計算機アーキテクチャ 125－19
（1997. 8. 21）

レジスタアクセスを軽減するための アーキテクチャサポートの検討

高木 秀樹† 李 鼎超‡ 石井 直宏†

†名古屋工業大学知能情報システム学科
‡名古屋工業大学情報処理教育センター
†E-mail: {takagi, ishii}@egg.ics.nitech.ac.jp
‡E-mail: liding@center.nitech.ac.jp

プログラム中で使用される変数の大部分は、その生存区間がわずか数命令であるという意味で、short である。スーパースカラプロセッサでは、これらの short な変数は、リオーダバッファ内で処理される。これらの変数は、レジスタファイルから読み出されないため、レジスタファイルに書き込むのは無駄である。本稿では、このような無駄な書き込みを軽減し、レジスタアクセスを抑制するためのアーキテクチャサポートの検討を行う。

Architecture Support to Reduce Register Access

Hideki Takagi † Dingchao Li ‡ Naohiro Ishii †

†Department of Intelligence and Computer Science, Nagoya Institute of Technology
‡Educational Center for information processing, Nagoya Institute of Technology
†E-mail: {takagi,ishii}@egg.ics.nitech.ac.jp
‡E-mail: liding@center.nitech.ac.jp

A significant number of program variables are short in the sense that their live range are only few instructions . In superscalar processors using the reorder buffer, their live range may occur within the reorder buffer. Because none of value produced by these variable, would be ever obtained from the register file, they do not need to be written back to the register file. In this paper, we present the architecture support to reduce the number of register access.

1 はじめに

命令レベルでの並列実行が可能な計算機アーキテクチャの代表的なものとして、スーパースカラアーキテクチャがある。スーパースカラアーキテクチャでは、動的スケジューリングや投機的な実行、レジスタリネーミングなどの手法によって動的に命令レベルでの並列性の抽出を行なう。このような、動的に命令レベルの並列性を抽出するために、スーパースカラプロセッサに実装されているハードウェアの1つに、リオーダバッファがある。

スーパースカラプロセッサでは、機能ユニットで生成された変数の値は一旦リオーダバッファに入れられる。そして、リオーダバッファからリタイアした後でレジスタに書き込まれる。そのため、リオーダバッファのエントリのサイズよりも生存区間が短い変数は、リオーダバッファ内でその生存区間を終え、レジスタからは読み出されない。これらの変数をレジスタファイルに書き込むことは、無駄な書き込みとなる。

CPU 内に実装されているレジスタは、計算機の所有する記憶装置の中で最も高速に動作する記憶装置であり、CPU の持つ限られた資源の一つである。命令レベルでの並列実行が可能な複数の機能ユニットを有する CPU において、機能ユニットを有効に利用するためには、機能ユニットの数に比例してレジスタを多く用意する必要がある。しかしながら、レジスタの実装には、物理的な制約が存在するので、実装可能なレジスタ数は制限されている。

本稿では、限られた資源であるレジスタをより効率良く利用するために、無駄な書き込みを軽減するための、アーキテクチャサポートの検討を行なう。

本稿は、以下 4 章から構成される。2 章では、無駄な書き込みをより詳しく説明し、無駄な書き込みを行なう変数である short-lived 変数の定義を述べる。3 章では、本稿で提案した、アーキテクチャサポートについて述べる。そして 4 章では、その有効性について評価を行う。最後に 5 章では、まとめと今後の課題について述べる。

2 無駄な書き込み

プログラム中で使用される変数の大部分は、その生存区間がわずか数命令であるという意味で short である。このことは非常に重要である。

なぜなら、リオーダバッファを利用したスーパースカラプロセッサにおいては、これらの生存区間が数命令の変数は、リオーダバッファ内にのみ存在することになるからである。

このような、生存区間がリオーダバッファ内にのみ存在する変数をレジスタファイルに書き込むことは、無駄な書き込みとなる。なぜなら、これらの変数はレジスタファイルから読み出されることはないからである。

