Vol.2009-ARC-185 No.6
2009/10/26

スケジュールド命令キャッシュを用いた高速な命令供給手法

三輪　忍[†1]　中條拓伯[†1]

本稿では，スケジュールド命令キャッシュを用いて，実行ユニットへ命令を高速に供給する手法を提案する．スケジュールド命令キャッシュは発行された命令列をそのままの順で保持するキャッシュである．それが保持する命令列は，フェッチと分解という，通常よりも低レイテンシな処理によって発行できる．高速な命令発行は分岐予測ミスの早期発見に繋がる．提案手法によって性能が最大 17.4% 改善された．

Fast Instruction Supply Method Using Scheduled Instruction Cache

Shinobu Miwa[†1] and Hironori Nakajo[†1]

We propose a fast instruction supply method using Scheduled Instruction Cache. Scheduled Instruction Cache is a cache which retains instructions in order as they were issued. An instruction in the cache is issued fastly because operations of issuing are quite simple : fetch and decompose. Fast instruction issue leads a branch mispredction to the fast detection. Proposal method improves performance by 17.4% or less.

1. はじめに

近年の Out-of-Order スーパスカラ・プロセッサでは，命令がフェッチされてから発行されるまでに多くのサイクルを必要とする．例えば命令キャッシュは，配線遅延の増大にともない，そのアクセスに複数サイクルを要するようになっている．デコーダやリネーマなどのユニットもまた，複数のパイプライン・ステージに分割されることが多い．その結果，命令がフェッチされてから発行されるまでには，12 サイクルも要することさえある[1)]．

こうした発行までのレイテンシは，具体的には，**分岐予測ミス・ペナルティ**としてプロセッサ性能に影響を与えている．発行までのレイテンシが大きいプロセッサでは，全ての命令がより遅いタイミングで発行される．その分，分岐結果も遅くに判明し，分岐予測ミスからの回復のタイミングも遅くなる．

分岐予測ミスを減らすさまざまな研究が行われているにも関わらず，ミス率はいまだに，アプリケーションによっては 10% を超えることもある[3)]．発行までのレイテンシを減らし，命令を高速に実行ユニットに供給できれば，プロセッサの性能はさらに向上するだろう．

命令供給の高速化を図った技術に，やや特殊な例ではあるが，Intel Pentium 4 の**トレース・キャッシュ**（**Trace Cache**．以下 **TC** とする）[4)] がある．Pentium 4 では，その内部で，x86 命令がマイクロ命令に変換されて実行される．変換処理には 1 サイクル以上要するが，TC がマイクロ命令を保持することによって，TC にヒットした場合にはその処理を省略できる[1)]．

しかし，すべての Out-of-Order スーパスカラ・プロセッサがマイクロ命令への変換を行うわけではない．一般のプロセッサにおいて発行までのレイテンシを短縮した例もある[7),8)]が，後述するように，それらは命令供給の高速化に主眼を置いていない．

本稿では，**スケジュールド命令キャッシュ**（**Scheduled Instruction Cache**．以下 **SIC** とする）を用いて，実行ユニットへ命令を高速に供給する手法を提案する．SIC は Out-of-Order に発行された命令をそのままの順で保持するキャッシュである．同時に発行できた命令群は，1 つの**命令グループ**[2)] として保持される．SIC が保持する命令列は，リネームもスケジューリングも済んだ状態である．そのような意味において，SIC は VLIW の命令キャッシュと同様である．

SIC にヒットした場合の動作もまた VLIW のそれとほとんど同じである．SIC からフェッチされた命令については，リネームやスケジューリングは必要ない．ほぼ直接実行ユニットに供給できる．そのため，SIC にヒットすれば，発行までのレイテンシを大幅に短縮できる．

本稿とは目的が異なるものの，Out-of-Order に発行された命令列をキャッシュに保持し再利用した例はある．そこでまず次章では，その従来の手法について詳しく述べる．続く 3 章，および，4 章では，我々が提案する，発行された命令列の再利用によってプロセッサを高速化する手法を説明する．3 章では手法の概念を説明し，4 章でその実装について述べている．評価は 5 章で行い，6 章でまとめる．

†1 東京農工大学
Tokyo University of Agriculture and Technology

Vol.2009-ARC-185 No.6
2009/10/26

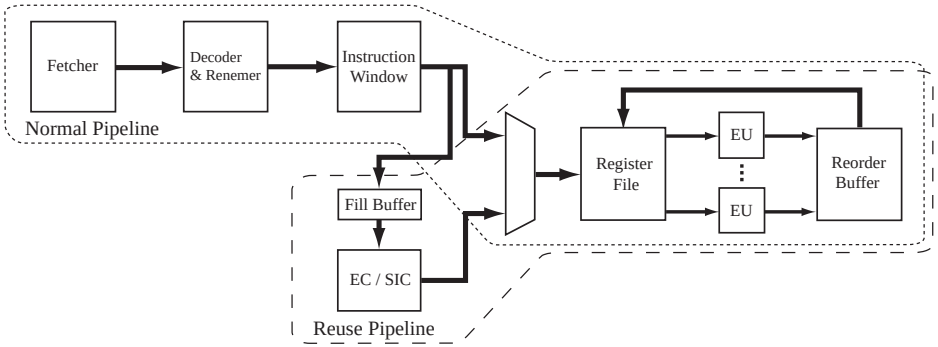

図 1 発行された命令列を再利用するプロセッサ

2. 従来の発行された命令列の再利用手法

Out-of-Order に発行された命令列の再利用は Talpes ら[7] も行っているが，その目的は我々とは異なる．彼らの研究は，過去に発行された命令列を高速に供給することが目的ではない．命令列を消費電力の少ないロジックから供給することで，フロントエンドの消費電力を削減することを狙っている[⋆1]．

