Vol.2010-ARC-190 No.25
2010/8/4

リオーダ・バッファの仮想的な拡大による先行実行

市原 敬吾[†1] 田中 雄介[†1,*1] 安藤 秀樹[†1]

データ・プリフェッチを実現する方法のひとつに，命令の先行実行がある．一般に，命令の実行タイミングは資源制約によって制限される．特にリオーダ・バッファ（ROB: Reorder Buffer）及び物理レジスタはプロセッサがサポートする in-flight 命令数を規定し，これが利用可能でなければ，命令の実行はフロントエンドでストールする．これらの資源制約を緩和し，本来の in-flight 命令数を越える命令を実行可能とすれば，先行実行を実現できる．本論文では仮想的に ROB を拡大し，実 ROB 及び物理レジスタを割り当てないまま命令を発行キューへ挿入し，先行実行させる方式を提案する．先行実行はこれらの資源を割り当てられていないため，実行を終了することはできないが，キャッシュ・ミスを早期に発生させることで，プリフェッチを実現できる．SPECfp2000 ベンチマークを用いて評価を行った結果，128 エントリの ROB を 8 倍に仮想的に拡大した場合，本手法を用いない場合に比べ 46%の性能向上を達成した．

Instruction Pre-Execution with a Virtual Reorder Buffer

Keigo Ichihara,[†1] Yusuke Tanaka[†1,*1]
and Hideki Ando[†1]

Instruction pre-execution is one of methods of data prefetching. In general, instruction execution is constrained by data dependences and resource constraints. In particular, reorder buffer (ROB) and physical registers are critical, because they strictly determine the number of in-flight instructions. If we can alleviate these constraints, we are allowed instructions to be executed beyond the determined number of in-flight instructions. This paper proposes a *virtual reorder buffer*, which allows pre-execution by allocating neither ROB nor physical registers. In our scheme, even if the ROB entries run out, renamed instructions are inserted into the issue queue with allocating neither ROB nor physical registers. These instructions are pre-executed when their source operands become available. If an pre-executed load causes a cache miss, it moves data to the cache, resulting a prefetch. Our evaluation results using SPECfp2000 benchmark show that our scheme improves performance by 46% over a processor without pre-execution.

1. はじめに

プロセッサとメモリの間の速度差は非常に大きく，メモリ・ウォールと呼ばれている．このメモリ・ウォールによるロードの長いレイテンシは，メモリ・インテンシブなプログラムの性能を大きく制限している．

ロード・レイテンシを短縮する方法のひとつに，データ・プリフェッチがある．これはロードされるデータを予めメモリ階層の上位へ移動しておく手法である．プリフェッチを実現する方法として，自動プリフェッチャがある[3),10),11)]．従来の自動プリフェッチャのほとんどは，ロードの規則的なアクセス・パターンを検出し，プリフェッチを行うものである．この方法は，単純なアクセス・パターンにしか効果がない．複雑なメモリ・アクセスにも対応可能な方法も提案されたが，それらは非常に大きな予測器を必要とする．

複雑なアクセスに対応可能な別のプリフェッチ手法として，命令の**先行実行**がある[4),5),16),18),19),21),22)]．これは，本来の実行に先駆けて，事前に命令を実行する手法である．実際に命令を実行するため，どのようなアクセス・パターンにも対応できる．これまで多くの先行実行手法が提案されたが，そのほとんどはマルチスレッド環境を必要とした．

そこで本論文では，単一スレッド環境で先行実行を実現し，プリフェッチを行う手法を提案する，一般に，命令の実行タイミングは依存と資源制約によって制限されている．特にリオーダ・バッファ（ROB: Reorder Buffer）はプロセッサがサポートする in-flight 命令数を規定し，これが利用可能でなければ，命令の実行はフロントエンドでストールする．また，デスティネーション・レジスタをとる命令には物理レジスタも割り当て可能でなければならず，これも in-flight 命令数を制限する要因となる．もしこれらの制約を緩和し，in-flight 命令数を越える命令を実行可能となれば，先行実行を実現できると考えられる．

本論文では，ROB 及び物理レジスタを割り当てることなく命令の先行実行を行う**仮想 ROB** と呼ぶ手法を提案する．本手法では ROB が不足している場合に，ROB 及び物理レジスタを割り当てないまま命令を発行キューへ挿入する．この命令はソース・オペランドが揃えば発行され，先行実行を行う．この実行では ROB 及び物理レジスタを割り当てていな

†1 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University
*1 現在，株式会社デンソー
Presently with Denso Corporation

© 2010 Information Processing Society of Japan

いため，実行を終了することはできない．しかし，先行実行されたロード命令がキャッシュ・ミスを起こせば，データをプリフェッチすることができる．

これら先行実行された命令は，後に ROB が利用可能となった時点で再びフェッチする．再フェッチされた命令は，通常の実行と同様，ROB 等の資源を割り当てられた上で発行キューへ挿入され，ソースが揃えば発行される．この実行を本実行と呼ぶ．先行実行によってプリフェッチが行われていれば，本実行でのロード・レイテンシは短縮される．

本論文では，まず 2 節で関連研究について述べる．次に 3 節で先行実行の効果を示し，4 節で提案する先行実行方式の詳細について説明する．5 節で評価を行い，6 節で本論文をまとめる．

