と投機的命令実行を実装し、その性能を評価した。投機的命令実行の際には、StoreSet を用いた参照アドレスの不一致予測と、投機ミス時のハードウェア状態の回復手法としての無効化処理を行った。その結果、分離方式、投機的な命令実行ともに、最大 2 倍を超える IPC の向上が得られた。また、StoreSet を用いた予測で、総ロード命令発行数に対する投機ミスをしたロード命令数の割合を、1%以下に抑制した。

本稿は、以下のように構成される。2 章では参照アドレスの曖昧性の問題について、及びメモリアクセス命令の分離方式について述べる。3 章では投機的命令実行について、加えて参照アドレスの不一致予測のための StoreSet の構成について、投機的命令実行時に起こる予測ミスによる回復処理としての三種類の無効化処理について述べる。4 章でそれらについて評価を提示し、5 章で結果についての考察とまとめを述べる。

第 2 章　分離方式

2.1　参照アドレスの曖昧性

ロード命令とストア命令の間には参照アドレスの曖昧性の問題が存在する。参照アドレスの曖昧性とは、ある時点においてメモリアクセス命令の参照するアドレスの値が決定されていないという事を指す。以下に、曖昧性によって生じる問題についての例を示す。

表 1 はプログラムの一例である。数字はフェッチされた順番を表す。プログラ

```
                ...
                ...
                ...
I0:  (100)  STORE  r0  M[X]
I1:  (101)   LOAD  r1  M[Y]
I2:  (102)    ADD  r1     1
                ...
                ...
                ...
```

表 1: 参照アドレスの曖昧性

ム実行中には、先行するストア命令よりも早くロード命令が発行できる状況が考えられる。I0 のストア命令が終了していない状況で I1 のロード命令が発行可能状態になったとする。このロード命令の参照するアドレスが、先行するストアのそれに対して依存関係を持つかどうかはこの時点では不明であり、決定できない。これが参照アドレスの曖昧性である。ここでは、その問題点を明確にするため、仮にロード命令を先に発行してその挙動を探る。

I1 が実行され、メモリアドレス M[Y] から値:A をロードしてきたと仮定する。その後も値:A を使って I2 等、いくつかの演算命令をこなす。数サイクルした後、I0 が実行され、M[X] に値:B がストアされた。ここで初めて M[X] の指す先が判明し、曖昧性が解消される。その結果、M[X] の指す先は M[Y] と同じであったことがわかったとする。I1 以下の演算命令は、I0 でストアされた値:B を用いて演算されなければならなかった。

こういった状況が考えられるため、先行するストアが終了し、参照アドレスの曖昧性が解消されるまでは、ロード命令を実行する事ができない事がわかる。これが参照アドレスの曖昧性によって生じる問題である。

2.2 メモリアクセス命令の分離

前節で述べた通り、メモリアクセス命令には参照アドレスの曖昧性による問題があり、ロード命令の発行が制限される。この問題の解決方法の 1 つとして、メモリアクセス命令の分離方式がある。曖昧性の問題点は上記の通り、参照ア

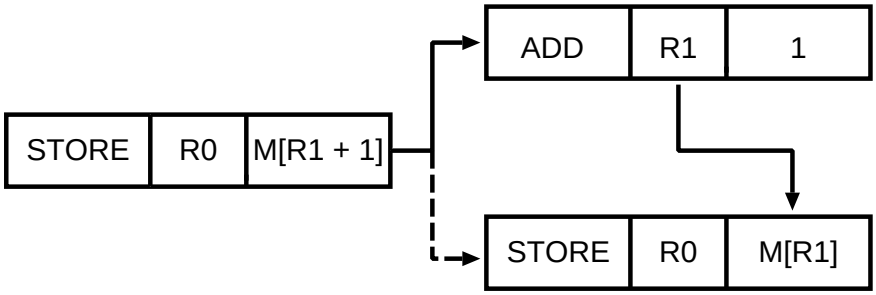


図 1: 分離方式

ドレスが同じだった場合に依存が生じる事にあった。従来のロード・ストア命令では、メモリアドレスの計算とメモリアクセスを同時に実行される。そのため曖昧性はストアの終了まで解消されない。これを解決するため、分離により参照アドレスの演算を早期に完了させる。分離により、参照アドレスはメモリアクセスとは独立に計算されるため、従来よりも早いタイミングでアドレスの値が求まるため曖昧性が早期に解消される。これにより、ロード命令発行のタイミングを早められると考えられる。

以降、具体的に分離の有効性を述べる。図 1 のようなストア命令であれば、図にあるように、`add r1 1` という演算命令と、`store r0 M[1]` というメモリアクセス命令の二つの別個の命令として認識し、それぞれを別々に実行する。ストア命令の参照アドレス演算部 `add r0 1` はメモリアクセスが開始される前に完了する。これは、曖昧性がストア命令の終了を待たずに、それよりも早い段階で解消されたことを示す。

次に、従来の方式とのロード命令の発行タイミングの違いを図 2 に示す。この図は、上部の白矢印でストア命令がフェッチされてから完了するまでの流れを表している。中央の矢印部分が、分離しない基本的な従来の方法 (basic) がロード命令を発行できるタイミングを、下の矢印部分が、分離方式の場合におけるロード命令が発行できるタイミングを示している。この図から、ストア命令の完了以降にしかロード命令を発行できない従来の方法と、ストア命令の参照アドレスの決定時に曖昧性が解消されるため、その時点からロード命令が発行可能になっている分離方式では、明らかにロード命令発行のタイミングに差があることがわかる。これは、分離方式によるメモリの曖昧性の問題に対する有効性を示したものである。

実装は、メモリアクセス命令を二つの独立した命令に分割した上で、メモリ

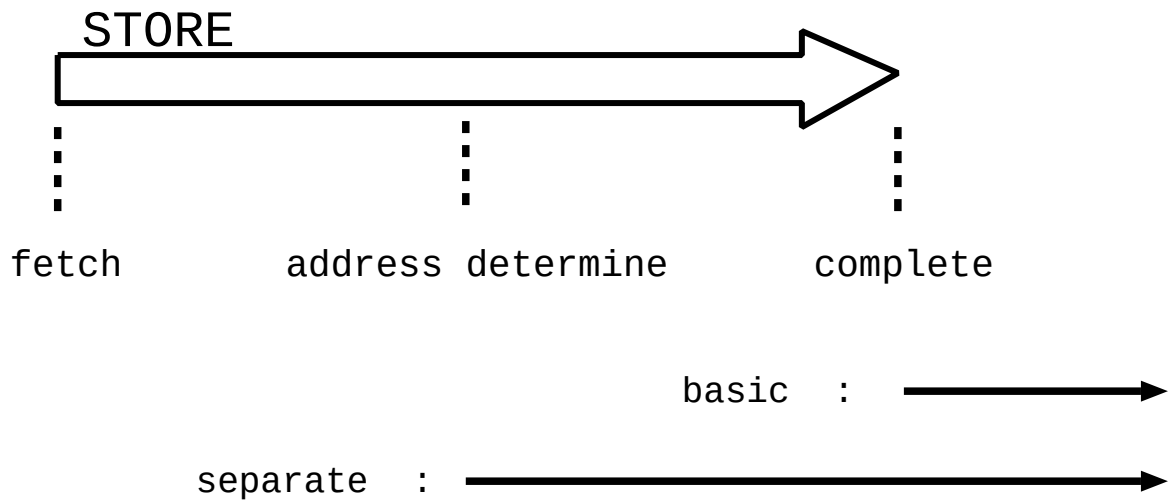


図 2: 分離方式におけるロード命令の発行のタイミング

アクセス命令をアドレス演算に依存させる。依存させる機構、依存関係にある命令の挙動に関しては、スーパスケーラの規格内にあるため、省略する。これらはハードウェア上では分離して実行され、独立した命令として動作する。プログラム上では一つの命令であるため、命令の終了は同時に行われる必要がある。命令ウィンドウのエントリに、その命令が終了できる状態にある時に立つ、comp ビットを用意する。参照アドレス演算の終了確認時に、メモリアクセス命令の comp ビットと合わせて同時に終了処理をする。

第 3 章　ロード命令の投機的実行

本章ではロード命令の投機的実行について述べる。通常はメモリの曖昧性問題により、ロード命令は先行するストア命令の完了を待って発行される必要がある。ロード命令の投機的実行とは、メモリの曖昧性問題を何らかの方法で解決し、ストア命令の完了以前にロード命令を発行する方法である。次節で投機的実行の概略を説明する。

3.1　ロード命令の投機的実行

ロード命令を、先行するストア命令よりも先に発行するためには、メモリの曖昧性問題を解決する必要がある。そのために、参照するメモリアドレスを予測し、それに基づいてロード命令を発行する方法がある。[1]

