Vol.2012-ARC-198 No.9
2012/1/20

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

稲垣貴範[†1]　塩谷亮太[†1]　安藤秀樹[†1]

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，単一スレッド環境で命令の先行実行を実現する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．この手法を用いれば，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，VROB 方式ではプロセッサ内に多くの先行実行命令を保持する必要があるため，素朴な実装では重要な資源（リオーダ・バッファ，レジスタ・ファイル，発行キュー，ロード/ストア・キュー（LSQ: load/store queue））のサイズを大きくする必要があり，クロック・サイクル時間に悪影響を与える．過去の研究では，この問題について，LSQ 以外では解決されていたが，LSQ だけは解決されていなかった．本論文では，先行実行ロードに対する in-flight ストアへの依存を無視することにより，先行実行のために必要であった LSQ を削除し，クロック・サイクル時間への悪影響を除去することを提案する．メモリ・インテンシブなプログラムが多い SPECfp2000 ベンチマークを用いて評価を行った結果，十分に大きな LSQ を持ち，正しく依存を守る場合に比べ，性能低下をわずか 1%に抑えられることがわかった．

1. はじめに

プロセッサとメモリ間の速度差は非常に大きく，最終レベル・キャッシュをミスしたときの性能低下は著しい．この問題を解決する手法として，データのプリフェッチは有効である．しかし，自動プリフェッチャ（例えば，5)，8)）は，過去のメモリ・アクセスの履歴を利用し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という欠点がある．これに対して，複雑なアクセス・パターンに対応可能な手法として，命令の先行実行がある（例えば，6)，11)）．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（以後，本実行と呼ぶ）に先駆けて，別スレッドとして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働く．しかし，これまで提案されたほとんどの手法は，マルチスレッド環境を必要とするという欠点があった（マルチスレッド環境は，マルチプログラミングにおける複数スレッドを実行するために使用する方がシステムのスループットは向上し，効果的である）．

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[15]．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る．資源制約では，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（以下，これらを重要な資源と呼ぶ）が，in-flight 命令数を規定し，実行タイミングに特に大きな影響を与える．もしこれらの資源を命令に割り当てないなどして制約を緩和できれば，本実行の in-flight 命令数を越える命令の実行が可能となり，先行実行を実現できるといえる．

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[15] を提案した．VROB では，次のようにして，上記重要資源による制約を緩和している．

- ROB: ハードウェアのサイズは変えず，仮想的に拡大し，先行実行命令には仮想エントリを割り当てる．
- RF: 先行実行命令には物理レジスタを割り当てない．実行結果の後続命令への受け渡しは，バイパス論理とこれを助けるフォワーディング・バッファ[2] と呼ぶ小さなバッファで行う．
- IQ: 本実行用の IQ とは別に，先行実行用に複数の FIFO で構成される IQ を用意する．本実行用の IQ は縮小し，FIFO IQ は，これにより低減された複雑度に相当する複雑度に抑える．

以上のようにして，ROB，RF，IQ については，複雑度を増加させず資源制約を緩和させることができた．しかし，LSQ については，これまで未解決であった．

一般に，LSQ は複雑なハードウェアである．この機構は，ロード-ストア間のメモリ依存を考慮し，メモリ命令のスケジューリングを行う．ロードと先行するストア間の依存チェックのために，アドレスによる連想検索機能が必要となる．このため，LSQ は一般に CAM で構成される．また，依存が見つかった場合，ロードは依存するストアから値を得るストア・データ・フォワーディング（SDF: store-data forwarding）という機能も必要である．

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

VROB のこれまでの研究[15] では，LSQ について，本実行用の LSQ とは別に，先行実行ロード/ストアを保持する十分に大きな LSQ を持つと仮定していた．通常のプロセッサでは，LSQ に要求されるサイズは，ROB がサポートする in-flight 命令数程度であるが，VROB の先行実行用 LSQ に要求されるサイズも同様に，仮想 ROB がサポートする先行実行 in-flight 命令の数程度である．この数は，我々のこれまでの評価では，通常の in-flight 命令の 7 倍程度（900 命令程度）にもなるため，CAM で構成される通常の LSQ を単純に拡大する素朴な実装は，クロック・サイクル時間の観点から許容できない．本論文では，この先行実行用 LSQ の複雑度を低減させることを目的とする．

