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

市原敬吾[†,☆] 田中雄介[†,☆☆] 安藤秀樹[†]

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．本論文では，リオーダ・バッファ（ROB: reorder buffer）及び物理レジスタによる資源制約を緩和し，単一スレッド環境において先行実行を実現する**仮想リオーダ・バッファ**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し，先行実行する．先行実行された命令は，後にこれらの資源が利用可能となった時点で再び実行する．先行実行においてロードがキャッシュ・ミスを起こせば，それがプリフェッチとなり，再実行時のロード・レイテンシは短縮される．SPECfp2000 ベンチマークを用いて評価を行った結果，実際には 128 エントリの ROB を仮想的に 8 倍に拡大し，本手法を用いない場合に比べ 35%の性能向上を達成した．

Instruction Pre-Execution through a Reorder Buffer Enlarged by Virtualization

Keigo Ichihara,[†,☆] Yusuke Tanaka[†,☆☆] and Hideki Ando[†]

Instruction pre-execution is one of the schemes for prefetching data. This paper proposes a scheme, called a *virtual reorder buffer*, that allows pre-execution in a single-thread environment by relaxing resource constraints on a reorder buffer (ROB) and physical register file without enlarging them. In our scheme, if the ROB is full, which typically occurs when L2 cache misses occur, instructions are pre-executed without allocating either ROB entries or physical registers. Relaxing resource constraints ensures that pre-execution is performed. Pre-executed instructions are later re-executed to build the architectural state once the necessary resources become available. If a pre-executed load causes a cache miss, it moves data to the cache from main memory, thereby allowing the same load to hit the cache in the later re-execution. Our virtual ROB scheme organizes pre-execution and re-execution efficiently. Evaluation results using the SPECfp2000 benchmark programs show that our scheme, with a real 128-entry ROB virtually enlarged 8 times, improves performance by 35% 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．これ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要求より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（例えば 3)，9)）．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（例えば 8)）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別のプリフェッチ手法として，命令の**先行実行**がある（例えば 4)，14)）．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）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，これら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．もしこれらのサイズを拡大せず制約を緩和し，in-flight

† 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University
☆ 現在，オークマ株式会社
Presently with Okuma Corporation
☆☆ 現在，株式会社デンソー
Presently with Denso Corporation

命令数を越える命令を実行可能となれば，クロック速度に悪影響を与えず先行実行を実現できる．

本論文では，資源制約を緩和することで，本来の実行を行いながら同時に命令の先行実行を行う**仮想リオーダ・バッファ**（VROB: virtual reorder buffer）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先行実行用の発行キューへ挿入する．この発行キューは，後述する本実行用の発行キューを含め，発行キュー全体の複雑度を増加させない構成をとっている．挿入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．先行実行結果は物理レジスタへ書き込むことはできないが，結果をバイパス論理もしくは**フォワーディング・バッファ**[2] と呼ぶ小さな一時バッファによって後続の命令に受け渡すことで，先行実行は連続的に行われる．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 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．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．

VROB 方式は，先行実行のために本来余分に必要とされる重要な資源のうち，ROB，レジスタ・ファイルのサイズを増加させず、また、発行キューの複雑度も増加させないという特徴がある．しかし現在のところ，ロード/ストア・キュー（LSQ: load/store queue）のサイズについては未解決である．これについては，現在検討中であり，本論文では LSQ は十分に存在すると仮定することとする．

本論文の残りの部分は次のような構成となっている．まず 2 節で関連研究について述べる．次に 3 節で VROB 方式による先行実行の効果を示し，4 節で VROB 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（例えば 4)，14)）．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という欠点がある．

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[10]．この手法は L2 キャッシュ・ミス（本論文では最終レベル・キャッシュを L2 とする）が生じると，プロセッサ状態をチェックポイントし，ミスが解決するまで runahead モードと呼ぶ特別のモードに移る．このモードでは，ミスした命令に依存していない命令を実行する．この時，ロードがキャッシュ・ミスを起こせば，データがプリフェッチされる．この手法の欠点は，runahead 実行中にはプロセッサ状態を更新する本来の命令実行を並行して行えないという点である．

