Vol.2018-ARC-232 No.3
2018/7/30

パイプライン構造の動的制御による命令フェッチ・スループットの向上

松尾 玲央馬[1,a)]　塩谷 亮太[2]　安藤 秀樹[1]

概要：近年のサーバー・アプリケーションや，ブラウザ上で実行される JavaScript では，従来のアプリケーションと比べて命令キャッシュ・ミスが非常に多く発生することが知られている．これに対し，命令キャッシュ向けのプリフェッチャが多く研究されており，非常に高いキャッシュ・ヒット率が達成されている．しかし，高い性能をもつ命令プリフェッチャほど，より大きな追加資源を必要とするという欠点がある．例えば，非常に高い命令キャッシュ・ヒット率を達成する Proactive Instruction Fetch は，L1 命令キャッシュそのものより大きなテーブルが必要になる．また，プリフェッチのミスに対するカバー率こそ高いものの，必要のない命令をプリフェッチし，電力を無駄に消費している．これに対し，本論文では命令プリフェッチのアプローチではなく，命令フェッチ・ステージのパイプライン構造の工夫でフェッチ・スループットを向上させる手法を提案する．本提案手法はプリフェッチャを用いない場合，従来よりも性能を低下させることなく命令キャッシュ・ミスによるストールを抑制し，フェッチ・スループットを向上させることができる．また，近年提案されている高性能なプリフェッチャと異なり，複雑な機構や大きなテーブルが必要ない．また，TPC-C ワークロードを用いて評価した結果，提案手法は従来と比べて 22.4%の性能向上が得られることを確認した

1. はじめに

近年のプロセッサでは，命令キャッシュ・ミスによる性能低下が問題となっている．これは，近年のアプリケーションの特性による．たとえばブラウザ上で実行される JavaScript やサーバーで用いられるようなアプリケーションは，命令ワーキングセットが非常に大きく，従来のアプリケーションと比べて命令キャッシュ・ミスが非常に多く発生することが知られている [1,2].

命令キャッシュ・ミスが発生すると，プロセッサはストールし，それにより性能が低下する．これは，近年の高性能なアウト・オブ・オーダー実行方式のプロセッサにおいても，命令キャッシュ・ミスにおけるストールは隠蔽することが難しいため，重要な問題である．

この問題に対し，命令キャッシュ・ミスを減らすアプローチとして，命令プリフェッチャがある．命令プリフェッチャは，ある命令が要求される前に，その要求を先読みして下位レベルのメモリへアクセスを行い，あらかじめその命令をキャッシュへと転送する．これを命令プリフェッチという．命令プリフェッチが成功した場合，命令キャッシュ・ミスを回避できるため，プロセッサのストールによる性能低下を抑えることができる．命令プリフェッチャは様々なものが提案されていて，たとえば，単純なものとしてネクストライン・プリフェッチャがある．他にも高度なプリフェッチャとして，分岐予測器を使用して命令プリフェッチを行う Fetch Directed Instruction Prefetching [3]，命令キャッシュ・ミスのストリームを記録して命令プリフェッチを行う Temporal Instruction Fetch Streaming [4]，およびそれを改良した非常に高いプリフェッチ効果をもつ Proactive Instruction Fetch (PIF) [5] などが提案されている．

しかし，これらのプリフェッチャは，有効性が高いものほど非常に大きなコストが必要になる．なぜなら，有効なプリフェッチを行うためには，複雑なキャッシュ・ミス・パターンを予測するアルゴリズムをハードウェアで実現する必要があるからである．特に PIF は，命令キャッシュ・ミスを 90%以上削減することが可能であるが，必要なストレージのサイズは一般的な L1 命令キャッシュよりも非常に大きい（L1 命令キャッシュが 32KB に対し，200KB 程度)．さらに，このようなプリフェッチャは，プリフェッチのミスに対するカバー率こそ高いものの，必要のない命令までプリフェッチすることもあり，その分電力を余分に消

[1] 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University
[2] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
[a)] matsuo@ando.nuee.nagoya-u.ac.jp

費してしまう．

このような命令キャッシュ・ミスそのものを減らすプリフェッチのアプローチに対して，本論文では命令フェッチ部のパイプライン構造を工夫することによって，命令フェッチのスループットを向上させる以下の手法を提案する．

- 従来の命令フェッチ･パイプラインでは, L1 命令キャッシュはヒットするものとして設計されており，フェッチ後は直ちに次のステージに命令が送られる．これに対し，提案手法では L1 命令キャッシュの**ミスを前提としたパイプライン**を提案する．このパイプラインでは L1 命令キャッシュにヒットした場合でも命令は必ず L2 キャッシュのレイテンシ分だけ待ってから次のステージへ送られる．このため，L1 命令キャッシュ・ミス時に L2 キャッシュ・アクセスを行う場合でもストールせずにフェッチを継続することができる．
- 単純にこの手法を適用した場合，L1 命令キャッシュ・ミス時にストールしなくなる反面，パイプライン長が大きく延びることにより分岐予測ミス・ペナルティが大幅に増加する．このため，従来のパイプラインとミスを前提としたパイプラインを動的に切り替えて利用する手法を提案する．

提案手法の利点は，近年提案されている高性能な命令プリフェッチャと異なり，複雑な機構や大きなテーブルが必要なく，低コストである．また，無駄なメモリ・アクセスを全く行わなず，低電力である．命令キャッシュ・ミスが多く発生すると知られている TPC-C ワークロードを用いて評価した結果，提案手法は従来と比べて 22.4%の性能向上が得られることを確認した．

