計算機アーキテクチャ 150－20
（２００２．１１．２８）

関数呼び出し時のレジスタの退避／復元に着目したメモリリネーミング手法

片山清和† 安藤秀樹† 島田俊夫†

関数呼び出し時にはレジスタの退避／復元を行うコードが実行される．退避と復元を行うコードは1対1に対応し，それらが参照するメモリアドレスはスタックポインタの指すアドレスからの相対位置として静的に決められている．本論文では，この性質を利用したメモリリネーミング手法を提案する．本手法は，動的に関数呼び出し時のレジスタの退避／復元を行う命令を検出し，退避／復元される論理レジスタに対し，退避前と復元後に同一の物理レジスタを割り当てることによって偽のデータ依存を切断するメモリリネーミングを行う．評価の結果，分岐予測が完全の場合，通常のスーパスカラ・プロセッサに対し，最大12.2%，平均4.7%の性能向上が得られることを確認した．また，データキャッシュのポート数を1ポートに制限した場合平均10.0%の性能向上が得られることを確認した．

Memory Renaming using Save/Restore of Registers on Procedure Invocation

Kiyokazu Katayama,† Hideki Ando† and Toshio Shimada†

On procedure invocation the save and the restore code of registers are executed. There is one-to-one correspondence between them. They refer to the relative address to the stack pointer that is decided statically by the compiler. In this paper we propose a memory renaming technique that exploits the character. Our scheme detects the save/restore codes of registers dynamically, assigns the same physical register to the logical register that is saved and restored, and cuts off false data dependency. Our evaluation results show that our scheme improves performance by a maximum of 12.2% or by an average 4.7% over conventional superscalar processor. We confirm that our scheme improves performance by an average 10.0% under 1-ported data cache over conventional superscalar processor with 4-ported data cache.

1. はじめに

プロセッサの性能向上を阻害する要因の１つに，命令間のデータ依存がある．データ依存はプログラムの命令レベル並列性を制限しており，近年，これを緩和する手法が研究されている．

値予測は，このデータ依存を緩和するのに効果的な手法であり，Lipasti らによって**最終値予測**[1] が提案されて以来，**ストライド値予測**[2]，**2 レベル値予測**[2),3] などが提案されている．値予測は命令の実行結果を実行前に予測するものである．予測が正しく行われれば，データ依存を削除することができ，命令レベル並列性は向上する．これまで多くの値予測器が提案されているが，ほとんどは過去の値の振舞いと将来の値の間に相関があることを利用するものである．

これに対し，メモリアクセスの RAW（Read-After-Write）の関係を予測し，ロード値予測に利用するものとして，**メモリリネーミング**[4)~6)] がある．メモリリネーミングは過去の RAW の関係と将来の RAW の関係の間に相関があることを利用するものである．

本論文では，関数呼び出し時における，プログラムの性質を利用したメモリリネーミング手法を提案する．一般的に，プログラムでは，関数呼び出しの際にレジスタの退避／復元を行う．そのため，ユーザが記述されたコードには存在しなかった依存がプログラムに発生する．これを我々は**偽のデータ依存**と呼ぶ．本手法は動的にレジスタの退避／復元を検出し，退避される値を保持している物理レジスタを開放せずに保存しておき，復元された値を使用する命令にメモリからの復元を行わず直接データを送ることによって，偽のデータ依存を削除し，プロセッサ性能を向上させる．

本論文の構成は以下の通りである．2 章では関連研究について述べる．3 章では関数呼び出し時のレジスタの退避／復元の性質について説明し，4 章でその性質を利用したメモリリネーミング手法を提案する．5 章で評価を行い，6 章で本論文をまとめる．

2. 関連研究

値予測の中で最も単純な手法は，Lipasti らによって提案された最終値予測[1] である．この予測手法では，同一値が繰り返されると予測する．他の主要な予測方式に，ストライド値予測[2]，2 レベル値予測[2),3] などがある．ストライド値予測は，一定の差分を持った値が繰り返されると予測するものである．前回および，前回の値と前々回の値の差分（ストライド）を保持しておき，命令がデコードされた時に前回の値