2.1 無駄な書き込みの例

機能ユニットによって作られた値の大部分は完全にリオーダバッファ内でのみ使われる。例として、livermoore のベンチマークプログラムを考える。図 1は C 言語で記述されたベンチマークプログラムを表す。図 2はそのプログラムをアセンブラコードに変換した結果である。図 3は、それぞれの変数のレジスタの割り付けとその生存区間を表している。○は定義ポイントを、×は使用ポイントを表す。

```
    i++;
x[i] = x[k]-v[k]*x[k-1]-v[k+1]*x[k+1];
```

図 1: C プロ グラム

命令番号	演算命令	
i1:	add	R1 R1 1
i2:	load	R3 v[R2]
i3:	sub	R4 R2 1
i4:	load	R4 x[R4]
i5:	mult	R4 R4 R3
i6:	add	R3 R2 1
i7:	load	R5 v[R3]
i8:	load	R3 x[R3]
i9:	mult	R3 R5 R3
i10:	sub	R3 R3 R4
i11:	load	R4 x[R2]
i12:	sub	R3 R3 R4
i13:	store	x[R1] R3

図 2: コード

これらから、いったん値が新しく定義されたならば、数命令後にはすぐに値の最後の使用ポイントが来るという事がわかる。

生存区間の長さを、定義ポイントから最後の使用ポイントまでの命令の数とすると、プログラム内で生成される値の最も長い生存区間の、長さは命令 i1 で定義される R1 の 12 命令である。

このプログラムを、16エントリのリオーダバッファを持つマシン上で実行するとすると、生成された値は、すべてリオーダバッファ内で処理される。この場合、これらの値をレジスタファイルに書き込む必要は無い。なぜなら、値がレジスタファイルから読み出されることは無いからである。

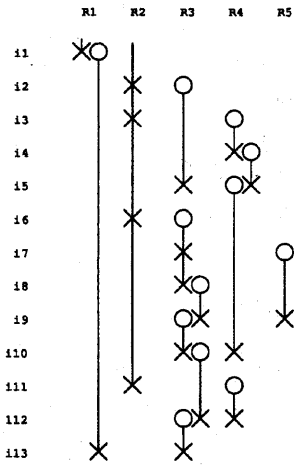

図 3: 生存区間

2.2 short-lived 変数の定義

リオーダバッファ内にのみに生存区間が存在する変数を short-lived 変数と呼ぶ、以下にその定義を示す。

- $d_0, d_1, d_2 \cdots d_n$ を変数の集合 $v_0, v_1, v_2, \cdots v_n$ の定義ポイントとする。
- $u_{i_0}, u_{i_1}, u_{i_2} \cdots u_{i_m}$ を変数 v_i の使用ポイントとする。
- v_i の生存区間を r_{i_j} とし $r_{i_j} = [d_i, u_{i_j}]$ とする。
- r_{i_j} の長さ $L(r_{i_j})$ を区間 $[d_i, u_{i_j}]$ 間の最も長いパスの命令の数とする。
- リオーダバッファのエントリの長さが LR、デコード幅が DB であるマシン M 上であるとすると、$L(r_{i_j}) \leq LR - DB$ ならば生存区間 r_{ij} は short である。
- v_i の全ての生存区間が short である場合、v_i は short-lived 変数である。

リオーダバッファはインオーダーでのリタイアを保証しているため、最後のエントリの命令が、最後の使用を待つ場合、他の命令のリタイアをストップさせてしまう。この場合、新たな命令をリオーダバッファに入れることができなくなってしまう。これを防ぐために、生存区間が short であると判断する基準を LR-DB とする。

3 アーキテクチャサポート

本節では、2章で説明した無駄な書き込みを減少させるためのアーキテクチャサポートの検討を行なう。

本アーキテクチャでは実行するコードとして、Luis A Lozano C らによって提案されたレジスタ割り付け [1] によって生成されたコードを採用する。このコードは short-lived 変数を仮のレジスタであるシンボリックレジスタに割り付けることでレジスタの使用数を減少させる。本稿では、このシンボリックレジスタを扱い、short-lived 変数を明確に破棄することで無駄な書き込みを減少させる。