本論文では，先行実行ロードのスケジューリングにおいて，先行する本実行 in-flight ストアおよび先行実行 in-flight ストアへの依存チェックを省略することを提案する．これにより，メモリ依存を違反することがあるが，先行実行は，データ・プリフェッチのためだけに行われるのであり，アーキテクチャ状態を更新しないから，プログラムの意味の維持の点では問題ない．先行実行 in-flight ストアへの依存チェックが省略されることにより，先行実行用 LSQ から連想検索機能を除去できる．さらに，先行実行用 LSQ では SDF は行う必要がなくなるので，先行実行ストアを保持する必要もない．以上の結果，先行実行ロードは，アドレス計算終了後，その終了順でデータ・キャッシュをアクセスすればよい（本論文では，ロード/ストア命令は，フロントエンドでアドレス計算とメモリ・アクセスに分離される分離ロード/ストアを仮定している）．以上より，先行実行用 LSQ を削除できることとなる．

この方式の欠点は，先に述べたように，メモリ依存に違反するロードが実行されることである．これは，有用なプリフェッチを減少させ，キャッシュを汚染する．ただし，この影響は限定的と予想される．**図 1** に，過去の VROB の研究で仮定していた先行実行用 LSQ において，先行実行ロードと本実行用 LSQ または先行実行用 LSQ 内ストアの間に依存が検出された頻度（コミットされたロードに対する割合）を示す（プロセッサ構成は，5 節の表 2 および表 3 に示す）．同図からわかるように，この頻度は一部のベンチマークを除いて概して低く，誤った値のロードによる問題は小さいと予想される．

本論文の構成は以下の通りである．まず 2 節で従来の LSQ について述べる．次に 3 節で VROB 方式を，4 節で先行実行用 LSQ の単純化について説明する．そして，5 節で評価を行う．6 節で関連研究について述べ，7 節で本論文をまとめる．

2. LSQ 概要

LSQ は，ロード/ストア命令をスケジューリングするハードウェアである．分離ロード/ストア方式では，デコードされたロード/ストア命令は，アドレス計算とメモリ・アクセス操作に分離される．アドレス計算は発行キューに，メモリ・アクセス操作は LSQ に挿入される．以下，説明の簡単のため，断りがない限り，ロード/ストア命令から分離されたメモリ・アクセス操作を，同じくロード/ストア命令と呼ぶ．LSQ へのロード/ストア命令の挿入はプログラム順に行われるので，LSQ に保持されたロード/ストア命令はプログラム順で並ぶこととなる．

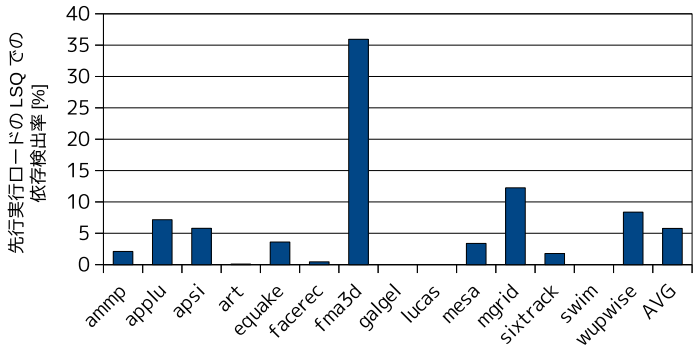

図 1 先行実行 in-flight ロード-ストア間依存検出の割合

ロードのアドレスが計算されたら，当該ロードが発行できるかどうかのチェックが行われる．具体的には，先行するストア（当該ロードより LSQ の前方のエントリに保持されているストア）に依存しているかどうかをチェックするため，アドレスにより連想検索が行われる．この連想検索のために，LSQ は CAM で構成される．検索の結果に応じて，以下のような処理がなされる．