投機的に実行することで、前章で述べた分離方式や、分離も投機も行わない

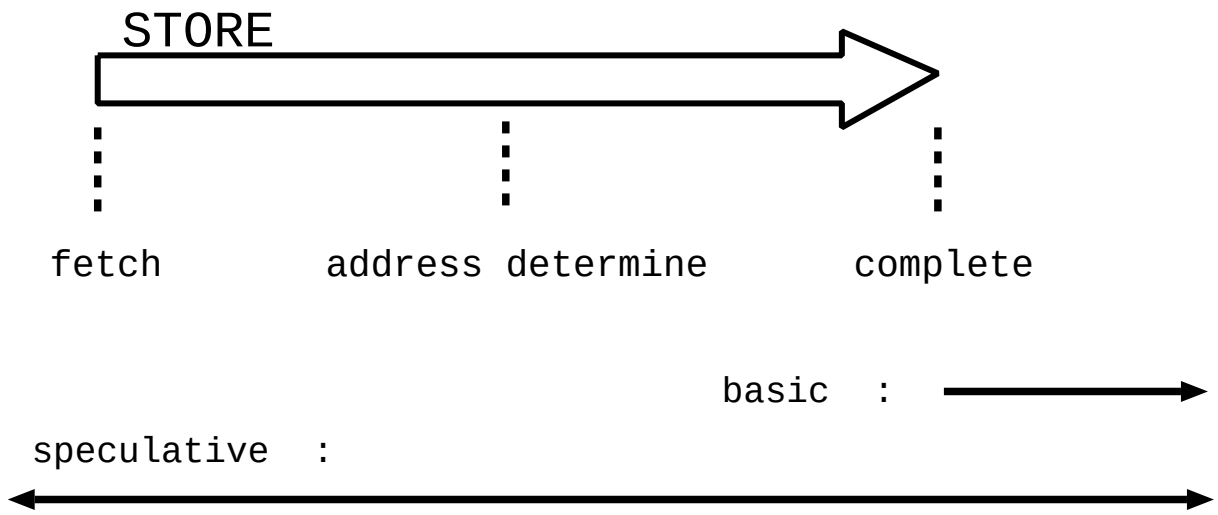


図 3: 投機的実行されるロード命令発行のタイミング

方法と比べて早い段階でロード命令を発行することができる。図 3 は、図 2 と同じくストア命令のフェッチから完了までの流れの中で、従来の方式 (basic) と投機的実行する場合の、ロード命令の発行のタイミングを示したものである。

投機的に実行したロード命令の参照するメモリアドレスが、後に発行されたストア命令のそれと同じだった事が発覚した場合 (以下、これを投機のミス、あるいは単にミスと呼ぶことにする)、ハードウェアの状態回復が必要である。それを解決する方法として命令の無効化処理がある。以下では参照するメモリアドレスを予測する方法と投機ミス時の無効化処理の方法について詳しく述べる。

3.2 予測方式

ロード命令を投機的に実行するためには、投機実行するロード命令の参照するメモリアドレスと、先行するストア命令のアドレスとの不一致を予測する必要がある。本節では StoreSet を用いた予測方式について述べる。

StoreSet のコンセプトは以下のような基礎的な考えに基づいている。複雑な依存関係にあるロード命令やストア命令のメモリの依存関係を把握するのは、命令実行において重要な要素である。そこで、依存関係を管理するテーブルの利用が考えられた。それが StoreSet である。

あるロードの StoreSet はそれと依存関係にある全てのストア命令から成り、それらは各命令の Program Counter によって区別される。プログラムの開始時、

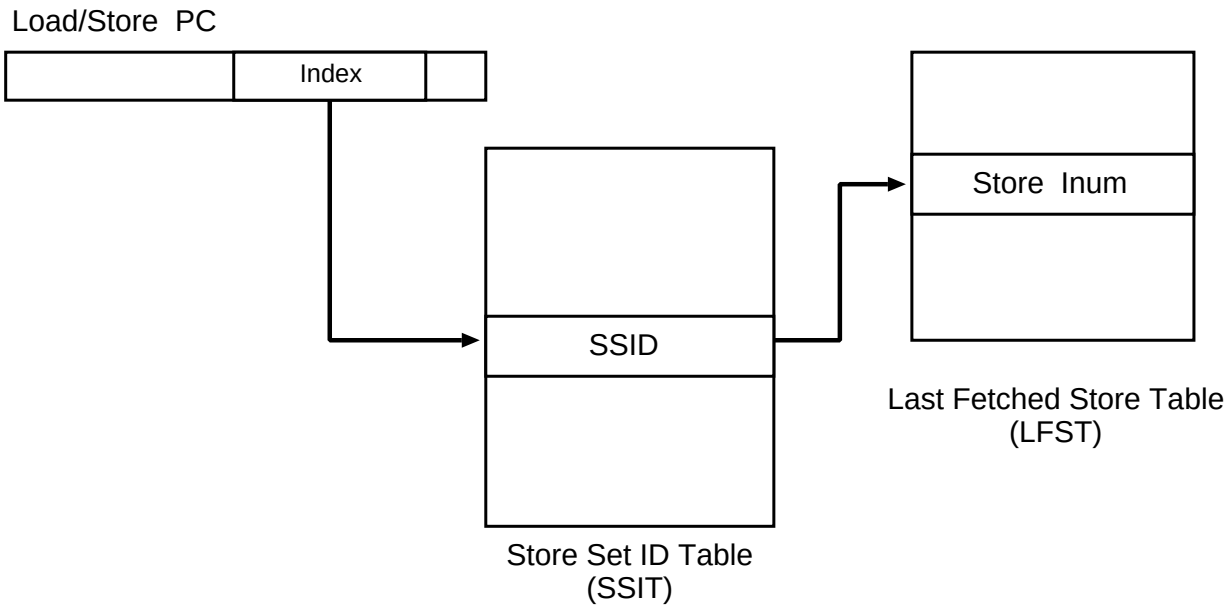


図 4: StoreSet を用いた予測方式

全てのロード命令の StoreSet は空である。投機的にロードが実行され、ミスが起った時にストアのロードの StoreSet に加えられる。同様に、同じロード命令が他のストアに対しても投機ミスをした場合は、さらにそのストアも追加される。次にこのロードがディスパッチされる時、そのストア命令は StoreSet の中にあった既存のストア命令の後に予測される必要がある。

次に、実装について述べる。

StoreSet は、その中に含まれるいくつかのストアに StoreSet を持つロードが依存しているというものである。よって StoreSet 内にあるストア命令が全て実行されるまでロードの発行を遅らせる必要がある。実際には、メモリの資源は有限であるため、上記のようなストア命令を一つのロード命令に対して複数個把握しておくような構造を構築するのは、ハードウェア上では非常に困難である。そこで George らは、 StoreSet の中にあるいくつかのストアのうち一つを選び、ロードに依存させる方式を提案する。[2]

George らの提案する StoreSet を用いた投機ミスのための予測には、図 4 に示す、二つのテーブルを用いる。一つ目は各命令の PC を index とした Store Set Identifier Table(SSIT) と呼ばれるものである。SSIT はロード命令とその StoreSet 内にあるストア命令とを共通のタグ (Store Set Identifier(SSID)) を使って StoreSet を維持している。二つ目は、Last Fetched Store Table(LFST) と呼ばれるもので

あり、これは StoreSet のために最近に fetch されたストア命令について動的な情報を保持している。そのテーブル内の情報とは、計算機内で各命令に固有のハードウェア上ポインタである、ストアの inum である。

フェッチされたロード命令は自らの PC を基に、まず SSIT にアクセスし、SSID を得る。もし、そのロード命令が有効な SSID を持っていれば、有効な StoreSet を持っている事になる。次に LFST にアクセスして StoreSet 内にある、最も近くフェッチされたストア命令の inum を得る。

新しくストア命令がフェッチされると、SSIT にアクセスする。もしそのストア命令が SSIT 内ですでに有効ならば、そのストアは有効な StoreSet に属す。この場合、行われるべき事は二つある。一つには、LFST にアクセスし、最新のストア命令の情報を StoreSet に得、そのストアに依存させる必要がある。二つ目には、自らが最新のフェッチされたストアであるため、テーブルの内容を更新しておく必要がある。

ストア命令は、発行された後に LFST にアクセスし、そのエントリを自らにのみ関係する場合無効化する。これにより、ロード命令とストア命令は未発行のストアにのみ依存させられる事が保証される。