本章では、リオーダバッファ、リザベーションステーションに対する、short-lived 変数を扱うための拡張を示す。次に、実行過程のアルゴリズムの修正を行なう。そして、short-lived 変数を破棄する方法について述べる。

3.1 ハードウェアの構成

本稿で提案するハードウェアの構成の一部を図 4に示す。命令は発行されると、一旦命令キューにいれられる。リザベーションステーションは、命令コード、オペランド、および、ハザード検出・解消に必要な情報を保持する。ロード・バッファは, 実行未完了のロード命令のロード結果を保持する。同様にストア・バッファはオペランド待ちによる実行未完了のオペランドアドレスを保持する。機能ユニットおよびロード・ユニットからの命令実行結果は、共通データバス (CDB) 経由でリオーダバッファ、リザベーションステーション、およびストア・バッファに送られる。ADD FU では加減演算が、MULT FU では乗除演算がそれぞれ実行されるものとする。retire check は、リオーダバッファからリタイアした変数のレジスタへの書き込み及び、破棄の判断を行なう。

3.1.1 リオーダバッファ

リオーダバッファの各エントリは、次の4つのフィールドから構成される。

- E_i ：エントリ番号。

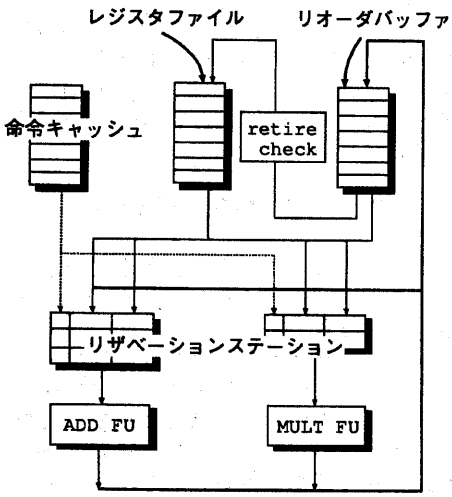

図 4: CPU ブロック図 (一部)

- R_i ：V_iに書き込みを行なう命令の定義レジスタの番号を格納する。
- V_i ：機能ユニットによって生成される値を格納する。
- Q_i ：値を生成するザベーションステーションの名称を格納 (オペランドフェッチ、結果書き込みに使用される)。E_iに書き込まれた実行結果の値を格納する。
- tag：short-lived であれば yes、そうでなければ no を格納する。

表 1: リオーダバッファ

E_i	R_i	V_i	Q_i	tag
E1				
E2				
E3				
⋮				
E16				

3.1.2 リザベーションステーション

リザベーションステーションは、次の 6 つのフィールドから構成される。

- **Busy**： 対応する機能ユニットおよび、当該リザベーションステーションが実行中であれば yes、そうでなければ no を格納する。
- $\mathbf{F_m}$： ソースオペランド S1,S2 に対して施す演算の種類を格納する。
- $\mathbf{Q_j, Q_k}$： ソースオペランドを生成するリザベーションステーションの番号 (値が 0 の場合は、当該オペランドがすでに V_jまたは V_kフィールドに存在していることを示す) を格納する。
- $\mathbf{V_j, V_k}$ ： ソースオペランドの値を格納する (各ソースオペランドに対しては、V フィールドと Q フィールドのどちらか一方の有効となる。即ち、V フィールドに値がある場合、Q フィールドは 0 となる)。

3.2 アルゴリズム

3.2.1 実行ステージ

実行ステージ数は、3 ステージとする。

- **命令発行 (issue)**
  リザベーションステーションに空きがあれば命令を発行する。また、同時にリオーダバッファにもこの命令のためのエントリを作成する。発行時に、既にソースオペランドが生成されていればオペランドも一緒にリザベーションステーションに格納する。
- **実行 (execute)**
  ソースオペランドが揃った時点で実行を開始する。揃うまではリザベーションステーションで待機する。
- **結果書き込み (write result)**
  実行を完了したら、実行結果をリオーダバッファに書き込む。同時に、(実行結果を必要としている) 各リザベーションステーションにも実行結果を送る。