- 一致するものがなければ，先行するどのストアにも依存していないことがわかるため，データ・キャッシュをアクセスし，値を得る
- 1 つでも一致すれば，その中で最も最近のストアに依存があることがわかるため，依存しているストアのデータを LSQ から読み出し，これをロード値として得る（ストア・データ・フォワーディング（SDF））
- もし，先行するストアの中にアドレスがまだ得られていないものがある場合，検索結果は不明となる．これは依存の有無が不明であることを意味する．この場合，基本的には，先行するすべてのストアのアドレスが得られるまで待ちあわせ，その後，依存

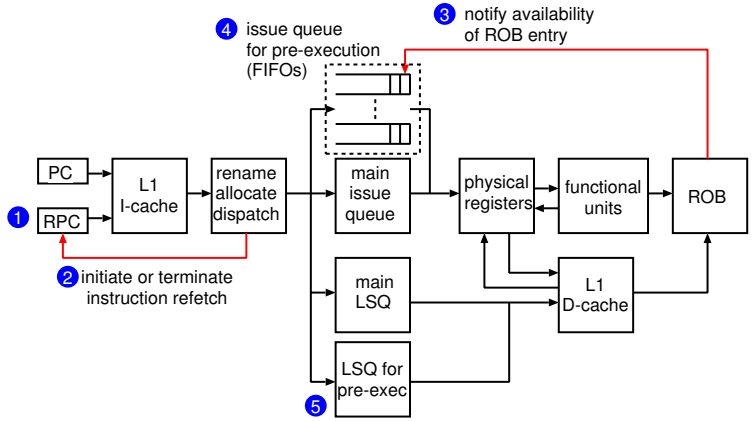

図 2 VROB プロセッサの構成

チェックの結果に応じた動作を行う．より積極的には，メモリ依存予測を行い，その予測にしたがって，投機的にロードを実行する場合もある．

ストアは，データとアドレスが計算され，ROB からリタイアした時点で，データをキャッシュに書き込む．

以上のように LSQ は複雑であり，クロック・サイクル時間に悪影響を与えず，単純にそのサイズを大きくすることは困難である．

3. 仮想リオーダ・バッファ方式

本節では，仮想リオーダ・バッファ（VROB: virtual reorder buffer）方式について説明する．**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 IQ 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 IQ が追加されている．この他，文献 15) の公開時点では，⑤先行実行用の LSQ が追加されている．これについては，4 節で述べる．

3.1 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，VROB 方式では 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 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，先行実行を行う．

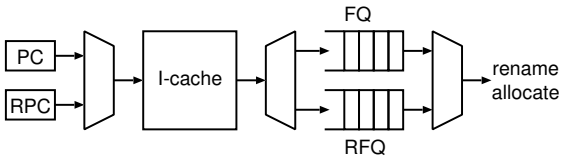

図 3 命令フェッチの構成

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③によって先行実行用 IQ から削除される（3.2 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本実行用 IQ 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．

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[10] 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．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 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．

3.2 先行ディスパッチ命令の削除

3.2.1 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先行実行用 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能となり，先行実行用 IQ 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先行実行用 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（3.2.2 節で述べる仮想エントリ番号）を先行実行用 IQ へ放送する．先行実行用 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，その命令を先行実行用 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 から本実行用 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，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．

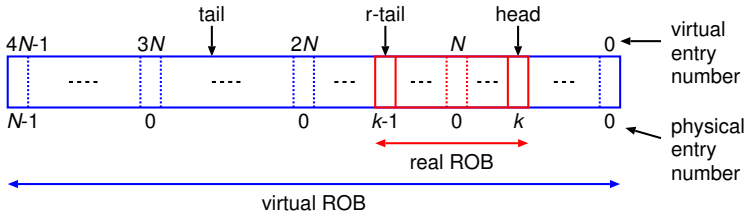

図 4　仮想 ROB（$M = 4$）

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失敗率は非常に低く，(2) の状態が生じることは稀である．したがって，これによる性能への影響は小さい（受け渡しの失敗による性能低下は約 2%[15]）．