以下、実際に例をとって StoreSet を用いたアドレス予測の方法を述べる。

プログラムが開始されると、SSIT のエントリは全て無効になっている。初めに、ロード、もしくはストア命令がテーブルにアクセスし、有効となっているメモリ依存情報がないことを知る。ロード命令が投機ミスをすると、StoreSet が SSIT の中に作られる。参照アドレスが同じロード命令とストア命令は SSID、”X” を割り当てられるとする。SSID、X は SSIT の中の二箇所に書きこまれる。一つ目はロード命令の PC を index とする場所であり、二つ目はストア命令の PC を index とする場所である。次にそのストア命令がフェッチされた時、その PC を使って SSIT を読む。SSID は有効となっているので、その値を用いて LFST にアクセスし、StoreSet “X” から最近フェッチされた命令は無く、前と状況が変っていないことを知る。よって、もう依存関係を新たに 持たされることはない。そのストアは LFST に自らの inum を書き込む。その後、ロード命令がフェッチされると、その命令はまず SSIT にアクセスし、次に SSID “X” を用いて LFST にアクセスする。LFST は、そのロード命令がそこで見つかったストア命令に依存する、という情報を渡す。

そのロード命令が後に、上のストア命令とは異なるストア命令とも同じ参照

アドレスを指すことになったとする。SSID “X” は、SSIT 内の新しいストア命令の PC を index としている場所にコピーされる。この段階で、SSIT 内の SSID “X” の指す場所には三つのエントリがある。次に、その二つのストアとロードがフェッチされると、二つ目のストアは一つ目のストアに、ロードは二つ目のストアに依存する関係を持つことになる。

3.3 投機ミス時の無効化処理

ロード命令の投機的実行の際には、参照アドレスの曖昧性によるミスが起こる可能性がある。よって、いくつかの命令はミスが起った場合に何らかの方法でハードウェアの状態を回復する必要がある。我々はそのための手法として命令の無効化処理を採用する。無効化が必要となる命令とは、投機ミスを起こしたロード命令と、その命令に依存する命令である。以下ではまず、無効化処理に必要な issue queue についての説明、選択的一括無効化方式と、非選択的無効化方式について述べ、最後に $wakeup\ logic$ を用いた方法について述べる。

3.3.1 issue queue

本節では、無効化処理の方法を述べるにあたり、まずは必要となる issue queue について説明する。[3]

命令がフェッチされた後、それらはデコードとリネームの処理を受ける。リネームされた命令は、そのソースオペランドが準備され、実行されるまで issue queue に留まる。命令は実行された後、完了したものとして ROB(reorder buffer) にマークされる。その後、その命令に Program Order で先行する全ての命令が、完了し、終了すると、その命令も終了する。

issue queue は、図 5 に示すような、命令間の依存を追跡するための依存行列を含む。これは、同時にスケジューリングする命令と同じ数の行をもち、物理レジスタと同じ数の列を持つ。列は、全ての行に接続され、列の信号が行にとどく様な配線である。それぞれの列はレジスタにあるデータの有効性をマークしている。列に接続されたシフトレジスタもしくはカウントダウン ・レイテンシ ・カウンタによってセットされる。

ある命令が issue queue のエントリ (行) に挿入されると、その命令のソースオペランドに関係する信号がセットされる。また、デスティネーション ・レジスタに関係するレイテンシ ・カウンタはその命令のレイテンシに初期化される。

依存行列の各交差点には、必要とされているソースオペランドが用意されてい

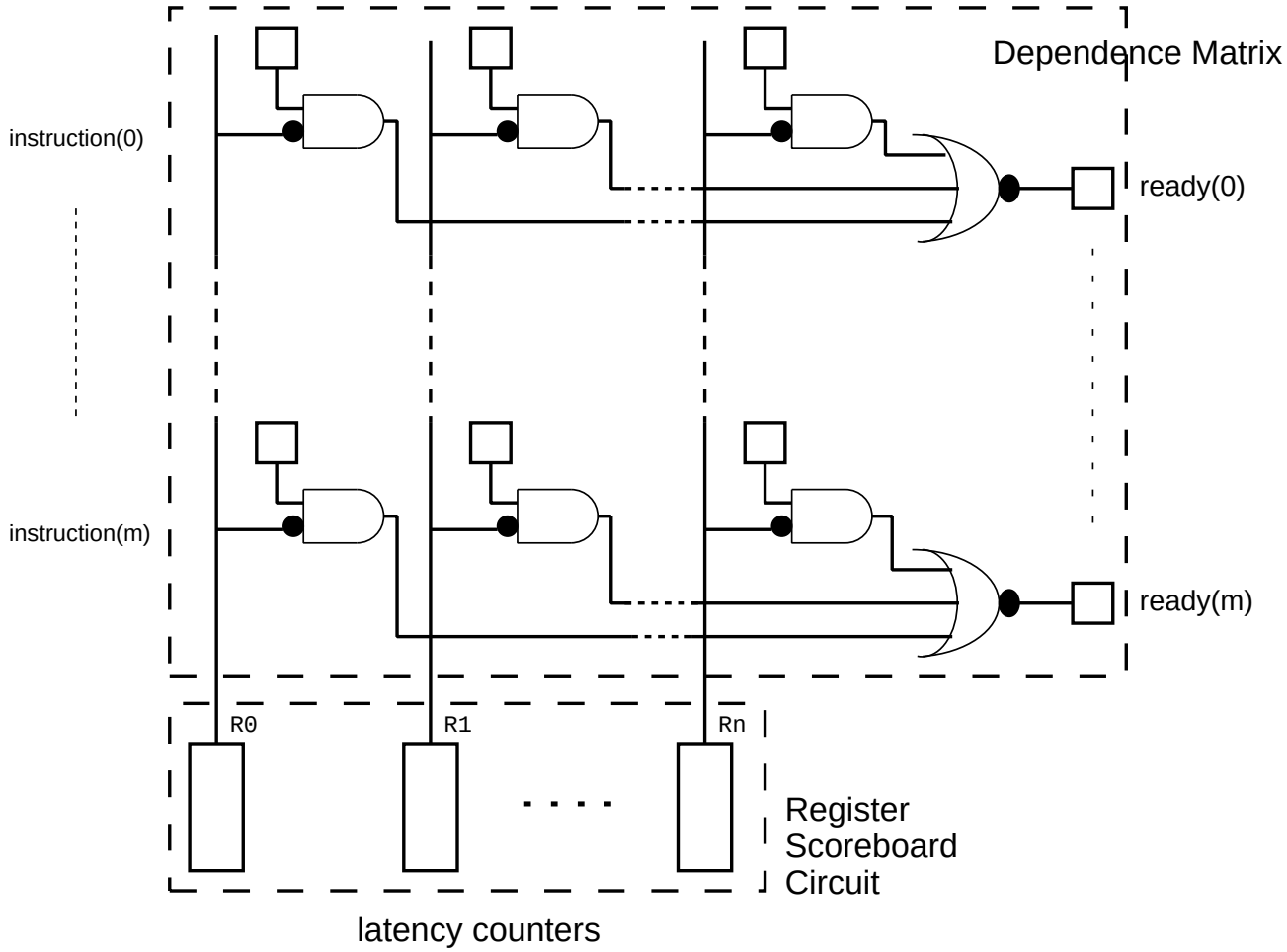


図 5: 依存行列の構造

るかを決定する論理回路が含まれている。各行では、もしその命令が $select\ logic$ に選択される準備ができているなら、これらの論理回路の出力は `Ready bit` を計算するのに使われる。`Ready bit` は毎サイクル評価される。

命令が発行されると、そのデスティネーション・レジスタに関係するレイテンシ・カウンタは毎サイクルごとに減少する。

通常の、投機ミスのない命令実行では、命令が発行されると即座に issue queue から除かれる。しかし、投機ミスが考えられる状況では、無効化処理を素早く行うために、その処理の必要が無いとわかるまで issue queue 内に発行された命令を残しておく。ロード命令が投機的に実行され、その参照アドレスが決まるまで、issue queue に残された命令は $select\ logic$ に発見されないようにする必要がある。そのために、`no-request bit` というフラグを、ロードの投機的実行が