2.2 ROB サイズの削減

Ponomarev らは ROB の不使用領域を非アクティブとする手法を提案した[13]．しかし，メモリ・インテンシブなプログラムにおいては ROB の需要は高いため，このような消極的方法だけでは ROB サイズを十分に削減できない．

Petit らは非投機状態となった命令をアウト・オブ・オーダでリタイアさせる validation buffer と呼ぶ方式を提案した[12]．この手法は L2 キャッシュ・ミスによって ROB が詰まる問題を軽減し，ROB サイズを削減できる．しかし，命令をリタイアさせるためには例外を生じないことが確認されなければならず，例外を発生し得る命令（分岐命令を含む）は多いため，その効果は限定的である．特に，そういった命令が L2 キャッシュ・ミスを起したロード命令に依存している場合，キャッシュ・ミスが解決されるまで後続の命令はリタイアできない．

2.3 チェックポイント回復

チェックポイント回復では，ROB を用いる代わりに，特定の間隔でチェックポイントと呼ばれるプロセッサ状態のコピーを作成する．この手法の欠点は，単純な実装法では多くのチェックポイントを保持するためにハードウェア量が増大することである．また，レジスタ・ファイル（もしくはマップ表）の復元を伴うため，チェックポイントからの状態回復のオーバーヘッドも大きい．これは特に分岐予測ミスからの回復において問題となる．

Akkary らは分岐予測の信頼性が低い場合のみ選択的にチェックポイントを採取する手法を提案した[1]．これによりハードウェア量の増大を抑制することはできるが，状態回復のオーバーヘッドの問題は解決されない．

Cristal らはチェックポイント回復を ROB と組み合わせる手法を提案した[5]．この手法では，実行が完了しているかどうかにかかわらず，ROB から命令を定期

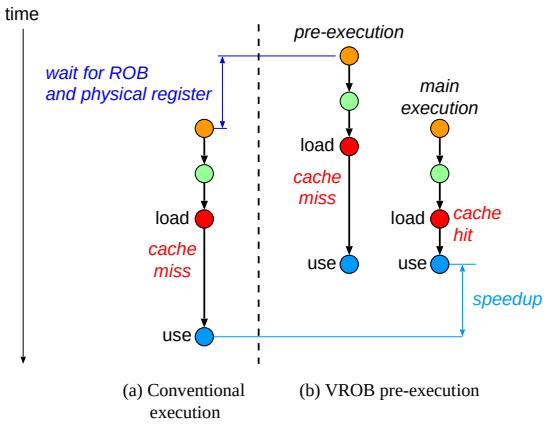


図 1 先行実行の効果

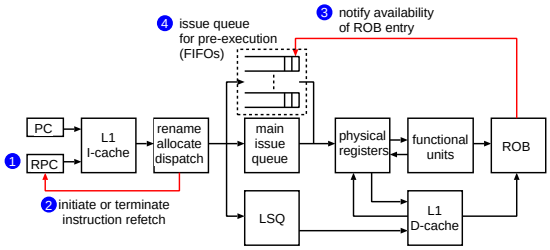


図 2 プロセッサの構成

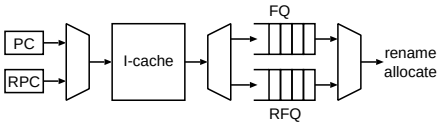


図 3 命令フェッチの構成

的に削除する．チェックポイントは実行が未完了の命令を削除する場合のみ採取する．ROB を併用することで，特に分岐予測ミスからの状態回復のオーバーヘッドを削減することができるが，ROB が小さいとチェックポイントも頻繁に採取しなければならなくなる．

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．

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. VROB 方式を用いた命令の先行実行

図 2 に VROB 方式を実装したプロセッサの構成を示す．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，③ROB から先行実行用発行キューへ ROB に空きエントリが生じたことを伝える信号，及び④先行実行用の FIFO 発行キューが追加されている．