本論文の残りの構成は次の通りである．まず，2 節で関連研究を示す．そして，3 節で本論文が提案するミスを前提としたパイプラインについて説明する．4 節ではミスを前提としたパイプラインと従来のパイプラインを切り替えて使用するアーキテクチャについて説明し，続く 5 節でその切り替えアルゴリズムについて述べる．そして 6 節で性能評価を行い，7 節でまとめる．

2. 関連研究

命令キャッシュ・ミスが性能に与える影響は大きく，そのため命令プリフェッチャの提案は広く行われてきた．分岐予測器を用いて命令プリフェッチを行うものとして，Fetch Directed Instruction Prefetching [3] がある．また，特定のワークロードやプロセッサに特化して命令プリフェッチを行う手法として，OLTP ワークロード用に特化した SLICC [6] や，コンパイラを用いてプリフェッチに有用な情報を記録する pTask [7] がある．

中でも，キャッシュ・ミス・ストリームの相関を利用する命令プリフェッチャは特に高い性能をもつことが知られている．このようなプリフェッチャとして，Ferdman らは Temporal Instruction Fetch Streaming [4] およびそれを改良した Proactive Instruction Fetch [5] を提案した．

しかし，ストリームベースの命令プリフェッチャは，高いプリフェッチ精度を達成するために非常に多くのメタデータを必要とする．そこで，少ないメタデータで高い精度を達成する命令プリフェッチャの研究も広く行われている．Kolli らは，RAS の情報を利用することで，PIF よりも少ないメタデータでそれに近いプリフェッチ精度を達成する RDIP [8] を提案した．Kaynak らは，マルチコア・プロセッサにおいて，共有 LLC にストリーム履歴を格納することによって，コア間でメタデータの共有を行う SHIFT [9] と，命令キャッシュと BTB の各プリフェッチャを統合する Confluence [10] を提案した．また，Kumar らはメタデータを最小限に抑える命令プリフェッチャとして，分岐予測器ベースの命令プリフェッチャを用いる Boomerang [11] と BTB の機構を工夫した Shotgun [12] を提案した．

本提案手法と同様，ミスを仮定したパイプラインを利用した手法として，塩谷らが提案した Non-Latency-Oriented Register Cache System [13] と Semi-Global Renamed Trace Cache [14] がある．これらの手法は，ともに従来の性能を保ちつつ，回路面積や消費エネルギーを削減することができるため，プロセッサのエネルギー効率を向上させる．

3. ミスを前提としたパイプライン

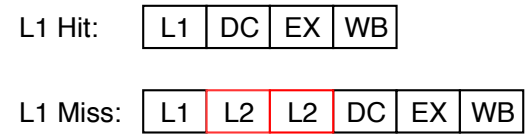
本論文では，既存と異なるパイプライン構造をもつミスを前提としたパイプラインを提案する．このミスを前提としたパイプラインは命令キャッシュ・ミスによるストールを削減するという特性があるため，命令キャッシュ・ミスが多く発生する場合は従来のパイプラインよりも性能が高くなる．以下では，ミスを前提としたパイプラインの構成，動作，構造，性能について順に説明する．その際，説明を簡単にするために以下の仮定をおく．

- L1 命令キャッシュ，L2 キャッシュのアクセス・レイテンシをそれぞれ 1, 2 サイクルとする
- L2 キャッシュ・アクセスは完全にパイプライン化されている
- L2 キャッシュは必ずヒットする

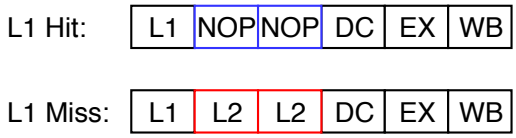
3.1 パイプライン構成

図 1 に，ヒットおよびミスを前提としたパイプラインの構成をそれぞれ示す．同図では L1 が L1 命令キャッシュ・アクセス，L2 が L2 キャッシュ・アクセス，DC がデコード，EX が実行，WB がライトバックの各ステージを表す．また，NOP はパイプライン・ラッチだけがあり，何もせずに命令を後ろに渡すだけのステージを表す．

図 1(a) のヒットを前提としたパイプラインは，通常のプ

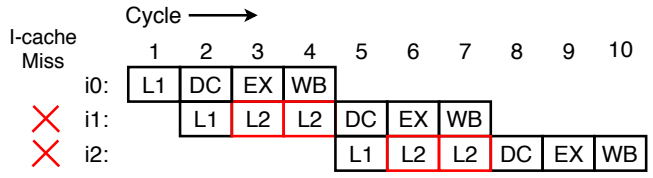


(a) ヒットを前提としたパイプライン

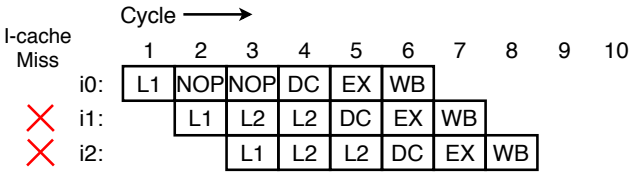


(b) ミスを前提としたパイプライン

図 1 パイプラインの構成



(a) ヒットを前提としたパイプライン



(b) ミスを前提としたパイプライン

図 2 命令キャッシュ・ミス時の動作

ロセッサで用いられているパイプラインである．このパイプラインでは L1 ステージの直後に DC ステージが置かれており，キャッシュ・ヒット時はフェッチして得られた命令を即座に次のデコード・ステージに送る．また，キャッシュ・ミス時は L2 キャッシュにアクセスするため，デコード・ステージには 2 サイクルのバブルが生じる．