3.2.2 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．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

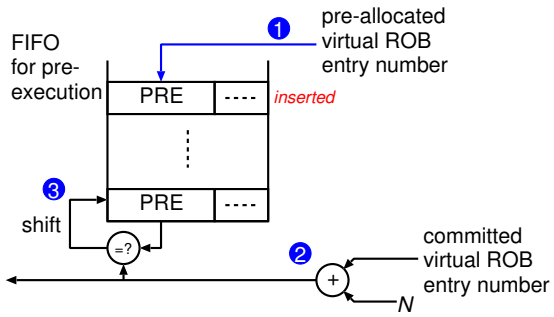

図 5 先行実行用 IQ への挿入及び削除（N は実 ROB サイズ）

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 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先行実行用 IQ へ挿入される（①）．仮想エントリ番号を保持するために，先行実行用 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 $((\text{仮想エントリ番号} + N) \bmod (N \times M))$ が先行実行用 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．先行実行用 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，その命令を先行実行用 IQ から削除する（③）．なお，3.1 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. VROB 方式における LSQ の単純化

4.1 VROB の過去の研究での LSQ

図 2 に示したように，以前我々が発表した文献 15) の時点では，本実行用 LSQ の他に，先行実行用の LSQ が用意されていた．その動作について説明する．

ロード/ストア命令には，先行ディスパッチ時に先行実行用 LSQ のエントリが割り当てられる．それらは，先行実行用 IQ 内の命令と同じく，実 ROB が利用可能となった時点で（図 2③と同様の信号を受ける）先行実行用 LSQ から削除され，改めて本実行用 LSQ が割り当てられ，本実行が行われる．

先行実行用 LSQ では，ロードは先行実行用 LSQ および本実行用 LSQ 内のストアに依存がないかをチェックし，発行される．依存があれば，いずれかの LSQ から SDF によって値を得，そうでなければキャッシュから得る．一方，ストアは，通常では ROB からリタイアするときにキャッシュに書き込みを行うが，先行実行ではリタイアという概念はなく，また，ストアをキャッシュに書き込むことはアーキテクチャ状態を破壊するので行われない．代わりに，先行実行用 LSQ に保持され続け，実 ROB が利用可能となったときに削除される．よって，先行実行ロード-ストア間にメモリ依存があった場合，先行実行用 LSQ 内での SDF により解決される．

我々の評価では，128 エントリの実 ROB に対し，仮想 ROB のサイズとしては，その 8 倍まで性能が向上することがわかっている．この場合，896（=128×7）エントリの先行実行用 LSQ が要求され，通常の構成で実装すると，クロック・サイクル時間に非常に大きな悪影響を与える．

4.2 先行実行用 LSQ の削除

本論文では，先行実行ロードは，本実行用 LSQ および先行実行用 LSQ を検索することなく，実行することを提案する．これにより，先行実行ロードは単にキャッシュから値を得るということになる．この実行は，先行する本実行 in-flight ストアおよび先行実行ストアに対して依存があった場合，不正な実行となるが，先行実行はプリフェッチのためだけに行われるのであるから，プログラムの意味の維持の点では問題ない．

先行実行用 LSQ の検索が不要になったことにより，これを CAM で構成する必要がなくなる．さらに，ストアはロードに対して SDF によりデータを供給する必要もなくなるから，先行実行ストア自体も保持する必要がない．なお，本実行用 LSQ の方は，本実行ロードのメモリ依存を満たすために，連想検索が必要であり，従来通り，CAM で構成しなければならない．