本章ではその手法について詳しく述べる．なお，文献 7) では彼らが実装したプロセッサがボトムアップに説明されているが，本稿では彼らの再利用手法に着目し，それをトップダウンに説明する．そのため，以下では，原文にはない用語や図がいくつか補われている．原文を参照される場合は注意されたい．

2.1 概　　要

発行された命令列を再利用するプロセッサの構成を**図 1** に示す．図は物理レジスタ・ファイルを用いてリネーミングする方式の Out-of-Order スーパスカラ・プロセッサを想定している．3 章で述べるように，本稿で提案するプロセッサも全体構成はまったく同じである．

図 1 のプロセッサは，命令発行部分を除き，通常のスーパスカラ・プロセッサと基本的には同じ構造をしている．発行された命令列は，命令ウィンドウに並置された **Execution Cache**（以下 **EC** とする）によって保持される．命令は，レジスタ・ファイルの手前に置かれたセレクタを通して，命令ウィンドウ，EC のどちらか一方から発行される．

2.1.1 再利用のための命令パイプライン

図 1 のプロセッサは，まったく異なる命令パイプラインを持つ 2 つのプロセッサが，レジスタ・ファイル以下のユニットを共有していると見なすと理解しやすい．

1 つは通常の Out-of-Order スーパスカラ・プロセッサそのものである．命令の発行は通常のフロントエンド——— フェッチャ，デコーダ，リネーマ，命令ウィンドウ———を通して行われる．このパイプラインで発行された命令列が EC によって保持され，必要に応じて再利用される．本稿では，このパイプラインを**ノーマル・パイプライン**と呼ぶことにする．

もう 1 つは保持している命令列を VLIW のように実行するプロセッサである．EC が保持する命令列は，以前に発行された状態の命令列，すなわち，論理レジスタに物理レジスタが割り当てられ，なおかつ，データ依存が（以前は）解決した状態の命令列である．そのため，EC からフェッチされた命令は，各論理レジスタに再び同じ物理レジスタが割り当てられていれば，基本的にはただちに発行できる（詳しくは 2.2.1 項で述べる）．このパイプラインを**再利用パイプライン**と呼ぶことにする．

再利用パイプラインで命令を実行している間は EC から命令が供給されるため，通常のフロントエンド部分は必要ない．その間，(クロック/パワー）ゲーティングすることができる．EC は数十 KB の CAM で実現されるため，その規模はフロントエンド全体と比べて小さい．そのため，再利用パイプラインで実行することによって消費電力を削減できる．

2.1.2 2 つのパイプラインの動作

2 つのパイプラインは**非投機的に**動作する．異なるパイプラインが同時に動作することはない．**図 2** の I_b と I_t のように，一方のパイプラインで命令がすべてコミットされた次のサイクルから，もう一方でのフェッチが始まる．本稿では，ノーマル/再利用パイプラインで命令が処理される時間を，それぞれ，**ノーマル/再利用モード**と呼ぶことにする．

2 つのモードは EC のヒット/ミスによってスイッチングする．ただし，スイッチングのために EC が参照されるのは以下のどちらかの場合である．

- 分岐予測ミスが検出された時
- （レジスタ間接分岐などの）予測ミスしやすい分岐命令がコミットされた時（後述）

この時 EC にヒットすれば再利用モードに，ミスすればノーマル・モードになる．結果として，図 2 の 1 回目のスイッチング（時刻 $t+4$）のように同じモードが続くこともある．

2.1.3 再利用の単位

上述の方針は，分岐予測ミスした（しやすい）命令の次の命令から，Inorder 上での次の

⋆1 文献には高速化を目指しているかのような記述もあるが，それが目的とは到底言いがたい．結果として，2.3 節で述べるように，性能が大きく低下しているからである．5 章で示すように，我々の手法はほとんどのプログラムで性能を改善することから，両者は目的が異なると言ってよい．