これに対し，図 1(b) のミスを前提としたパイプラインでは，L1 命令キャッシュ・ヒット時には L1 ステージの直後に 2 つの NOP ステージが設けられている．このパイプラインではキャッシュ・ヒット時に命令を次の DC ステージに直ちに送るのではなく，2 つの NOP ステージを介した後に DC ステージに送る．また，キャッシュ・ミス時はヒットを前提としたパイプラインと同様に 2 サイクルかけて L2 キャッシュから命令を得る．

3.2 動作例

図 2 は，命令キャッシュ・ミスが発生する命令を含む命令列をヒットおよびミスを前提としたパイプラインで実行した際のパイプライン・チャートである．命令列 i0〜i2 のうち，i1 と i2 で命令キャッシュ・ミスが発生すると仮定している．同図 (a) のヒットを前提としたパイプラインでは命令キャッシュ・ミスが発生するたびに次の命令のフェッチがストールしている．そのたびに次の命令のフェッチが遅れるため，全体の実行時間は 10 サイクルとなっている．

一方，同図 (b) のミスを前提としたパイプラインでは命令キャッシュ・ミスが発生してもストールせずに次の命令のフェッチを行っており，全体の実行時間は 8 サイクルとなっている．このように，ミスを前提としたパイプラインは命令キャッシュ・ミスが発生しても命令のフェッチがストールしないため，性能は低下しない．そのため，命令キャッシュ・ミスが多く発生する場合はヒットを前提としたパイプラインよりも性能が高くなる．

3.3 構造

図 3 は，ミスを前提としたパイプラインの構造を表すブロック図である．同図では PC がプログラム・カウンタ，L1 が L1 命令キャッシュ，L2 が L2 キャッシュ，DC がデコード・ステージ，D がパイプライン・ラッチ，MUX がマルチプレクサをそれぞれ表している．

ミスを前提としたパイプラインでは，L1 命令キャッシュに加えて，L2 キャッシュからも命令を受け取る必要があるため，2 つのパスが存在する．その際，どちらのパスを通っても，命令をデコード・ステージに送るタイミングが同じになるように，L1 命令キャッシュから命令を受け取るパスに L2 キャッシュのヒット・レイテンシ分のパイプライン・ラッチを挿入する．そして，命令の流れるパスが 2 通りあるため，最終的にどちらのパスの命令をデコード・ステージに渡すかを選択するマルチプレクサが必要になる．

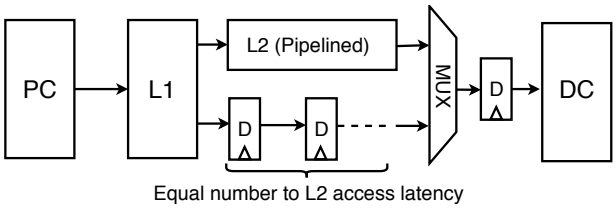


図 3 ミスを前提としたパイプラインのブロック図

3.4 性能

ヒットおよびミスを前提としたパイプラインの間で性能差が生じる状況として，命令キャッシュ・ミスが発生した場合，あるいは分岐予測ミスが発生した場合の 2 通りがある．命令キャッシュ・ミスが発生した場合は，3.2 節で述べたようにミスを前提としたパイプラインの方が性能が高くなる．一方，分岐予測ミスが発生した場合はヒットを前

提としたパイプラインの方が性能が高くなる．以下では，これらのミスにより 2 つのパイプラインの間で性能差が生じる理由について述べる．

3.4.1 命令キャッシュ・ミスによって生じる性能差

命令キャッシュ・ミスの発生によって 2 つのパイプラインの間で性能差が生じる理由を図 2 を用いて説明する．命令キャッシュ・ミスが発生すると，L2 キャッシュ・アクセスが必要であるため，ヒットおよびミスを前提としたパイプラインのどちらもパイプラインの長さが 6 段になり，パイプラインの構成が同じになる．そのため，命令キャッシュ・ミスが発生した時点では，2 つのパイプラインの間で性能差は生じない．これは，図 2(a)(b) において i1 が実行されるタイミングを見比べると確認できる．

しかし，キャッシュ・ミスを生じた命令の後の命令の処理は，2 つのパイプラインで異なる．ヒットを前提としたパイプラインでは，キャッシュ・ミスを生じた命令の次の命令のフェッチは，キャッシュ・ミスの処理が終わった後に行われる．これは，命令キャッシュのヒットおよびミスによってパイプラインの長さが異なるため，キャッシュ・ミスの処理が終わる前に次の命令のフェッチを行うと，命令の実行順序が変わることがあるからである．一方，ミスを前提としたパイプラインでは，キャッシュ・ミスを生じた命令の次の命令のフェッチを，キャッシュ・ミスの処理が終わるのを待たずに即座に行うことができる．なぜなら，パイプラインの長さが命令キャッシュのヒットおよびミスによらず一定のため，キャッシュ・ミスの処理を待たずにフェッチを継続しても命令の実行順序が変わることがないからである．

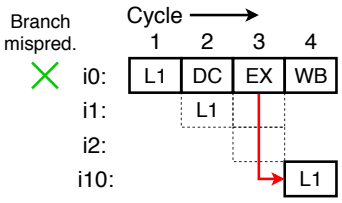
このように命令キャッシュ・ミスが発生すると，ヒットを前提としたパイプラインでは以降の命令のフェッチが 2 サイクル遅れ，命令の実行完了も 2 サイクル遅れる．一方，ミスを前提としたパイプラインでは性能低下はない．そのため，命令キャッシュ・ミスが発生した場合は，ミスを前提としたパイプラインの方が性能が高くなる．ただし，ミスを前提としたパイプラインは NOP ステージの追加によって命令の処理が元々 2 サイクル遅れているため，命令キャッシュ・ミスが 1 回のみ発生した場合は，ヒットおよびミスを前提としたパイプラインの性能は同じである．

