Vol.2014-ARC-212 No.16
2014/10/7

仮想リオーダ・バッファ方式における性能改善技術

蘇 翰[1]　塩谷 亮太[1]　安藤 秀樹[1]

概要：データ・プリフェッチを実現する方法の一つに命令の先行実行がある．過去に我々は，単一スレッド環境による命令の先行実行手法として仮想リオーダ・バッファ（VROB：virtual reorder buffer）方式を提案した．これまでの研究により，VROB 方式では，多くのロード命令のレイテンシが短縮され，大きな性能向上を達成できることを示した．しかし，まだ以下のような問題が残されている．(1) 性能向上に寄与しない命令も多く先行実行され，これによる資源競合により性能向上が妨げられている．(2) 命令間のオペランド受け渡しに，小容量のフォワーディング・バッファ（FB: forwarding buffer）を用いているが，受け渡しに失敗することがあり，性能向上が妨げられている．これらの問題を解決するために，本論文では二つの改善方法を提案する．(1) 性能向上に寄与しない浮動小数点命令の先行実行を省略することにより，資源競合を削減する．(2) FB の代わりに先行実行にレジスタ・ファイルを追加する．このレジスタ・ファイルは先行実行時の性質を利用することにより，小容量化する．SPEC2000 および SPEC2006 のメモリ・インテンシブなベンチマークを用いて評価を行った結果，提案手法を導入することにより，従来の VROB プロセッサに対し，8%の性能向上を達成した．この結果，通常のプロセッサに対し，27%の性能向上を達成することができることを確認した．

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．最終レベル・キャッシュ (LLC: last-level cache) のミスによって発生する長いロード・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一つに，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要求される前に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 [1], [9]．しかし，自動プリフェッチャは，過去のメモリ・アクセスの履歴に基づいてプリフェッチを行うため，一般に，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手法 [4], [8], [10], [13] 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・パターンに対応可能なプリフェッチ手法として，命令の先行実行がある [3], [5], [6], [12], [15], [16]．この手法は，典型的には，プログラムから一部の命令を抜きだし，本来の実行（本実行と呼ぶ）に先駆けて，事前に命令を実行する手法である．先行実行されたロードがキャッシュ・ミスを起こせば，それは本実行に対するプリフェッチとして動く．また，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．しかし，これまで多くの先行実行手法が提案されたが，ほとんどはマルチスレッド環境を必要とした．

これに対して，我々は，単一スレッド環境で命令の先行実行を実現する方式について研究を行ってきた [17], [18]．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），レジスタ・ファイル（RF：register file），発行キュー（IQ：issue queue），ロード/ストア・キュー（LSQ：load/store queue）といった資源が，インフライト命令数を規定し，命令の実行タイミングに大きな影響を与える．もしこれらのサイズを拡大せず資源制約を緩和できれば，本実行のインフライト命令数を越える命令を実行可能になり，先行実行を実現できると言える．

以上のような考えに基づき，我々は過去に，**仮想リオーダ・バッファ**（VROB：virtual reorder buffer）と呼ぶ方式を提案した [17], [18]．この手法では ROB の解放がキャッシュ・ミスを起こしたロード命令によってブロックされ，エントリが不足している場合，ROB，物理レジスタ及び LSQ を割り当てないまま，命令を先行実行する．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

[1] 名古屋大学大学院工学研究科電子情報システム専攻

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，ROB など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・オペランドが揃えば発行される．この実行を本実行と呼ぶ．本実行では，通常の実行と同様にプロセッサ状態を更新する．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．

これまでの研究 [17], [18] により，VROB 方式によって有効にプリフェッチが行われることが確認されているものの，依然として，以下のような性能向上を制限する問題がある．

- プリフェッチに貢献しない命令も先行実行される．これにより，資源が無駄に使用され，資源競合によりプリフェッチに有効な命令の早期実行を妨げている．
- コスト削減のため先行実行用にはレジスタ・ファイルを持たない．この代わりに，命令間のオペランドの受け渡しにバイパスを補助する小容量の**フォワーディング・バッファ**(FB: forwarding buffer) [2] と呼ぶバッファを用意している．しかし，FB は限られた容量という制限から，オペランドの受け渡しを完全に行えるものではなく，失敗することがある．この場合，先行実行は停止し，性能向上が妨げられる．

そこで，本論文では，上述の問題を解決するため，それぞれ，以下の手法を提案する．

- プリフェッチはロード命令により行われるが，ロードのアドレス生成に関与する命令は全て整数命令である．よって，浮動小数点 (FP: floating-point) 命令の先行実行は無駄である．そこで，FP 命令の先行実行を省略する．
- 先行実行用にレジスタ・ファイルを用意する．ただし，コスト削減のため，先行実行においては例外らの状態回復を行わないことを利用して，レジスタの生存について最短に近いタイミングで解放を行う．これにより，オペランドの受け渡しを完全にすることを，非常に少ないレジスタで行うことが可能となる．