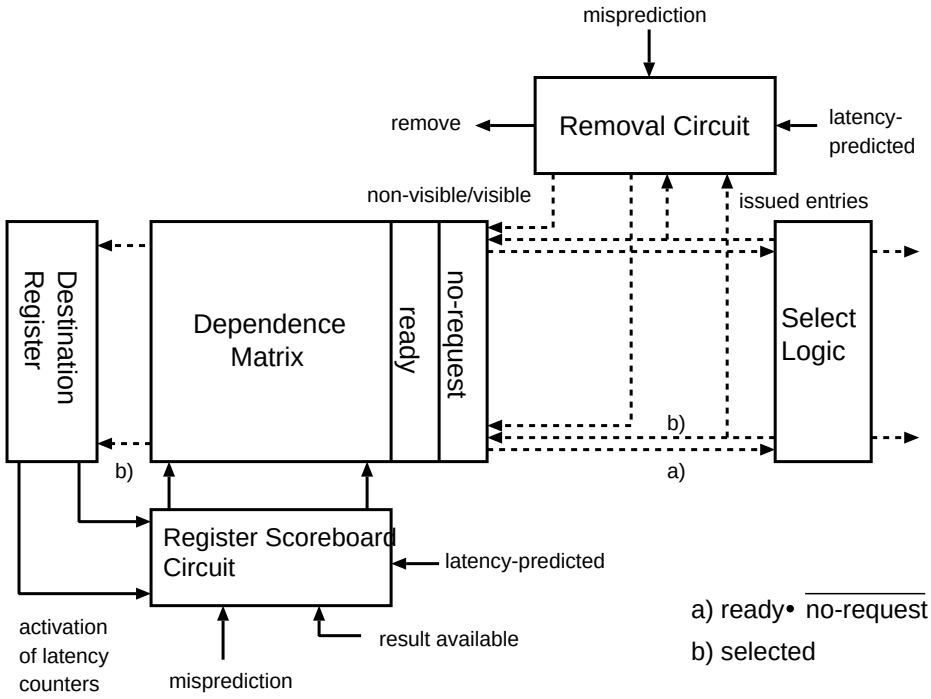


図 6: issue queue の構造

された後にフェッチされた各命令が立てる事で対処する。すでに発行された命令が無効化の可能性がないとわかった時、その命令は issue queue から除かれる。ミスが起ったときはその命令を無効にする必要がある。フラグを落とし、$select\ logic$ に発見されるようにして無効化処理を優先して行うようにする。

これらの処理をするための二つの制御回路について述べる。(図 6)

一つ目は、removal circuit といい、issue queue から (それが排除してよければ) 発行された命令を排除し、同様に無効化の必要があれば $select\ logic$ に発見されるようにフラグを立てるための回路である。二つ目の register-scoreboard circuit は発行された命令と無効化された命令に対して働く回路である。`Ready bit` は毎サイクル再評価されるので、無効化された命令は自らの `ready bit` を再評価し、無効化されている命令に依存する全ての命令は眠った状態に保たれる。

毎サイクルごとに、removal circuit が発行された命令を検出し、register-scoreboard circuit はその参照アドレスを知る。

以下では非選択的、および選択的無効化処理の方法について詳しく述べる。図 7 は、命令の依存関係を表したデータフローグラフである。アルファベット順にフェッチされているものとする。

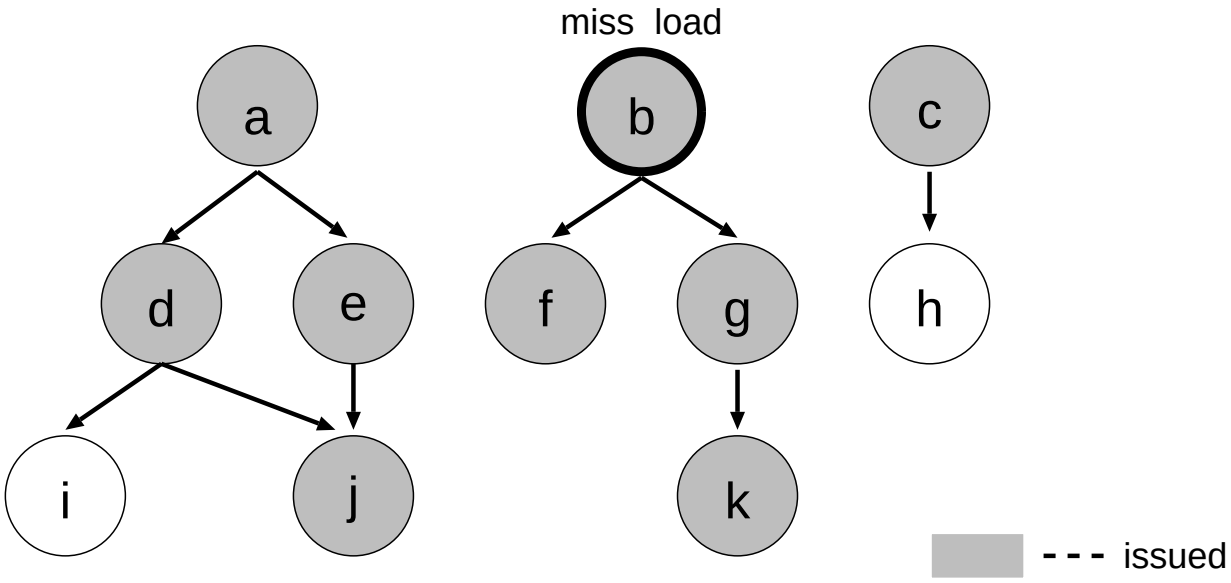


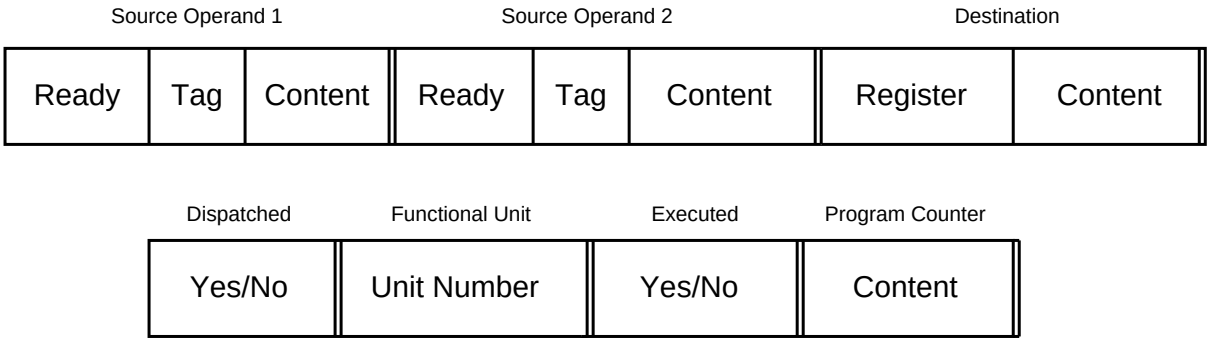
図 7: データフローグラフ

3.3.2 非選択的無効化方式

最も単純な無効化の方法は、無効化が必要となる可能性のある命令を全て再発行する方法である。投機ミスを起こしたロード命令の後、フェッチされた全ての命令を無効化する。

ロード命令が投機的に実行された後に発行された全ての命令はissue queueの中に残っている。その後、投機ミスが無いと判断された時、それらの命令はissue queueの中から排除される。投機ミスであった時、それらの全ての命令は無効化され、無効化処理が施される。removal circuitとregister-scoreboard circuitが参照アドレスの確保と毎サイクルissue queueに残った命令の発行を行う。

投機ミスが生じると、removal circuitとregister-scoreboard circuitは、無効化をすべき命令に関連する情報を集める。その後、register-scoreboard circuitは、無効化されるべき命令の目的レジスタに関する、依存行列の行をクリアし、removal circuitはこれらの命令のissue queueエントリの`no-request bit`を0にもどす。その上、register-scoreboard circuitは無効化すべき命令の目的レジスタそのものもクリアする。無効化された命令の間には、無効化すべき命令に依存していない無関係な命令もある。それらの命令は、ソースオペランドがまだ有効となっ



Register Update Unit.

図 8: レジスタ更新ユニット (RUU)

ているので、突然に発行される場合がある。

この方法では、図 7 の例で言えば、b,c,d,e,f,g,j,k の八つが無効化処理される。

3.3.3 選択的一括無効化方式

今節では、本来投機的命令実行のミスをした時に無効化しなければならない命令のみを選択し、1 サイクルでそれらの命令に対して無効化処理を行う方法について述べる。本来無効化しなければならない命令とは、投機ミスをしたロード命令に依存する命令のことであり、図 7 で言えば b,f,g,k の四つがそれに当たる。そのため、前節で述べた無効化処理は、ハードウェア的な処理は単純であったが、非効率的であった。

当方法では、b,f,g,k の四つに再発行処理を施す。これらの命令に対する無効化処理を投機ミスが発覚した時点で、同一サイクル内に行い、次サイクルでは無効化準備が整った状態にする。この方式は、最も理想的であり、無効化に伴うミスペナルティは最も少ないと考えられる。しかしこの方式をハードウェア上で実現する事は非常に困難である。

3.3.4 *wakeup logic* を用いた無効化方式

本節では、RUU(register update unit) の命令発行アルゴリズムの一つである *wakeup logic* を用いた命令無効化方式について述べる。まず、RUU とそれを用いた動的命令スケジューリングを説明する。続いて命令無効化を可能にする拡張レジスタ更新ユニットについて述べ、実際の再発行を説明する。[4]

$$
\begin{array}{lccc}
I1: & r11 & \leftarrow & f_1(r10) \\
I2: & r12 & \leftarrow & f_2(r11) \\
I3: & r13 & \leftarrow & f_3(r12) \\
I4: & r14 & \leftarrow & f_4(r13) \\
I5: & r15 & \leftarrow & f_5(r14) \\
I6: & r16 & \leftarrow & f_6(r15)
\end{array}
$$

