社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2005－ARC－163 (6)
2005／5／31

命令ウィンドウをレベル0キャッシュと見做すマイクロアーキテクチャ

古川浩史 †　　大津金光 ‡

† 無所属
‡ 宇都宮大学工学部情報工学科

要旨

今日のスーパースカラプロセッサにおいては、高い命令レベル並列度と深いパイプラインのために、分岐予測ミスによる性能低下が深刻になっている。分岐予測の精度はすでに十分高く、これ以上精度を上げることは難しいため、今後の性能向上のためには分岐予測ミスのペナルティそのものを低減させる必要がある。
分岐予測ミスペナルティの低減には、二通りの手段が考えられる。一つは予測ミスした場合の命令フェッチのレイテンシを最小化である。
もう一つは control independence の利用である。予測ミスしたパスに沿って投機的に実行された control independent な命令を再利用することにより、不要な命令の再実行を最小化する。
本研究では、命令ウィンドウをレベル0キャッシュと見做すことにより、(ヒットした場合の) 命令フェッチのレイテンシの最小化および、不要な命令の再実行の削減を行う。命令がキャンセルまたはコミットされた後も命令ウィンドウの状態を保持しておき、直前の実行との差分のみを再実行する。

Instruction Window Considered as a Level-0 Cache

Hiroshi Furukawa †　　Kanemitsu Ootsu ‡

†Unaffiliated
‡Department of Information Science, Faculty of Engineering, Utsunomiya University

Abstract

Branch misprediction penalty brings severe performance degradation of today's superscalar processor with high ILP and deep pipeline. Branch prediction accuracy is high enough as there seems to be no room for drastic improvement of the accuracy, thus it is necessary to reduce branch misprediction penalty itself for further performance improvement.
Two techniques are supposed to reduce the penalty. One is to minimize instruction re-fetch latency on branch misprediction.
The other is to utilize control independence. Branch mispredcition penalty can also be reduced by reusing control independent instructions issued along wrong path.
This study propose to consider instruction window as a level-0 cache to minimize re-fetch latency (on cache hit) and unnecessary re-execution of control independent instructions. Instruction window keeps its state even after instructions are canceled or committed, and performes differential execution to its previous execution.

1 はじめに

今日のスーパースカラプロセッサはIPCの増大およびクロックサイクルの短縮化を高速化の手法としている。このようなプロセッサは、有効な命令の並列性を抽出するために大きな命令ウィンドウを持ち、また短いクロックサイクルを達成するために深いパイプラインを実装している。

大きな命令ウィンドウはより多くの命令の投機的実行を許し、また、深いパイプラインは命令フェッチのレイテンシを増大させる。つまり、これらの手法はいずれも分岐予測ミスのペナルティを増大させ、実効的なIPCの向上が頭打ちとなる最大の要因となっている。したがってさらなるIPCの向上のためには、分岐予測の精度を上げる、または分岐予測ミスペナルティを削減する、のいずれかまたは両方が必要である。

しかし、分岐予測精度を向上させる手法については数多くの研究がなされすでに極めて高い精度が達成されているため、これ以上の劇的な改善は見込めない。さらに、本質的に予測が困難な分岐が主に非数値計算プログラムにおいて数多く見られる。そのため、分岐予測ミスペナルティそのものを削減するための手法が重要である。

分岐予測ミスペナルティを削減する方法の一つに、命令フェッチのレイテンシの最小化がある。このために、予測困難な分岐について両方のパスを同時に実行する複数パス同時実行が提案されている。分岐が確定すると、選択されなかったパスについては実行をキャンセルする。選択されたパスについてはすでに実行が開始されているため、命令フェッチのレイテンシが最小化され、ペナルティの削減になる。こちらの方法は深いパイプラインによるペナルティの増大に対応する。

もう一つの方法は、命令のcontrol independenceを利用するものである。分岐の再合流地点以降の命令は、分岐の結果如何にかかわらず、常に実行される。このような命令を分岐に対してcontrol independentであると言う。通常のプロセッサでは、分岐予測ミス時には後続の命令をすべてキャンセルし、正しいパスに沿って命令を再フェッチする。つまり、control independentな命令であっても一度キャンセルされ再フェッチされることになる。したがって、無効なパスに沿ってフェッチされたcontrol independentな命令を何らかの方法により再利用し無駄な命令の再実行をなくせば、分岐予測ミスペナルティが削減される。こちらは大きな命令ウィンドウによるペナルティの増大に対応する。