2. 関 連 研 究

2.1 先 行 実 行

先行実行を利用したプリフェッチ手法はこれまでにも多く研究されている[4),5),18),19),22)]．しかし，これらの手法は本手法と異なり，いずれも先行実行のために別スレッドを生成する必要がある．このため，SMT（Simultaneous Multithreading）や CMP（Chip Multiprocessor）といったマルチスレッド環境が必要となる．

単一スレッド環境において先行実行を行う手法として，Mutlu らは runahead 実行を提案した[16)]．この手法はキャッシュ・ミスによってメモリまでロード要求が至った時に，プロセッサが長時間ストールする期間を利用している．メモリからブロックが返ってくるまでアーキテクチャ状態をチェックポイントし，キャッシュ・ミスを起こした命令に依存していない命令を実行する．これにより，依存関係にない多くのキャッシュ・ミスを並列に処理することができる．

また，単一スレッド環境において先行実行を実現する他の手法として，山本らは物理レジスタ 2 段階解放を提案した[21)]．この手法では，物理レジスタの解放を割り当て許可及び書き込み許可の 2 段階に分割する．書き込みが許可されないレジスタも早期に解放し，後続の命令へ割り当て可能とすることで，物理レジスタの不足によってリネームがストールすることを回避する．また，ソース・オペランドが揃った命令を，デスティネーション・レジスタへの書き込み許可を待たずに一度だけ実行することで，先行実行が実現される．

2.2 in-flight 命令数の拡大

非常に多くの in-flight 命令をアウト・オブ・オーダ実行することで，MLP を利用し，ロード・レイテンシを隠蔽できることが文献 6) で述べられている．

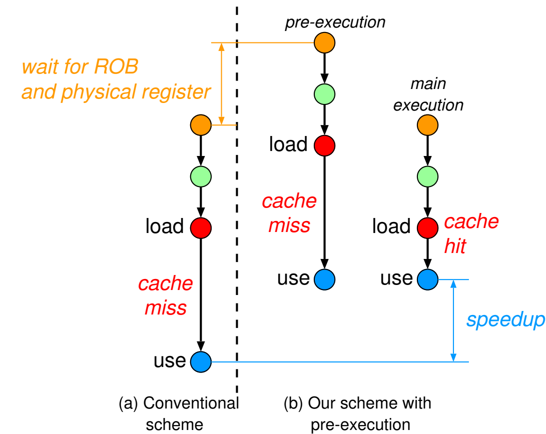


図 1 先行実行の効果

in-flight 命令を増加する手法として，Cristal らは kilo-instruction processor を提案した[7)]．この手法は，チェックポイントを用いて ROB を従来より早期に解放し，再利用するものである．この手法では，ROB の先頭エントリの命令は，実行が完了しているかどうかにかかわらず定期的に削除される．実行が未完了の命令を削除する場合のみ選択的にチェックポイントを取ることで，チェックポイント採取のオーバヘッドを削減している．また，ROB を併用することでチェックポイントからの回復を削減し，特に分岐予測ミスからの回復におけるオーバヘッド増加を抑制している．Martínez らも同様にチェックポイントを利用した資源の早期解放手法 Cherry を提案している[15)]．また，Petit らは投機的でないと判明した時点で ROB の先頭エントリを解放することで，アウト・オブ・オーダで命令をリタイアさせる手法を提案した[17)]．Latorre らは ROB を複数のセグメントに分割し，実行が完了した命令で占められているセグメントの情報を圧縮する手法を提案している[13)]．

3. 先行実行の効果

図 1 に本手法における先行実行の効果を示す．図 1(a) は，従来のプロセッサにおいて命令が実行されるタイミングを表している．図中の `load` はキャッシュ・ミスを起こすロード命令である．図に示されているとおり，ROB 及び物理レジスタが不足している場合，従来のプロセッサではそれらを割り当て可能となるまで，命令はフロントエンドでストールする．

© 2010 Information Processing Society of Japan

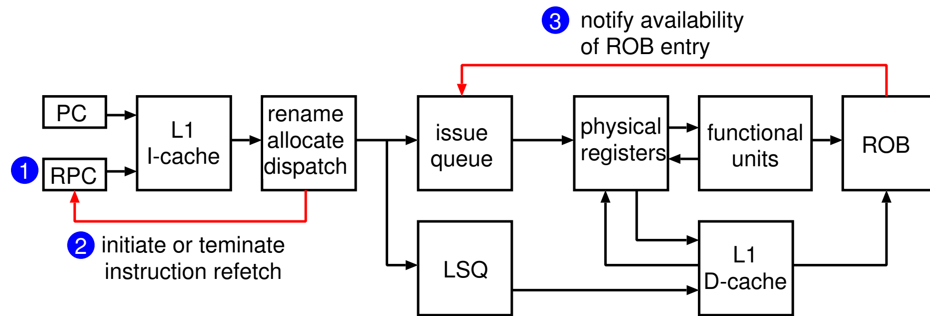


図 2 プロセッサの構成

一方，図 1(b) に示されているとおり，本手法においては ROB 及び物理レジスタを割り当てないまま命令を先行実行できる．これにより，`load` のキャッシュ・ミスは従来より早期に発生する．これがプリフェッチとなり，データをメモリ階層の上位へ移動させることができる．先行実行された命令は，後にこれらの資源を割り当てられた上で本実行されるが，その際には `load` はキャッシュ・ヒットするため，レイテンシは短縮され，性能は向上する．