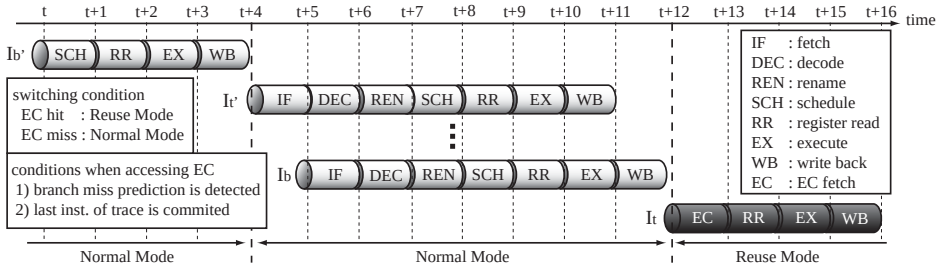

図 2 EC を用いたプロセッサの命令パイプライン

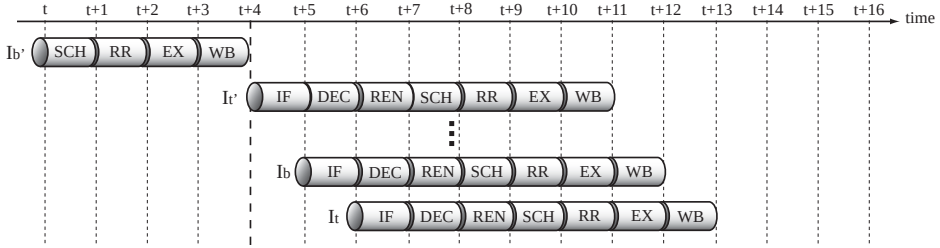

図 3 普通に実行して I_b の分岐予測がヒットした場合の命令パイプライン

分岐予測ミスした（しやすい）命令までをひとまとまりとみなしている．すなわち，Talpes らの手法では，そのような**トレース**の Out-of-Order な命令列を EC によって保持し，そのトレースが再び実行されると予測される場合にそれを再利用するのである．

例えば，図 2 において，命令 $I_{b'}$ を予測ミスした命令，$I_{t'}$ を正しいパスの先頭命令，I_b を次の予測ミスしやすい命令とする．$I_{b'}$ の分岐予測ミスによってスイッチングが起こると，$I_{t'}$ から I_b までのトレースはひとまとまりのノーマル・モード（時刻 $t+4$ から $t+12$）で処理される．この間，Out-of-Order に発行された命令列が EC に登録される．

$I_{b'}$ が再び実行されて再び予測ミスしたとしよう．すると，次に実行すべき（$I_{t'}$ を先頭とする）トレースが EC に存在するかどうか検索が行われる．検索の結果，EC にヒットすれば，リネーミングやスケジューリングを行うことなく，先程登録したそのトレースの発行順を得ることができる．以降は，保持しているトレースを分解し，分解された各命令をその順序で発行する．

このような単位で発行された命令列を再利用するのは，EC のヒット率を高めつつ，パイプラインが非投機的に動作することによるペナルティを緩和するためである．

再利用の単位とヒット率

トレースが長くなると，一般に，トレースの種類は増加する．トレース内の命令が Inorder に並んでいるか Out-of-Order に並んでいるかは関係がない．例えば，ある基本ブロックを起点として，基本ブロック N 個からなるトレースを作るとしよう．各分岐には成立/不成立の 2 つのパスがあるとすると，そのようなトレースは 2^N 個存在することになる．

トレースの種類が多くなると，それを保持する EC では競合の発生確率が増加してしまう．したがって，競合を抑えるにはトレースは短い方がよい．

非投機的実行のペナルティ

前節で述べたように，スイッチングの契機となる分岐命令が現れるとパイプラインは一端停止する．図 2 における I_b と I_t のように，その分岐（I_b）がコミットされてようやく，次の命令（I_t）のフェッチが始まる．そのため，分岐予測がヒットする場合には，上述のように非投機的に実行することによってペナルティを被ってしまう．

図 2 の命令列を普通にノーマル・パイプラインで実行し，I_b の分岐予測がヒットした場合のパイプライン・チャートを**図 3** に示す．普通に投機を行えば，I_t は時刻 $t+11$ には実行できる．これは図 2 のように実行するよりも 3 サイクルも早い．

このペナルティを緩和するため，Talpes らは，分岐予測ミスした命令，および，分岐予測ミスしやすい命令をスイッチングの契機としている．

2.2 発行された命令列を再実行する場合の問題点

前述のように，再利用モードでは，以前にノーマル・パイプラインでリネーミングされ，スケジューリングされた命令列が EC から供給される．このような命令列を実行した場合でも，当然，普通に実行した場合とまったく同じ演算結果が得られなければならない．そのためには以下の点を考慮する必要がある．

(1) 再利用を開始する時点での論理レジスタと物理レジスタのマッピング
(2) リオーダ・バッファへの命令の割り当て

この他にも，文献には明記されていない[⋆1]が，キャッシュにミスする命令の存在を考慮する必要がある．その詳細は 3.3 節で述べるとして，以下では上 2 つについて説明する．

2.2.1 レジスタ・マッピング

EC には，ノーマル・パイプラインで発行された状態，すなわち，リネーミングが済んだ状態の命令が登録される．各命令のオペランドは，ノーマル・パイプラインでは，物理レジ

⋆1 明記はされていないが，2.3 節で述べる実装からは，彼らもこの問題を認識していたことが伺える．

スタ番号に変換された上で発行される．EC はそうした変換済みの命令を保持する．

EC が保持する命令は物理レジスタに変換済みではあるが，だからといって単純に物理レジスタを読み出し，実行してよいというわけではない．参照する物理レジスタは，変換の際とまったく同じ論理レジスタに割り当てられているとは限らないからである．異なる場合に以前の割り当ての物理レジスタを参照すると，その命令が本来参照すべき論理レジスタではなく，まったく的外れの論理レジスタを参照することになってしまう．