3.4.2 分岐予測ミスによって生じる性能差

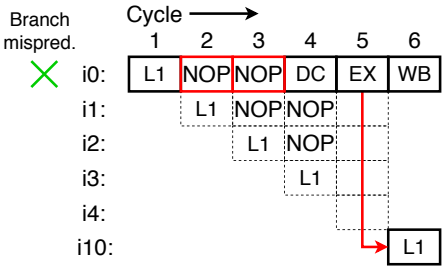
分岐予測ミスの発生によって 2 つのパイプラインの間で性能差が生じる理由を図 4 を用いて説明する．同図は，ヒットおよびミスを前提としたパイプラインで分岐予測ミスが発生する命令列を実行した際のパイプライン・チャートであり，命令列 i0〜i4, i10 のうち，i0 で分岐予測ミスが発生すると仮定している．

分岐予測ミスが発生すると，パイプライン中の誤ったパスの命令がフラッシュされ，PC に次の正しい命令のアドレスが格納される．この際，パイプラインが長いほど取り消される命令が増えるため，より分岐予測ミスによる性能低下が大きくなる．ミスを前提としたパイプラインは，命令キャッシュ・ヒット時のパイプラインが従来よりも長いので，分岐予測ミスが発生した際に大きな性能低下を被る．

同図 (a) に示すように，ヒットを前提としたパイプラインでは分岐予測ミスの解決が 3 サイクル目に行われている．しかし，同図 (b) に示すように，ミスを前提としたパイプラインでは NOP ステージを 2 サイクル設ける分，分岐予測ミスの解決が遅れ，5 サイクル目に行われている．これにともない，次の正しい命令のフェッチも 2 サイクル遅れている．このように，分岐予測ミスが発生した場合は，ヒットを前提としたパイプラインの方が性能が高くなる．



(a) ヒットを前提としたパイプライン



(b) ミスを前提としたパイプライン

図 4 分岐予測ミス時の動作

4. 2 つのパイプラインをもつアーキテクチャ

3.4 節で述べたように，ミスを前提としたパイプラインと従来のパイプラインでは，命令キャッシュ・ミス，分岐予測ミスに対する性能への影響が異なる．そこで，本論文ではミスを前提としたパイプラインと従来のパイプラインを同時に持ち，それらを動的に切り替えて使用するアーキテクチャを提案する．このアーキテクチャは，使用するパイプラインを適切に切り替えることで，将来発生するミスによる性能低下を抑制することができる．本節では，このアーキテクチャの構造，動作について説明する．なお，パイプラインの切り替えアルゴリズムについては 5 節で述べる．

4.1 構造

図 5 に，本節で提案するアーキテクチャの構造を表すブ

ロック図を示す．同図に示すように，本提案手法の構造は単純で，基本的には 2 つのパイプラインを並列に配置するだけである．具体的には，L1 命令キャッシュからの出力をヒットおよびミスを前提としたパイプラインの入力へとつなぎ，パイプラインの最終段にどちらのパイプラインの命令をデコード・ステージに送るかを選択するマルチプレクサを配置する．

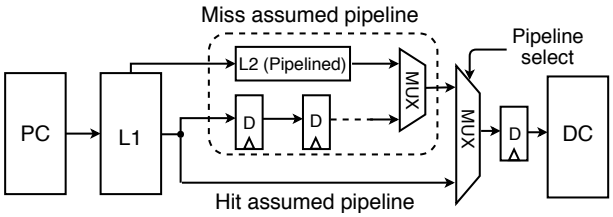


図 5　提案するアーキテクチャのブロック図

4.1.1　FIFO バッファによる実装

図 5 に示した NOP ステージの実装は，多くの命令をパイプライン・ラッチ上で移動させる必要があるため，ラッチの数が多く消費電力が大きくなる．この NOP ステージは，命令の移動を伴わない FIFO バッファによって実現できる．図 6 に FIFO バッファを用いて実装した場合のブロック図を示す．この実装においては，FIFO バッファに命令を挿入した後に，L2 キャッシュのレイテンシだけ待ってから命令を取り出す．

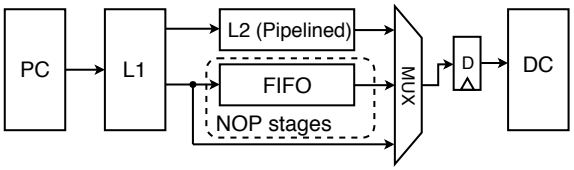


図 6　FIFO を用いた設計

4.2　動作

本節で提案するアーキテクチャは，ヒットを前提としたパイプラインのみを使用するモードとミスを前提としたパイプラインのみを使用するモードの 2 つのモードがあり，この 2 つのモードの間を何らかのアルゴリズムにしたがって遷移させ，命令のフェッチを行う．

その際，命令の正しい実行順序を保証するために，パイプラインを切り替える際に必要な動作がある．以下では，これについて述べる．なお，以降の説明ではヒットを前提としたパイプラインを使用するモードおよびミスを前提としたパイプラインを使用するモードをそれぞれヒット前提およびミス前提と呼ぶことにする．