4. 仮想 ROB を用いた命令の先行実行

本節では，ROB 及び物理レジスタを割り当てないまま命令を発行キューに挿入し先行実行する方法，及び先行実行した命令を本実行のために再フェッチする方法を説明する．

図 2 に本手法を実装した場合のプロセッサのブロック図を示す．図では通常の構成要素に加え，(1) 再フェッチ用 PC（RPC: Refetch PC），(2) ディスパッチ・ステージから RPC へ再フェッチの開始・停止を指示する信号，及び (3)ROB から発行キューへ ROB に空きエントリが生じたことを伝える信号が追加されている．(1)，(2) の詳細については 4.1 節で，(3) については 4.2 節において示す．

4.1 先行実行・本実行

従来のプロセッサでは，命令に ROB を割り当てることができない場合には命令はストールする．これに対し，本手法では ROB が不足している場合には，ROB 及び物理レジスタを割り当てないまま命令を発行キューへ挿入する．これを**先行ディスパッチ**と呼ぶ．先行ディスパッチされた命令は，ソース・オペランドが揃えば発行され，先行実行を行う．

先行実行命令は物理レジスタを割り当られていないため，実行結果を保持することはできないが，バイパス論理を経由して後続の依存命令に結果を渡すことはできる．ただしバイパス論理による結果の受け渡しは，実行後 1 サイクルの間しか有効でない．この制約を緩和するため，フォワーディング・バッファ（FB: Forwarding Buffer）[2) を用いる．FB はオペランド・タグで連想検索可能な小さなバッファであり，最近の先行実行の結果を保持している．バイパス論理による実行結果の受け渡しに失敗した場合でも，FB にその結果があれば，後続の依存命令を先行実行できる．FB からも結果値を得られなかった場合は，これらの命令は発行できず，後に本節の冒頭で示した信号 (3) によって発行キューから削除される（4.2 節）．

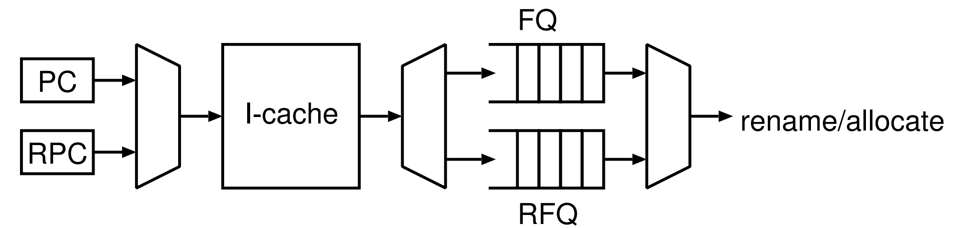


図 3 命令フェッチの構成

命令の先行ディスパッチを開始したら，直ちにそれらの命令の再フェッチを開始し，本実行に備える．再フェッチは，再フェッチ用の PC（**RPC**: Refetch PC）を用いて行う．RPC は先行ディスパッチを開始した際に，その最初の先行ディスパッチ命令の PC で初期化する（図 2）．再フェッチした命令は再フェッチ・キュー（RFQ: Refetch Queue）へ格納する．なお，PC によってフェッチされた命令はフェッチ・キュー（FQ）へ格納される（**図 3**）．

再フェッチは，先行ディスパッチされた命令を全て本実行するまで継続する．再フェッチを終了するタイミングを検出するため，**先行ディスパッチ・カウンタ**と呼ぶカウンタを用意する．このカウンタは本実行されるべき命令数を表し，命令を先行ディスパッチした際にインクリメントする．一方，再フェッチした命令を発行キューへ挿入した際にはデクリメントする．カウンタ値が 0 となった場合，必要な本実行は全て行われることが確定するため，再フェッチを終了し，RFQ をフラッシュする．

命令フェッチ及び再フェッチは時分割で行う．再フェッチを優先して行い，RFQ が満杯となった場合に PC によるフェッチを行う．また，リネーム・ステージにおける FQ ／ RFQ からの読み出しも，同様に時分割で行う．まず RFQ の先頭の命令について，リネーム及び資源割り当てが可能かを確認し，可能であれば RFQ から命令を読み出す．不可能であれば

ⓒ 2010 Information Processing Society of Japan

FQ から読み出す．

4.2 先行ディスパッチ命令の削除

先行ディスパッチされた命令は，以下の場合においては発行される前に発行キューから削除されなければならない．

(1) 先行実行する前に，本実行に必要な資源が利用可能となった場合．この場合，命令は本実行可能となるため，もはや先行実行を行う必要はない

(2) バイパス論理及び FB による先行実行結果の受け渡しに失敗した場合．この場合，後続の依存命令は発行不能となり，発行キューに取り残される（4.1 節）

(1) の場合に対処するため，次のようにして資源の利用可能性を発行キュー内の命令に伝達する．ROB から命令がコミットされ空きエントリが生じたら，そのエントリ番号を発行キューへ放送する．発行キューでは，先行ディスパッチされた各命令が，放送されてきたエントリ番号が，もし空いていれば自身が割り当てられるはずであったエントリのものかどうかを判断する．もしそうであれば，その命令を発行キューから削除する．削除された命令は必要な資源を割り当てられた上で，RFQ から発行キューへ再びディスパッチされる．なお，厳密には ROB が利用可能であっても，物理レジスタが割り当て可能であるとは限らず，直ちに再ディスパッチ可能であることは保証されない．しかし，ROB と物理レジスタ数がバランスがとれた設計においては，ほぼ良い近似を示すと考えられる．そのため，ここでは ROB が利用可能となった時点で削除を行っている．