本論文の以降の部分は次のような構成となっている．まず 2 節では，過去の研究における VROB 方式について述べる．3 節では，本論文で着目する VROB における問題について説明し，4 節で問題の解決手法を説明する．5 節で評価を行い，6 節で本論文をまとめる．

2. 仮想リオーダ・バッファ方式

本節では，VROB 方式における先行実行の効果を説明した後，過去の研究において提案した VROB の構成及び動作について具体的に説明する．

2.1 先行実行の効果

図 1 に VROB 方式における先行実行の効果を示す．図 1

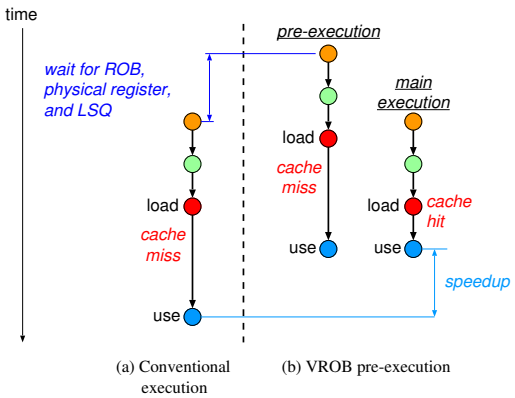


図 1 先行実行の効果

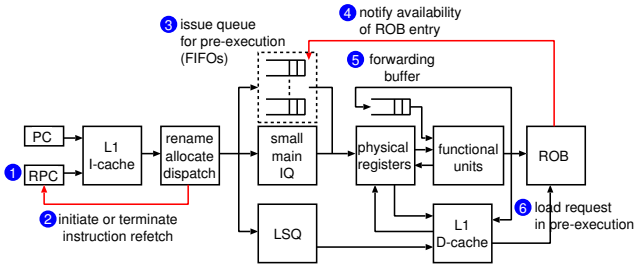


図 2 プロセッサの構成

(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．図に示すように，ROB，物理レジスタ，LSQ のいずれかが不足していた場合，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．

一方，図 1 (b) に示すように，本手法においては ROB，物理レジスタ，LSQ を割り当てないまま命令を先行実行する．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` はキャッシュにヒットするため，レイテンシは短縮され，性能は向上する．

2.2 これまでの研究における VROB プロセッサの構成

図 2 に，これまでの研究における VROB 方式を実装したプロセッサの構成を示す．従来のプロセッサと比べ，以下の構成要素が異なっている．

- 先行実行された命令を再フェッチし，本実行として再実行するための PC（RPC: refetch PC）を備える（図 2①）．
- 命令の再フェッチの開始・停止を指示する信号を，ディスパッチ・ステージから RPC へ出力する（図 2②）．
- 本実行用の発行キュー (M-IQ: main issue queue) の他に，先行実行用の発行キュー（P-IQ：pre-execution issue queue）を備える．P-IQ は，複雑度を抑制する

ため，FIFO で構成する [11] （図 2③）．
- P-IQ に挿入されたが，まだ先行実行されていない命令に対し，本実行のための ROB の空きエントリが生じたことを伝える信号を伝送する（図 2④）．
- 先行実行用にはレジスタ・ファイルはなく，代わりに，バイパスを補助する小容量のフォワーディング・バッファを備え，先行実行命令間のオペランドの受け渡しを行う（図 2⑤）．
- 先行実行用に LSQ はなく，メモリ依存をチェックすることなく，機能ユニットよりデータ・キャッシュをアクセスする（図 2⑥）．

2.3 先行実行・本実行

従来のプロセッサでは，命令に ROB，物理レジスタ，LSQ を割り当てることができない場合，命令の実行はストールする．これに対し，VROB 方式では ROB が不足している場合には，これらを割り当てずに命令を P-IQ 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，先行実行される．

先行実行は物理レジスタを割り当てられていないため，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命令に受け渡すことはできる．ただし，バイパス論理による結果の受け渡しは実行後 1 サイクルしか有効でない．この制約を緩和するために，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④によって先行実行用 IQ から削除されれ (2.4 節で詳述)，先行実行は停止する．

先行実行した命令はアーキテクチャ状態を更新しないため，実行した後，再実行する必要がある．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，それ用の PC である RPC を用いて行う．図 2②に示すとおり，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．再フェッチした命令は，**図 3** に示すとおり再フェッチ・キュー（RFQ: refetch queue）と呼ぶ一時バッファへ格納する．一方，PC によってフェッチされた命令は，フェッチ・キュー（FQ: fetch queue）と呼ぶ別のバッファへ格納される．

再フェッチは，先行ディスパッチされた命令が全て本実行されるまで継続する．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の

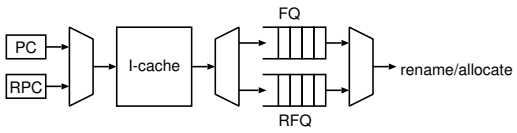


図 3 命令フェッチの構成

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される．一方，再フェッチした命令を本実行用 IQ へ挿入した際にデクリメントされる．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．これは本実行のスループットの方が，先行実行よりも性能において重要となるからである．また，リネーム・ステージにおける FQ または 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，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 FQ から読み出す．

