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

2F－1

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

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。

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。

2 short-lived 変数

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。

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 変数と呼び、以下にその定義を示す。

- 変数 v_iの定義ポイントを d_i、使用ポイントを $u_{i_0}, u_{i_1}, u_{i_2} \cdots u_{i_m}$ とする。
- v_iの生存区間を r_i とし $r_{i_j} = [d_i, u_{i_j}]$ とし、その長さ $L(r_{i_j})$ 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 であるとすると、$L(r_{i_j}) \leq LR - DB$ ならば生存区間 r_{i_j} は short である。全ての生存区間が short である時、変数 v_i は short-lived 変数である。

3 アーキテクチャサポート

本章では、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 らにより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。

An Investigation of Architecture Support for Reducing Unnecessary Register Access
Hideki Takagi, Dingchao Li,Naohiro Ishii
Nagoya Institute of Technology, Gokiso-cho, Syowa-k, Nagoya 466, Japan

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 1に示す。命令は発行されると、一旦命令キューに入れられる。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。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。

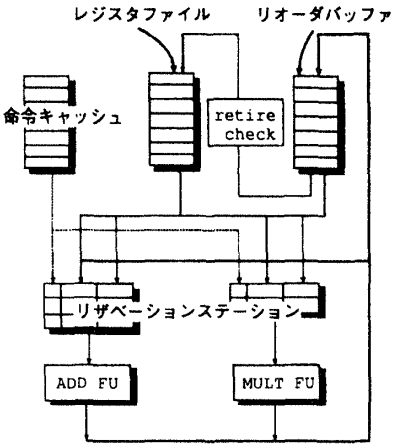

図 1: CPU ブロック図 (一部)

3.1.1 リオーダバッファ

リオーダバッファは R_i, V_i, Q_i, tag の 4 つのフィールドから構成される。

R_iには命令の定義レジスタの番号を、V_iには生成される値を格納する。そして、Q_iには値を生成するザベーションステーションの名称を格納する。tag には short-lived であれば yes、そうでなければ no を格納する。

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、Busy, F_m, Q_j, Q_k, V_j, V_k の 6 つのフィールドから構成される。

Busy には実行中であれば yes, そうでなければ no を格納する。F_mにはソースオペランド S1,S2 に対して施す演算の種類を、Q_j, Q_kには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を格納する。V_j, V_kには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)。

3.2 アルゴリズム

3.2.1 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。実行ステージ数は、3 ステージとする。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 は格納されるリオーダバッファの番号を示す。また、RS, Buffer, Store はそれぞれリザベーションステーション、リオーダバッファ、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 の使用権を確保。

- **動作および管理情報の更新作業**

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

3.2.2 オペランドフェッチのアルゴリズム

オペランドフェッチは命令の発行時に行なう。

リオーダバッファからの値の読み出しは Q フィールドによって行なわれる。値はリオーダバッファ内で探され、ない場合のみレジスタファイルから読み出される。

```
if (Buffer[x].Ri = S1)   [x > E]
    最小のエントリ番号 i を選ぶ
      if (Buffer[Ei].Vi ≠ no)
        RS[r].Vj ← Buffer[Ei].Vi
        RS[r].Qj ← 0
      else
        RS[r].Qj ← Buffer[Ei].Qi
  else
      RS[r].Vj ← Register[S1]
      RS[r].Qj ← 0
```

3.3 short-lived 変数の破棄

retire check のハードウェアは、リオーダバッファからリタイアした値に対して以下のような判断を行なう。

値が Short-lived ならば破棄する (すなわち、$Buffer[E16].tag = yes$ 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を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 個のループにから、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。

表 1はそれぞれのベンチマークプログラムにおいて、捨てることができる変数のパーセンテージを表している。リオーダバッファのサイズはそれぞれのベンチマークプログラムに対して、8, 16, 32 エントリの 3 つの場合に対して評価を行なった。

こ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。

表 1: 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%

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ハードウェアを利用したため、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 p292 ～p302