この方法の欠点としては，(2) の場合の命令の削除としてはタイミングが遅いことが挙げられる．この場合においては本来，命令は先行実行結果の受け渡しに失敗した時点で削除されるべきである．しかし，実際には結果受け渡しの成功率は高く（5.3 節），(2) の状態が生じることは稀である．従って，これによる性能への影響は小さい．

4.2.1 ROB の利用可能性の伝達

発行キュー内の命令が ROB が利用可能となったことを検出できるようにするため，先行ディスパッチの際には，もしも空いていたなら割り当てられたはずの ROB のエントリを命令に**先行割り当て**することとする．これにより，仮想的に ROB を拡大することができる．

図 4 に仮想的に拡大された ROB の概念図を示す．この図では，実エントリ数 N の ROB を $M = 4$ 倍に拡大した場合を例示している．図において，ROB の上部の数字は仮想的に拡大された ROB 全体の論理エントリ番号を表し，下部の数字は ROB を循環バッファで実装した時の物理エントリ番号を表している．論理エントリ番号で 0 番から $(N-1)$ 番までのエントリを実 ROB，残りの部分を仮想 ROB と呼ぶ．循環バッファであるので，エントリの割り当ては実 ROB から仮想 ROB に向かって modulo N で行われる．つまり，1 つの物理エントリには 1 つの実エントリと $(M-1)$ 個の仮想エントリがマッピングされる．

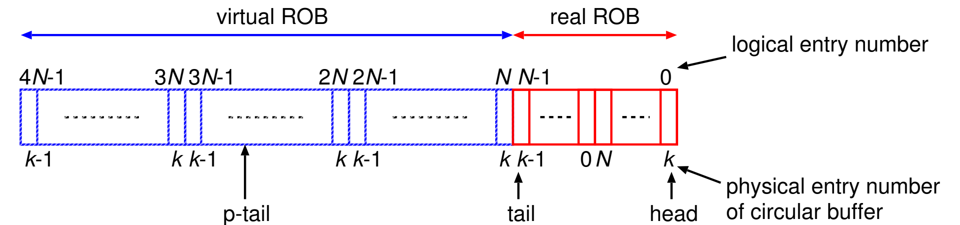


図 4 p-tail による PRE 割り当て

仮想 ROB を管理するため，実 ROB を管理する通常の head 及び tail ポインタに加え，仮想 ROB の末尾のエントリを指すポインタ p-tail（Pre-allocation Tail）を用意する．リネーム・ステージにおいて ROB が不足した場合，命令には p-tail が指す仮想 ROB のエントリを割り当てる．これを先行割り当てと呼ぶ．先行割り当てを行ったら p-tail をインクリメントし，以後同様に後続の命令に先行割り当てを行う．

前述のとおり，1 つの物理エントリには最大で $(M-1)$ 命令が仮割り当てされる．各物理エントリに現在いくつの命令が先行割り当てされているかを特定するため，ROB の各エントリに **MC**（Multiplicity Counter）と呼ぶカウンタを追加する．MC は，実 ROB エントリに通常どおりに命令を割り当てた場合に 0 で初期化する．もし ROB が不足しており，先行ディスパッチを行う場合には，先行割り当てされた仮想 ROB エントリの MC をインクリメントする．

ROB を先行割り当てされた命令は，割り当てられたエントリの物理エントリ番号とそのエントリの MC の値と共に発行キューへ挿入される．このために，発行キューの各エントリに PRE（Pre-allocated ROB Entry）及び MC の 2 つのフィールドを追加する．PRE は CAM で構成される．前述のとおり，資源の利用可能性を伝達するため，ROB から命令がコミットされ空きエントリが生じたら，その物理エントリ番号が発行キューへ放送される（図 2）．発行キューでは，ROB を先行割り当てされた命令を保持するエントリについて，その PRE フィールドと放送されてきた ROB エントリ番号とを比較する．一致した場合には，そのエントリの MC をデクリメントする．もしカウンタ値が 0 となれば，その命令は先行割り当てされた ROB のエントリが利用可能となったため，当該エントリを無効化してその命令を発行キューから削除する．なお，命令の再フェッチは通常のフェッチと並行して

ⓒ 2010 Information Processing Society of Japan

行われるため，多くの場合において削除された命令は既に RFQ 内に存在している．ROB の MC は，命令のコミットの際にデクリメントされる．

5. 評　価

5.1 評価環境

以下のモデルについて評価を行った．

- **BASE**: 実 ROB の割り当てのみを行うモデル
- **PE**: $M = 8$ の先行割り当てを行うモデル

評価には，SimpleScalar Tool Set Version 3.0a[20] をベースに提案手法を実装したシミュレータを用いた．命令セットは，MIPS R10000 を拡張した SimpleScalar/PISA である．ベンチマーク・プログラムとして，数値計算プログラムからなる SPECfp2000 から 8 本を使用した．**表 1** に使用したベンチマーク・プログラム及びその L1 データ・キャッシュ・ミス率，L2 キャッシュ・ミス率，メイン・メモリ・アクセス率を示す．バイナリは，gcc ver.2.7.2.3 を用いて-O6 -funroll-loops のオプションでコンパイルした．入力には ref 入力を用い，シミュレーション時間が過大とならないよう，命令のスキップと実行命令数の制限を行った．