変換時のレジスタ・マッピングと再利用時のそれとを一致させる最も単純な方法は，両者の開始の時点でのそれらを一致させることである⋆1．すなわち，再利用を開始する時点で，すべての論理レジスタに以前と同じ物理レジスタを割り当てる．そうすれば，開始前に定義された論理レジスタについては，以前の物理レジスタ番号で正しい論理レジスタを参照できる．開始後に定義される論理レジスタについても，命令列を再利用することが以前と同じ物理レジスタを割り当てることに相当するため，以前の番号で正しく参照できる．

開始時のマッピングを一致させるため，Talpes らはレジスタ・ファイルとレジスタ・プールを改造した．しかし，詳細は紙面の都合で省略するが，彼らの実装では，論理レジスタごとにレジスタ・プールを分散したことにより，各論理レジスタに割り当て可能な物理レジスタが 4 個程度にまで減っている．これは性能低下の大きな要因となる．

我々は性能低下を起こさない実装によりこの問題を解決した．その方法は 4.2 節で述べる．

2.2.2 リオーダ・バッファへの割り当て

EC から Out-of-Order に供給される命令列は，Out-of-Order にコミットしてよいわけではない．再利用モードでも分岐予測ミスや例外は発生する．分岐予測ミスや例外から正しく復帰するためには，再利用モードでもリオーダ・バッファは必要である．

このため Talpes らは，ノーマル・モードで各命令に割り当てられたリオーダ・バッファのエントリ番号も EC に記録している．EC から命令を発行する際は，その番号をもとにリオーダ・バッファのエントリを割り当てるようにする．

2.3 実　　装

EC 周辺の回路構成を**図 4** に示す．EC 自体は TC によく似た構成をとる．

タグ・アレイは以下のフィールドをもつ．

タグ　トレースの先頭命令のアドレス．ここで，先頭命令とはトレース内の命令を（Out-of-Order ではなく） Inorder に並べた際のそれを指す．

パス情報　トレース内の分岐命令の分岐結果を Inorder に並べたもの．成立を 1，不成立を 0 で表す．ただし，ハードウェア量削減のため，分岐結果は最初の 1 ～ 2 命令分しか保持しない．このようにしても性能はほとんど悪化しないと Talpes らは述べている．

有効ビット　トレースの有効/無効を表したビット．1 が有効，0 が無効に対応する．

カウンタ　再利用が中断された回数を数える 2 b 程度のカウンタ．0 を初期値とし，再利用が分岐予測ミスによって中断されるとインクリメントされる．カウンタが最大値に達するとトレースは無効になる．このような無効化を行うのは，再利用モードが中断される，電力削減効果の小さいトレースをキャッシュから除外するためである．

ヘッド（テール）　データ・アレイ上のトレースの先頭（末尾）を指すポインタ．

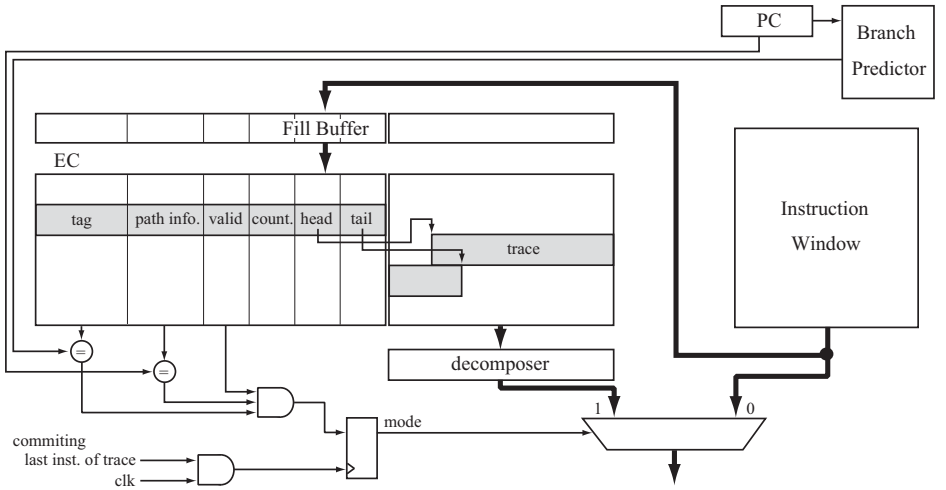

図 4　EC 周辺の回路構成

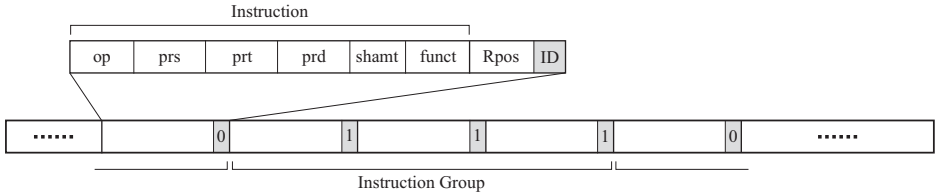

図 5　EC が保持するトレースのフォーマット

⋆1 この方針は単純ではあるが保守的である．再利用開始前に定義された論理レジスタのうち，開始後に参照されるものについてのみマッピングが一致していれば十分である．参照されないものについてはどうでもよい[6]．