2 節で述べたように，ロードは LSQ にプログラム順で割り当てられていたが，先行実行ロードは，アドレス計算の終了順にキャッシュにアクセスするのであるから，プログラム順で割り当てる合理性はない．そこで，LSQ は削除し，単にキャッシュのポートが利用可能になるのを待ち合わせる FIFO （以下，**LRB**: load request buffer と呼ぶ）をおくこととす

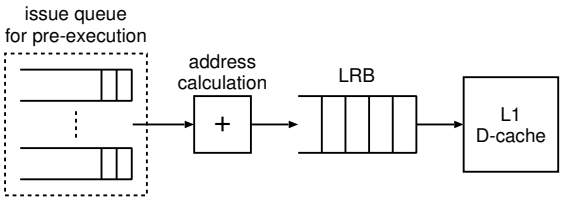

図 6 先行実行ロードの処理の流れ

る[⋆1]．**図 6** に，先行実行ロード（アドレス計算とメモリ・アクセスを含む）の処理の流れを示す．

この方式の欠点は，省略した依存チェックのために，ロードはキャッシュから不正なデータを得てしまうことがあることである．不正なデータを得た命令に依存する後続命令も，そのデータを得て実行されてしまうため，不正な先行実行は先行実行中の依存連鎖が途切れるまで続くことになる．このような不正ロードの実行は，有用なプリフェッチを減少させ，キャッシュを汚染する可能性がある．この結果，従来の研究における先行実行用 LSQ の場合に比べて，本実行時のキャッシュ・ヒット率を低下させる．しかし，5 節で示すように，このような不正ロードの発生頻度は平均的には低い．

5. 評　　価

5.1 評 価 環 境

評価には，SimpleScalar Tool Set Version 3.0a[13] 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 DEC Alpha ISA を用いた．ベンチマーク・プログラムとして，メモリ・インテンシブなプログラムが多い SPECfp2000 を使用した．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ものである．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．

以下のモデルについて評価を行った．

- **Base**: 先行実行を行わない通常のプロセッサ
- **VROB_ideal**: 十分に大きな先行実行用 LSQ を持つ従来研究における VROB 方式
- **VROB_simple**: 先行実行用 LSQ を削除し，LRB に置き換えた提案の VROB 方式

⋆1 キャッシュ・アクセスは，本実行用 LSQ からも行われるため，ポートの調停が必要で，アクセスには待ち合わせが生じる可能性がある．

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	21.9	0.7	0.2%	no
applu	24.0	17.0	4.4%	yes
apsi	4.6	1.1	0.3%	no
art	122.5	11.4	3.9%	yes
equake	71.8	27.0	5.9%	yes
facerec	4.0	1.7	0.5%	no
fma3d	15.6	12.1	2.8%	moderately
galgel	18.9	1.1	0.3%	no
lucas	26.3	21.9	7.6%	yes
mesa	1.4	0.6	0.2%	no
mgrid	19.6	6.7	1.8%	moderately
sixtrack	0.9	0.3	0.1%	no
swim	39.7	20.1	7.2%	yes
wupwise	5.6	2.8	1.0%	moderately

ベース・プロセッサの構成を**表 2** に示す．また，VROB に固有の構成を**表 3** に示す．VROB_simple モデルにおいては，LRB は 8 エントリとしている．4.2 節で述べたように，LRB はキャッシュ・ポート調停の待ち合わせのためだけのものなので，1 命令がエントリを占有する期間は非常に短く，少ないエントリで十分である．

表 1 に，ベンチマーク・プログラムごとのロードの 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．

5.2 不正な値を得た先行実行ロードの割合

これまで述べたように，メモリ依存のチェックを省略したため，先行実行において不正な値をロードする場合がある．**図 7** に，VROB_simple において，コミットされたロード数に対する不正な値を得た先行実行ロードの割合を示す．先に 1 節の図 1 に，VROB_ideal における先行実行ロードと in-flight ストアの依存の存在割合を示したが，その割合が高かったベンチマークでは，不正な値をロードする割合も高くなっている．これは，VROB_ideal で in-flight ロード-ストア間の依存があったロードは，VROB_simple では不正な値をロードしてしまうためである．加えて，不正な値でアドレス計算を行ったロードの結果もまた不正な値となるため，図 1 に示した割合より，VROB_simple において不正な値を得た割合の方が多い．不正な値をロードする割合は，多くのベンチマーク・プログラムでは，10%以下で少ないが，fma3d，mgrid，wupwise ではこれをこえている．これらでは，性能への影響が懸念される．