† 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University

とストライドを加算して，今回の予測値とする．2 レベル値予測は，現在の値は過去の値の振る舞いのパターンに対して相関があることを利用して予測を行うものである．この方式では，**値履歴表**（VHT : Value History Table）と**パターン履歴表**（PHT : Pattern History Table）を用いる．VHT は命令アドレスをインデクスとして参照する．各エントリには命令の最近数回の実行結果とその出現の履歴パターンを保持する．PHT は履歴パターンをインデクスとして参照する．各エントリには VHT の 1 つのエントリに格納する値の数だけのカウンタを用意し，VHT の中のどの値が次に現れるかを記録し，これを予測に用いる．

こうした値予測はデスティネーションを定義する全ての命令に対して用いることができるが，ロード命令のロード値を予測する手法として**メモリリネーミング**という手法がある．この手法は過去の RAW(Read-After-Write) の関係を基に，現在の RAW の関係を予測することで，ロード値予測を行うものである．メモリリネーミングには，**メモリリネーミング**[4]，**メモリクローキング**[5]，**2 ホップアドレス名前替え**[6] がある．これらの予測手法では，**依存検出表**と，**リネーム表**と，**シノニム・ファイル**を用いる．依存検出表は，メモリ依存を検出する表で，メモリアドレスをインデクスとして参照する．各エントリには，当該メモリアドレスに対して，書き込みを行ったストア命令の命令アドレスを保持する．リネーム表は依存関係にあるロード／ストア命令を後述するシノニム・ファイルのエントリに関連づける表であり，ロード／ストア命令の命令アドレスをインデクスとして参照する．各エントリはシノニム・ファイルのエントリ番号をタグとして保持する．シノニム・ファイルはリネーム表に保持されているタグをインデクスとする表で，各エントリにはストア命令が書き込んだ値を保持しており，ロード命令の予測値として使用される．

3. レジスタの退避／復元の性質

本章では，まず，レジスタの退避／復元の性質について述べる．次に，レジスタの退避／復元のために生じる偽のデータ依存関係について述べる．

3.1 レジスタの退避／復元

一般的に，プロセッサが仮定している論理レジスタの数は，プログラムが必要とする変数の数に対して極めて少ない．そのため，1 つの論理レジスタは複数の変数によって共有される．論理レジスタがある変数の値を保存しているときに，別の変数の値を書き込むと，以前の変数の値が破壊される．そこで，この値の破壊を防ぐために，コンパイラはレジスタの退避／復元を行うコードを追加する．この退避／復元は関数呼び出しに伴うもの（以降，単に save ／ restore と記す）と spill-in ／ spill-out に分類できる．

save ／ restore は，関数呼び出し時のレジスタ値の破壊を防ぐために行われる．退避と復元は，それぞれ，関数のプロローグとエピローグで行われる．save ／ restore は関数内で更新する可能性のある全ての論理レジスタに対して行われる．一方，spill-in ／ spill-out は関数内でのレジスタ値の破壊を防ぐために行われる．退避と復元はメインブロック内で行われる．spill-in ／ spill-out は関数内で必要なレジスタ数がプロセッサで仮定されている論理レジスタ数を越えた場合のみ行われる．退避と復元には次の性質がある．

- 退避と復元は必ず 1 対 1 に対応する．
- 退避される論理レジスタと復元される論理レジスタは一致する．
- 退避と復元は，それぞれストア命令とロード命令で実現される．

```
decompress()                    getcode()
{                               {
        .                               .
        .                               .
        .                               .
    while (getcode()) {         }
        .
        .
        .
    }
        .
        .
        .
}
```

(a) C 言語でのプログラム

```
decompress()                    getcode()
{                               {
    .                               addiu $29, $29, -24
    .                               sw $16, 16 ($29)
    .                               sw $31, 20 ($29)
L1:                                     .
  jal getcode                           .
    .                                   .
    .                               lw $31, 20 ($29)
    .                               lw $16, 16 ($29)
  j L1                              addiu $29, $29, 24
    .                               jr $31
    .                           }
    .
}
```