BASE モデルのプロセッサ構成を**表 2** に示す．本評価では，ROB を拡大することによる性能への影響を評価するため，ROB 以外の主要な資源（発行キュー，LSQ）は十分に存在すると仮定した．具体的には，発行キュー及び LSQ のエントリ数を 実 ROB エントリ数 $N \times$ 拡大倍率 M とした．また，物理レジスタ数は実 ROB エントリ数と等しくしている．これは本評価において仮定しているものと同様のレジスタ・リネーム方式を採用している商用プロセッサ（Intel Pentium 4[9]，Alpha 21264[12]）における ROB と物理レジスタ数のバランスに従ったものである．なお，fp レジスタは実 ROB エントリ数の 2 倍としている．これは上記商用プロセッサの fp レジスタは 64 ビットであり，倍精度の値を 1 レジスタに格納できるのに対し，我々のシミュレータの fp レジスタは 32 ビットであり，2 レジスタを要するためである．従って fp レジスタ・ファイルを 2 倍とすることで，実 ROB エントリ数と同数とすることと等価となる．これら ROB 以外の主要な資源の拡大手法については文献 1)，14)，21) などにおいて研究されている．

PE モデルにおいては 8 エントリの FB を用いた．FB の容量を有効に利用するため，エントリの置換ポリシとして non-bypass caching[8] を用いた．このポリシでは，バイパス経由で読み出されなかった結果のみを FB に置く．2 回以上参照されるオペランドは少ないため，これにより読み出されないオペランドによる FB の浪費を抑制できる．

表 1 キャッシュの MPKI 及びメモリ・アクセス率

program	MPKI		memory
	DL1	L2	access rate
ammp	29.5	9.5	2.6%
applu	10.0	4.9	2.2%
apsi	1.2	0.4	0.1%
art	147.5	64.7	19.5%
equake	15.5	6.0	1.5%
mesa	2.7	1.1	0.2%
mgrid	11.2	3.3	0.9%
swim	34.3	12.3	4.8%

表 2 BASE モデルの構成

Pipeline width	4-instruction wide for each of fetch, refetch, decode, issue, and commit
ROB	128 entries
Issue queue	1024 entries
LSQ	1024 entries
Physical regsiter	128 for int, 256 for fp
Function unit	4 iALU, 2 iMULT/DIV, 2 Ld/St, 4 fpALU, 2 fpMULT/DIV/SQRT
L1 I-cache	64KB, 2-way, 32B line
L1 D-cache	64KB, 2-way, 32B line, 2 ports, 2-cycle hit latency, non-blocking
L2 cache	2MB, 4-way, 64B line, 12-cycle hit latency
Main memory	300-cycle min. latency, 8B/cycle bandwidth
Branch prediction	6-bit history gshare, 8K-entry PHT, 10-cycle misprediction penalty

5.2 性　能

図 5 に各ベンチマークについての性能変化を表すグラフを示す．縦軸には IPC をとっている．図の右端の「G.M.」は 8 本の幾何平均を示している．この図から，多くのベンチマークにおいて PE モデルは BASE モデルに対して性能向上していることがわかる．特に art，swim では大きな性能向上がみられる．平均では 46%の性能向上が得られた．これは先行実行によってロード・レイテンシを短縮できたためである．

図 6 にロード命令のレイテンシを示す．縦軸にはコミットされたロードの平均レイテン

© 2010 Information Processing Society of Japan

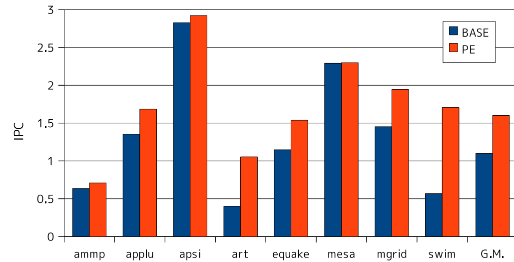


図 5 IPC

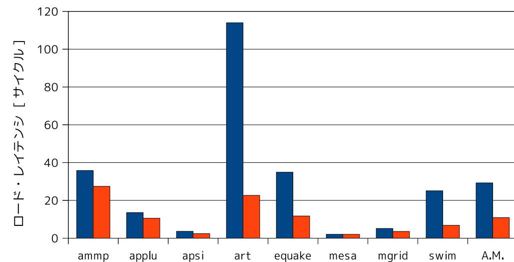


図 6 ロード命令のレイテンシ

シをとっている．図の右端の「A.M.」は 8 本の算術平均である．

図に示されているとおり，本手法を用いることで全てのベンチマークにおいてロード・レイテンシは短縮されている．特に art，swim では大きく短縮されていることがわかる．最も効果が高かった art では，BASE モデルに対する PE モデルのレイテンシ短縮率は 80%であった．平均では 63%である．これは先行実行によって，データがキャッシュへプリフェッチされたためである．先行実行時にプリフェッチされれば，本実行においてはキャッシュ・ヒットとなるため，レイテンシは短縮される．

一方，もともとレイテンシが小さい apsi，mesa では，性能はあまり向上していない．これは，この 2 ベンチマークがメモリ・インテンシブではないためである．メモリ・インテンシブでないプログラムでは，先行実行によるプリフェッチは性能改善に寄与しない．