4.2.1　切り替え時に必要な動作

提案手法では，命令フェッチ部のパイプラインの動的な切り替えを行うが，このときデコード・ステージに送られる命令の順序が保たれていなければならない．パイプラインの切り替えは，ヒット前提からミス前提へ切り替える場合と，ミス前提からヒット前提へ切り替える場合の 2 通りがあるが，このうちデコード・ステージに送られる命令の順序が変わる可能性があるのは，パイプラインが短くなるミス前提からヒット前提へ切り替える場合のみである．そこで，ミス前提からヒット前提に切り替える際には，次のような動作とする．

まず，ヒット前提に切り替えると同時に命令のフェッチを停止させる．そして，ミスを前提としたパイプラインから全ての命令が排出されるのを待つ．命令の排出が完了したら，フェッチを再開させる．

5.　パイプラインの切り替えアルゴリズム

本節では，4 節で提案したアーキテクチャで用いるパイプラインの切り替えアルゴリズムについて述べる．この切り替えアルゴリズムとしては，将来，命令キャッシュ・ミスと分岐予測ミスのどちらが発生するかを予測し，事前にパイプラインを切り替える方法が直感的には考えられる．しかし，そのような予測を導入せずに，常に最適な切り替えを行うことができるアルゴリズムを提案する．以下では，この切り替えアルゴリズムについて説明する．

5.1　提案する切り替えアルゴリズム

図 7 に，提案する切り替えアルゴリズムの状態遷移図を示す．

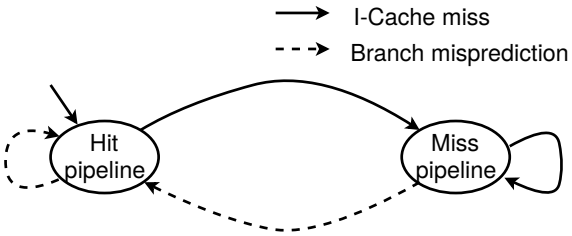


図 7　提案手法の切り替えアルゴリズムの状態遷移図

この切り替えアルゴリズムが意味するのは次の通りである．まずヒット前提で命令のフェッチを行うが，命令キャッシュ・ミスが発生したら，ミス前提へ切り替える．これにより，以降に発生する命令キャッシュ・ミスはミス前提の特性により性能を低下させずに実行することができる．つまり，この間で命令キャッシュ・ミスが発生した分だけ，従来よりも性能が向上することになる．そして，分岐予測ミスが発生したらヒット前提へ戻る．

この切り替えアルゴリズムは，提案するアーキテクチャにおける性能低下を最小に抑える最適な切り替えを行う．

以下では，その理由を説明する．

5.2 最適な切り替えを行うことができる理由

提案するアーキテクチャの切り替えアルゴリズムによって性能差が生じる場面は，ヒット前提で実行中に命令キャッシュ・ミスが発生する場合と，ミス前提で実行中に分岐予測ミスが発生する場合の 2 通りである．提案する切り替えアルゴリズムは，これらの場面で発生する性能の低下を最小に抑えることができる．

5.2.1 ヒット前提時に発生する命令キャッシュ・ミス

まず，ヒット前提で実行中に命令キャッシュ・ミスが発生する場合について説明する．この場合は，命令キャッシュ・ミスが発生した直後にミス前提へ切り替えることで性能低下を回避することができる．なぜなら，3.4.1 節で述べたように，命令キャッシュ・ミスが発生するとすぐに 2 つのパイプラインの間で性能差が生じるわけではなく，命令キャッシュ・ミスの後の動作によって性能差が生じるからである．

図 8 は，命令キャッシュ・ミスが発生する命令列を実行する際に (a) 事前に命令キャッシュ・ミスを予測してミス前提に切り替えた場合と (b) 命令キャッシュ・ミスが発生した直後にミス前提に切り替えた場合におけるパイプライン・チャートである．同図では命令列 i0〜i3 のうち，i2 で命令キャッシュ・ミスが発生すると仮定している．また，同図右側の矢印は，その命令を実行している際のモードを表している．

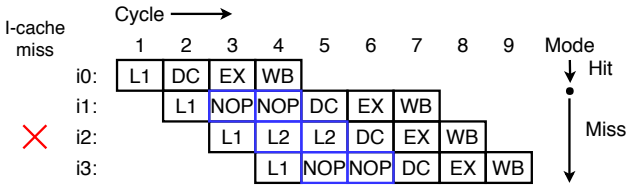
同図 (a) では，i2 で命令キャッシュ・ミスが発生すると予測して，i1 をフェッチした時点でミス前提へ切り替えている．一方，同図 (b) では命令キャッシュ・ミスが発生した直後にミス前提へ切り替えている．ここで，どちらの場合においても全体の実行時間は 9 サイクルであり，性能が同じであることが分かる．

5.2.2 ミス前提時に発生する分岐予測ミス

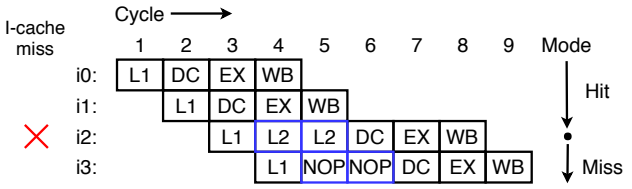
次に，ミス前提で実行中に分岐予測ミスが発生する場合について説明する．この場合は，どのように切り替えを行ったとしても，性能低下を回避することはできない．なぜなら，ミス前提からヒット前提への切り替えには，4.2.1 節で述べたように，ミスを前提としたパイプラインから全ての命令が排出されるのを待つ必要があり，分岐命令の実行タイミングは切り替えを行っても行わなくても結局変わらないからである．