(b) アセンブリ言語でのプログラム

図 1 関数呼び出しの例

- 退避／復元で使用されるメモリアドレスはコンパイル時にスタックポインタからの相対位置で決定される．

図 1 に save ／ restore の例を示す．(a) はC言語で記述された SPECint95 の compress95 のコードの一部であり，decompress 関数から getcode 関数を呼び出している．(b) は (a) のコードをコンパイルした結果である．getcode 関数では，関数の先頭で 29 番レジスタ (スタックポインタ) を更新し，その後 16 番，31 番レジスタをそれぞれ，スタックポインタの指すメモリアドレスから 16，20 大きいメモリアドレスに退避している．また，関数から出る直前に，退避しておいた 16 番，31 番レジスタを復元し，スタックポインタの値を関数が呼び出される直前の値に更新している．

3.2 レジスタの退避／復元で生じる偽のデータ依存関係

図 1(b) のコードで，論理レジスタ 16 番に関するデータ依存関係を考える．**図 2**(a) に論理レジスタ 16 番と関数呼び出しに関連する命令のみ着目したコードを示し，同図 (b) に論理レジスタ 16 番に関するデータ依存関係を示す．矢印は実行時における真のデータ依存を示している．図では命令 `i1` から命令 `i5`，命令 `i6` から命令 `i7`，命令 `i8` から命令 `i3` にデータ依存がある．ここで，破線で示した命令 `i6` から命令 `i7` へのデータ依存はユーザが記述したコードに，もともと存在したデータ依存である．しかし，太い実線の矢印で表されている命令 `i1` から命令 `i5`，命令 `i8` から命令 `i3` へのデータ依存は，save ／ restore のために生じた依存であり，ユーザが記述したコードには存在しないものである．以降，この依存を偽のデータ依存と表記する．なお，spill-in ／ spill-out でも同様に，偽のデータ依存が生じる．

4. 関数呼び出し時の save ／ restore を利用した偽のデータ依存削除手法

本章では，レジスタ・リネーミングを利用して，偽のデータ依存を削除するメモリリネーミング手法について説明する．4.1 節でレジスタ・リネーミングを利用した偽のデータ

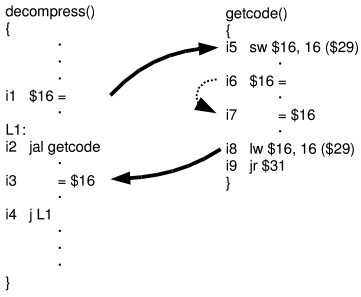


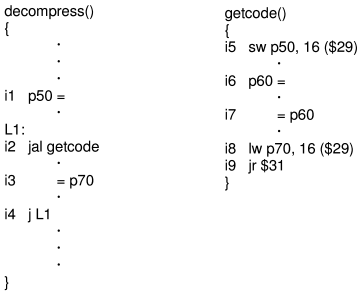
図 2 $16 のデータ依存関係

依存の削除方法について説明する．そして 4.2 節で本手法を実現する上で検討すべき事項をあげ，議論する．4.3 節で偽のデータ依存を削除するメモリリネーミング機構の構成とその動作について説明し，4.4 で本手法の利点と欠点について述べる．

4.1 レジスタ・リネーミングを利用した偽のデータ依存の解消

図 2 のコードにおいて，論理レジスタ 16 番を，物理レジスタにマッピングした場合について考える．通常のレジスタ・リネーミング手法によって物理レジスタにマッピングした例を**図 3** に示す．同図 (a) は論理レジスタを物理レジスタにマッピングしたコードで，同図 (b) は命令ごとの物理レジスタの定義と参照と解放を表している．図では，命令 `i1`，命令 `i6`，命令 `i8` のデスティネーションレジスタは，それぞれ，物理レジスタ 50 番（p50），60 番（p60），70 番（p70）に割り当てられている．