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，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先行実行用の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てられていないため，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 節で詳述）．

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 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）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．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．

発行キューは，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発行キューはクロック速度に悪影響を与える．そこで，先行実行用には複雑度が低い依存ベースの FIFO 発行キュー[11] を用いる．この方法では，発行キューを複数の 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．ただし，依存している命令が存在しない場合には空の 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発行キュー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 で構成した発行キューを用いる．先行実行によるロード・レイテンシの短縮により，本実行時の発行キューへの圧迫は小さくなっており，小さな発行キューでも十分となる．

4.2 先行ディスパッチ命令の削除

4.2.1 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．

(1) の場合に対応するため，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，そのエントリの ID 番号（4.2.2 節で述べる仮想エントリ番号）を先行実行用の FIFO 発行キューへ放送する．発行キューでは，各 FIFO の先頭の命令において，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 から本実行用の発行キューへ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，すべ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．

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 5.4 節で示すとおり結果受け渡しの成功率は非常に高く，(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．

4.2.2 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．

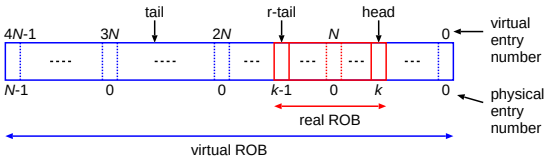


図 4 仮想 ROB（$M = 4$）

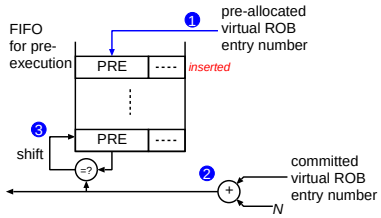


図 5 先行実行用発行キューへの挿入及び削除
（N は実 ROB サイズ）

従って，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先行実行用発行キュー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発行キューへ挿入される（①）．仮想エントリ番号を保持するために，発行キューの各エントリに **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 $((\text{仮想エントリ番号} + N) \bmod (N \times M))$ が先行実行用発行キューへ放送される（②，図 2③も参照）．この値は，解放された物理エントリが次に割り当てられる仮想エントリの番号を表している．先行実行用の発行キューでは，各 FIFO の先頭の命令について，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を比較する．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 のエントリが利用可能となったことを意味する．この場合，その命令を発行キューから削除する（③）．なお，4.1 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，すでに RFQ の先頭で待ち合わせている．

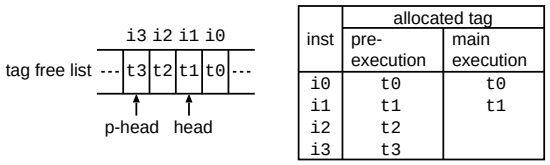
4.3 オペランド・タグの割り当て

先行実行命令は物理レジスタを割り当てられないため，オペランド・タグとして物理レジスタ番号を用いることはできない．代わりに任意のユニークな番号を用いることとする．先行ディスパッチのデータ依存関係を管理するため，**DT**（dependence table）と呼ぶ表を用意する．DT は論理レジスタ番号でインデクスされる表で，先行実行命令が生成するオペランドのタグを持つ．先行実行命令は，リネーム時に通常のマップ表と同様の方法で DT を参照し，更新する．

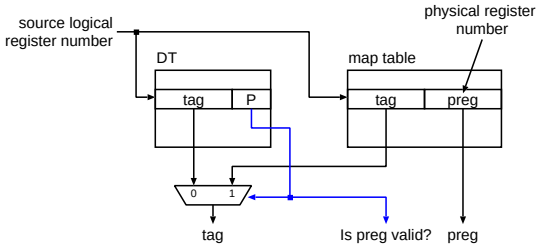
一方，本実行命令には物理レジスタが割り当てられるため，タグとして通常通り物理レジスタ番号を使用することが可能である．しかしそのように本実行と先行実行で別々のタグを用いると，生産者が本実行命令で消費者が先行実行命令の場合，関係が認識されず，先行実行にオペランドが受け渡されないという問題が生じる．これには 2 つの場合がある．1 つは，ROB が満杯になる前の命令の結果を満杯後の先行実行命令が使用できない場合である．もう 1 つは，4.2 節で述べたように，先行ディスパッチされた命令が，実 ROB を利用可能となったために実行前に発行キューより削除され本実行された場合，それに依存している命令がその結果を先行実行時に使用できない場合である．このような問題を解決するために，本実行命令に使用するタグも先行実行と同じく任意の番号とし，かつ 1 つの命令の動的インスタンスには先行実行の場合も本実行の場合も同一のタグを割り当てることとする．具体的には，以下のようにして本実行命令のタグ割り当てを行う．