図 9 は，分岐予測ミスが発生する命令列を実行する際に (a) 事前に分岐予測ミスを予測してヒット前提に切り替えた場合と (b) 切り替えを行わない場合におけるパイプライン・チャートである．同図では命令列 i0〜i5, i10 のうち，i1 で分岐予測ミスが発生すると仮定している．

同図 (a) では，i1 で分岐予測ミスが発生すると予測して，i1 の実行前にヒット前提へ切り替えている．一方，同図



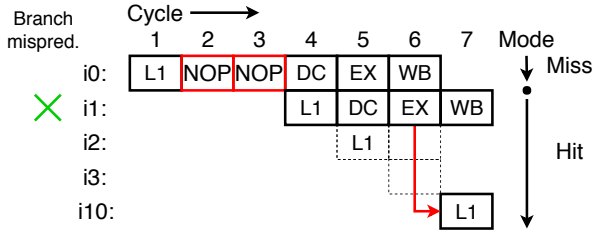
(a) 事前にミス前提に切り替える場合



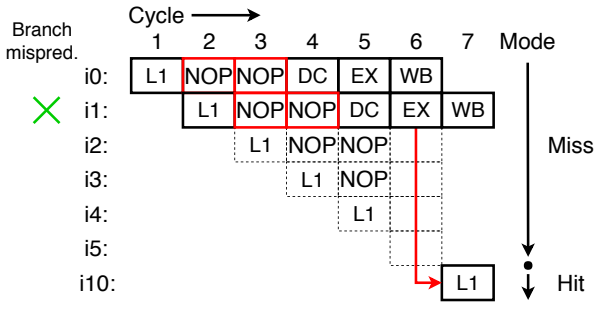
(b) 命令キャッシュ・ミスの直後にミス前提に切り替える場合

図 8 ヒット前提で実行中に命令キャッシュ・ミスが発生する場合

(b) ではヒット前提への切り替えは行わず，ミス前提のままで分岐予測ミスが発生している．同図 (a)(b) を見比べると，どちらも分岐予測ミス後の正しい命令のフェッチは 7 サイクル目に行われているため，性能は変わらないことが分かる．



(a) 事前にヒット前提に切り替える場合



(b) 切り替えを行わない場合

図 9 ミス前提で実行中に分岐予測ミスが発生する場合

i10 でフェッチ方向を変えた後には，以後の分岐予測ミス，命令キャッシュ・ミスで発生する性能低下を最小に抑えるために，ヒット前提へ切り替える．ヒット前提では分

岐予測ミスが発生した場合，3.4.2 節で述べた通り生じる性能低下は最小である．また，命令キャッシュ・ミスが発生した場合，5.2.1 節で述べたように命令キャッシュ・ミスの直後にヒット前提からミス前提へ切り替えることで，性能低下を回避することができる．一方，切り替えずに，ミス前提のままにすると，命令キャッシュ・ミスが発生した場合は 3.4.1 で述べた通り性能低下は発生しないが，分岐予測ミスが発生すると，本節で述べたように必ず性能が低下してしまう．

以上に述べたように，この切り替えアルゴリズムは命令キャッシュ・ミスおよび分岐予測ミスによって発生する性能低下を最小に抑えるため，提案手法における最適な切り替えアルゴリズムといえる．

6. 評価

6.1 評価環境

評価には，提案手法を実装した gem5 シミュレータ [15] を用いた．性能の評価に用いたプロセッサの基本構成を表 1 に示す．

評価では，SPEC CPU 2006 ベンチマーク・プログラムと，TPC-C を用いた．SPEC CPU 2006 プログラムおよび TPC-C ベンチマーク・プログラムは gcc ver.4.9.3 でコンパイルし，コンパイル・オプションには -O3 を用いた．SPEC CPU 2006 プログラムの入力セットには ref データ・セットを使用し，プログラムの先頭 16G 命令をスキップした後の 100M 命令について測定した．TPC-C の評価では，gem5 のフルシステム・シミュレーションを用いて，OS の動作も含めてシミュレーションを行った．測定区間は，ベンチマーク・プログラムの先頭 10G 命令をスキップした後の 100M 命令である．データベース・サーバーは，MySQL Server 5.5.59 を用いた．

6.2 評価モデル

次に，評価に用いた 4 種類のプロセッサ・モデルを示す．

- **Base**: 提案手法も命令プリフェッチャも適用していないモデル
- **NL-x**: プリフェッチ深さ x のネクストライン・プリフェッチャを適用したモデル
- **Proposed**: 提案手法を適用したモデル
- **Proposed&NL-x**: 提案手法とプリフェッチ深さ x のネクストライン・プリフェッチャを適用したモデル
- **Perfect**: 命令キャッシュが必ずヒットするモデル

なお，ネクストライン・プリフェッチャ（以下 NL とも表記する）のプリフェッチ深さは，1，2，4 の 3 通りについてシミュレーションした．ただし，グラフを簡単にするため，ネクストライン・プリフェッチャを適用したモデルは，その評価において最も高い性能を記録したプリフェッチ深さの場合（深さ 1）のみをグラフに記載した．