5.3 先行実行率

図 7 に PE モデルおける，コミットされたロード命令の次に示すカテゴリでの分類を示す.「先行実行」は実際に先行実行されたロード命令,「バイパス失敗」はバイパス論理もし

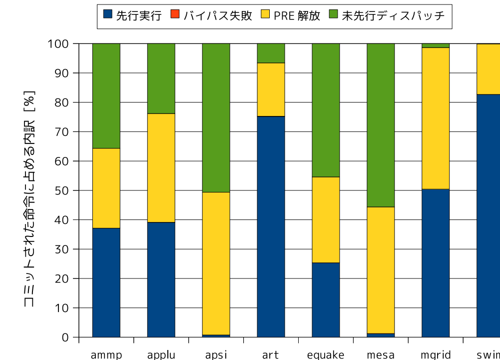


図 7 コミットされたロード命令の内訳

くは FB による実行結果の受け渡しがうまく行えなかったために，先行実行が取り消されたロード命令,「PRE 解放」は先行実行される前に ROB に利用可能な空きが生じ，先行実行が取り消されたロード命令,「未先行ディスパッチ」は先行ディスパッチ自体が行われなかったロード命令の割合をそれぞれ表している．

このグラフから，概ね半数程度のロード命令が先行実行されていることがわかる．コミットされたロード命令に対して，それらのうち先行実行されたものの割合を**先行実行率**と定義する．PE モデルでは，平均で 40%程度の先行実行率が得られた．特に art，swim の先行実行率は高く，75%以上のロード命令が先行実行されている．これらのベンチマークはレイテンシ短縮率及び性能向上率も大きく，先行実行によって性能向上が得られたことがわかる．

一方，apsi，mesa の先行実行率は低い．メモリ・インテンシブでないプログラムではキャッシュ・ミスによって ROB がブロックされることが少ないため，ROB が不足しにくく，先行実行が行われにくいものと考えられる．

また，この図において「バイパス失敗」が非常に少ないこともわかる．これは先行実行結果の受け渡しの成功率が非常に高いことを示しており，4.2 節で述べた (2) の状態は稀にしか発生しないことがわかる．

5.4 ROB の拡大倍率と性能

PE モデルでは ROB を 8 倍に仮想的に拡大し，先行割り当てを行っている．本節では，この先行割り当ての倍率 M を変化させた場合の性能評価を行う．

図 8 に倍率 M を変化させた場合の性能変化のグラフを示す．apsi，mesa を除くベンチ

© 2010 Information Processing Society of Japan

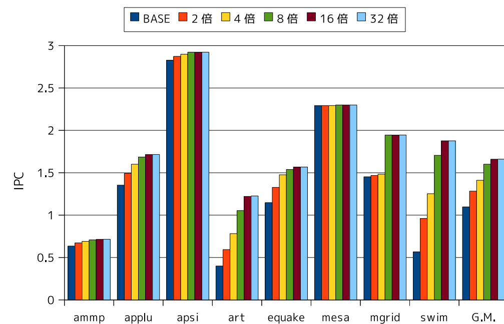


図 8　仮想 ROB の拡大倍率と性能

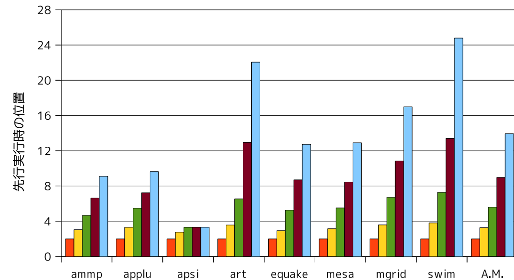


図 9　先行実行時に ROB の何重目にいたか

マークでは，M を 2 から増加させることで性能は向上している．これは M を大きくすることによって，in-flight な先行実行命令数が増加し，より早期にロードの先行実行を開始できるようになったためである．

図 9 はロードの先行実行が，発行された時に ROB の何重目にいたかを表すグラフである．このグラフでは，本実行として実 ROB を割り当てられた状態を 1 重目と数えている．この図からわかるとおり，全てのベンチマークにおいて，仮想 ROB の拡大倍率を増加することで発行タイミングは早くなっている．$M = 2$ の場合，先行実行は必ず 2 重目からでなければ発行できないが，$M = 8$ では平均で 6 重目，$M = 32$ では 14 重目と非常に早期に発行されるようになっている．このように早期にロードが発行されることでデータ・フェッチのタイミングが早まり，ロードのレイテンシを隠蔽し性能を向上できると考えられる．

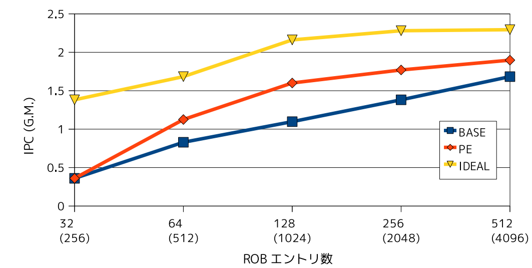


図 10　実 ROB エントリ数を変化させた場合の性能

ただし，図 8 に示されているとおり，倍率 M を拡大することに伴う性能向上は $M = 8$ 程度でピークとなり，以降は飽和している．これは次のように考えられる．本評価ではメモリ・アクセス・レイテンシを 300 サイクルとしている．IPC を 1.6（これは $M = 8$ における IPC の幾何平均である）と仮定すると，もし仮想 ROB がおよそ 480（$= 300 \times 1.6$）エントリ以上存在すれば，キャッシュ・ミスを処理するのに十分な先行度が得られると考えられる．いま実 ROB エントリ数は 128 なので，$M = 8$ で仮想 ROB エントリ数は $128 \times 7 = 896$ となり，初めてこの条件を満たす．一方で，in-flight な先行実行命令が増え過ぎれば，早すぎるプリフェッチによってキャッシュを汚染することで性能低下を引き起こし得る．このようなことから，$M = 8$ においてピークを示したものと考えられる．