表 2: 命令シーケンスの例

3.3.5 RUU を用いた命令スケジューリング

RUU を用いた命令ウィンドウを図 8 に示す。命令ウィンドウの各エントリは、2 つのソースオペランドフィールド (`Source Operand`)、デスティネーションフィールド (`Destination`)、ディスパッチビット (`Dispathcd`)、機能ユニットフィールド (`Functional Unit`)、実行ビット (`Executed`)、そしてプログラムカウンタフィールド (`Program Counter`) から構成される。もしソースオペランドがいまだ得られないときには、レディビット (`Ready`) がリセットされソースオペランドが利用不可能であることを示す。同時に、そのオペランドを示すタグ (`Tag`) がセットされる。オペランドが利用可能になると、ソースレジスタの値が `Content` フィールドにセットされ、`Ready` ビットがセットされる。`Tag` 情報をともなったデスティネーションレジスタ番号は `Destination` フィールドの `Register` フィールドに保持され、命令の実行結果は `Content` に保存される。`Dispatch` ビットは `Functional Unit` フィールドで指定される機能ユニットに命令がディスパッチされているかどうかを示す。`Executed` ビットは命令が完了するとセットされる。`Executed` ビットがセットされていると、この命令とデータ依存関係にある後続命令はディスパッチ可能になる。最後に `Program Counter` フィールドは、分岐予測失敗からのプロセッサ状態の回復や正確な割り込みを実現する他、前述した StoreSet における `index` に使われる。

例を用いて命令の発行過程を説明する。表 2 に示す。命令シーケンスを用いて説明する。理解を容易にするために以下の仮定をおく。まず、演算 $f_1 - f_6$ のオペランドは 1 つとする。また、命令 $I3$ はロード命令とし、その演算レイテンシはアドレス計算 1 サイクルとメモリアクセス 1 サイクルからなるものとする。最後に、残りの命令のレイテンシはすべて 1 サイクルとする。

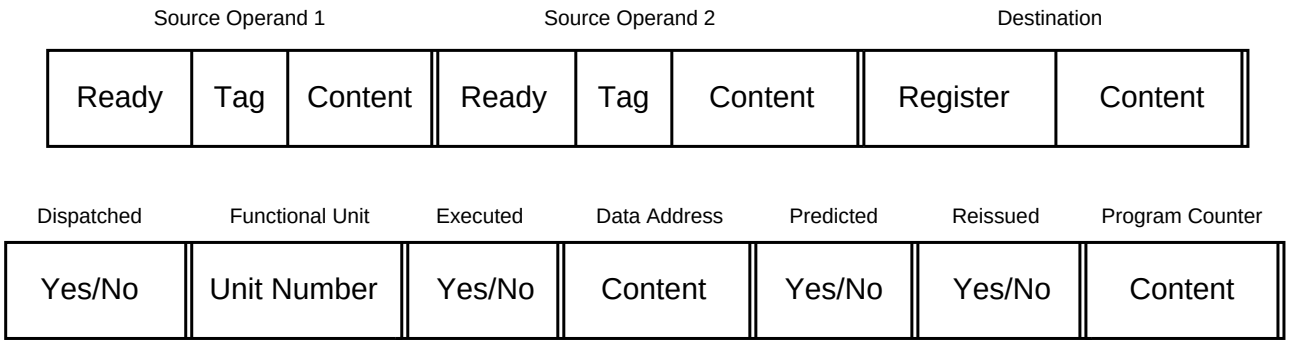
理解を容易にするために以下の仮定を加える。プロセッサのフェッチ幅とディスパッチ幅は、ともに3とする。機能ユニットの数は無限とする。命令のコミットは説明から省略する。最後に、タグ情報を含んだデスティネーションレジスタ番号は、アーキテクチャレジスタ番号と同じであると仮定し、レジスタ$r10$はすでに利用可能となっているものとする。以上の仮定に基づいて、命令スケジューリングを説明する。

まず最初のサイクルで命令$I1 - I3$がRUUに発行される。$r10$が利用可能なので命令$I1$のReadyビット(r)がセットされ、さらに命令$I1$は機能ユニットにディスパッチされてDispatchedビット(d)がセットされる。2サイクル目で命令$I4 - I6$が発行される。命令$I1$が演算を終えExecutedビット(e)をセットする。$r11$が利用可能になり、命令$I2$がディスパッチされる。3サイクル目で、命令$I2$が演算を終え$r12$が利用可能になり、命令$I3$がディスパッチされる。命令$I3$はロード命令なので、結果が得られるまでに2サイクル必要になる。したがって、4サイクル目にディスパッチされる命令はなく、5サイクル目で、ロード命令$I3$が$r13$を読み出し、命令$I4$がディスパッチされる。6サイクル目で、命令$I4$が演算を終え$r14$が利用可能になり、命令$I5$がディスパッチされる。7サイクル目で、命令$I5$が演算を終え$r15$が利用可能になり、命令I$I6$がディスパッチされる。最後に8サイクル目で命令$I6$が演算を終え、表2の例の実行が終わる。

以上のように、この例の命令シーケンスはデータ依存のためにシーケンシャルに実行せざるを得ない。そのため実行には8サイクルを要する。

3.3.6 命令の無効化処理

図8に示したRUUを拡張して命令無効化を可能にした命令ウィンドウを図9に示す。各エントリにはデータアドレスフィールド(Data Address)と予測ビット(Predicted)、さらに無効化ビット(Reissued)が付加されている。Reissueビットは対応する命令が無効化されたことを表わしている。前節の説明と同様に、予測アドレスはData Addressフィールドに保持され、それを用いて読み出されたデータはDestinationのContentフィールドに保持される。予測アドレスに対して実際のアドレスが計算されると、両者を比較して予測成功の判定が行われる。このときPredictedビットはリセットされる。もし比較結果が一致しなければ予測は失敗であり、Reissuedビットがセットされる。命令が終了し結果が得られると、従来の命令スケジューリングと同様にデスティネーショ



Extended RUU for instruction reissue.

図 9: レジスタ更新ユニット (RUU)

ンタグと実行結果が放送される。同時に再発行ビットも放送される。以降この信号を無効化信号と呼ぶことにする。予測に成功した場合は従来のスケジューリングと同様に、デスティネーションタグとソースオペランドタグの一致した後続の命令がオペランドを獲得する。一方、予測に失敗したときには、デスティネーションタグとソーズオペランドタグの一致した命令は無効化される可能性がある。タグの一致した命令の Dispatched ビットがすでにセットされている場合には、その命令は誤ったソースオペランドを用いて演算を実行しているため無効化されなければならない。Dispathched ビットと Executed ビットはリセットされ、Reissued ビットがセットされる。無効化された命令の実行が終了した場合にも無効化信号が放送される。したがって、無効化信号は、アドレス予測に失敗した命令あるいは再発行された命令が実行を終了したことを表わしている。以降は上述の説明と同様に、タグが一致し、かつ Dispatched ビットがセットされた命令が無効化される。こうして Reissued ビットが伝搬していくことで、無効化されなければならない命令が順に検出される。また、無効化されるべき命令がなくなると Reissued ビットが消えるので、たとえ RUU 内に命令が存在したとしても無効化は起こらない。本機構は従来の命令ウィンドウと比較して、完了命令 1 命令あたりの無効化信号が 1 本増えただけであり、ハードウェアの増大とサイクルタイムの延長を防いでいる。無効化されるべき命令を検出するためにタグを放送する機構は、動的にスケジューリングを行うプロセッサにはすでに *wakeup logic* として存在している。そのため、この機構がボトルネックにはならない。一方で同じ命令が複数回発行される可能性があるが、依

存関係にある命令間の距離が大きかったり機能ユニットの数の制限があったりするため、必ずしも演算結果を予測けいた命令が投機実行されるわけではなく、冗長に発行される命令の数も制限されると思われるので深刻な問題ではない。