表 1　プロセッサの基本構成

Pipeline width	8 instructions wide for each of fetch, decode, issue, and commit
ROB	224 entries
Issue queue	96 entries
LQ	72 entries
SQ	56 entries
Branch prediction	L-TAGE [16] 12-cycle misprediction penalty 4K-set 4-way BTB
Function unit	6 iALU, 2 iMULT/DIV, 2 Ld/St, 4 fpALU, 2 fpMULT/DIV/SQRT
L1 D-cache	32KB, 8-way, 64B line 2-cycle hit latency
L1 I-cache	32KB, 8-way, 64B line 2-cycle hit latency
L2 cache	256KB, 16-way, 64B line 12-cycle hit latency
Main memory	300-cycle latency
ISA	ARMv8

6.3 評価結果

6.3.1 ネクストライン・プリフェッチャとの性能比較

まず，提案手法およびネクストライン・プリフェッチャを適用した場合の性能向上について評価を行った．図 10 は，TPC-C と SPEC CPU 2006 ベンチマークにおける各モデルの Base に対する性能向上率を表したグラフである．なお，SPEC のベンチマークは命令キャッシュ・ミスが最も多く発生した 3 種類のベンチマークのみをグラフに記した．また，図 11 は各ベンチマークを Base で実行した際の命令キャッシュの MPKI である．

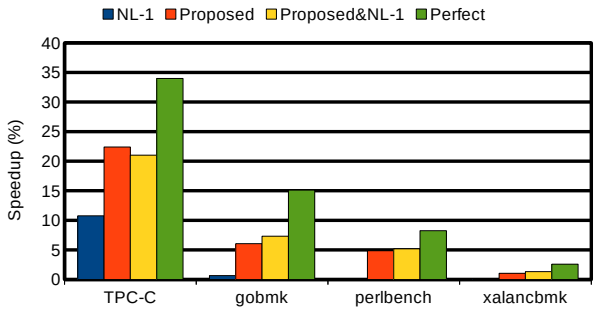


図 10　各モデルの性能向上率

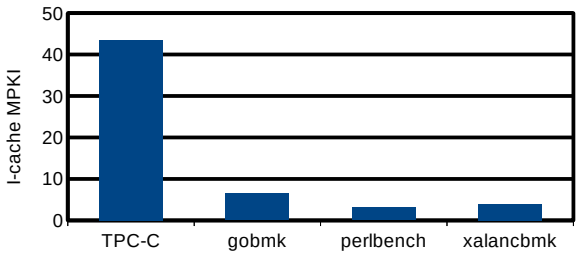
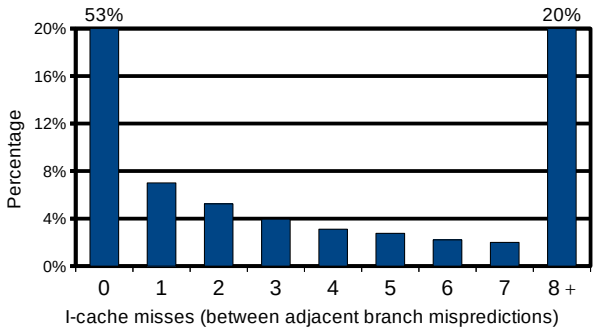
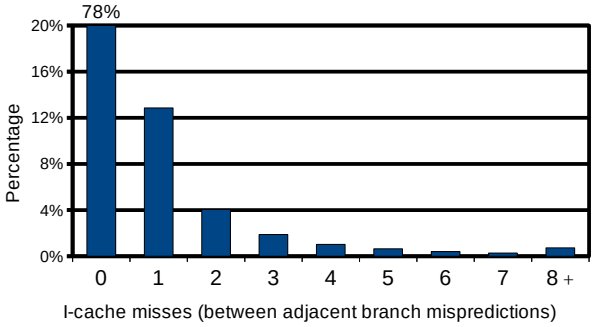


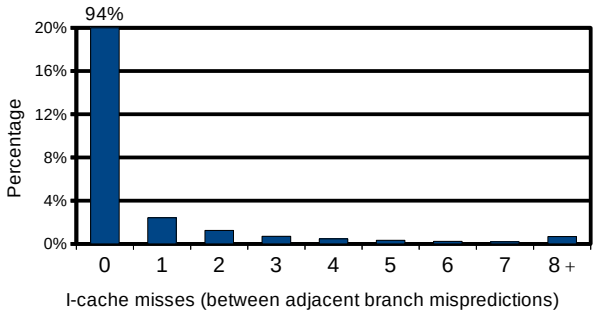
図 11　各ワークロードの命令キャッシュの MPKI



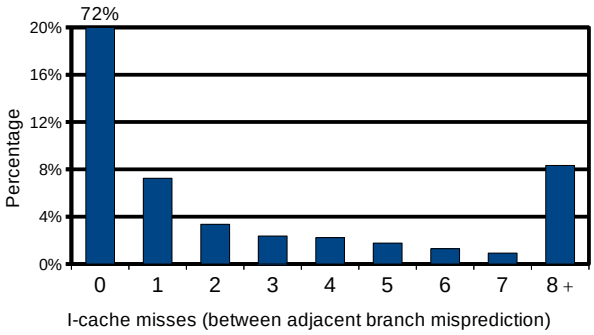
(a) TPC-C



(b) gobmk



(c) perlbench



(d) xalancbmk

図 12 命令キャッシュ・ミスの連続発生数の統計

図 10 から，Proposed は NL よりも大きな性能向上が得られていることが分かる．特に，命令キャッシュ・ミスが非常に多く発生する TPC-C においては，NL-1 の性能向上率は 10.7%であるのに対し，Proposed は 22.4%と，非常に大きな性能向上が得られることを確認した．しかし，性能の上限を表すモデルである Perfect の性能向上率は 34.0%であり，Proposed の性能は理想的な命令キャッシュを持つ場合には及ばないことが分かる．