3.2.2 実行過程のアルゴリズム

ここでは、実行過程のアルゴリズムを示す。まず、ここで使用する記号を説明する。

D は定義レジスタ番号、S1,S2 はソースレジスタ番号 、r はリザベーションステーションの番号、E は自分が格納されるリオーダバッファの番号を示す。また、RS はリザベーションステーションの管理情報、Buffer はリオーダバッファの管理情報、Store はストアバッファの管理情報を表す。

命令発行 (issue)

- 次のステージに進むための条件
  リザベーションステーションまたはバッファに空きがあること。
- 動作および管理情報の更新作業
  $Buffer[E].Q_i = r$
  $Buffer[E].R_i = D$
  オペランドフェッチ作業を行なう
  $RS[r].Busy \leftarrow yes$

実行 (execute)

- 次のステージに進むための条件
  $RS[r].Q_j = 0$ かつ $RS[r].Q_k = 0$
- 動作および管理情報の更新作業
  なし

結果書き込み (write back)

- 次のステージに進む判定条件
  実行が完了し、CDB の使用権が確保できていること。
- 動作および管理情報の更新作業

$\forall x\ if\ ((Buffer[x].Q_i = r) \wedge (Buffer[x].V_i = no))$
$Buffer[x].V_i \leftarrow result$
$\forall x\ if\ (RS[x].Q_j = r)$
$RS[x].V_j \leftarrow result$
$RS[x].Q_j \leftarrow 0$
$\forall x\ if\ (RS[x].Q_k = r)$
$RS[x].V_k \leftarrow result$
$RS[x].Q_k \leftarrow 0$
$\forall x\ if\ (Store[x].Q_i = r)$
$Store[x].V \leftarrow result$
$Store[x].Q_i \leftarrow 0$
$RS[r].Busy \leftarrow no$

3.2.3 オペランドフェッチのアルゴリズム

ここでは、オペランドフェッチのアルゴリズムを示す。オペランドフェッチは命令キャッシュからリザベーションステーションへ命令が発行される時点で行われる。ソースオペランドは、最初リオーダバッファ内で探される、リオーダバッファに該当するエントリがない場合のみ、レジスタから値が読み出される。また、値を生成中の場合は、リオーダバッファの Q フィールドを参照することにより、生成中のリザベーションステーションの名称を判断することができる。

```
if (Buffer[x].R_i = S1)   [x > E]
    最小のエントリ番号 i を選ぶ
      if (Buffer[Ei].V_i ≠ no)
        RS[r].V_j ← Buffer[Ei].V_i
        RS[r].Q_j ← 0
      else
        RS[r].Q_j ← Buffer[Ei].Q_i
  else
      RS[r].V_j ← Register[S1]
      RS[r].Q_j ← 0
```

3.3 short-lived 変数の破棄

リオーダバッファからリタイアした命令に対して、その値をレジスタファイルに書き込むか、破棄するかの判断する必要がある。
値の破棄の判断は、以下の様に行う。

- 値の破棄の条件
  値が Short-lived ならば破棄する (即ち、$Buffer[E16].tag = yes$ ならば、値は破棄する)。そうでなければ、値をレジスタに書き込む。

4 評価

ここでは、プログラム中の変数のどの程度破棄できるかを調べ、本稿で提案したアーキテクチャの有効性の評価を行なう。

4.1 破棄できる変数

livermoore の 24 個のループに対して、14 個を選び、それぞれにループアンローリングを 5 回行なったものに対して、実験を行った。実験の主要なパラメータを表 2に示す。
表 3の結果から分るようにリオーダバッファが 16 エントリのの時には 90%近くが short-lived 変数である。また、32 エントリでは 90% 以上が short-lived 変数である。このことから、プログラム中の変数の大部分が short-lived 変数であるという考察は正しいと言える。