次に、命令無効化の流れについて説明する。表 2 で動的命令スケジューリングを説明したのと同じ命令と仮定を用いる。今回はさらに、命令 $I3$ がロードデータアドレスを予測し、命令 $I3$ とデータ依存関係にある命令が投機実行されるとする。まず、命令 $I1 - I3$ が RUU に発行される。$r10$ が利用可能なので、命令 $I1$ が機能ユニットにディスパッチされ、Ready ビットと Dispatched ビットがセットされる。命令 $I3$ はロードデータアドレスを予測し、Predictecd ビット (p) をセットする。予測されたアドレスは Data Address フィールドに保存されている。次のサイクルで、命令 $I4 - I6$ が発行される。命令 $I1$ が演算を終え $r11$ が利用可能になり、命令 $I2$ がディスパッチされる。命令 $I3$ はメモリから $r13$ を読み出し、それを Distination の Content フィールドに保持する。命令 $I4$ はその $r13$ を用いてディスパッチされる。演算を終えた命令 I1 と $I3$ は Executed ビットをセットする。3 サイクル目で命令 $I2$ と $I4$ が演算を終えて $r12$ と $r14$ が利用可能になるため、命令 $I3$ と $I5$ がディスパッチされる。次のサイクルでは 2 通りの場合が考えられる。すなわち、命令 $I3$ がアドレス予測に成功した場合と失敗した場合である。アドレス予測に成功した場合は以下の通りである。命令 $I3$ は演算を終え予測ビットをリセットする。同時に命令 $I5$ も演算を終え命令 $I6$ がディスパッチされる。最後のサイクルで命令 $I6$ が演算を終え、この例の命令シーケンスは 5 サイクルで完了する。投機的実行を行わない場合と比べて、アドレス予測に成功すると実行時間が 3 サイクル短縮できる。一方、予測に失敗した場合は、命令 $I3$ の Predicted ビットはリセットされ、Reissued ビット (i) がセットされる。命令 $I3$ はもう一度メモリにアクセスする必要がある。同時に命令 $I5$ が演算を終え命令 $I6$ がディスパッチされる。次のサイクルで、命令 $I3$ がメモリアクセスを終え $r13$ が利用可能になる。命令 $I3$ の Reissue ビットがセットされているので無効化信号が放送される。タグの一致した命令 $I4$ は値の間違った $r13$ を用いてディスパッチされているので、$r13$ の放送によって無効化も対象として検出される。命令 $I4$ の Reissued ビットがセットされ、無効化されるべき命令の検出が伝搬する。6 サイクル目で命令 $I4$ は演算を終え、命令 $I5$ が無効化される。理由は命令 $I4$ の無効化と同様である。続くサイクルで命令 $I5$ が演算を終え、命令 $I6$ が無効化される。そして次のサイクルで $I6$ が演算を

終えて、この例の命令シーケンスは 8 サイクルで完了する。この例では、データ値の予測に失敗した場合に必要なサイクル数は、データ依存の投機実行を行わない場合に必要なサイクル数と同じである。

第 4 章　性能評価

4.1　評価環境

基本となるプロセッサ ・モデルはアウト ・オブ ・オーダー実行を行うスーパースカラ ・プロセッサである。アウト ・オブ ・オーダーを実現するためには RUU を採用している。各演算器の数は以下の通りである。

整数加算器が 8、整数乗算器が 1、メモリポートが 8、浮動小数点加算器が 2、浮動小数点乗算器が 1 で評価を行った。

実際の評価には、Simple scalar ツールセット (ver.2.0) を用いた。ベンチマークプログラムには SPEC95 を利用し、評価した。

以上の環境で、メモリアクセス命令の分離方式、ロード命令の投機的実行、StoreSet による参照アドレス不一致予測、非選択的無効化処理、選択的一括無効化処理、*wakeup logic* を用いた無効化処理、を実装し、各方式について評価した。次節でこれらの評価結果を述べる。

4.2　評価結果

図 10 は本稿の 2 章で述べた分離方式、3 章で述べた投機的実行、及びその両方の方法を組み合わせたものについて評価したものである。ただし、投機的実行については投機ミスの際のミスペナルティを実装をしていない。

default となっているのは、非分離 ・非投機で実行させた時の IPC(Instruction per cycle) であり、図 10 ではそれを 1 とした時の各手法による IPC 比率がどのようになったかを示している。

この結果から、分離方式を実装したものとロード命令の投機的実行、ともに IPC の向上に対する有効性を確認した。また、投機的実行と、分離かつ投機をしたものの間に大きな差がみられなかったのは、次のように考えられる。

図 2 と図 3 を見ればわかるように、分離方式のロード命令発行のタイミングは投機的実行のロード命令発行のそれに被覆される。非分離 ・非投機である従来の方法に対して、参照アドレスの曖昧性を解消する事を考えた場合、投機的

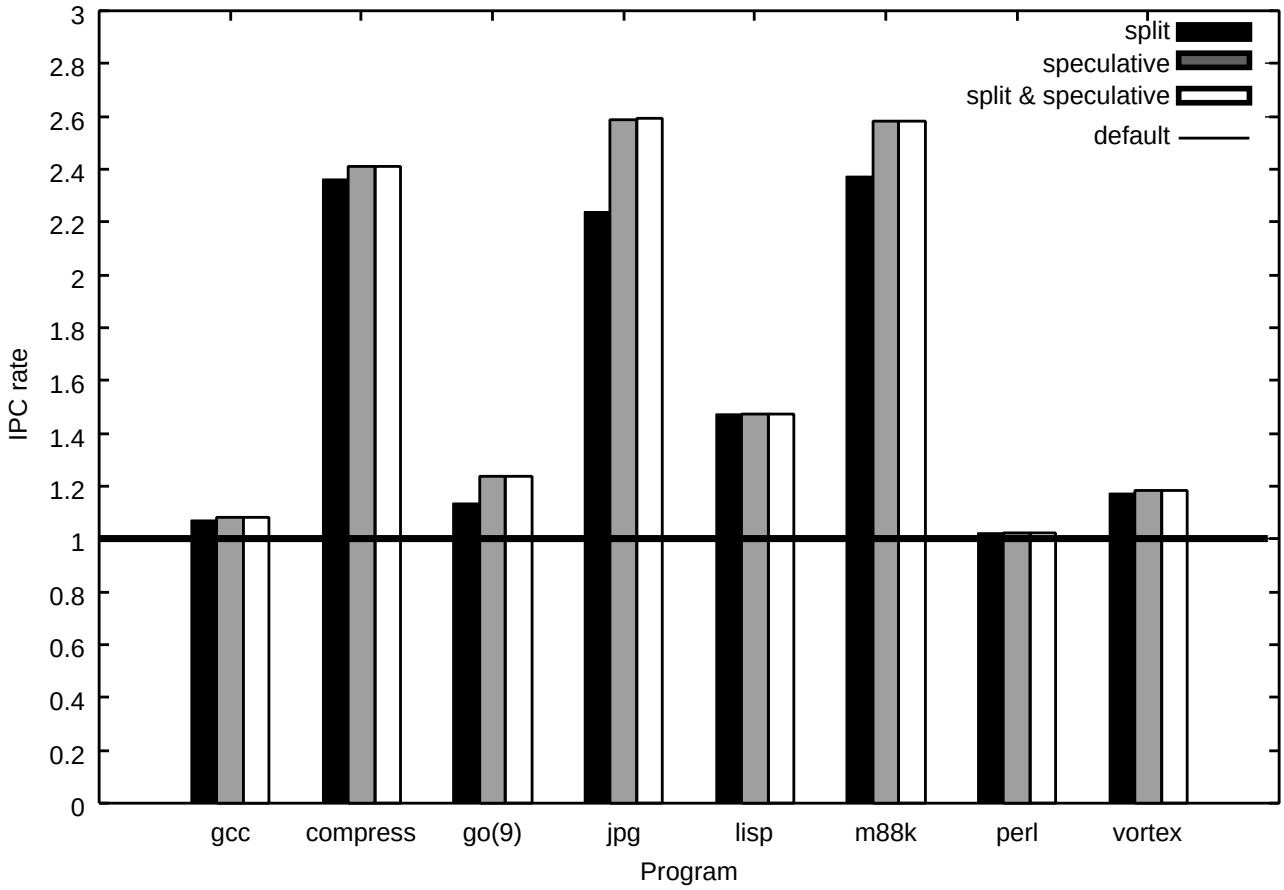


図 10: 分離、および投機的命令実行の効果

実行にミスペナルティが無い今回の評価では、いずれの方法を取っても結果に変化はない。その理由として以下のように考えられる。

分離方式において、先行ストア命令のメモリアドレス部分の命令がすでに終了しており、かつストア命令自体は完了していないとする。この時、参照するメモリアドレスの曖昧性はすでに解消されており、ロード命令は発行可能となっている。一方、曖昧性が解消された結果、ロード命令が発行できなかった場合を考える。本来なら分離されているためにロード命令は発行されないが、非分離であった場合は投機的に実行される。当然投機ミスが起こり、無効化処理をする必要があるが、今回に限ってはペナルティを課さない仕様で評価しているため、それがIPCの低下となって現れない。そのために、分離されていれば無効化処理を回避した分のIPC向上が期待できたが、それがないために投機的な命令実行の有効性に被覆された格好となった、と考えられる。