また，Proposed&NL の性能向上率は Proposed とほとんど同じで，ワークロードによってわずかに上下するという傾向が見られた．これはプリフェッチによるキャッシュ汚染のためと考えられる．

6.3.2 連続して発生する命令キャッシュ・ミス数の統計

本節では，各ワークロードにおいて，2 つの隣り合う分岐予測ミスの間に命令キャッシュ・ミスが何回連続して発生したかを調べた．図 12 にその統計を示す．同図の横軸は，隣り合う分岐予測ミスの間で命令キャッシュ・ミスが連続して発生した回数を表しており，8+ は 8 回以上命令キャッシュ・ミスが発生した場合を意味している．縦軸は全体に占める割合を表している．

TPC-C では，命令キャッシュ・ミスの連続発生数が広く分布している（同図において 0 の割合が小さく，8+ の割合が大きい）ため，5 節で述べた提案手法の利点（命令キャッシュ・ミスが連続して発生すると，より大きな性能向上が得られる点）が多く活用されているといえる．一方，その他のワークロードでは，命令キャッシュ・ミスの連続発生数の分布は狭い（同図において 0 の割合が大きく，8+ の割合が小さい）．これは，そもそも命令キャッシュ・ミスの発生回数が少ないためであり，このようなワークロードでは提案手法の性能向上率は小さくなっている．

7. まとめ

本論文では，命令キャッシュ・ミスが発生してもストールしない特性をもつミスを前提としたパイプラインを提案した．また，このミスを前提としたパイプラインを既存のパイプラインと組み合わせ，状況に応じて使い分けることで性能向上を図るアーキテクチャを提案した．本提案手法はプリフェッチャを用いない場合，従来よりも性能を低下させることなくフェッチ・スループットを向上させることができる．また，近年提案されている高性能な命令プリフェッチャと異なり，複雑な機構や大きなテーブルが必要なく，低コストで実現できる．TPC-C ワークロードを用いて評価した結果，提案手法は従来と比べて 22.4%の性能向上が得られることを確認した．

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 16K00070），及び科学研究費補助金 若手研究 (A) (課題番号 16H05855) による補助のもとで行われた．

参考文献

[1] Zhu, Y., Richins, D., Halpern, M. and Reddi, V. J.: Microarchitectural Implications of Event-driven Server-side Web Applications, *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 762–774 (2015).

[2] Ranganathan, P., Gharachorloo, K., Adve, S. V. and Barroso, L. A.: Performance of Database Workloads on Shared-memory Systems with Out-of-order Processors, *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 307–318 (1998).

[3] Reinman, G., Calder, B. and Austin, T.: Fetch Directed Instruction Prefetching, *Proceedings of the 32nd International Symposium on Microarchitecture*, pp. 16–27 (1999).

[4] Ferdman, M., Wenisch, T. F., Ailamaki, A., Falsafi, B. and Moshovos, A.: Temporal instruction fetch streaming, *Proceedings of the 41st International Symposium on Microarchitecture*, pp. 1–10 (2008).

[5] Ferdman, M., Kaynak, C. and Falsafi, B.: Proactive Instruction Fetch, *Proceedings of the 44th International Symposium on Microarchitecture*, pp. 152–162 (2011).

[6] Atta, I., Tozun, P., Ailamaki, A. and Moshovos, A.: SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads, *Proceedings of the 45th International Symposium on Microarchitecture*, pp. 188–198 (2012).

[7] Kallurkar, P. and R. Sarangi, S.: pTask: A smart prefetching scheme for OS intensive applications, *Proceedings of the 49th International Symposium on Microarchitecture*, pp. 1–12 (2016).

[8] Kolli, A., Saidi, A. and Wenisch, T. F.: RDIP: Return-address-stack Directed Instruction Prefetching, *Proceedings of the 46th International Symposium on Microarchitecture*, pp. 260–271 (2013).

[9] Kaynak, C., Grot, B. and Falsafi, B.: SHIFT: Shared History Instruction Fetch for Lean-core Server Processors, *Proceedings of the 46th International Symposium on Microarchitecture*, pp. 272–283 (2013).

[10] Kaynak, C., Grot, B. and Falsafi, B.: Confluence: Unified Instruction Supply for Scale-out Servers, *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 166–177 (2015).

[11] Kumar, R., Huang, C. C., Grot, B. and Nagarajan, V.: Boomerang: A Metadata-Free Architecture for Control Flow Delivery, *IEEE International Symposium on High Performance Computer Architecture (2017)*, pp. 493–504 (2017).

[12] Kumar, R., Grot, B. and Nagarajan, V.: Blasting Through the Front-End Bottleneck with Shotgun, *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 30–42 (2018).

[13] Shioya, R., Horio, K., Goshima, M. and Sakai, S.: Register Cache System Not for Latency Reduction Purpose, *Proceedings of the 43rd International Symposium on Microarchitecture*, pp. 301–312 (2010).

[14] Shioya, R. and Ando, H.: Energy efficiency improvement of renamed trace cache through the reduction of dependent path length, *IEEE 32nd International Conference on Computer Design*, pp. 416–423 (2014).

[15] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D. and Wood, D. A.: The Gem5 Simulator, *SIGARCH Comput. Archit. News*, Vol. 39, No. 2, pp. 1–7 (2011).

[16] Seznec, A.: A 256 Kbits L-TAGE branch predictor, *Championship Branch Prediction-2* (2006).