本研究では、命令ウィンドウそのものをレベル0キャッシュと見做すことを提案する。レベル0キャッシュにヒットした場合には命令フェッチのレイテンシが最小化されることが期待される。

また、直前の実行と比較してデータ依存関係に変化があった命令のみを再実行することにより、control independence、ループおよび関数呼び出しにおいて命令の再利用を統一的に行う。

本稿の構成は以下の通りである。2章で提案する命令ウィンドウの構成について述べる。3章で関連研究を示し、4章でまとめる。

2 命令ウィンドウの構成

命令ウィンドウを命令キャッシュと見做して再利用するためには、以下のような解決すべき問題点がある。

1. 命令の連想検索。命令フェッチアドレスにより、命令ウィンドウ内を連想検索しなければならないが、命令ウィンドウのエントリ数が多くなると各エントリごとに検索するのは効率的ではない。
2. データ依存関係の変更。通常のプロセッサにおいては、同じ静的な命令であっても、実行パスが異なれば動的には異なる物理レジスタを参照する可能性がある。
3. 命令の選択的キャンセル·再実行。命令のcontrol independenceを最大限に利用するためには、両パスにおいてデータ依存関係が変わらない、つまりdata independentな命令については再実行せず、data dependentな命令のみを選択的にキャンセルや再実行できることが望ましい。

これらの問題点を解決するために、本研究ではセグメント化された命令ウィンドウを考える。

まず、[6]と同様に、命令ウィンドウを8から32命令程度のセグメントに分割する。各セグメントには命令を基本ブロックまたはトレース[5]を単位として割り当て、各命令のブロック内での位置を、命令ウィンドウセグメントの各エントリに対応させる。

基本ブロックまたはトレース内でのデータ依存関係の表現には依存行列 [10] を用いる。但し後述する理由のため、直接的な依存関係だけでなく、間接的な依存関係をふくめ推移的に求めた行列を用いる。各命令と命令ウィンドウエントリの対応は固定されているため、依存行列はブロックのフェッチ時に作成し、キャッシュしておくことも可能である。

ブロック内で参照されるレジスタのうち、そのブロックの外で値が定義されるものを live in レジスタと呼ぶ。Live in レジスタについては、データ依存関係は実行パスにより変化するため、通常の命令ウィンドウと同様の、物理レジスタタグによる連想的な機構によりデータの待ち合わせをおこなう。ブロック間のデータ依存関係は live in レジスタを用いて表わされる。

各命令ウィンドウセグメントが持つステートは、命令ウィンドウエントリのテーブルである分割された命令ウィンドウ本体、および各セグメントが固有に持つ付加的なステートからなる。命令ブロックの割り当て状況やブロック間のデータ依存関係がセグメント固有のステートとなる。

命令ウィンドウの各セグメント固有のステートを図 1 に示す。

tag	state	i0 tag	...	*in* tag

図 1: 命令ウィンドウセグメント固有のステート

tag は各ウィンドウが保持するブロックを表わし、命令アドレスまたはトレース ID を保持する。state はセグメントの状態を保持する。i*x* tag は各 live in レジスタに対応する物理レジスタ番号、または無効な割り当てを表す特別な値を保持する。(live in レジスタマップ)

図 2 にセグメント化された命令ウィンドウのエントリの構成を示す。

IR	D(row)	V	I	S
...	...	...	...	...

図 2: 命令ウィンドウエントリ

命令ウィンドウの各エントリは、命令レジスタ (IR)、依存行列 (D の一行)、および有効フラグ (V)、スコアボード (S)、発行許可フラグ (I) の 3 つのビットを持つ。有効フラグは、そのエントリが有効な命令を保持しているか否かを表す。スコアボードは、その命令の実行が完了し、有効な結果を保持しているか否かを表わす。発行許可フラグは、データが揃った場合にその命令を発行するか否かを表わす。発行許可フラグは、命令がエントリに新たに挿入された場合、および命令がキャンセルされた場合にセットされ、命令が発行されるとクリアされる。

命令ウィンドウの各セグメントは以下の 3 つの状態を取る。

1. 無効。セグメントにはブロックは割り当てられていない。
2. 実行中。セグメントにはブロックが割り当てられており、in flight な状態である。
3. 有効。セグメントにはブロックが割り当てられているが、コミットしたかキャンセルされた。