次に、図 11 で投機ミス時の三種類の無効化処理に対する分離方式の効果を

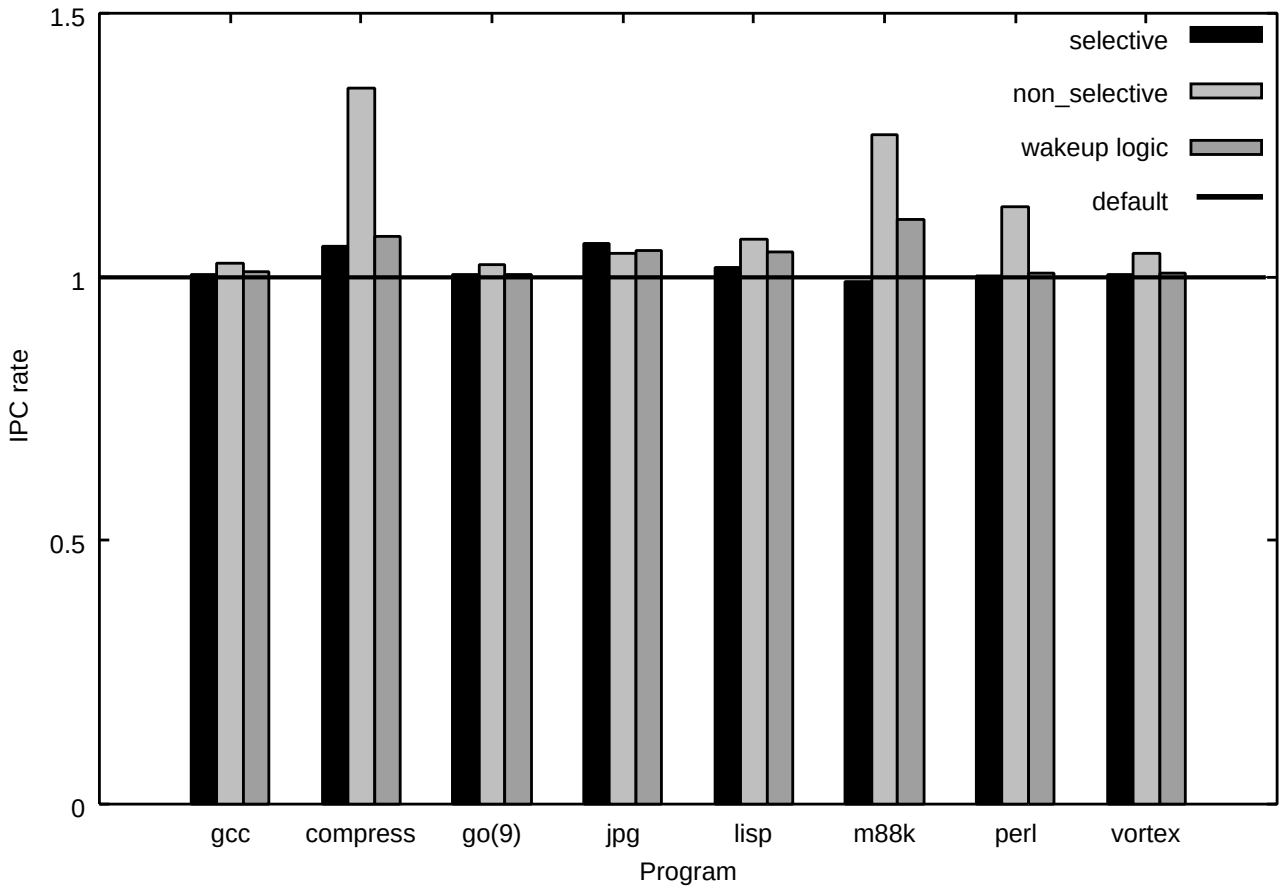


図 11: 分離 ・投機 (無効化処理) の評価

示す。

ここに見られるように、非選択的無効化処理を中心にある程度の効果が出ていると思われる。その理由として、分離によって全体的に命令実行速度が底上げされており、無効化処理の時に無効化すべき命令数が多いこの手法に特に効果が出ていると考えられる。しかし、より現実的な手法と言える $wakeup\ logic$ を用いた方法に対して効果的ではない。

図 12 は、3 章で述べた、投機ミスをした時の無効化処理の方法三種類それぞれの、非投機 ・非分離である default 対する有効性を IPC 効率にして示したものである。

また、投機的実行をする際のアドレス不一致予測に関して、予測をしないもの (blind) と、3 章で述べた StoreSet を用いたものの 2 種類で評価した。

ベンチマークにより大きな差が生じたが、おおむね効果があると言える。3

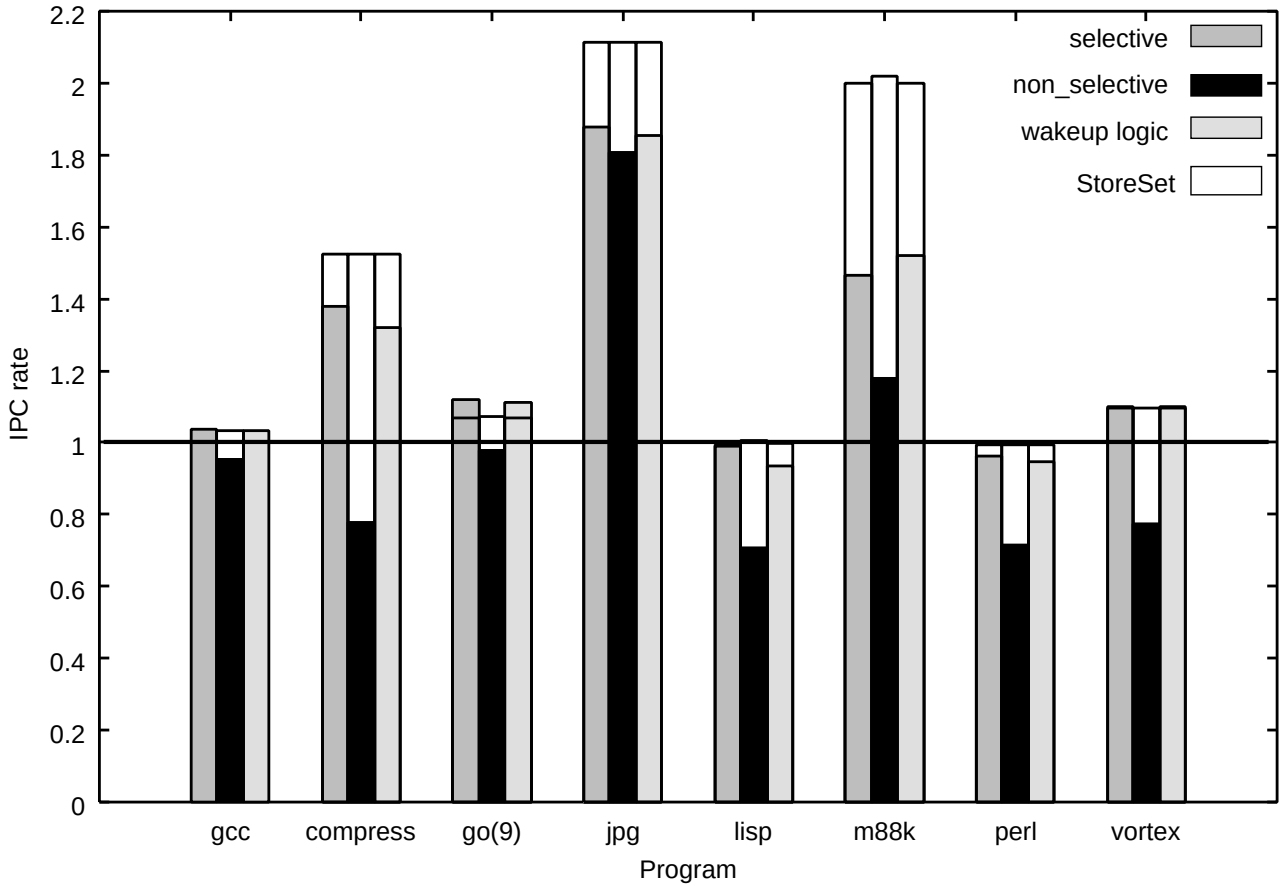


図 12: 無効化処理をした場合の評価 (分岐予測なし)

章で述べた通り、非選択的無効化処理は本来しなくてもよい命令に対しても再発行処理をしているため、blind で評価した場合に、他の二種類の方法に比べて IPC が低下していると考えられる。