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, 4B/cycle bandwidth
Branch prediction	16-bit history gshare, 64K-entry PHT 10-cycle misprediction penalty

表 3　VROB_ideal，VROB_simple モデルの構成

VROB common	1024-entry virtual ROB, 64-entry main IQ, 64 16-entry FIFOs pre-execution IQ, 8-entry forwarding buffer
VROB_ideal	896-entry pre-execution LSQ
VROB_simlple	8-entry LRB

5.3 性　　能

図 8 に各モデルの性能（IPC）を示す．同図より，多くのベンチマークにおいて VROB_simple は VROB_ideal と同等の高い性能を達成できていることがわかる．VROB_ideal の Base に対する性能向上率は 41%であるのに対し，VROB_simple の性能向上率は 39%であり，その低下率は 1.4%とほとんど変わらない．このように，先行実行 in-flight 命令間のメモリ依存を無視しても高い性能向上を維持できたのは，5.2 節で述べたように，不正な値を得るロードの割合が平均的には少ないからである．

これに対し，fma3d，mgrid，wupwise では，性能向上率は有意に低下している．これらのプログラムでは，図 7 に示したように，不正な値を得るロードの割合が高いからである．しかし，依然として Base に対して高い性能向上率を示している．

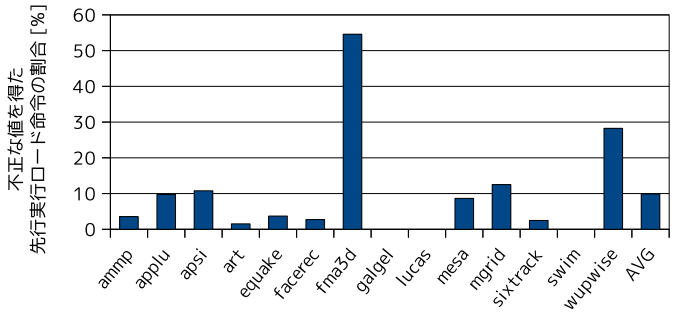

図 7　不正な値を得た先行実行ロードの割合

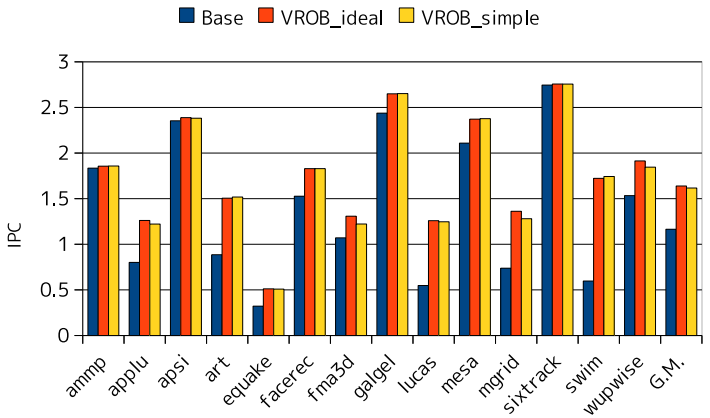

図 8　IPC

6. 関 連 研 究

6.1 先 行 実 行

先行実行によりプリフェッチを実現する手法はこれまでにも多く研究されている（例えば 6)，11)）．しかし，これらの手法では，先行実行のために別スレッドを生成する必要があ

り，SMT（Simultaneous Multithreading）や CMP（Chip Multiprocessor）といったマルチスレッド環境が必要であるという欠点がある．これに対し，VROB では単一スレッド実行環境において，先行実行を実現することが可能である．

