CC-001

# 予測ベースの先行実行による In-Order プロセッサの高速化

灘 洋太郎[1,a)]　小泉 透[2]　塩谷 亮太[1]　入江 英嗣[1]　坂井 修一[1]

**概要**：In-Order (InO) プロセッサはプログラムオーダに従ってプログラムを実行するプロセッサであり，ハードウェアが単純であるため消費電力や回路面積が小さい．このため InO プロセッサは消費電力や回路面積が重視される環境で広く用いられているものの，その一方で性能の低さが課題となる．これに対し，本研究では InO プロセッサの単純さを維持しつつ，その性能を改善することを目指す．我々は，ロード命令の結果を使用するコンシューマ命令が InO プロセッサにおいて多くのストールを引き起こしていることに着目した．この観察に基づき，我々は InO プロセッサにおけるロード命令の投機的な先行発行を提案する．提案手法では発行可能性を予測してロード命令をあらかじめ投機的に発行しておくことにより，そのコンシューマ命令のストールを効果的に取り除く．これにより，Out-of-Order 実行のような複雑な機構を導入することなく，非常に小さな追加コストで大きな性能向上を実現する．シミュレーションによる評価の結果，提案手法を実装したプロセッサは既存の InO プロセッサと比べて 15% 高い性能を示した．

## 1. はじめに

In-Order (InO) プロセッサは，その名が示すようにプログラムオーダに従って InO にプログラムを実行するプロセッサである．InO プロセッサは，Out-of-Order (OoO) 実行を行うプロセッサと比べてハードウェアが単純であるため，消費電力や回路面積が小さい．このため，InO プロセッサはモバイル分野や組み込み分野などの，消費電力や回路面積の制約が強い用途で広く用いられている．また，近年では big.LITTLE と呼ばれるアーキテクチャにおいても，InO プロセッサが採用されている [2]．この big.LITTLE は高性能な OoO プロセッサと高電力効率な InO プロセッサを組み合わせて使用し，性能と電力効率の両立を図るものである．これらの用途のために，近年でも様々な InO プロセッサが開発され，新しく登場している [3], [4], [5].

本研究では，このような InO プロセッサの単純さを維持しつつ，その性能を改善することを目指す．InO プロセッサは一般に OoO プロセッサと比べて性能が低く，ある程度の性能が求められる用途では採用が難しい．これに対し，InO プロセッサの性能を改善することで，小型で電力効率の高い InO プロセッサの利用範囲をより要求性能が高い用途にまで広げることができる．また，InO プロセッサの性能改善は，big.LITTLE における電力効率の改善にも繋がる．big.LITTLE では要求性能の高さに応じて OoO と InO のプロセッサを切り替えるため，InO プロセッサの性能を改善することにより，より要求性能が高い状況でも電力効率の高い InO プロセッサを使用することができる．

InO プロセッサの性能を改善するため，我々はロード命令の結果を使用するコンシューマ命令に着目した．InO プロセッサでは命令は必ず InO に発行されるため，特定の命令が依存元命令の完了を待つ場合にはその後続の命令は一切発行できず，パイプラインがストールする．我々は，このストールをロードの命令のコンシューマが頻繁に引き起こすことに着目した．一般にロード命令はキャッシュヒットした場合でも複数サイクルのレイテンシを持つため，そのコンシューマはロード命令の完了を待つ事が多い．プログラム内ではロード命令とそのコンシューマが連続して配置される事が多いためこのような待ち合わせが頻発し，それによるストールは大きな性能低下を招く場合がある [6]. 静的な命令スケジューリングによる対処も試みられているが，分岐や可変レイテンシの命令といった動的なイベントに対応できない [7], [8].