「無効」は初期状態である。「実行中」は in flight な命令を保持している状態である。「有効」は、命令ウィンドウ内には有効な命令を保持しているが、すべての命令がキャンセルされた、またはコミットされた状態であり、in flight な命令を保持してない状態である。

各命令ウィンドウセグメントのヒット・ミス判定は以下のように行なう。

プロセッサのフロントエンドは命令フェッチアドレス (またはトレース ID) を各セグメントのタグと比較し、タグが一致し、かつセグメントの状態が「有効」である場合は、セグメントにヒットし、そのセグメントを再利用することができる。セグメントにヒットした場合には、まず、そのセグメントの状態を「実行中」にする。また、最新のレジスタマップを用いてセグメントの live in レジスタマップを更新する。物理レジスタの割り当てに変更があった Live in レジスタすべてについて、そのレジスタに直接または間接的に依存している命令は再実行される必要がある。

それ以外の場合は、ミスした場合である。「無効」または「有効」なセグメントを一つ選択し、フェッチした命令ブロックを割り当て、レジスタマップから live in レジスタマップを設定し、状態を「実行中」にし、実行を開始する。ここで、対象となるセグメントを選択するアルゴリズムは、キャッシュのリプレースメントポリシーに相当し、様々なものが考えられる。

すべてのセグメントが「実行中」であることも考えられるが、この場合には、命令供給をストールさせる。もしくは、フロントエンドがout of orderなフェッチを行なう場合にはデッドロックの可能性があるため、投機実行中のセグメントをキャンセルするといったことが必要である。

セグメントは実行コンテクストを保持するため、タグが一致した場合でも、状態が「実行中」であれば、多重にブロックを割り当てることはできない。

「有効」なセグメントに再割り当てした場合には、キャッシュのフラッシュに相当する動作を行う必要がある。対応する物理レジスタを開放し、当該レジスタの無効化をブロードキャストする。各セグメントはブロードキャストされたレジスタ番号をlive inレジスタマップと比較し、一致するものについて無効化を行なう。

分岐予測ミス時には、予測ミスした分岐命令以降を含むセグメントの実行をキャンセルし、状態を「有効」にする。キャンセルされたセグメントの命令ウィンドウの各エントリの状態はそのまま保持する。

命令のコミットもセグメント単位で行なう。セグメントがコミットされると状態は「有効」になり、キャンセル時と同様に命令ウィンドウエントリの状態はそのまま保持される。

以下で例2を用いた説明を行なう。

	D							
	i1	i2	r1	r2	r3	r4	I	S
i1								1
i2								1
r1 = i1 + 1	1						0	1
r2 = r1 - i2	1	1	1				0	1
r3 = r2 + r1	1	1	1	1			0	1
r4 = i2 * 5		1					0	0
r5 = r4 + r4		1				1	1	0

D: 依存行列
I: 発行許可フラグ
S: スコアボード
有効フラグは省略した。

図 3: 命令ウィンドウの動作例

i1およびi2がLive inレジスタ、r1からr5までがブロック内で定義されるレジスタである。r3を定義する命令までが実行を完了し、r4の命令が発行済みで未完了、r5の命令が未発行である。

依存行列Dの求め方は以下の通りである。

$$D(n) \leftarrow \{n_l, n_r\}|D(n_l)|D(n_r)$$

$D(n)$は依存行列のn番目の行を表わし、n_lとn_rはそれぞれn番目の命令の左右オペランドを表わす。依存行列はレジスタリネーミングと同様の機構で求めることができる。

発行可能な命令は以下の通りに求めることができる。

$$W = (D * (S|C))\&I$$

ここでWは発行可能な命令の集合、Cはそのサイクルで実行完了した命令の集合のビットベクトルである。行列とビットベクトルの積については、行列の各行の1である要素に対応するビットベクトルの要素がすべて1であるときに、積は1であり、それ以外は0とする。これは、各命令について、直接または間接的に依存するデータがすべて揃っている場合に発行可能とすることを意味する。例2において、$D(r2)$は、i1,i2,r1に対応する要素が1であるが、ビットベクトルSの対応する要素はすべて1であるため、積$D(r2) * S$は1となる。$D(r5)$では、i2とr4に対応する要素が1であるが、Sのr4に対応する要素は0であるため、積$D(r5) * S$は0となる。積が1となったもののうち、発行許可フラグがクリアされているものは、すでに発行済みであるため、発行可能な命令からは除外する。