図 6(a) に示すように，タグのフリー・リストには本実行における先頭を指すポインタ head と，先行実行における先頭を指すポインタ p-head の 2 つを設ける．タグの割り当てはプログラム順で行われるため，本実行におけるリネーム時には，先行実行におけるリネーム時と同一の順番でタグが割り当てられる．例えば，図において先行実行命令 `i0,...,i3` にそれぞれタグ `t0,...,t3` が割り当てられたとする．本実行命令は，先行実行命令と同順でリネームされるため，`i0,i1` にはタグ `t0,t1` がそれぞれ割り当てられる．

タグとして物理レジスタ番号を用いていないが，本実行命令に物理レジスタを割り当てなければならないため，マップ表は必要である．タグ割り当てと物理レジスタ割り当てを統合して行えるよう，**図 6(b)** に示す



(a) タグの割り当て



(b) ソース・タグの取得

図 6 オペランド・タグ

ように，マップ表に本実行命令用のタグも保持する．また，前述したように先行実行命令のソース・オペランドが先行する本実行命令の結果である場合があるから，先行実行命令も物理レジスタ番号を得なければならない状況がある．この場合に対応するため，先行実行命令のソース・オペランドにすでに物理レジスタが割り当てられていることを示すフラグ P を，DT のエントリに付加する．命令がリネームされる際に，デスティネーション・レジスタに対応する P フラグは，本実行の場合セットされ，先行実行の場合クリアされる．図 6(b) に示すように，命令のリネーム時に，ソース・レジスタ番号で DT 及びマップ表を参照し，P=0 ならば，生産者は先行実行命令であり，DT からのタグのみを得る．P=1 ならば，生産者は本実行命令であり，マップ表からタグ及び物理レジスタ番号を得る．

5. 評　　価

5.1 評 価 環 境

評価には，SimpleScalar Tool Set Version 3.0a[15] をベースに提案手法を実装したシミュレータを用いた．命令セットには Compaq Alpha ISA を用いた．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 を使用した．**表 1** に使用したベンチマーク・プログラム及びロードの L1 データ・キャッシュ・ミス率（MPKI: misses per kilo-instructions），L2 キャッシュ・ミス率，主記憶アクセス率を示す．バイナリは，Compaq C 及び Fortran コンパイラを用いて-fast -O4 のオプションでコンパイルした．入力には ref 入力を用い，SimPoint[7] によって選択した 100M 命令を実行した．

表 1 キャッシュの MPKI 及びメモリ・アクセス率

program	MPKI		memory	memory
	L1 data	L2	access rate	intensive?
ammp	22.2	0.7	0.2%	no
applu	33.5	17.0	4.4%	yes
apsi	6.7	1.1	0.3%	no
art	124.1	11.4	3.9%	yes
equake	74.3	27.1	5.9%	yes
facerec	5.3	1.6	0.5%	no
fma3d	19.1	12.1	2.8%	moderately
galgel	22.4	1.1	0.3%	no
lucas	41.2	21.9	7.6%	yes
mesa	1.9	0.6	0.2%	no
mgrid	22.8	6.7	1.8%	moderately
sixtrack	0.9	0.3	0.1%	no
swim	56.6	20.1	7.2%	yes
wupwise	6.0	2.8	1.0%	moderately