この観察に基づき，我々は InO プロセッサにおけるロード命令の投機的な先行発行を提案する．提案手法ではフロントエンドでロード命令がフェッチされた際に，それがいつ発行できるかを予測する．この予測に基づき，ロード命令は，先行する命令がまだ発行されていない場合でも それを追い越して投機的に発行される．投機的に発行されたロード命令は後から検証され，依存を満たしておらず投機

[1] 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
[2] 名古屋工業大学情報工学科
Department of Computer Science, Nagoya Institute of Technology
[a)] nada@mtl.t.u-tokyo.ac.jp

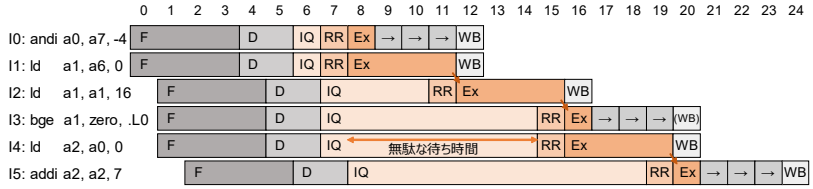


**図 1** 一般的な InO プロセッサのパイプラインチャート．F はフェッチステージ，D はデコードステージ，IQ は命令キューに入っている期間，RR はレジスタ読み出しステージ，Ex は実行ステージ，WB はレジスタ書き込みステージである．→は待機ステージで，ステージング [1] による InO 書き込みの実現のためのものである．

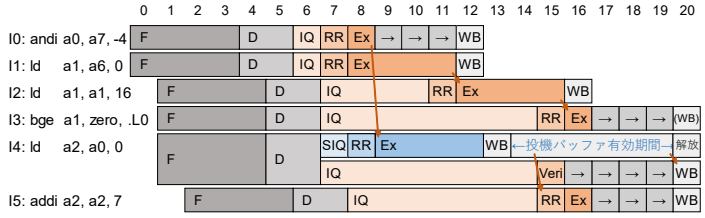


**図 2** 提案手法を搭載したプロセッサのパイプラインチャート．I4: ld a2, a0, 0 が二行になっているのは，投機実行の動作と本実行の動作を分けて記述するためである．SIQ は投機命令キューに入っている期間，Veri は投機検証ステージである．それ以外は図 1 と同様である．図は投機の検証が成功した場合を示している．この場合、I5: addi a2, a2, 7 以降のすべての命令の実行が 4 cycle 早まることとなる．

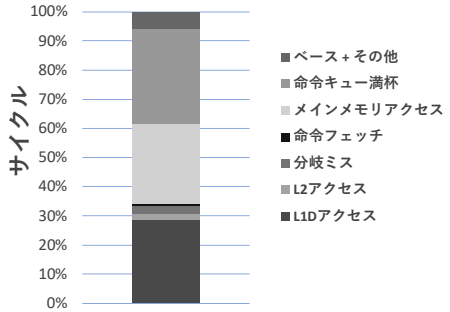


**図 3** SPEC CPU 2017 Int を実行した際に費やしたサイクルの内訳

が失敗している場合には再度実行される．このようにしてロード命令をあらかじめ投機的に発行しておくことにより，その完了を待つコンシューマ命令のストールを効果的に取り除くことができる．

このようなロード命令とコンシューマがある場合でも，OoO プロセッサであればコンシューマの後続にある依存がない命令を先に実行しロード命令のレイテンシを隠蔽できる．しかし，OoO 実行のためには複雑なハードウェアが必要であり，消費電力や回路面積を大幅に増やしてしまう．これに対し提案手法は，ロード命令のみを単純な機構を用いて投機的に発行することにより，OoO 実行と比べて非常に小さな追加コストで効率的に性能を向上させる．

以下に，本研究の貢献をまとめる：

- 我々は，InO プロセッサにおいてロード命令のコンシューマが発行できないことによるストールが性能を大きく低下させている事を見出した．
- この観察に基づき，我々は InO プロセッサにおけるロード命令の投機的な先行発行を提案した．提案手法は OoO 実行のような複雑な機構を導入することなく，非常に小さな追加コストで大きな性能向上を実現する．
- 提案手法をサイクルアキュレートなシミュレータ Sniper [9] に実装し評価した．評価の結果，提案手法は SPEC CPU 2017 Speed [10] ベンチマークにおいて，既存の InO プロセッサと比べて 15% 性能が向上した．また，理想的な予測器や検証機構を用いた場合には 28% 性能が向上することが確かめられ，予測器などの改良によるさらなる性能向上の余地が示された．

## 2. 背景とモチベーション

InO プロセッサは，命令キュー内の一番古い命令のみを発行するという制約を持つ．この制約により，OoO プロセッサと比べて遥かに単純な機構で命令の発行可能性を判定できる．しかし一方で，この制約があるために InO プロセッサの性能は OoO プロセッサより低くなる．これは，キュー内で一番古い（キュー先頭の）命令が発行できない時に命令発行が止まってしまうためである．