IQ は，単純には先行実行を余分に行うため大きくする必要があるが，CAM で構成された大きな IQ はクロック・サイクル時間に悪影響を与える．そこで，先行実行用には複雑度が低い依存ベースの FIFO IQ [11] を用いる．この方法では，IQ を複数の FIFO によって構成する．ディスパッチの際には，依存している命令が格納されている FIFO におけるその命令の直後のエントリに書き込む．これにより FIFO 内の命令はイン・オーダで発行すればよく，アウト・オブ・オーダ発行のためのウェイクアップ及び選択は各 FIFO の先頭の命令のみを対象とすればよくなる．だだし，依存している命令が存在しない場合には空きの FIFO に書き込まなければならないため，空の FIFO がなければストールする．しかし，先行実行は本実行と異なり命令処理のスループットを決定するものではないから，多少のストールは寛容できる．一方，本実行用には，追加した先行実行用 IQ の複雑度だけ軽減した，すなわち，従来より少ないエントリ数の CAM で構成した IQ を用いる．先行実行によるロード・レイテンシの短縮により，本実行時の IQ への圧迫は小さくなっており，小さな IQ でも十分となる．

2.4 先行ディスパッチ命令の削除

2.4.1 概要

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 P-IQ 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

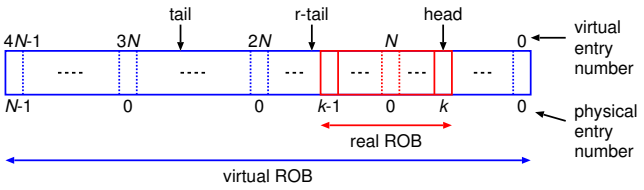


図 4 仮想 ROB（$M = 4$）

能となり，先行実行用 IQ に取り残される．

これらの場合に対応し，次のようにして資源の利用可能性を P-IQ 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，そのエントリの ID 番号（2.4.2 節で述べる仮想エントリ番号）を P-IQ へ放送する．P-IQ では，先行ディスパッチされた命令について，そのエントリが，もし ROB に空きがあれば自身が割り当てられたはずのエントリかどうかを判断する．もしそうであれば，その命令を P-IQ 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 M-IQ へ再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，その他の資源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，全ての資源の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．

2.4.2 ROB の利用可能性の伝達

リネーム時に ROB が満杯でエントリを割り当てることができなければ，もしも空いていたとするなら割り当てられたはずの ROB のエントリを命令に割り当てる．これを**先行割り当て**と呼ぶ．これは概念的には ROB を仮想的に拡大したことに相当し，ROB に関する資源制約を緩和し先行実行を可能とする．

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 全体の仮想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．（仮想エントリ番号 mod N）が物理エントリ番号となる．したがって，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．以後，仮想的に拡大された ROB 全体を仮想 ROB，実在の ROB を実 ROB と呼ぶ．

仮想 ROB の先頭と末尾を，それぞれ head，tail ポインタが指す．一方，実 ROB の先頭は仮想 ROB と同一であり head ポインタが指すが，末尾は別途 r-tail（real tail）と呼ぶポインタが指す（これらのポインタは全て仮想エントリ番号を持つ）．リネーム・ステージにおいて ROB が満杯の場合，命令には仮想エントリを割り当て，tail ポインタのみを更新する．これが前述した先行割り当てである．

図 5 に，先行ディスパッチされる命令の P-IQ への挿入及び削除の様子を表す．命令は，先行割り当てされた ROB エントリの仮想エントリ番号と共に P-IQ へ挿入される（①）．仮想エントリ番号を保持するために，P-IQ の各エントリに PRE（pre-allocated ROB entry）フィールドを追加する．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たら，その $((\text{仮想エントリ番号} + N) \bmod (N \times M))$ が P-IQ へ放送される（②，図 2③も参照）．この値は，解放された物理エントリが対応する仮想エントリの番号を表している．P-IQ では，先行ディスパッチされた命令について，その PRE フィールドが保持している仮想エントリ番号と放送されてきたエントリ番号と比較する．一致すれば，そのエントリに割り当てられた命令に先行割り当てされた ROB エントリが利用可能となったことを意味する．この場合，その命令を P-IQ から削除する（③）．なお，2.3 節で述べたように，先行ディスパッチされた命令は即座に再フェッチされるため，削除された命令は多くの場合，既に RFQ の先頭で待ち合わせている．