5.5　ROB 実エントリ数を変化させた場合の評価

図 10 に拡大倍率 M を一定とし，実 ROB エントリ数を 32 から 512 まで変化させた場合の各モデルにおける性能変化のグラフを示す．縦軸は表 1 の 8 本のベンチマークの IPC の幾何平均で，横軸は実 ROB エントリ数である．測定においては実 ROB エントリ数に応じて，本節の冒頭で示したバランスに従って ROB 以外の資源（発行キュー，LSQ，物理レジスタ）の量も変化させている．なお，図において IDEAL とは，PE モデルにおける拡大された ROB の総エントリ数と等しい数の実 ROB エントリをもつ理想モデルを表している．この IDEAL モデルの実 ROB エントリ数は括弧内に示した．

このグラフから，本手法は異なる実 ROB エントリ数においても有効であることがわかる．PE モデルの性能は，ROB 32 エントリのときを除いて常に BASE モデルを上回っている．BASE に対する性能向上率は ROB 128 エントリのときが最大で，平均で 46%であった．なお，ROB 32 エントリにおいては性能向上が得られていないが，これは論理レジスタ

ⓒ 2010 Information Processing Society of Japan

数に対する物理レジスタ数の比が小さくなるためである．このため ROB よりも先に物理レジスタが不足しやすくなり，先行実行が起こりにくくなっていると考えられる．

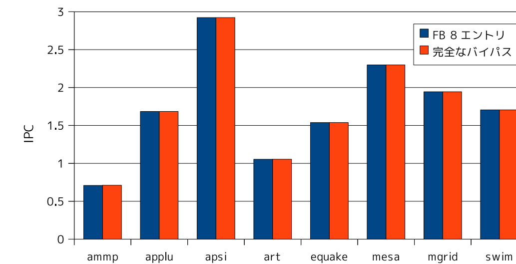
PE モデルは BASE モデルに対して大きく性能向上しているものの，IDEAL モデルには届いていない．これは本実行命令数の差が原因と考えられる．IDEAL では全ての in-flight 命令は本実行命令であるのに対し，$M = 8$ の仮割り当てを行っている PE においては，総 in-flight 命令の 7/8 は仮想 ROB を仮割り当てされただけの先行実行命令であり，本実行命令は残りの 1/8 のみである．先行実行はプロセッサ状態を更新できないため，直接に性能向上に寄与するのは，キャッシュをミスするロード命令の先行実行だけである．キャッシュ・ミスと無関係の先行実行は性能向上に寄与しない．従って有効な in-flight 命令数は PE の方が少なく，IDEAL の方が多くの命令レベル並列性を利用することができる．このために，保持できる最大の in-flight 命令の総数は等しいにも関わらず，PE モデルの性能は IDEAL を下回っている．

5.6 先行実行に物理レジスタを割り当てない影響

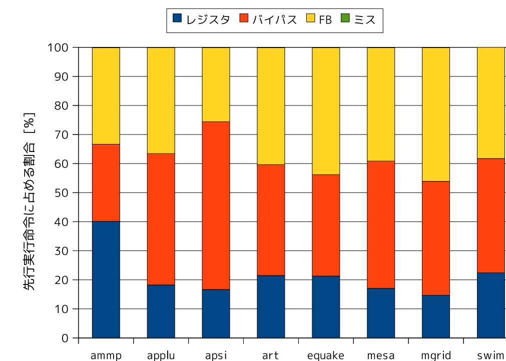
本手法では，命令を先行ディスパッチする際には ROB の他に物理レジスタも割り当てない．これによるデメリットとして，先行実行の結果を保持できないために，後続の依存命令が発行できなくなる可能性が生じることが挙げられる．先行実行の結果はバイパス論理もしくは FB 経由で受け渡されるが，バイパスは実行後 1 サイクルの間しか有効ではなく，また FB は小さいため全ての先行実行結果を保持することはできない．実行結果が失われれば，後続の依存命令は先行実行できなくなる．

図 11(a) に 8 エントリの FB を使用した場合と，完全なバイパス論理を仮定した場合の性能を比較したグラフを示す．ここで完全なバイパス論理とは，先行実行結果の受け渡しがどのようなタイミングであっても可能な理想的なバイパス論理を指す．これは先行実行結果を保持しておけることを意味するため，先行実行に物理レジスタを割り当てる場合と等価である．この図から，8 エントリの FB を用意することで，完全バイパス論理とほとんど等しい性能が実現できることがわかる．従って，先行実行に物理レジスタを割り当てないことによる性能への悪影響はほとんどないと言える．