特に，キュー先頭の命令が複数サイクルのレイテンシを持つ命令に対して真に依存しているとき，命令発行が止まりやすい．図 1 に，このような依存関係によって発行が止まる様子の一例を示す．この例では，命令 I3 はロード命令 I2 に依存しており，キューの先頭で発行待ちとなる．ロード命令 I4 は実行準備が整っているが，命令 I3 が発行されるまでは発行されない．このように命令 I4 以降の命令はすべて，ロード命令 I2 の結果を参照しない場合であっても，ロード命令 I2 の完了を待つこととなる．

なかでも，ロード命令のコンシューマが発行できないことによるストールは InO プロセッサの主要な性能ボトルネックの 1 つとなっている．InO プロセッサで SPEC CPU 2017 Speed を実行したときのサイクル数の内訳を図 3 に示す．なお，測定環境は 5 節の評価で用いたものと同じである．図 3 は，InO プロセッサにおいて全実行時間の約 30%もの時間が，L1D キャッシュからデータが得られるのを待つためだけに費やされていることを示している．

これに加えて我々は，ロード命令の中には OoO 発行可

能なタイミングを簡単に予測できるものが一定数あることを見出した．例えば，「命令キューに挿入された瞬間にすでに発行可能になっていた」が全実行期間を通じて成り立つロード命令は，全ロード命令の 24.9%を占めていた．

これらの観察に基づき我々は，InO プロセッサの性能を改善するために，ロード命令の発行可能性を予測し，先行発行する手法が有効ではないかと考えた．ただし，ロード命令の発行可能性が予測できたとしても，それを投機発行する機構と検証する機構が軽量でなければ，本研究の目的を達成できない．そこで，本研究では投機発行と検証を行う軽量な機構を提案する．本稿の構成は以下のとおりである．3.1 節では，InO プロセッサで OoO 発行を行う既存手法はいずれも，高コストな OoO 発行機構を用いていることを見る．3.2 節では，軽量な投機検証機構に転用できると我々が着目した手法を紹介する．

## 3. 関連研究

### 3.1 InO プロセッサで OoO 発行を行う既存手法

#### 3.1.1 CASINO

CASINO [11] は，2 つの FIFO な命令キュー（投機命令キュー，通常の命令キュー）を用いて，OoO な命令発行を行うプロセッサアーキテクチャである．フロントエンドからディスパッチされた命令はまず投機命令キューへ格納される．投機命令キューからは命令を FIFO な順で発行してゆくが，まだ発行できない命令があったときはストールせずに，通常の命令キューへその命令を転送する．通常の命令キューは転送されてきた命令を FIFO な順で発行してゆく．そして，まだ発行できない命令に発行順が回ってきた場合は適宜ストールして発行可能になるのを待つ．

CASINO の問題点は，投機命令キュー内の命令を OoO 実行するために，複雑・高消費電力な機構を用いることにある．CASINO は，リザベーションステーション方式の機構で投機命令キューから発行する命令をリネームしている．この方式は，wakeup ロジックに複雑かつ高消費電力な CAM を必要とする．また，さらに，OoO プロセッサの Store Queue と同様の複雑な機構で動的なメモリ曖昧性解決を行っており，これも消費電力や回路面積を大きくしている．

#### 3.1.2 Load Slice Core

Load Slice Core [12] は，CASINO と同様に 2 つの FIFO な命令キュー（バイパス命令キュー，通常の命令キュー）を用いて OoO な命令発行を行うマイクロアーキテクチャである．Load Slice Core は，ロード命令とそのアドレスを生成する命令を早期にバイパス命令キューから OoO 発行することで，メモリレベル並列性を抽出することを主眼としている．

Load Slice Core の問題点は，OoO 発行するために全命令に対してレジスタリネーミングを適用していることにある．このため Load Slice Core は物理レジスタ方式のリネームを採用しており，複雑・高消費電力な RMT やその復帰用のログ，また大きな物理レジスタファイルを用いてしまっている．