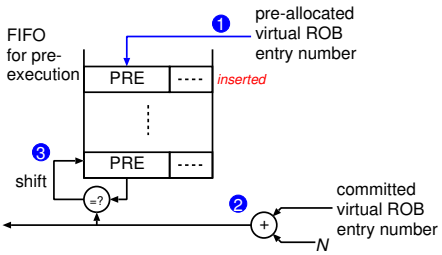


図 5 発行キューへの挿入及び削除
（N は実 ROB サイズ）

3. 性能向上を制限している問題

過去の研究により，VROB は有効な手法であることを示した [17], [18]．しかし，依然として，性能向上を制限している問題がある．本節はこれらの問題について説明する．

3.1 無用な先行実行命令

VROB では，先行実行によりロードを早期に実行し，プリフェッチ効果を得る．したがって，ロードが依存しない命令を実行することは資源を無駄に消費するだけであり，資源競合により性能向上に悪影響を与える．ここで，ロードのアドレスは整数であり，その生成に関与する命令も全て整数命令である．したがって，FP 命令は先行実行する価値のない無駄な命令と言える．

図 6 に，従来の VROB において，先行ディスパッチされた命令に占める整数 (INT: integer) 命令と FP 命令の割合を示す（プロセッサの構成については，**表 2** および**表 3** を参照）．同図より，INT 系のベンチマークでは，先行ディスパッチされた命令の中に，FP 命令はほとんど存在しないが，FP 系ベンチマークの場合，平均で 58%の先行ディスパッチ命令が FP 命令であり，資源を非常に無駄に消費していることがわかる．FP 系プログラムにおいては，FP

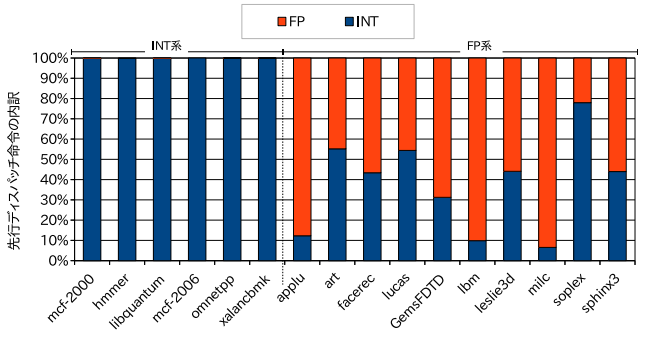


図 6 先行ディスパッチされた命令が整数系か浮動小数点系かの分類

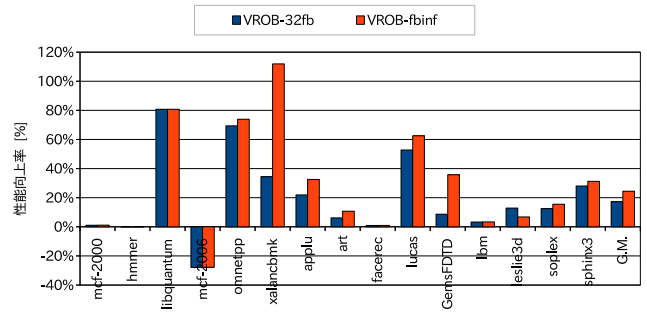


図 7 32 エントリと無限サイズの FB を持つ VROB の性能比較

命令の先行実行を省略すれば，資源競合が緩和され，ロードがより早期に実行され，その結果，さらなる性能改善が得られると考えられる．

3.2 オペランド受け渡しの失敗

2.3 節で述べたように，従来の VROB では，先行実行時の命令間のオペランドの受け渡しは，バイパス論理とそれを補助する FB で行われていた．しかし，限られたサイズの FB は生きている変数を全て保持することはできず，受け渡すべきオペランドが存在しないことがある．これによりオペランドの受け渡しに失敗すれば，先行実行は停止し，性能向上の機会が失われる．

図 7 に，オペランド受け渡し失敗による性能への悪影響を評価した結果を示す．各ベンチマークについて 2 本ある棒グラフは，FB のエントリ数が 32 の場合 (従来の VROB)(VROB-32fb) 及び FB が無限大の場合 (オペランドの受け渡しに失敗しない場合)(VROB-fbinf) の VROB の基準プロセッサ (通常のプロセッサ) に対する性能向上率である (プロセッサの構成は表 2，表 3 を参照)．同図からわかるように，FB のサイズが 32 の場合，性能向上率は平均 17%であるのに対して，FB を無限大にすると，性能向上は 24%に大幅に増加する．これより，オペランド受け渡しの失敗が大きく性能を制限していることがわかる．

4. 性能改善技術

本節は，前節で述べた問題を解決する手法について説明する．

4.1 先行実行における FP 命令の削除

FP 命令は，P-IQ にディスパッチする際に削除し，P-IQ には挿入しない．ここで，削除する FP 命令とは，具体的には，ソースまたはデスティネーション・レジスタが浮動小数点型である命令とする．ただし，浮動小数点レジスタに値をロードするロード命令は，プリフェッチとして働くので，削除しない．