4.2 実行例

図 1を実行する場合を例にとって有効性を示す。i7 のロード命令の書き込みが完了し、i8, i9, i10 の命令がリザベーションステーションに発行された時点に注目する。表 4は発行前のリオーダバッファの状態、表 5, 表 6は発行後のリオーダバッファ、リザベーションステーションの状態を表している。
まず、i8, i9, i10 のためのエントリが E3, E2, E1 に作られ、E8 に達するエントリはリタイ

表 2: パラメータ

パラメータ	値
リオーダバッファ	16 サイズ
デコード幅	1 サイクル 4 命令
メモリ遅延	2 サイクル
メモリアクセス	1 サイクル 1 命令
レジスタ数	16 個

表 3: 評価結果

Benchmark	8 entries	16 entries	32 entries
L1	77.8%	84.7%	91.7%
L2	71.1%	81.9%	93.9%
L3	66.7%	87.2%	89.7%
L4	55.8%	86.0%	86.0%
L5	86.8%	96.2%	96.2%
L6	75.0%	82.4%	91.2%
L7	71.8%	93.3%	93.9%
L8	80.9%	87.1%	90.2%
L9	62.1%	94.8%	94.8%
L10	70.6%	93.6%	93.6%
L11	70.7%	90.7%	97.3%
L12	72.2%	88.8%	94.4%
L18	74.4%	87.2%	93.6%
L23	69.1%	88.2%	91.6%
average	71.3%	89.8%	92.9%

アする。即ち、E6, E7 に存在したエントリがリタイアする。リタイアした値は retire check によって tag が判断される。ここでは E6 の値は破棄され、E7 の値はレジスタに書き込まれる。

次に、ソースオペランドのフェッチが行われる。i8, i9 のソースオペランドは全て実行中のため、リザベーションステーションの Q フィールドに、実行中のリザベーションステーションの値がセットされる。どのリザベーションステーションで実行を行っているかは、リオーダバッファの Q フィールドから判断できる。また、値の書き込みもこの Q フィールドを調べることで行う。即ち、リオーダバッファからの値のやりとりは、全て Q フィールドで行う。このため、シンボリックレジスタと物理レジスタを同じに扱うことができる。

以上の結果から、本稿で提案したアーキテクチャは、無駄な書き込みを行う変数を明確に破棄でき、シンボリックレジスタを扱うことができることが分る。

5 まとめ

本稿では、無駄な書き込みを軽減するためのアーキテクチャサポートを行った。これによ

表 4: リオーダバッファの状態 1

E_i	R_i	V_i	Q_i	tag
E1	SR3		load1	yes
E2	SR1	値	add3	yes
E3	R3		mult1	no
E4	SR2	値	load2	yes
E5	SR2	値	load2	yes
E6	SR1	値	load1	yes
E7	R1	値	add1	no

表 5: リオーダバッファの状態 2

E_i	R_i	V_i	Q_i	tag
E1	SR1		add1	yes
E2	SR1		mult2	yes
E3	SR1		load2	yes
E4	SR3		load1	yes
E5	SR1	値	add3	yes
E6	R3		mult1	no
E7	SR2	値	load2	yes
E8	SR2	値	load2	yes

表 6: リザベーションステーション

名称	Busy	F_m	V_j	V_k	Q_j	Q_k
add1	yes	sub			mult1	mult2
add2	yes	sub	値	値		
add3	yes	add	値	値		
mult1	yes	mult	値	値		
mult2	yes	mult			load2	load1

り、short-lived 変数を明確に破棄し、無駄な書き込みを軽減することができる。サポートは、通常スーパースカラプロセッサで使われているリオーダバッファとリザベーションステーションを利用して行った。このため、ハードウェアの拡張のためのコストは非常に少なくてすむ。

今後の課題としては、オペランドフェッチのアルゴリズムの改良、そして、ループの実行時における short-lived 変数の破棄などが挙げられる。

参考文献

[1] Luis A.Lozano C and Fuang R. Gao "Exploiting Short-Lived Variables in Superscalar Precessors" IEEE Proceedings of MICRO-28,1995

[2] James K.Pickett, David G.Mayer, and PhD. "Enhanced superscalar hardware : The schedule table." *Supercomputing*, Nov 1993.