トレースのフォーマットを**図 5** に示す．ISA は MIPS を想定している．トレースは，命令ごとに区切られた，いくつかのフィールドからなる．各フィールドの構成を以下に示す．

命令本体　前述のように，各オペランドは物理レジスタ番号（prs, prt, prd）で表される．

Rpos　ノーマル・パイプラインで実行された時のリオーダ・バッファのエントリ番号

ID　命令グループ，すなわち，同時に発行された命令群を識別するための 1 b の ID．同じ命令グループには同じ ID が割り振られる．図のように境界で ID を反転することにより，命令グループを区別する．

トレースはノーマル・モードで作成される．ノーマル・モード時に命令ウィンドウから発行された命令が，そのままの順序で**フィル・バッファ**に記録される．フィル・バッファ上のトレースは，ノーマル・モードが終わる際，EC に書き込まれる[⋆1]．

モードのスイッチングは，前述のように，分岐予測ミスした（しやすい）命令がコミットされた際（図では “commiting traces’s last inst. = 1” に対応）に行われる．その際，複数分岐予測を行い，その予測結果と次の PC とによって EC を参照する．EC にヒットし，有効ビットに 1 がセットされていればそのトレースは有効である．モード（図のフリップ・フロップ）は再利用モード（図では “1”）になる．

再利用モードでは，ヘッドとテールを用いてデータ・アレイからトレースを読み出す．読み出されたトレースは，分解機構で各々の命令に分解される．分解された命令列は，先頭から順に，グループ単位で毎サイクル発行される．前述のように，発行と同時にリオーダ・バッファへの割り当ても行う．発行された命令がまだ書き込まれていないレジスタを参照する場合は，パイプラインをストールさせる．

結　　果

文献 7) によると，50KB の EC を用いた場合，SPEC95/2000 ベンチマーク・プログラムにおいて，消費エネルギーが平均 29% 減少する．ただし，性能は 9.8% も悪化することが知られている．

3. スケジュールド命令キャッシュを用いた高速な命令供給手法

提案するプロセッサの構成は，基本的には，Talpes らのそれ（図 1）と同じである．後述するように，キャッシュやレジスタ・ファイル周辺の実装にやや違いはあるものの，全体構成に変わりはない．提案するプロセッサもやはり，ノーマル・パイプライン，再利用パイプラインという 2 系統の命令パイプラインをもつ．

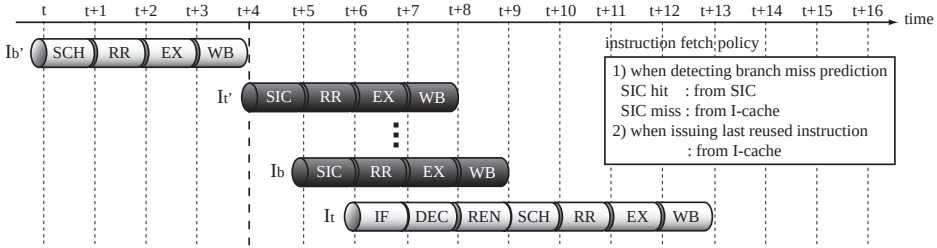

図 6　SIC を用いたプロセッサの命令パイプライン

最大の違いは 2 つのパイプラインが**投機的に**動作する点にある．提案手法には 2.2 節で述べた非投機的実行のペナルティが存在しない．そのため，提案手法には**性能上のデメリットがほとんどない**と言える．以下，詳しく述べる．

3.1 パイプラインの動作

提案手法における 2 つのパイプラインの動作を**図 6** に示す．分岐予測ミスした命令（「予測ミスしやすい命令」ではない）がコミットされた時にキャッシュを参照し，それにヒットした場合に再利用が始まる点は従来と同じである．

異なるのは再利用パイプラインからノーマル・パイプラインに切り替えるタイミングである．従来手法では，再利用が分岐予測ミスによって中断されなければ，トレースのすべての命令がコミットされてようやく，次の命令のフェッチがノーマル・パイプラインで始まっていた．図 2 においては，命令 I_t のフェッチは，I_b がコミットされた次の時刻 $t+12$ であった．それに対して提案手法では，トレースの最後の命令がフェッチされた時点で分岐予測を行い，ノーマル・パイプラインでの処理を投機的に開始するのである（図 6 の時刻 $t+6$）．

図 6 のパイプライン・チャートを図 3 のそれと見比べてほしい．I_b の分岐予測がヒットした場合には，I_t は，普通に実行した場合とまったく同じタイミングで実行される．すなわち，2.2 節で述べたペナルティは提案手法にはない．

一方，分岐予測ミスを早期に発見する効果は依然ある．図 6 のように実行した場合，I_b の分岐結果は時刻 $t+7$ に判明する．これは普通に実行した場合よりも 3 サイクルも早い．

提案手法のパイプラインが停止するのは，普通に実行した場合と同様，分岐予測ミスが発生した場合だけである．従来手法のように，分岐予測ミスしやすい命令が出現したからと

⋆1 文献では実装上の工夫として 16 命令ごとに書き込みを行っている．しかし，再利用時の正しい実行を保証するためには，ノーマル・モードで発行された（間違ったパスの命令以外の）すべての命令を EC に書き込む必要がある．そこで本稿では，簡単のため，書き込みはまとめて行われるものとして説明している．