評価におけるベース・プロセッサの構成を**表 2** に示す．

VROB プロセッサにおいては，本実行用の発行キューのエントリ数及び先行実行用の FIFO の本数は，それぞれベース・プロセッサの発行キューのエントリ数の半分とした．依存ベースの発行キューにおいては，命令のウェイクアップ及び選択は FIFO の先頭の命令のみを対象とすればよいため，この構成はベース・プロセッサの発行キューの複雑さと同等である．すなわち，ウェイクアップ論理のタグ比較器の数は等しく，選択論理は共に 128 から 4 を選ぶ論理と等しい．また，各 FIFO はそれぞれ 16 エントリとした．なお，FIFO は単純なシフト・レジスタであるので，FIFO の長さがクロック速度に悪影響を与えることはない．

また，LSQ は十分に存在すると仮定し，エントリ数を仮想 ROB エントリ数と同一とした．この仮定は現実的ではないが，LSQ サイズに関する問題は，1 節で述べたとおり本論文では考慮しない．

VROB プロセッサにおける FB は 8 エントリとした．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[6] 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，読み出されることのないオペランドによる FB の容量の浪費を抑制できる．

以下の評価結果においては，ROB の拡大倍率 $M = 8$ とした．一般に，M を大きくするほど性能は向上するが，およそ 8 で飽和する．M に対する性能の感度評価は，紙面の都合上割愛する．

5.2 性　　能

図 7，**図 8** にベース及び VROB の性能（IPC），及びコミットされたロード命令の平均レイテンシをそれ

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
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	hybrid of 16-bit history 64K-entry PHT gshare + 2K-entry bimodal with 4K-entry choice predictor, 10-cycle misprediction penalty

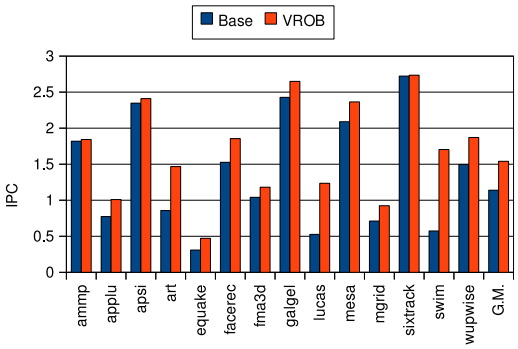


図 7 IPC

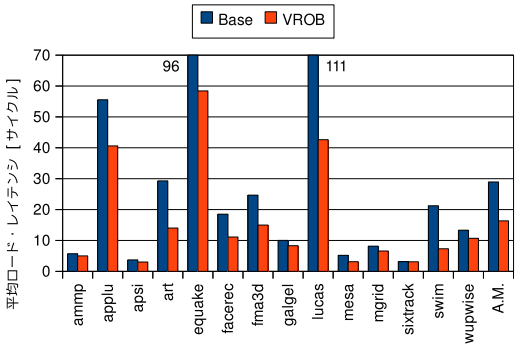


図 8 ロード命令のレイテンシ

ぞれ示す．図 7 に示すとおり，全てのプログラムにおいて VROB はベースより高い性能を示した．平均の性能向上率は 35%である．また，2 つのグラフの間には相関がみられ，プリフェッチの効果が確認できる．当然ながら，メモリ・インテンシブなプログラム（表 1 参照）で効果が大きく，特に lucas，swim などで効果が高い．