図 3(a) のコードにおいて，命令 `i1` から，命令 `i3` へデータを直接受け渡すことができれば，命令 `i8` から命令 `i3` への偽のデータ依存を削除することができ，命令 `i3` の実行開始を早めることができる．また，命令 `i8` のロード値を使用する命令が存在しなくなるため，命令 `i8` の実行も不要となる．そこで，命令 `i8` に割り当てる物理レジスタを，命令 `i1` に割り当てられた物理レジスタと同じものにすることを考える．この時の割り当てた結果を**図 4** に示す．同図 (a) は物理レジスタにマッピングした例で，同図 (b) は命令ごとの物理レジスタの定義と参照と解放を表している．この割り当ては，物理レジスタを退避後でも解放しないようにし，1 つの物理レジスタを退避前と復元後の論理レジスタに割り当てることで実現できる．このように，物理レジスタへのマッピング手法を変更することによって，レジスタ・リネーミングで偽のデータ依存を削除することができる．

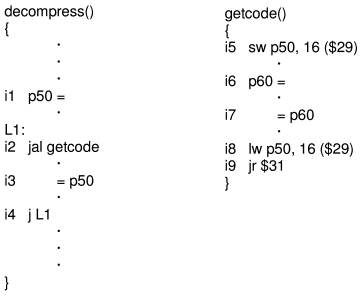


(a) 物理レジスタ番号にマッピングした例

命令	定義	参照	解放
i1	p50		
i5		p50	
i6	p60		p50
i7		p60	
i8	p70		p60
i3		p70	

(b) 物理レジスタの定義と参照と解放

図 3 従来のマッピング例



(a) 物理レジスタ番号にマッピングした例

命令	定義	参照	解放
i1	p50		
i5		p50	
i6	p60		
i7		p60	
i8	p50		p60
i3		p50	

(b) 物理レジスタの定義と参照と解放

図 4 偽のデータ依存を解消するマッピング例

4.2 検討すべき事項

レジスタ・リネーミングを利用して偽のデータ依存を削除するには次のことが必要となる．

- save ／ restore 命令の検出方法．
- save 命令の記録．
- save ／ restore 命令が属する関数の識別方法．
- 物理レジスタ解放の抑制方法．

これらについて順に議論する．

4.2.1 save ／ restore 命令の検出方法

save 命令は，各退避レジスタ（saved register）に対し，関数内で定義される前にそのレジスタをストアする命令であり，以下の形式をとる．

`sw` r`,` $offset$ `($sp)`

よって，save 命令の検出は，各退避レジスタについて，関数が呼び出された後，最初に出現する上記の形式をとるストア命令を検出すればよい．

restore 命令のデスティネーションとオフセットは，対応する save 命令の退避レジスタ番号とオフセットに一致する．つまり，先の save 命令に対応する restore 命令は，以下の形式をとる．

`lw` r`,` $offset$ `($sp)`

よって，restore 命令の検出は次のように行えばよい．まず，各退避レジスタについて，save 命令のオフセットと属する関数を記録しておく．上記の形式のロード命令に対し，記録してある save 命令の中で，対応する restore 命令がまだ出現しておらず，最も近い save 命令のオフセットと属する関数が，ロード命令のオフセットと属する関数が，それぞれ一致するかどうかを判定し，ともに一致すれば restore 命令であり，そうでなければ restore 命令ではないとする．

4.2.2 save 命令の記録

本手法は，restore 命令が出現したときに，対応する save 命令が退避した物理レジスタを使用するものである．よって，記録しておく save 命令の情報は対応する restore 命令がま

だ出現していない save 命令だけでよい．また，3.1 節で述べたように，save ／ restore 命令は関数呼び出しに伴って出現するため，必ず入れ子構造をとる．このような特徴より，save 命令の情報を記録するには，RAS（Return Address Stack）と同様，スタックが効率的である．

4.2.3 save ／ restore 命令が属する関数の識別方法

4.2.1 節で述べたように，本手法では，restore 命令の検出のために，記録されている save 命令と出現したロード命令が同一の関数に属しているかどうか判定する必要がある．また，4.2.2 節で述べたように，記録されている save 命令は，対応する restore 命令が出現していないものだけであり，save ／ restore 命令は入れ子構造をとる．よって，save ／ restore 命令が属する関数を識別するには，関数呼び出しの深さを用いればよい．