いって，投機が止まることはない．そのため，3.3 節で述べる点を除き，提案手法には性能を低下させる要因はほとんどない．

3.2 再利用の単位

上述のように，提案手法では，分岐予測ミスした次の命令から，Inorder 上で次の予測ミスする命令までをひとまとまりとみなす．すなわち，そのようなトレース内の命令が Out-of-Order に発行された際の命令列を再利用の対象とする．

したがって，提案手法のトレースは従来手法のそれよりも長い．従来手法では，分岐予測ミスした命令，または，ミスしやすい命令の次の命令をトレースの起点とする．そして，そこから次の予測ミスする命令，または，ミスしやすい命令までを再利用の対象としていた．

2.1.3 項で述べたように，トレースが長いほど，それを保持するキャッシュ上で競合が発生しやすい．そのため，SIC は EC よりもキャッシュ・ミスを起こしやすい．

しかし，もし SIC にミスしたとしても，命令は普通にノーマル・パイプラインで実行されるだけである．ミスしたからといって，普通に実行した場合と比べて性能が低下するわけではない．単に分岐予測ミスを早期に発見する機会を逃しただけである．

3.3 性能上のデメリット

提案手法に性能上のデメリットがまったくないわけではない．ただし，そのデメリットは 2 つのパイプラインの動作方法に起因するのではない．命令を以前に発行された順序で再び実行しようとすること自体に起因する．

例えば，トレースを作成した際にキャッシュ・ミスしたロード命令が，そのトレースを再利用した際はキャッシュにヒットしたとする．その場合，トレース作成の際は，ロード命令に依存する命令はミスのタイミング，すなわち，下位のキャッシュからデータを取得できるタイミングでスケジューリングされる．そのため，そのトレースを再利用した際も，依存する命令はミスのタイミングで発行されてしまう．ロード命令がキャッシュにヒットして実行可能になったとしても，依存する命令はヒットのタイミングでは発行できない．

ほぼ同様のことが命令キャッシュ・ミスについても言える．普通に実行すれば命令キャッシュにヒットするにも関わらず，命令キャッシュにミスするタイミングでスケジューリングされたトレースを再利用すると損をしてしまう．

逆のケースは深刻である．以前はキャッシュにヒットしたロード命令が再利用時にはミスしたとする．すると，依存する命令は，以前はヒットのタイミングで発行できたため，再利用時にもヒットのタイミングで発行してしまう．ところが実際は，ロード命令がキャッシュ・ミスしているために，ヒットのタイミングではまだ依存が解決できていない．

前者のケースによる性能低下は，キャッシュ・ミスした命令を含むトレースを再利用の対象から外すことで緩和できる．ノーマル・パイプラインでロード命令がキャッシュ・ミスした場合，あるいは，命令キャッシュ・ミスが発生した場合は，フィル・バッファを無効化する．そうすれば，このような命令を含むトレースは SIC には書き込まれない．

後者のケースは，従来と同様，ストールによって対処せざるを得ない．この点は性能低下の要因となり得るが，5 章で述べるように，実際には一部のプログラムを除いてストールはほとんど発生しない．そのため，性能上のデメリットはほとんどないと言ってよい．

4. 実　　装

発行された命令列を保持するキャッシュ，SIC を実装した．また，性能低下を引き起こさない実装によって，2.2.1 項で述べたレジスタ・マッピングの問題を解決した．以下，それぞれについて述べる．

4.1 スケジュールド命令キャッシュ

SIC 周辺の回路構成を**図 7** に示す．SIC は EC（図 4）とほとんど同じ構成をとる．違いを以下に挙げる．

- タグ・アレイには，再利用を中断した回数を数えるカウンタは存在しない．
- 遅延を減らすためヘッド（テール）・ポインタは用いない．データ・アレイが保持するデータは固定長とする．遅延が少ない一方で，固定長のラインによって可変長のトレースを保持するため，アレイの利用効率は悪い．この点は今後改良の余地がある．
- 分岐予測ミスの早期発見という観点からは，分岐予測ミスによって再利用が中断されないトレースの価値は低い．そこで，SIC から発行された命令がすべてコミットされた場合は，そのトレースを無効化する．

再利用パイプラインへの切り替えは，分岐予測ミスした命令がコミットされた際（図では “commiting branch miss prediction inst. = 1” に対応）に行われる．EC と同様，その際に複数分岐予測を行い，SIC を参照する．SIC にヒットし，有効ビットがセットされていればそのトレースは有効である．以降は，SIC から命令が発行されるようになる．

フリップ・フロップはリセット機能付きのものとする．リセットは，SIC から最後の命令が発行された時に行われる．最後の命令が発行された次のサイクルから，(可能であれば）命令ウィンドウから命令が発行される．このようにして，SIC から発行された命令がすべてコミットされる前に，命令ウィンドウからの発行を再開する．