4.2 先行実行用レジスタ・ファイル

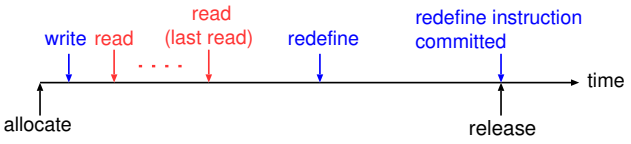
FB を廃止し，先行実行用のレジスタ・ファイルを設けることによりオペランドの受け渡しを完全に行えるようにする．このとき，VROB における先行実行の特徴を利用すれば，レジスタ・ファイルの必要サイズを大幅に抑えられることを示す．

まず，4.1 節で述べたように，FP 命令は先行実行しないので，浮動小数点レジスタ・ファイルは不要である．つまり，整数レジスタ・ファイルしか必要でない．よって，必要サイズを半分にできる．ここで，浮動小数点ロードは先行実行するため，通常，物理レジスタが割り当てられなければならないが，割り当てないとする．その理由は，1) 浮動小数点デスティネーション・レジスタを参照する命令は，削除され，実際に参照されることはない．2) 浮動小数点ロードの実行は，プリフェッチの機能を果たせばよく，値のレジスタ書き込みには意味はない．

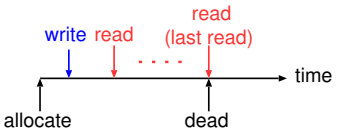
次に，先行実行においては，分岐予測ミスや例外 (以下，まとめて例外と呼ぶ) からの回復の必要がない (正確には，それらを行うことができない) ので，レジスタの開放タイミングを従来のプロセッサでのタイミングより大幅に早期化できる．これにより，同時に保持すべきレジスタ値が減少し，その結果，レジスタ・ファイルを小さくできる．

例外からの回復を行わなければならない実行においては，物理レジスタの開放は，それを再定義する命令がコミットされるまでである．しかし，回復を必要としなければ，最短で，物理レジスタの最後の参照時に開放できる．そこで，我々は，物理レジスタの生存情報を管理することによりレジスタの開放を早期に行う手法を提案する．これにより，レジスタ・ファイルの必要サイズは，さらに小さくすることができ，最大必要サイズの 12.5%に削減できる．

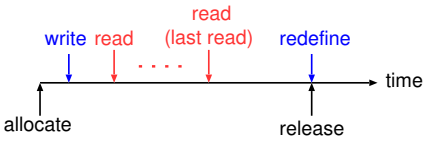
本節の以降の部分では，まず，物理レジスタの生存期間と開放タイミングについて説明する．次に，生存情報を利用する物理レジスタの開放について提案する．



(a) 従来のプロセッサでの物理レジスタ開放のタイミング



(b) 理想的な物理レジスタ開放のタイミング



(c) 提案する物理レジスタ開放のタイミング

図 8 物理レジスタの生存期間と開放のタイミング

4.2.1 物理レジスタの生存期間と開放のタイミング

従来のプロセッサでは，例外からの回復のために ROB を使っている．それは，次のように行われている．まず，レジスタ・リネーミング時に，論理デスティネーション・レジスタに割り当てられていた古い物理レジスタ番号を ROB に書き込む．例外を起こした命令がコミットされるとき，ROB の末尾から先頭に向かって，上述した古い物理レジスタ番号を使って，レジスタのマッピングを元に巻き戻す．こうすることにより，アーキテクチャ・レジスタ状態は，例外を起こした命令がリネーミングされる直前にまで回復される．一方，例外を起こしていない命令がコミットされるときは，古い物理レジスタは解放される．この手法では，ある物理レジスタが解放されるのは，それがマップされていた論理レジスタを再定義する命令がコミットされる時である．**図 8** (a) は，そのタイミングを説明するものである．最初に論理デスティネーション・レジスタに物理レジスタが割り当てられ (allocate)，使用が始まる．その後，何回か読み出される．そして，同じ論理デスティネーション・レジスタを再定義する命令が現れ，それがコミットされるときに，最初に割り当てられた物理レジスタは解放される．

この方式は，物理レジスタの使用の開始から開放までが，レジスタの生存期間に比べて非常に長い．図 8 (b) において，「dead」と示した点が，すなわち，最後の読み出しが行われた時点がレジスタ生存の終点である．その後，再定義命令の出現とそのコミットを待たなければレジスタは解放されない．

これに対して，VROB の先行実行では，例外が発生した時には，それを無視し，回復も行わず，先行実行を停止する．先行実行には実 ROB が割り当てられていないので，回復する手段を持たないからである．回復を行わないので，理想的には，最後の読み出しが終わった時点で開放が可能である．しかし，ハードウェアには，ある読み出しが最後かどうかの判断は困難である．そこで，図 8 (c) に示すように，再定義命令が現れた時点で解放するとする．この手法は，レジスタの使用期間が，生存の終点で解放する理想状態よりは長いが，従来よりは大幅に短い．