4.2.4 物理レジスタ解放の抑制方法

本手法では，save 命令が退避した物理レジスタの解放を抑制しなければならない．これを実現するために，命令のデスティネーションレジスタに現在割り当てられている物理レジスタが，save 命令によって退避されたものかどうか判定する必要がある．

一般的に，物理レジスタは同時に複数のレジスタに割り当てられることはない．よって，物理レジスタ番号が一致しているかどうかの判定は，デスティネーションレジスタ番号に対する save 命令に対してのみ行えばよい．

したがって，命令のデスティネーションレジスタ番号でレジスタリネーミングのマップ表を参照し，現在割り当てられている物理レジスタ番号を得る．また，同時にデスティネーションレジスタ番号に対して，対応する restore 命令がまだ出現していない，最も近い save 命令が退避した物理レジスタ番号も得る．この両方の物理レジスタ番号が一致するか判定し，一致した場合，物理レジスタを解放しないようにすればよい．

4.3 機構と動作

本手法は関数レベルカウンタと save 命令スタックより構成される．関数レベルカウンタで現在の命令が属している関数の呼び出しの深さを記録し，save 命令スタックで save ／ restore を検出する．それぞれの構成と動作を以下に詳しく説明する．

4.3.1 関数レベルカウンタ

関数レベルカウンタは現在の関数呼び出しの深さを記憶するカウンタである．関数呼び出し命令をデコードした場合，カウンタ値を 1 増加させ，関数復帰命令をデコードした場合，カウンタ値を 1 減少させる．

4.3.2 save 命令スタック

save 命令スタックは，対応する restore 命令がまだ出現していない save 命令の情報を保持するスタックである．スタックは深さ n の循環スタックであり，退避レジスタの数だけ存在し，スタックと退避レジスタは対応している．スタックの 1 つのエントリは有効ビット（valid），命令の関数レベルカウンタ値（cntr），オフセット（offset），物理レジスタ番号（preg）からなる．

最初にベースレジスタとしてスタックポインタを持ち，退避レジスタをストアする sw 命令をデコードした場合の動作について説明する．ストアするレジスタの論理レジスタ番号に対応する save 命令スタックのトップを参照し，cntr フィールドの値と現在の関数レベルカウンタ値が一致するか判定する．一致した場合，この sw 命令は save 命令ではないため，何もしない．一致しない場合，この sw 命令は save 命令であるため，sw 命令のオフセット，関数レベルカウンタ値，ストアする物理レジスタ番号をスタックにプッシュし，有効ビットをセットする．なお，このとき，スタックの有効ビットがセットされていたら，保持されている物理レジスタを，sw 命令がコミットされる時に解放するようにする．

次にベースレジスタとしてスタックポインタを持ち，デスティネーションレジスタが退避レジスタである lw 命令をデコードした場合の動作について説明する．デスティネーションレジスタ番号に対応する save 命令スタックのトップを参照し，cntr フィールドの値と offset フィールドの値が，それぞれ現在の関数レベルカウンタ値と lw 命令のオフセットと一致するか判定する．どちらか一方でも一致しない場合，この lw 命令は restore 命令ではないため，何もしない．両方とも一致した場合，この lw 命令は restore 命令であるため，preg フィールドに記録されている物理レジスタ番号を，デスティネーションレジスタの物理レジスタに割り当て，スタックをポップする．さらに，この lw 命令は実行を行わず，状態を実行完了とする．

最後にそれ以外の命令のデコード時の動作について説明する．命令のデスティネーションレジスタ番号に対応する save 命令スタックのトップを参照する．preg フィールドに記録されている物理レジスタ番号と，現在そのレジスタに割り当てられている物理レジスタ番号が一致するか判定する．一致しない場合，この命令がコミットされる時に，現在割り当てられている物理レジスタを解放するようにする．一致した場合，物理レジスタを解放しないようにする．

4.4 本手法の利点と欠点

本手法の利点と欠点は次の点である．

- 低コストである．
- save ／ restore の性質を利用しているため，予測ではない．
- 検出できた restore 命令は実行する必要がないため，データキャッシュポート数要求が減少する．
- save した物理レジスタの解放を抑制するため，物理レジスタ要求が増加する．