図 8 に SIC が保持するトレースのフォーマットを示す．トレースのフォーマットも EC

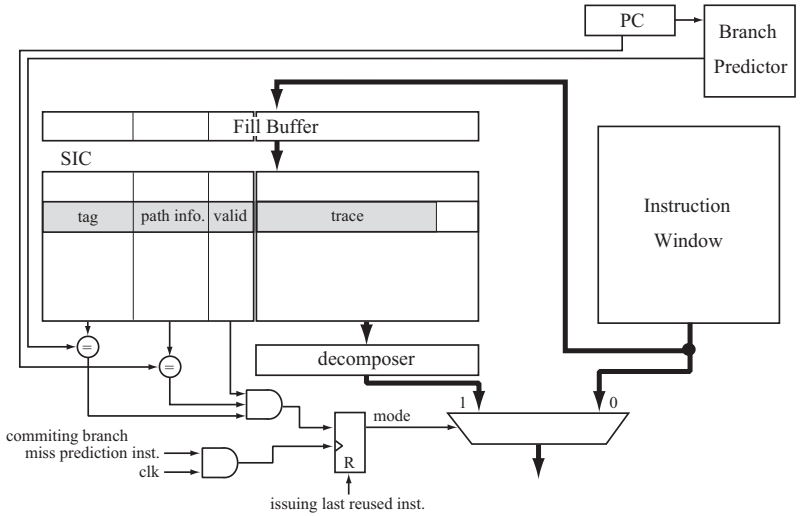

図 7 SIC 周辺の回路構成

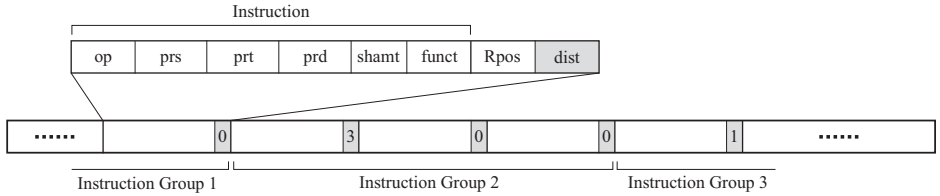

図 8 SIC が保持するトレースのフォーマット

のそれ（図 5）とほぼ同じである．

ただし，命令グループの表現に若干の違いがある．SIC のトレースでは，命令ごとのフィルードの最後の部分（dist）が，前の命令との間の発行間隔（サイクル数）を表す．”dist = 0” はその命令が 1 つ前の命令と同じサイクルに発行されたこと，すなわち，前の命令と同じグループであることを意味する．図 8 では，グループ 1 と 2 の発行間隔が 3 サイクルであったことを表している．再利用の際は，これをもとにグループ単位で命令を発行する．

4.2 レジスタ・ファイル周辺の構成

トレースの作成を始めた時点とそのトレースの再利用を始める時点とで，各論理レジスタに常に同じレジスタが割り当てられるようにするため，**論理レジスタ・ファイル**を用意する．命令がコミットされる際は，結果をデスティネーションに割り当てられている物理レジスタへ書き込むとともに，対応する論理レジスタ・ファイルにも書き込むようにする．

また，各論理レジスタに論理レジスタ・ファイルの各レジスタを割り当てた状態を表す，特殊なチェック・ポイントを用意する．そして，分岐予測ミスした命令がコミットされる際は，常にこのチェック・ポイントに回復する．そうすれば，トレースの作成を始める場合でも再利用を始める場合でも常に，各論理レジスタには論理レジスタ・ファイルの各レジスタを割り当てた状態になる．

提案手法では，再利用パイプラインからノーマル・パイプラインに投機的に切り替わることもある．その場合でも，再利用パイプラインで実行した命令が書き込んだレジスタを，ノーマル・パイプラインで実行する命令が正しく参照できなければならない．

このため再利用中もマップ表の更新は行う．ただし，マップ表の更新は，上述のように，パイプラインを切り替えた場合の実行の正しさを保証するために行われる．命令をリネーミングするためではない．そのため，マップ表の更新は命令の分解処理後に行えばよく，これによって SIC から命令を発行する処理の遅延が増加することはない．

5. 評　　価

提案手法を SimpleScalar ツールセット (ver. 3.0)[5] の sim-outorder シミュレータに実装し，評価を行った．以下ではその結果を述べる．

5.1 評 価 方 法

SIC のセット数，パス情報として用いる分岐の数を変えて評価を行った．セット数は 8〜32 の場合について測定した．分岐の数は 2 〜 8 とした．以下では，セット数 `x`，分岐数 `y` の SIC を用いるモデルを `SIC-Sx-By` と表すことにする．

3.3 節で述べたように，キャッシュ・ミスした命令を含むトレースを再利用すると性能が低下する可能性がある．そこで，L2 キャッシュにミスしたロード命令，または，命令キャッシュにミスした命令を含むトレースは再利用の対象外とした．

評価したプロセッサのパラメタを**表 1** に示す．分岐予測ミス・ペナルティは 20 サイクルとし，その内，フェッチから発行までのサイクル数を 12 とした[1]．

データ・アレイは，なるべく長いトレースを保持できるよう，1 ライン 4KB という横長のものを用いる．図 8 のフォーマットにしたがうと，dist が 7b の場合，命令ごとに必要な領域は 55b （表 1 より prs, prt, prd に各 7b，Rpos に 6b 必要）となる．すなわち，このラインにより 595 命令からなるトレースまで保持できる．横長なデータ・アレイが遅延に