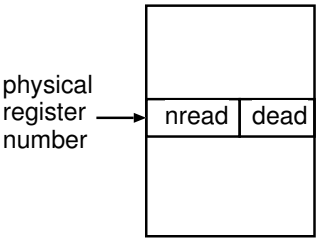


図 9 先行実行用の物理レジスタの生存情報を管理する表 (live_tab)

4.2.2 物理レジスタの早期解放

4.2.1 節で説明した方式をプロセッサに実装するには，以下の 2 点が必要となる．

- プログラム順の命令の流れから，再定義を見つける．
- アウト・オブ・オーダの命令の流れから，最後の読み出しを見つける．

上記のようにして，再定義と最後の読み出しが見つかれば，物理レジスタを開放できる．まとめると，以下の条件が成立した時，ある論理レジスタ `lreg` にマップされた物理レジスタ `preg` を開放できる．

> フロントエンド (リネーム・ステージ) に，`lreg` の再定義命令が現れた後であって，それに先行する `lreg` の全ての読み出しが，レジスタ読み出しステージにおいて，完了すること．

この実装のために，物理レジスタの生存情報を管理する `live_tab` と呼ぶ表 (**図 9** 参照) を用意する．`live_tab` の各エントリは，物理レジスタに対応し，以下の 2 つの情報を保持する．

- `nread`: ペンディングとなっている読み出し予定回数を保持するカウンタ
- `dead`: 先行する全ての読み出しは完了していないが，再定義命令がフロントエンドに現れたことを示すフラグ

論理デスティネーション・レジスタを新たな物理レジスタにマップした時に，`nread`, `dead` ともに 0 に初期化する．その後の，物理レジスタ開放のアルゴリズムを，**図 10** と **図 11** を用いて説明する．同図において，`lsreg`，`ldreg` は，それぞれ，論理ソース・レジスタ番号，論理デスティネーション・レジスタ番号であり，`psreg`，`pdreg` は，それぞれ，上記論理レジスタにマップされている物理ソース・レジスタ番号，物理デスティネーション・レジスタ番号である (`pdreg` は，新たにマップされた物理レジスタではな

```
1 foreach lsreg {
2   live_tab[psreg].nread++;
3 }
4 if (instruction has ldreg) {
5   if (live_tab[pdreg].nread == 0)  // check if all reads are completed
6     release pdreg;
7   else
8     live_tab[pdreg].dead = 1;  // redefined, but reads are not completed
9 }
```

図 10 物理レジスタ開放のアルゴリズム (リネーム時)

```
1 foreach lsreg {
2   live_tab[psreg].nread--;
3 }
4 // check if lsreg is redefined and all reads are completed
5 if (live_tab[psreg].nread == 0 && live_tab[psreg].dead)
6   release psreg;
7 }
```

図 11 物理レジスタ開放のアルゴリズム (レジスタ読み出し時)

く，古い物理レジスタである).

- **リネーム時**(図 10 参照): 論理ソース・レジスタ lsreg にマップされている物理レジスタ psreg に対応する live_tab のエントリの nread をインクリメントする (2 行目). 一方，論理デスティネーション・レジスタ ldreg にマップされていた物理レジスタ pdreg に対応する live_tab のエントリの nread がゼロならば，当該物理レジスタは全ての読み出しを完了しているので，pdreg を解放する (5，6 行目). そうでなければ，live_tab の pdreg に対応するエントリの dead フラグをセットし，当該物理レジスタの全ての読み出しが完了したら開放できることを示す (8 行目).
- **レジスタ読み出し時**(図 11 参照): 論理ソース・レジスタ lsreg にマップされている物理レジスタ psreg に対応する live_tab のエントリの nread をデクリメントする (2 行目). もし，その値がゼロで，dead フラグがセットされているなら，再定義命令が現れており，かつ，以後その物理レジスタを読み出す命令はない (4 行目). よって，psreg を解放する.

先行実行用レジスタは，本実行と同様，フロントエンドにおいて割り当てられる. レジスタ・ファイル・サイズが小さければ，割り当てるレジスタが不足し，ストールするが，FB の場合のように，オペランドの受け渡しに失敗することはない.

5. 評価

5.1 評価環境

評価には，SimpleScalar Tool Set Version 3.0a [14] をベースに提案手法を実装したをシミュレータ用いた. 命令セットには Alpha ISA を用いた. ベンチマーク・プログラムとして，SPEC2000 と SPEC2006 の中のメモリ・インテンシブなプログラムを使用した. 閾値を 10 サイクルとして，平均ロード・レイテンシが 10 サイクル以上のプログラムをメモリ・インテンシブとし，それ未満のプログラムを計算インテンシブとした. **表 1** に，この基準で分類したベンチマーク・プログラムを示す. VROB 方式の目標はメモリ・アクセスのレイテンシを短縮するためであるから，メモリ・インテンシブなプログラムのみを選択して評価を行った. バイナリは，SPEC2000 のプログラムについては，Compaq C 及び Fortran コンパイラを用い-fast -O4 のオプションでコンパイルした. SPEC2006 のプログラムについては，gcc 4.5.3 を用い-O3 のオプションでコンパイルした. 入力には ref 入力を用い，SPEC2000 のプログラムの場合は SimPoint [7] によって選択した 100M 命令を，SPEC2006 のプログラムの場合は 16G の命令をスキップして 100M の命令を実行し，評価した.