5. 評　　価

本章では，最初に評価環境について述べ，次に評価結果について述べる．

5.1 評 価 環 境

SimpleScalar Tool Set[7] Version 3.0a の中の，スーパスカラ・プロセッサ用シミュレータをもとに，本機構を組み込んだシミュレータを作成し，測定を行った．命令セットは MIPS R10000[8] を拡張した SimpleScalar/PISA である．ベンチマーク・プログラムは，SPECint95 の全 8 種類を使用した．ベンチマーク・プログラムのバイナリは，GNU GCC Version 2.7.2.3（コンパイルオプション：-O6 -funroll-loops）を用いて作成した．

表 1 に入力セット，動的ロード命令に対する restore 命令

表 1 ベンチマーク・プログラム

プログラム	入力	restore 命令の割合 [%]
compress95	30000 e 2231	7.1
gcc	genoutput.i	30.8
go	6 9 2stone9.in	12.2
ijpeg	specmun.ppm	4.5
li	train.lsp	29.6
m88ksim	ctl.in	12.8
perl	scrabble.in	26.7
vortex	vortex.in	34.3

表 2 プロセッサモデル

命令フェッチ幅	最大 8 命令
命令デコード幅	最大 8 命令
命令発行幅	最大 8 命令
命令コミット幅	最大 8 命令
命令ウィンドウ	RUU256 エントリ，LSQ128 エントリ
機能ユニット	iALU 5，iMULT/DIV 1，Ld/St 4， fpALU 2，fpMULT/DIV/SQRT 2
パイプライン段数	12 段
物理レジスタ数	int256 本，fp256 本
分岐予測機構	完全
命令キャッシュ	256KB 4 ウェイ・セットアソシアティブ， ライン幅 32 バイト， ヒットレイテンシ 4 サイクル， ミスペナルティ10 サイクル
データキャッシュ	256KB 4 ウェイ・セットアソシアティブ， ライン幅 32 バイト，4 ポート ヒットレイテンシ 4 サイクル， ミスペナルティ10 サイクル
二次キャッシュ	統合，1MB 16 ウェイ・セットアソシアティブ， ライン幅 64 バイト， ミスペナルティ68 サイクル

の割合を示す．シミュレーション時間が過大にならないようにするために，関数の出現頻度をほぼ維持しつつ，入力のパラメータを調整している．また，ijpeg では最初の 50M 命令をスキップした後，450M 命令を実行し，vortex では最初の 100M 命令をスキップした後，80M 命令を実行した．ijpeg, vortex 以外のベンチマーク・プログラムでは，命令のスキップを行わず，最後まで実行した．なお，以降のグラフでのベンチマーク・プログラム名の表記において，compress95, m88ksim，vortex は，それぞれ comp，m88k，vort と表すこととする．

表 2 にプロセッサモデルを示す．表に示すように分岐予測機構は完全とした．

また，比較対象として，2 レベル値予測器を用いた．予測器の VHT と PHT は，ともに 4K エントリとした．また保持する履歴数は 4 とした．この数は文献 2) を参考にして決めた．また，予測器には予測値の信頼性を表す信頼性カウンタが付加されている．信頼性カウンタはライトバック時に投機的に更新した．信頼性カウンタの値は，予測が正しい時に 1 増加させ，予測に誤った時には 15 減少させ，30 以上で予測が信頼できるとした．これらは文献 9) を参考にして決めた．また，コミット時に，値予測に誤ったかどうかを判定し，予測に誤り，後続命令が誤った予測値を使用した場合のみ，命令をフラッシュし，再フェッチした．

5.2 評価結果

5.2.1 検出可能な restore 命令

図 5 に save 命令スタックの深さを変化させた場合の検出可能な restore 命令の割合を示す．横軸はベンチマーク・プログラムで，縦軸は動的 restore 命令に対する割合である．各ベンチマーク・プログラムにつき，4 本の棒グラフがある．左からスタックの深さが 1，2，4，8 の場合の結果である．図より深さを 1 とした場合でも平均 75%の restore 命令が予測可能である．また，深さを増加させると予測できる割合も増加し，深さ 4 でほぼ飽和していることが分かる．このとき，平均 95%の restore 命令が予測可能である．この結果より，save 命令スタックの深さは 4 で十分であることが分かる．