表 1 プロセッサの各パラメタ

parameter	remarks	parameter	remarks
Way	4	Branch Predictor	32KB g-share
Instruction Window	64	BTB	4K entry, 4-way
Load/Store Unit	32	RAS	8
Register File	INT : 128, FP : 128	L1 I-Cache	32KB, 2-way
Execution Unit	IALU : 2, MULT : 1		64B/line, 2 cycle
	FPALU : 1, MULT : 1	L1 D-Cache	32KB, 2-way
	LD : 1, ST : 1		64B/line, 2 cycle
Reorder Buffer	64	L2 Cache	1MB, 8-way
Pipeline Depth	20	(Unified)	64B/line, 10 cycle
	(fetch ∼ issue : 12)	Main Memory	200 cycle

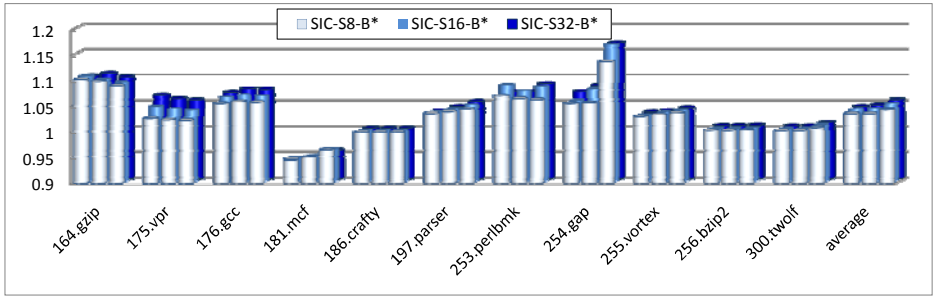

図 9 性能向上率

与える影響が懸念されるが，これはマルチバンク化することで解消できると考えている．

SIC の連想度は 4 とした．したがって，セット数が 8 のキャッシュのデータ・アレイ・サイズは 128KB，32 のそれは 512KB である．なお，SIC から命令を発行する際のレイテンシは 3 サイクル（フェッチ：2 サイクル，分解：1 サイクル）としている．

測定には，SPEC CINT2000 の（*253.eon* を除く） 11 本のプログラムを使用した．入力セットには train を用いた．プログラムは，最初の 1G 命令をスキップし，続く 256M 命令を実行した．ISA は PISA とした．

5.2 評価結果

普通に実行した場合に対する性能向上率を**図 9** に示す．グラフの横軸はプログラム名（一番右は平均），縦軸は性能向上率である．プログラム毎の横に並んだ 3 本の棒グラフは，左から順に，分岐数 2,4,8 の場合を表している．また，奥方向に並んだ 3 本の棒グラフは，手前の色が薄い方から順に，セット数 8,16,32 の場合に対応している．

グラフより，*181.mcf* を除くすべてのプログラムで性能が向上している．特に `SIC-S32-B8` においては，平均 6.01%，最大 17.4% （*254.gap*）の性能向上が見られた．

mcf で性能が悪化するのは，3.3 節で述べたストールが発生するためである．*mcf* では，全体の 9.2% のサイクルでストールが起きていた．この点は今後改良の余地がある．

しかし，残りの 10 本の平均では，ストールしたサイクルの割合は 1.10% であった．このように，ほとんどのプログラムではストールは問題にならない．

6. まとめ

本稿では，発行された命令列を再利用することによってプロセッサを高速化する手法を提案した．本手法によって，平均 6.01%，最大 17.4% 性能が改善できた．今後は，提案した回路を実際に実装し，面積・遅延等を評価する予定である．

謝辞 本研究の一部は共生情報工学推進経費による．

参考文献

1) Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A. and Roussel, P.: The Microarchitecture of the Pentium 4 Processor, Technical Report Q1, Intel Technology Journal (2001).
2) Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H. and Zahir, R.: Introducing the IA-64 Architecture, *IEEE Micro*, Vol.20, No.5, pp.12–23 (2000).
3) Ishii, Y.: Fused Two-Level Branch Prediction with Ahead Calculation, *The Journal of Instruction-Level Parallelism*, Vol.9 (2007).
4) Rotenberg, E. and Bennet, S.: Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching, *Proc. 29th Annual Int'l Symp. on Microarchitecture* (1996).
5) SimpleScalar LLC: http://www.simplescalar.com/.
6) Stephen, M. and Yale, P.: Enhancing instruction scheduling with a block-structured ISA, *International Journal of Parallel Programming*, Vol.23, No.3, pp. 221–243 (1995).
7) Talpes, E. and Marculescu, D.: Execution Cache-Based Microarchitecture for Power-Efficient Superscalar Processors, *IEEE Transactions on VLSI Systems*, Vol.13, No.1, pp.14–26 (2005).
8) 一林宏憲，塩谷亮太，入江英嗣，五島正裕，坂井修一：逆 Dualflow アーキテクチャ，情報処理学会論文誌コンピューティングシステム， Vol.1, No.2, pp.22–33 (2008).