評価におけるベース・プロセッサの構成を表 2 に示す. ベース・プロセッサは先行実行を行わない通常のプロセッサである. また，従来の VROB 方式における構成を表 3 に示す. 以後，従来の VROB 方式のプロセッサを，VROB ベースと呼ぶ.

5.2 FP 命令を先行実行から削除する効果についての評価

図 12 に，FP 命令を先行実行から削除することの効果の評価結果を示す. 各ベンチマークについて 2 本の棒グラフがあるが，それぞれ，VROB ベース，FP 命令を先行実行

表 1　プログラムの分類

		memory-intensive	compute-intensive
SPEC2000	INT	mcf	bzip, crafty, eon, gap, gcc, gzip, parser, perlbmk, twolf, vortex, vpr
	FP	applu, art, farcerec, lucas	ammp, apsi, equake, fma3d, galgel, mesa, mgrid, sixtrack, swim, wupwise
SPEC2006	INT	hmmer, libquantum, mcf, omnetpp, xalancbmk	astar, bzip2, gcc gobmk, h264ref, perlbench, sjeng
	FP	GemsFDTD, lbm, leslie3d, milc, soplex, sphinx3	bwaves, cactusADM, dealII, gamess, gromacs, namd, povray, tonto, zeusmp

表 2　ベース・プロセッサの構成

Pipeline width	4-instruction wide for each of fetch, decode, issue, and commit
Real ROB	128 entries
Fetch queue	16 entries
Issue queue	128 entries
LSQ	128 entries
Physical regsiter	128 for int and fp
Branch prediction	16-bit history gshare, 64K-entry PHT 10-cycle misprediction penalty
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
Data prefetcher	stride-based, 4K-entry, 4-way table, 16-data prefetch to L2 cache on miss

表 3　VROB ベース・モデルの構成

Virtual ROB	1024 entries (ROB multiplicity, $M = 8$)
Main issue queue	64 entries
Pre-exec issue queue	64 FIFO buffers, 4 entries each
Refetch queue	16 entries
Forwarding buffer	32-entry CAM, LRU replacement
Load request queue	8 entries

から削除した場合 (VROB-fpremove) の，ベースに対する性能向上率である．同図よりわかるように，FP 命令をほとんど含まない INT 系のプログラムでは，FP 命令の実行を削除する効果はほとんどないが，FP 系のプログラムでは，ほとんどのプログラムで VROB ベースに対して性能向上を達成している．ベースに対する性能向上率は，VROB

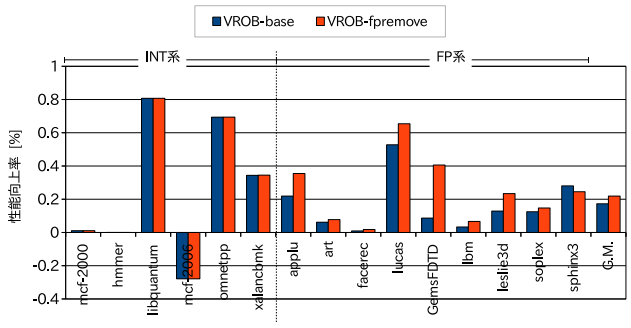


図 12　先行実行から FP 命令を削除した場合の性能向上率

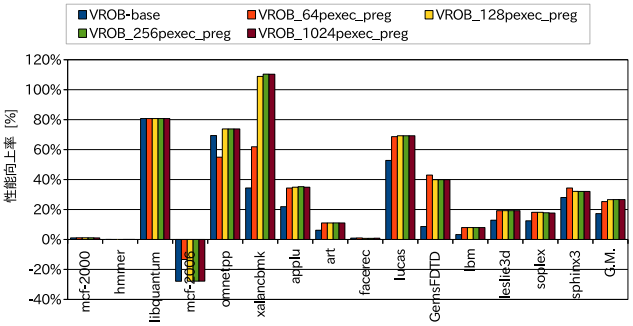


図 13　先行実行用のレジスタ・ファイル・サイズを変えた場合の性能向上率

ベースでは 17%であるが，VROB-fpremove では 22%を達成している．

5.3　先行実行用の物理レジスタの評価

図 13 に，VROB ベース，および，種々のサイズの先行実行用レジスタ・ファイルのサイズの VROB の，ベースに対する性能向上率を示す．凡例の「VROB_Xpexec_preg」は，先行実行用レジスタ・ファイルが X 個のレジスタを持つことを示す．仮想 ROB のサイズが 1024 なので，必要レジスタ数の最大値は 1024 である．つまり，このときが最大性能を示す．本節の評価では，FP 命令の削除も行っているので，先行実行用レジスタ・ファイルは，整数用のみである．