図 13 は、上記のものに対して分岐予測を行ったものである。これにより、IPC 効率は全体的に前のものと比べて抑制されている。その理由として、分岐予測ミスによるペナルティが全体の IPC を制限しているのではないかと考えられる。分岐予測のミスペナルティは予測ミスした後の命令をフェッチからやり直すというもので、今回投機ミスに採用した無効化処理と比べるとその手間は大きい。よって、分岐予測ミスのペナルティが投機的実行や分離方式による参照アドレスの曖昧性に対する問題を解消する効果に比べて大きかったと考えられる。

StoreSet を用いた場合については、全体的に効率の底上げがなされている。これは、投機ミス率の低下が原因と考えられ、その徴候として無効化処理別での

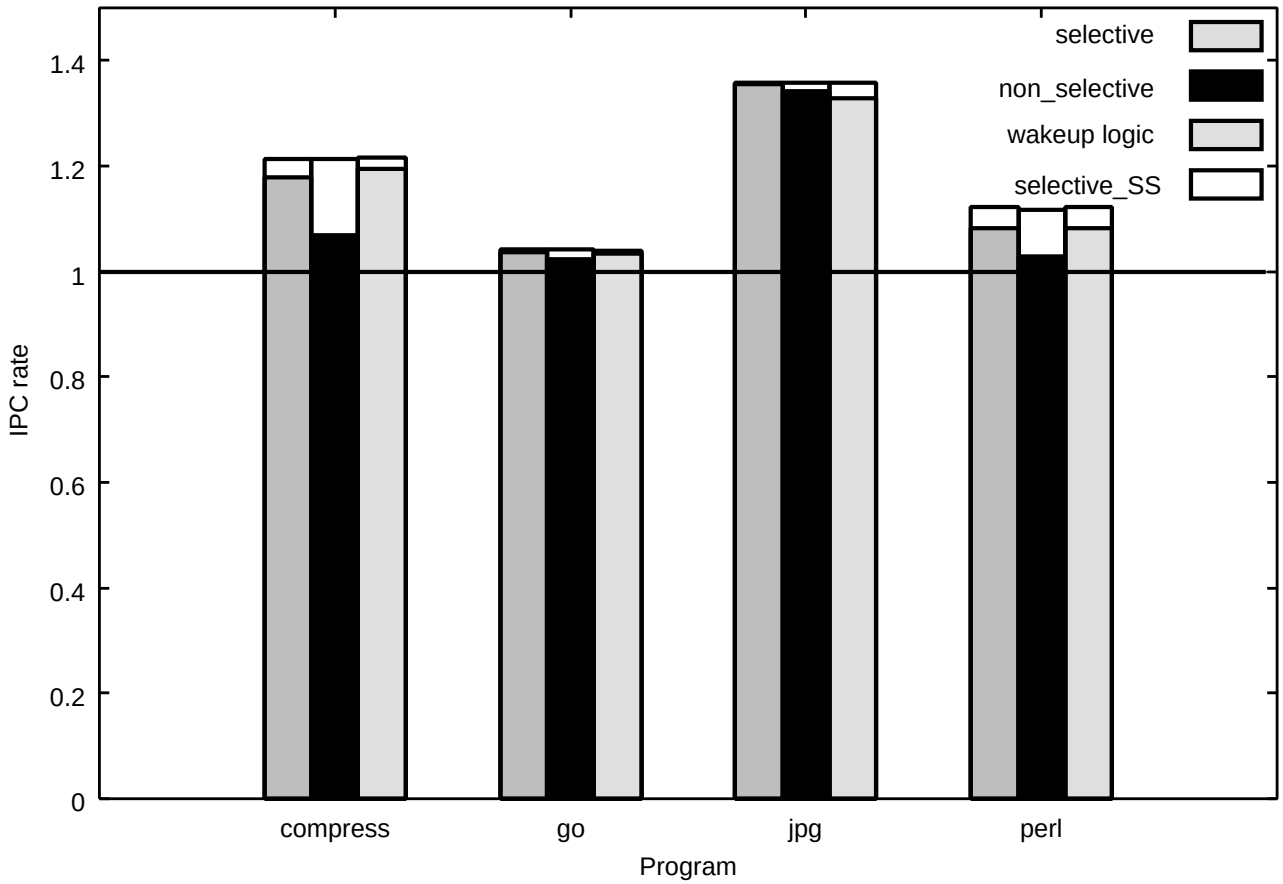


図 13: 無効化処理をした場合の評価 (分岐予測あり)

効率の変化がみられないことが挙げられる。無効化処理時の負担の違いにより、blind での評価では差がついたにも関わらず、StoreSet では評価に差があまりみられないのは、投機ミス自体の量が激減し、無効化処理のペナルティによる IPC の低下が無視できる程になっているからだと考えられる。

それを確かめるため、一部ベンチマークプログラムの各無効化処理方法について blind の場合と StoreSet を用いた場合で、投機的実行をした際の全体のロード命令発行回数に対する投機ミスを起こしたロード命令の数の割合を調べたところ、表 3 の様になった。

この表からもわかるように、確かに StoreSet を用いた場合には、投機ミス率を一律して 1%以下に抑える事に成功している。十分に投機ミス率が低下しているため、無効化処理によるミスペナルティが IPC に与える影響はほとんどなくなったと言える。その結果、上記の通り各無効化処理での隔差はなくなり、かつ全体的な IPC の向上につながった。

表 3: 各無効化方式における blind、および StoreSet の投機ミス命令の割合

	非選択的無効化		選択的一括無効化		*wakeup logic*	
	blind(%)	StoreSet(%)	blind(%)	StoreSet(%)	blind(%)	StoreSet(%)
compress	5.484	0.019	17.54	0.041	17.73	0.034
go	3.804	0.261	1.139	0.168	3.358	0.260
jpg	4.690	0.032	16.80	0.042	13.42	0.040
perl	2.931	0.039	7.066	0.050	6.974	0.049

第5章　おわりに

本稿では、ILP 向上の妨げとなる、参照アドレスの曖昧性を解消するための方法として、ロード・ストア命令の分離方式とロード命令の投機的実行の手法について述べた。ロード命令の投機的実行に際して、ロード命令の依存するストア命令を管理する StoreSet を用いてメモリの曖昧性の問題を解決する方法を述べた。投機的実行をミスした場合に必要となるハードウェア回復手法のうち、命令無効化処理の方法 3 種類についても述べた。この 3 種類の方法とは、実装が容易であるが必要以上の命令を無効化する、非選択的無効化方式、実装は困難だが必要最低限の命令を 1 サイクルで無効化処理が可能な、選択的一括無効化方式、動的命令スケジューリングに既存の *wakeup logic* を用いてサイクルごとに無効化すべき命令を選択していく方式、の三つである。これらは上記の StoreSet を用いた参照アドレス不一致予測をしなかった場合に顕著にその効果の違いがみられた。

前章でそれらについて評価を行った結果、分岐予測が完全に当たるとした場合には多少の例外は認められるものの、概して本研究の主題である参照アドレスの曖昧性を解消する事による ILP 向上は達せられていると考えられるが、分岐予測が外れる場合には、当研究で評価した方法は、分岐予測ミスのペナルティによってその効果を抑制されてしまっている事がわかった。

分離方式は、非分離のものに比べて平均的に IPC 効率で 1.5 倍程度であった。投機的命令実行では、投機ミスによる無効化処理でパフォーマンスが低下するものも見られたが、全体的に分離方式と同様に平均 1.5 倍程度の結果が得られた。

StoreSet を用いた予測は効果が高い。投機ミスの割合を、一律 1%以下まで抑

制する結果が得られた。これにより、上記の三種類の無効化処理によるパフォーマンスの低下割合が平坦化され、結果的に、ペナルティの負荷を十分に軽減することとなった。全体的なIPCの向上という観点からも満足のいく結果が得られたと考えられる。

今後の課題として、まず分岐予測を実装した上で、全てのベンチマークに対するこれらの評価、また無効化処理時にissue queue内で処理するのではなく、Recovery bufferを用いて行うことなどが挙げられる。

謝辞

本研究を進めるにあたり、四月より、一年を通じ多大な御指導を賜りました富田眞治教授に、深く感謝の意を表します。そして、日頃より熱心に指導していただき、本報告書の作成に対して、折に触れ多大なる助言を頂きました五島正裕氏に心から感謝致します。また、日頃より様々に支援して下さり、数多くの気兼ね無い助言を頂いた富田研究室の皆様、その他研究に関わった皆様に感謝します。

参考文献

[1] Glenn Reinman and Brad Calder "Predictive Techniques for Aggressive Load Speculation" (1998)

[2] George Z.Chrysos and Joel S.Emer "Memory Dependence Prediction using Store Sets" (1998)

[3] Enric Morancho,Jose Maria Llaberia and Angel Olive "Recovery Mechanism for Latency Misprediction" (2001)

[4] 佐藤 寿倫"命令再発行機構によるデータアドレス予測に基づく投機実行の効果改善" (1999)