また，**図 11(b)** は先行実行された命令において，そのソース・オペランドの供給元について分類したものである．測定は完全なバイパス論理を仮定して行っている．図において「FB」「バイパス」「レジスタ」はそれぞれ FB，通常のバイパス論理，物理レジスタからソース・オペランドを取得した命令の割合を表す．「ミス」は FB をミスし，本来は（完全なバイパス論理を仮定していなければ）発行できなかった命令の割合を示しているが，実際には図



(a) IPC



(b) ソース・オペランドの供給元

図 11 物理レジスタを割り当てないことの性能への影響

から読み取ることができないほど少なかった．測定の結果，平均で 38%の命令は FB からオペランドの供給を受けていた．このことから，先行実行結果の受け渡しを円滑に行うためには，FB が有効であることがわかる．

6. ま と め

データ・プリフェッチを実現する方法のひとつに，命令の先行実行がある．一般に，資源制約を緩和することによって，先行実行を実現することができる．本論文では ROB を仮想的に拡大し，実 ROB 及び物理レジスタを割り当てないまま命令を発行キューへ挿入し，先行実行させる方式を提案した．SPECfp2000 ベンチマークを用いて評価を行った結果，128

ⓒ 2010 Information Processing Society of Japan

エントリの ROB を 8 倍に仮想的に拡大した場合，本手法を用いない場合に比べ 46%の性能向上を達成した．

謝辞 本研究の一部は，日本学術振興会 科学研究費補助金基盤研究 (C)（課題番号 22500045）による補助のもとで行われたものである．

参 考 文 献

1) Akkary, H., Rajwar, R. and Srinivathan, S. T.: Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors, *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pp.423–434 (2003).
2) Borch, E., Manne, S., Emer, J. and Tune, E.: Loose Loops Sink Chips, *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pp.299–310 (2002).
3) Chen, T.-F. and Baer, J.-L.: Reducing Memory Latency via Non-blocking and Prefetching Caches, *Proceedings of the 5th International Conference on Architectural support for Programming Languajges and Operating Systems*, pp.51–61 (1992).
4) Collins, J.D., Tullsen, D.M., Wang, H. and Shen, J.P.: Dynamic Speculative Precomputation, *Proceedings of the 34th International Symposium on Microarchitecture*, pp.306–317 (2001).
5) Collins, J. D., Wang, H., Tullsen, D. M., Hughes, C., Lee, Y.-F., Lavery, D. and Shen, J. P.: Speculative Precomputation: Long-range Prefetching of Delinquent Loads, *Proceedings of the 28th International Symposium on Computer Architecture*, pp.14–25 (2001).
6) Cristal, A., Santana, O. J., Cazorla, F., Galluzzi, M., Ramírez, T., Pericàs, M. and Valero, M.: Kilo-Instruction Processors: Overcoming the Memory Wall, *IEEE Micro*, Vol.25, No.3, pp.48–57 (2005).
7) Cristal, A., Santana, O. J. and Valero, M.: Toward Kilo-instruction Processors, *ACM Transactions on Architecture and Code Optimization*, Vol.1, No.4, pp.368–396 (2004).
8) Cruz, J.-L., González, A., Valero, M. and Topham, N.P.: Multiple-Banked Register File Architectures, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp.316–325 (2000).
9) Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A. and Roussel, P.: The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal*, Vol.5, No.1, pp.1–13 (2001).
10) Joseph, D. and Grunwald, D.: Prefetching Using Markov Predictors, *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp.252–263 (1997).
11) Jouppi, N.P.: Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp.364–373 (1990).
12) Kessler, R.E.: The Alpha 21264 Microprocessor, *IEEE Micro*, Vol.19, No.2, pp. 24–36 (1999).
13) Latorre, F., Magklis, G., González, J., Chaparro, P. and González, A.: Building a Large Instruction Window through ROB Compression, *Proceedings of the 2007 Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, pp.41–48 (2007).
14) Lebeck, A.R., Koppanalil, J., Li, T., Patwardhan, J. and Rotenberg, E.: A Large, Fast Instruction Window for Tolerating Cache Misses, *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp.59–70 (2002).
15) Martínez, J.F., Renau, J., Huang, M.C., Prvulovic, M. and Torrellas, J.: Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors, *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pp. 3–14 (2002).
16) Mutlu, O., Stark, J., Wilkerson, C. and Patt, Y.N.: Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors, *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pp.129–140 (2003).
17) Petit, S., Sahuquillo, J., López, P., Ubal, R. and Duato, J.: A Complexity-Effective Out-of-Order Retirement Microarchitecture, *IEEE Transactions on Computers*, Vol.58, No.12, pp.1626–1639 (2009).
18) Purser, Z., Sundaramoorthy, K. and Rotenberg, E.: A Study of Slipstream Processors, *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pp.269–280 (2000).
19) Roth, A. and Sohi, G.S.: Speculative Data-Driven Multithreading, *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pp. 37–48 (2001).
20) : . `http://www.simplescalar.com/`
21) Yamamoto, A., Tanaka, Y., Ando, H. and Shimada, T.: Two-Step Physical Register Deallocation for Data Prefetching and Address Pre-Calculation, *IPSJ Transactions on Advanced Computing Systems*, Vol.1, No.2, pp.34–46 (2008).
22) Zilles, C. and Sohi, G.: Execution-based Prediction Using Speculative Slices, *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp.2–13 (2001).

© 2010 Information Processing Society of Japan