VROB と同様，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[9]．この手法では最終レベル・キャッシュ・ミスが生じると，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 モードと呼ばれる特別なモードに移る．このモード中に，ミスしたロード命令に依存の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．しかしこの手法には，runahead モード中にはプロセッサ状態を更新する本来の命令実行を進められないという欠点がある．

6.2 LSQ の単純化

従来の LSQ では，依存チェックのための連想検索が必要となるため，スケーラビリティに乏しい．過去に提案された，LSQ の複雑度低減化の手法には，次のようなものがある．

Akkary らは階層化 LSQ を提案した[1]．この手法では，LSQ を小さく高速な L1 LSQ と，大きく低速な L2 LSQ の二層に階層化する．L1 LSQ にはより最近の命令を保持し，そこでの依存チェックの結果に応じてロード命令を投機的に実行する．一方，L2 LSQ での依存チェックは，後に時間をかけて投機を確認するために行う．メモリ依存は比較的近い命令間で発生することが多いため，投機の成功確率は高い．しかし，大きな L2 LSQ が必要であり，コストと消費電力が大きくなるという欠点がある．

Cain らは value-based replay と呼ばれる手法を提案した[3]．この手法では，ロード命令はメモリ依存予測器の予測に基づき投機的に実行され，投機の確認をロードのコミット時に，ロードを再実行することにより行う．投機時のロード値とコミット時のそれが等しければ，投機は成功である．この手法は，LSQ から依存チェックのための連想検索機能を除くことができ，スケーラビリティを向上させることができる．しかし，ロードを 2 度実行しなければならず，キャッシュ・バンド幅を圧迫するという欠点がある．

value-based replay によるメモリ依存違反の検出を利用した LSQ 単純化手法は，他に文献 4)，12)，14) などがある．

7. 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．過去の VROB の研究では，多くの先行実行を行うために必要な重要な資源（ROB，IQ，RF，LSQ）のうち，LSQ を除く資源については，仮想化による拡大，あるいはハードウェアの単純化などにより，複雑度の問題は解決されていた．

これに対し，本論文では残る LSQ について，その複雑度を下げる手法を提案した．本提案手法では，先行実行ロードの in-flight ストアへの依存を無視することにより，先行実行用 LSQ を削除する．メモリ・インテンシブなプログラムが多い SPECfp2000 ベンチマークを用いて評価を行った結果，理想的な LSQ をもつモデルと比較して，提案手法では 1%程度の性能低下で先行実行用 LSQ の複雑さを大幅に下げられることを示した．

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．

参 考 文 献

1) Akkary, H., Rajwar, R. and Srinivasan, S.T.: Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors, *Proceedings of the 36th 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) Cain, H.W. and Lipasti, M.H.: Memory Ordering: A Value-Based Approach, *Proceedings of the 31st International Symposium on Computer Architecture*, pp.90–101 (2004).
4) Castro, F., Pinuel, L., Chaver, D., Prieto, M., Huang, M. and Tirado, F.: DMDC: Delayed Memory Dependence Checking through Age-Based Filtering, *Proceedings of the 39th International Symposium on Microarchitecture*, pp.297–308 (2006).
5) 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 Languages and Operating Systems*, pp.51–61 (1992).
6) 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).
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) 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).
9) Mutlu, O., Stark, J., Wilkerson, C. and Patt, Y.N.: Runahead Execution: An Alter-

native 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).
10) 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).
11) 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).
12) Sha, T., Martin, M. M.K. and Roth, A.: NoSQ: Store-Load Communication without a Store Queue, *Proceedings of the 39th International Symposium on Microarchitecture*, pp.285–296 (2006).
13) `http://www.simplescalar.com/`.
14) Subramaniam, S. and Loh, G.H.: Fire-and-Forget: Load/Store Scheduling with No Store Queue at All, *Proceedings of the 39th International Symposium on Microarchitecture*, pp.273–284 (2006).
15) 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).