同様に、キャンセル・再実行すべき命令は以下の通りである。

$$R = \sim (D* \sim C_{inv})$$

Rはキャンセル・再実行すべき命令の集合、C_{inv}は無効化されたレジスタの集合である。これは、各命令について、その命令が直接または間接的に依存しているデータのうちの一つでも無効化された場合にキャンセル・再実行することを表わす。例2において、live inレジスタi1のレジスタマッピングが変更されたとすると、それに依存する命令r1,r2,r3がキャンセル・再実行の対象となる。

ここで無効化されるレジスタはlive inレジスタに限定されない。ブロック内で値を定義されるレジスタであっても、ロード命令のレイテンシ予測やメモ

リ依存性予測、値予測などのミスにより、無効化・再実行される可能性がある。

発行許可フラグは以下の様に更新される。

$$I \leftarrow (I\& \sim select(W))|R$$

ここで $select(W)$ は、発行可能な命令のうち、実際に発行されたものを示す。すでに発行された命令については発行許可フラグをクリアし、再実行すべき命令についてはセットする。

以下で本提案の問題点について若干の考察を行なう。

提案する命令ウィンドウの構成は、[10] を若干修正したものであるから、実装上の大きな困難は特にないと思われる。ただし、Live in レジスタについてはタグとの比較の後段に依存行列との演算による待ち合わせ回路 (wired OR) がくるために、遅延が大きくなる可能性がある。また、タグ数の削減のため、live in レジスタの最大数は小さいことが望ましいが、これはブロックサイズの低下を招き、命令ウィンドウの利用効率が低下する。

命令ウィンドウの高い再利用率が可能になるためには、正しい位置でブロックまたはトレースを区切らなければならない。ブロックやトレースを生成するのはフロントエンドであるので、具体的なアルゴリズムについては本稿では述べないが、一般的には再利用性を高めようとするとブロックが細切れになり、やはり効率が低下する傾向にある。

3 関連研究

Cher ら [1] は、Skipper と呼ばれるアーキテクチャを提案している。これは予測困難な分岐命令を実行後、再合流地点までスキップしそこから実行を再開する。分岐の確定後に正しいパスを実行する。再合流地点以降の命令が、スキップされる命令にデータ依存する場合のために、物理レジスタの preassign という技法を用いる。スキップされた領域をフェッチする際、領域から live out する論理レジスタについては、preassign された物理レジスタにマップすることにより、データ依存関係を保証する。再合流地点やスキップされる領域についての情報は、テーブルを用いて実行時に学習する。

Gandhi ら [3] は exact convergence と呼ばれる場合について control independence を利用するアーキテクチャを提案している。Exact convergence とは、以下のようなプログラムにおいて、分岐しないと予測したが、実際は分岐したため B において再合流する場合を表す。

```
if (cond) {
  A;
}
B;
```

分岐の確定時に命令ウィンドウ内の A に相当する命令列を無効化し、B から再実行を行うが、A を経由しないことによって再合流地点におけるレジスタマッピングが異なり、偽の依存関係が発生する。そこで、A 内のレジスタを定義する命令を move 命令に変換して再実行することにより、偽の依存関係を解決する。再合流地点の検出には fully associative なバッファを用いる。また、偽の依存関係が発生した命令およびそれに依存する命令についてのみ選択的に再実行することも提案されている。

Chou ら [2] は、リオーダバッファに似た Dynamic Control Independence(DCI) Buffer という機構を提案している。DCI Buffer は、リオーダバッファの各エントリと一対一に対応するエントリを持ち、通常実行時にはリオーダバッファと同じく確保される。分岐予測ミスが起きると、正しいパスを実行しつつ、命令フェッチアドレスをキーに再合流地点を DCI Buffer 内に検索する。再合流地点が見つかった場合、正しいパスと過ったパスの両方について、それぞれのパスで定義されるレジスタ集合を求める。いずれのパスにおいても定義されないレジスタのみに依存した命令は control independent かつ data independent であるので、再実行の必要はない。