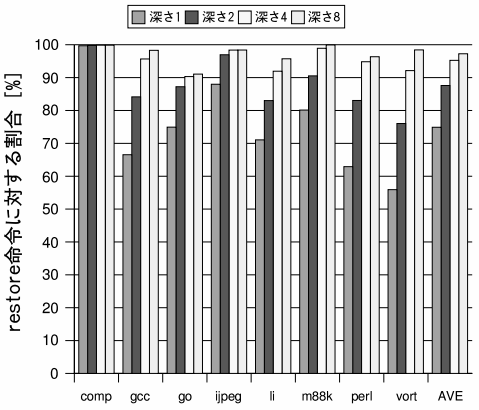


図 5 検出可能な restore 命令数の割合

5.2.2 性能向上率

図 6 に save 命令スタックの深さを変化させた場合の性能向上率を示す．同図 (a)，(b) はそれぞれ，物理レジスタ数が 1024，256 の場合の結果である．各グラフの横軸はベンチマーク・プログラムで，縦軸は性能向上率である．各ベンチマーク・プログラムにつき，5 本の棒グラフがある．左の 4 本は本手法の結果であり，右が 2 レベル値予測の結果である．本手法の結果は左からスタックの深さが 1，2，4，8 の場合の結果である．図より物理レジスタ数が 1024 本の場合には，どのベンチマーク・プログラムでも本手法のエントリ数を増加させるにしたがって，性能向上率が増加している．深さが 8 の場合に，平均 5.0%の性能向上が得られた．一方，物理レジスタ数を 256 本とした場合には，スタックの深さを 8 にすると go では大きな性能低下が見られる．これは，本手法によって，物理レジスタの生存期間が延び，物理レジスタ数要求が増加したためである．深さが 4 の場合に平均の性能向上率が最も高く，平均 4.7%の性能向上が得られた．

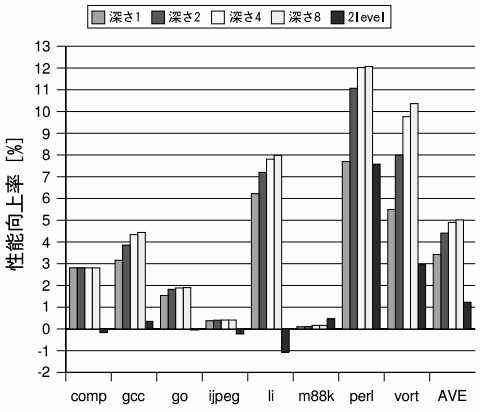
また，物理レジスタ数 256 の場合，save 命令スタックの深さが 4 でのハードウェア量は，321byte である．これは，これまで提案されている値予測器と比較して十分小さい．

5.2.3 データキャッシュポート数

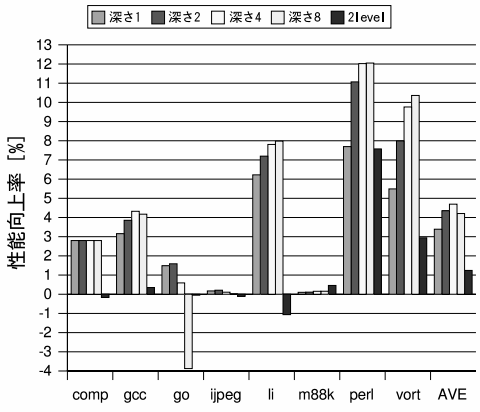
図 7 にデータキャッシュポート数を変えた場合の性能向上率を示す．グラフの横軸はデータキャッシュポート数で，縦軸は性能向上率である．実線が save 命令スタックの深さ 4 における本手法の結果であり，破線が 2 レベル値予測の結果である．図よりデータキャッシュポート数を制限すると，値予測では性能向上はほとんど認められない．これは，値予測では，予測が正しいかどうか検証するためのロード命令を実行する必要があり，予測を適用しても実行するロード命令数は変わらないからである．これに対し，本手法では，ポート数を制限するほど，性能向上が増加している．2 ポートでは 7.0%，1 ポートでは 10.0%の性能向上が得られている．これは，本手法では，検出できた restore 命令は実行する必要がなくなり，実行されるロード命令数が減少するためである．