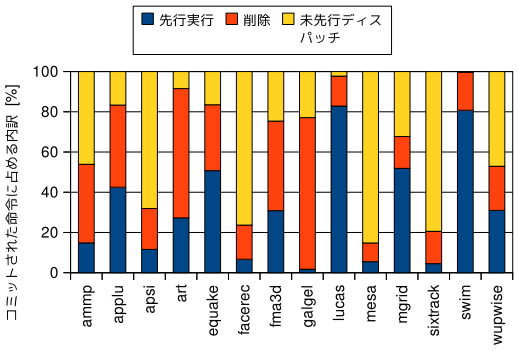


図 9 コミットされたロード命令の内訳

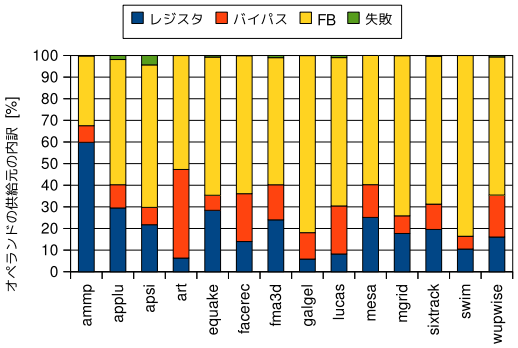


図 10 ソース・オペランドの供給元

5.3 先行実行率

図 9 に，VROB において，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命令，「削除」は先行ディスパッチされたが発行キューの中で削除された命令，「未先行ディスパッチ」は先行ディスパッチされなかった命令の割合をそれぞれ表している．

この図から，メモリ・インテンシブなプログラムにおいては多くのロード命令が先行実行されていることがわかる．これは，こういったプログラムでは L2 キャッシュ・ミスによって ROB が満杯となりやすいためである．一方で，メモリ・インテンシブでないプログラムにおいては，ROB が不足しにくいため先行実行も行われにくい．すなわち，先行実行はそれが有効である時に行われやすいと言える．

5.4 先行実行における結果の受け渡し

VROB の先行実行では物理レジスタが割り当てられないため，実行結果を必ずしも後続命令に受け渡すことができない．**図 10** は先行実行された命令のソース・オペランドの供給元を分類したものである．この測定

は完全なバイパス論理を仮定して行っている．すなわち，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としている．従って，実際には受け渡しに失敗する場合でも，先行実行は継続されるが,「失敗」に分類している．

この図からわかるとおり，受け渡しの失敗は極めて少ない．また，バイパス論理だけでは不十分であり，先行実行結果の十分な受け渡しには FB が必要であることがわかる．なお，受け渡しの失敗による性能低下は，測定の結果，平均 2.0%と小さかった．

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 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 35%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] H. Akkary, R. Rajwar, and S. T. Srinivasan, Checkpoint processing and recovery: Towards scalable large instruction window processors, In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pp. 423–434, December 2003.

[2] E. Borch, S. Manne, J. Emer, and E. Tune, Loose loops sink chips, In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pp. 299–310, February 2002.

[3] T.-F. Chen and J.-L. Baer, Reducing memory latency via non-blocking and prefetching caches, In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 51–61, October 1992.

[4] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, Dynamic speculative precomputation, In *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 306–317, December 2001.

[5] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramírez, M. Pericàs, and M. Valero, Kilo-instruction processors: Overcoming the memory wall, *IEEE Micro*, Vol. 25, No. 3, pp. 48–57, May–June 2005.

[6] J.-L. Cruz, A. González, M. Valero, and N. P. Topham, Multiple-banked register file architectures, In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 316–325, May 2000.

[7] G. Hamerly, E. Perelman, J. Lau, and B. Calder, SimPoint 3.0: Faster and more flexible program phase analysis, *The Journal of Instruction-Level Parallelism*, Vol. 7, pp. 1–28, September 2005.

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

[9] N. P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 364–373, May 1990.

[10] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, Runahead execution: An alternative to very large instruction windows for out-of-order processors, In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pp. 129–140, February 2003.

[11] S. Palacharla, N. P. Jouppi, and J. E. Smith, Complexity-effective superscalar processors, In *Proceedings of 24th Annual International Symposium on Computer Architecture*, pp. 206–218, June 1997.

[12] S. Petit, J. Sahuquillo, P. López, R. Ubal, and J. Duato, A complexity-effective out-of-order retirement microarchitecture, *IEEE Transactions on Computers*, Vol. 58, No. 12, pp. 1626–1639, December 2009.

[13] D. Pomarev, G. Kucuk, and K. Ghose, Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources, In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pp. 90–101, December 2001.

[14] A. Roth and G. S. Sohi, Speculative data-driven multithreading, In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pp. 37–48, January 2001.

[15] `http://www.simplescalar.com/`.