Sodani ら [8] は動的な命令の再利用を提案している。オペランドのレジスタの値そのものをキーにするもの、レジスタ名をキーにするもの、およびレジスタ名と dependence chain を用いたものの三種類がある。自然な形で control independence を利用できるが、実現の困難なハードウェアを必要としている。本稿で提案した機構はレジスタ名と dependence chain を用いた命令の再利用と考えることもできる。

Rotenberg ら [7] は、Trace Processors[6] における control independence の利用について述べている。オリジナルの Trace Processors に対する変更点および新しいトレース生成アルゴリズムが提案され

ているが、選択的再実行を実現する機構については特に述べられていない。

Morancho ら [4] は、レイテンシ予測ミスが起きた場合の選択的再実行を実現する機構を提案している。レイテンシ予測ミスが起きる可能性のある命令それぞれについて、その命令に直接または間接的に依存する命令を表すビットベクトルを推移的に構成し、予測ミス時の選択的な再実行を実現する。また、命令ウィンドウ内での依存関係の表現には五島ら [10] のものと同様な依存行列を用いている。本稿で提案されているものと似た機構であるが、control independence の利用はできず、また、命令ウィンドウがキャッシュとして機能するわけでもない。

Multiscalar[9] や SKY[11] も control independence に着目したアーキテクチャであるが、いずれもコンパイラによるサポートが必要である。

4 まとめ

分岐予測ミスペナルティを削減するために、命令ウィンドウをレベル 0 キャッシュと見做すマイクロアーキテクチャを提案した。提案される機構は、Trace Processors[6] などと同様に命令ウィンドウをセグメント化する。

各セグメントは命令アドレスまたはトレース ID のタグを持ち、命令フェッチアドレスと一致し、かつセグメントが in flight な命令を保持していない場合は、有効な命令を保持する命令ウィンドウを再利用ことにより、命令フェッチのレイテンシを最小化する。

ブロック間の命令の依存関係は、実行パスにより依存関係が変化するため、通常の命令ウィンドウと同様に物理レジスタタグによりデータの待ち合わせを行う。ブロック内の命令の依存関係は固定されたものであるので、依存行列を用いた待合せを行う。

さらに、依存行列で直接または間接的な依存関係を推移的に表わすことにより、データ依存関係の変化や予測ミスなどによる命令の選択的なキャンセルおよび再発行を高速に行なえるようにした。

問題点としては、連想タグ比較の後に wired OR によるデータの待ち合わせ回路が入るため、遅延が大きくなる可能性があることと、高速化のために連想タグ数を減らすとブロックサイズの低下を招き、効率が下がること、命令ウィンドウの再利用率を高めるようなフロントエンド戦略もブロックサイズを低下させる傾向にあることである。

スペースの都合により性能評価が行えなかったため、今後の課題とする。また、命令ブロックのセグメントへの割り当てアルゴリズムについては単純な LRU ポリシー以外にも、予測困難な分岐以降を優先的に割り当てるといったものも考えられるので、検討を加えたい。

参考文献

[1] Chen-Yong Cher and T. N. Vijaykumar. Skipper: A microarchitecture for exploiting control-flow independence. In *Proceedings of MICRO 34*, 2001.

[2] Yuan Chou, Jason Fung, and John Paul Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of ICS'99*, 1999.

[3] Amit Gandhi and Haitham Akkary. Reducing branch misprediction penalty via selective branch recovery. In *Proceedings of HPCA'04*, 2004.

[4] Enric Morancho, José María Llabería, and Àngel Olivé. Recovery mechanism for latency misprediction. In *Proceedings of PACT'01*, 2001.

[5] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceesings of MICRO 29*, 1996.

[6] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *Proceedings of MICRO 30*, 1997.

[7] Eric Rotenberg and Jim Smith. Control independence in trace processors. In *Proceedings of MICRO 32*, 1999.

[8] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of ISCA 24*, 1997.

[9] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of ISCA 22*, 1995.

[10] 五島正裕, 西野賢悟, グェン・ハイハー, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治. スーパースケーラのための高速な命令スケジューリング方式. 情報処理学会 HPS 論文誌, Vol. 42, No. SIG 9(HPS 3), pp. 77–92, 2001.

[11] 小林良太郎, 小川行宏, 岩田充晃, 安藤秀樹, 島田俊夫. 非数値計算応用向けスレッド・レベル並列処理マルチプロセッサ・アーキテクチャsky. 情報処理学会論文誌, Vol. 42, No. 2, pp. 349–366, 2 2001.