同図よりわかるように，平均では，レジスタ数が 64〜1024 の範囲で，性能向上率に大きな差は見られないが，64 の場合，一部のプログラム (*omnetpp*，*xalancbmk*) で 128 以上の場合に比べて大きな性能低下が見られる．よって，性能を最大化するには，128 のレジスタが必要であることがわかる．また，どのプログラムにおいても，レジスタ数 128 でほぼ最大性能に達しており，4.2 節に述べた手法により，必要レジスタ数は，最大必要数の 12.5%に削減されていることがわかる．VROB ベースのベースに対する性能向上率は，17%であるのに対して，128 の先行実行用レジスタを持つ VROB は，27%の性能向上率を達成している．

6. まとめ

データ・プリフェッチを実現する方法の 1 つに命令の先行実行がある．過去に我々は，単一スレッド環境において先行実行を実現する手法として VROB 方式を提案した．これまでの研究により，VROB 方式は有効なプリフェッチ手法であることを示したが，VROB 方式にはまだ性能向上を抑制している問題が存在する．その 1 つは，プリフェッチに寄与しない FP 命令の実行による不要な資源競合の発生であり，もう 1 つは，FB 利用による命令間のオペランド受け渡しの失敗に起因する先行実行の停止である．

これに対し，本論文ではこれらの性能低下問題を改善する手法を提案した．1 つは，FP 命令を先行実行から削除することであり，もう 1 つは，FB の代わりに，先行実行用のレジスタ・ファイルを用意することである．後者においては，物理レジスタの再利用を加速する手法により，必要なレジスタ・ファイル・サイズを大幅に削減した．SPEC2000 と SPEC2006 のメモリ・インテンシブなプログラムを用いて評価を行った結果，提案手法を導入することにより，従来の VROB プロセッサに対し，8%の性能向上を達成した．この結果，通常のプロセッサに対し，27%の性能向上を達成できることを確認した．

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 25330057）による補助のもとで行われた．

参考文献

[1] Baer, J. L. and Chen, T. F.: An effective on-chip preloading scheme to reduce data access penalty, *Proceedings of the 1991 Conference on Supercomputing*, pp. 176–186 (1991).

[2] Borch, E., Tune, E., Manne, S. and Emer, J. S.: Loose Loops Sink Chips, *Proceedings of the 8th Annual International Symposium on High Performance Computer Architecture*, pp. 299–310 (2002).

[3] Chappell, R., Stark, J., Kim, S., Reinhardt, S. and Patt, Y.: Simultaneous Subordinate Microthreading (SSMT), *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 186–195 (1999).

[4] Collins, J. D., Sair, S., Calder, B. and Tullsen, D. M.: Pointer Cache Assisted Prefetching, *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pp. 62–73 (2002).

[5] Collins, J. D., Tullsen, D. M., Wang, H., Lee, Y., Lavery, D., Shen, J. P. and Hughes, C.: Speculative Precomputation: Long-Range Prefetching of Delinquent Loads, *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 14–25 (2001).

[6] Collins, J. D., Tullsen, D. M., Wang, H. and Shen, J. P.: Dynamic Speculative Precomputation, *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pp. 306–317 (2001).

[7] Hamerly, G., Perelman, E., Lau, J. and Calder, B.: SimPoint 3.0: Faster and More Flexible Program Phase Analysis, *The Journal of Instruction-Level Parallelism*, Vol. 7, pp. 1–28 (2005).

[8] Joseph, D. and Grunwald, D.: Prefetching using Markov Predictors, *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 252–263 (1997).

[9] 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).

[10] Lai, A., Fide, C. and Falsafi, B.: Dead-Block Prediction and Dead-Block Correlating Prefetchers, *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 144–154 (2001).

[11] Palacharla, S., Jouppi, N. P. and Smith, J. E.: Complexity-Effective Superscalar Processors, *Proceedings of 24th Annual International Symposium on Computer Architecture*, pp. 206–218 (1997).

[12] 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).

[13] Roth, A. and Sohi, G. S.: Effective Jump-Pointer Prefetching for Linked Data Structures, *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 111–121 (1999).

[14] *http://www.simplescalar.com/*.

[15] Zilles, C. and Sohi, G. S.: Master/Slave Speculative Parallelization, *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pp. 85–96 (2002).

[16] Zilles, C. B. and Sohi, G. S.: Execution-Based Prediction using Speculative Slices, *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 2–13 (2001).

[17] 稲垣貴範，塩谷亮太，安藤秀樹：仮想リオーダ・バッファ方式におけるロード/ストア・キューの単純化，2012 年先進的計算基盤システムシンポジウム SACSIS 2012，pp. 262–269 (2012).

[18] 市原敬吾，田中雄介，安藤秀樹：仮想化により拡大したリオーダ・バッファによる先行実行，2011 年 先進的計算基盤システムシンポジウム SACSIS 2011，pp. 64–71 (2011).