6. まとめ

プログラムの多くの変数で，限られたレジスタを利用するために，プログラムにはレジスタの退避／復元を行う命令が存在している．レジスタの退避／復元を行う命令は 1 対 1 に対応し，使用するメモリアドレスはスタックポインタからの



(a) 物理レジスタ数 1024



(b) 物理レジスタ数 256

図 6 性能向上率

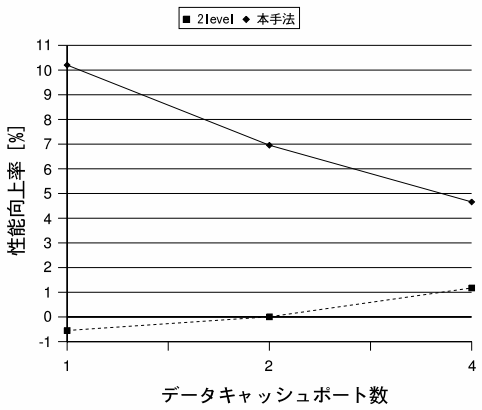


図 7 データキャッシュポート数を変えた場合の性能向上率

相対位置としてコンパイラによって決定されている．

本論文では，関数呼び出し時におけるレジスタの退避／復元に着目したメモリリネーミング手法を提案した．本手法では，動的に save 命令とそれに対応する restore 命令を，各退避レジスタに対応したスタックを用いて検出する．検出された save 命令では save する物理レジスタ番号を保存し，restore 命令では保存しておいた物理レジスタをデスティネーションレジスタに割り当てることで，save ／ restore によって生じる偽のデータ依存を解消する．また，検出された restore 命令の実行は不要になるため，データキャッシュポート数要求も緩和できる．

評価の結果，1 つの退避レジスタに対し深さが 4 のスタックで平均 95%の restore 命令が予測可能であることがわかった．これによって，物理レジスタ要求が最も厳しい分岐予測完全の場合，最大 12.2%（perl），平均 4.7%の性能向上が得られることを確認した．このときのコストは 321byte であった．また，データキャッシュのポート数を制限した場合，本手法では，7.0%（2 ポート），10.0%（1 ポート）の性能向上が得られることを確認した．

参 考 文 献

1) Lipasti, M. H., Wilkerson, C. B. and Shen, J. P.: Value Locality and Load Value Prediction, *Proc. Seventh Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, October 1996.

2) Wang, K. and Franklin, M.: Highly Accurate Data Value Prediction using Hybrid Predictors, *Proc. 30th Int'l Symp. on Microarchitecture*, pp. 281–290, December 1997.

3) Sazeides, Y. and Smith, J.E.: The Predictability of Data Values, *Proc. 30th Int'l Symp. on Microarchitecture*, pp. 248–258, December 1997.

4) Tyson, G. S. and Austin, T. M.: Improving the Accuracy and Performance of Memory Communication Through Renaming, *Proc. 30th Int'l Symp. on Microarchitecture*, pp. 218–227, December 1997.

5) Moshovos, A. and Sohi, G. S.: Streamlining Inter-operation Memory Communication via Data Dependence Prediction, *Proc. 30th Int'l Symp. on Microarchitecture*, pp. 235–245, December 1997.

6) 佐藤寿倫: 2 ホップアドレス名前替えを用いたロード命令の投機的実行, 情報処理学会論文誌, Vol. 40, No. 5, pp. 2109–2118, May 1999.

7) Burger, D. and Austin, T. M.: The SimpleScalar Tool Set, Version 2.0, Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

8) MIPS Technologies, Inc.: *MIPS R10000 Processor User's Manual, Version 2*, October 1996.

9) Reinman, G. and Calder, B.: Predictive Techniques for Aggressive Load Speculation, *Proc. 31st Int'l Symp. on Microarchitecture*, pp. 127–137, December 1998.