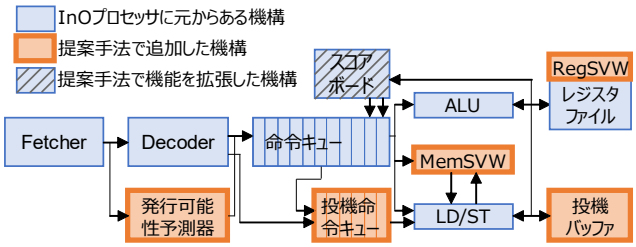


**図 4** 予測ベース先行実行を実現するマイクロアーキテクチャ

### 3.2 ロードストアの順序違反を軽量に検出する既存手法

#### 3.2.1 Store Vulnerability Window

Store Vulnerability Window (SVW) [13] は，ロードの投機実行から投機が検証されるまでの間に，同じアドレスに書き込んだストアが存在したかを判定する機構である[*1]．同様の目的を達成する機構として load queue が存在するが，load queue は content-addressable memory (CAM) で構成されるのが一般的であり，回路面積と消費電力が大きくなりがちである．SVW は RAM を使って実装できるため，load queue を用いる場合と比べてハードウェアを単純化できるという利点がある．

SVW は，正確には，「同じアドレスに書き込んだストアが存在した**可能性があるか**」を判定する．すなわち，実際には同じアドレスに書き込んでないにも関わらず，「存在した可能性がある」と誤判定することがある（偽陽性）．このような偽陽性を許容することで，SVW はハードウェアの大幅な簡素化を実現している [13]．

SVW の検証の仕組みは，次のようなものである．まず，RAM で作られたテーブル $svw$ を用意する．また，メモリアドレスを入力すると $svw$ のインデクスを出力するハッシュ関数 $hash$ も用意する．さらに，ストア命令それぞれに，フロントエンドでユニークな番号 (store sequence number, ssn) を振る．このとき，プログラム中の順番が早ければ早いほど，小さな ssn を振られるようにする．また，最後に実行されたストアの ssn を記録する変数 $SSN_{retire}$ を用意し，0 で初期化する．これらの準備を行った上で，ロード・ストアを実行するとき以下の工程を行うことで，順序違反を検出する．

- ストアを実行するとき，$svw[hash(\text{ストア先アドレス})]$ に，ストアの ssn を書き込む．また，SSN_retire にも書き込む．
- ロードを投機実行するとき，SSN_retire を読み出し，

[*1] 誤ったストアフォワーディングの検出にも使われるが，本論文には関係しないため詳細は省いた．

記録しておく．

- ロードを検証するとき，$svw$[hash(ロード先アドレス)] に書かれた番号を読み，投機実行開始時に記録した値よりも大きければ，ロードストアの順序違反が発生したと判定する．

SVW はこのような動作をするため，アドレスの異なる（すなわち，絶対に順序違反とならない）ロードとストアのペアについて，それらのアドレスのハッシュ値が偶然一致した場合に偽陽性を報告する．

#### 3.2.2 Bloom-like SVW

SVW は load queue より軽量な機構でロードストア順序違反を検出できる一方で，ハッシュ値の衝突による偽陽性が発生する．このような偽陽性を減らす手法として，Bloom-like SVW が提案されている [14]．Bloom-like SVW は異なるハッシュ関数を用いる複数の SVW を並列に動作させ，すべてが順序違反を検出した時のみ，全体として順序違反であると判定する．総テーブル容量を固定した場合，複数のハッシュ関数を用いる手法は単一のハッシュ関数を使う手法と比べ，偽陽性の確率を低くできる利点を持つ．

## 4. ロードの投機実行を行うマイクロアーキテクチャ

### 4.1 ベースラインの InO プロセッサ

InO プロセッサでは，フロントエンドから供給される命令はまず FIFO な命令キューへ格納される．そして毎サイクル，命令キューに入ったタイミングが早い命令から順に発行可否を判定してゆく．発行できない命令があった場合は，そこでそのサイクルの発行を打ち切る．

発行可否の判定は，スコアボードというテーブルを用いて行う．スコアボードは各論理レジスタの値が ready か書かれている表である．発行可否を判定する際には，判定対象の命令のソースレジスタそれぞれに対応するスコアボードのエントリを読み出し，そのレジスタの値が ready か判定する．全てのソースレジスタの値が ready である場合は，その命令を発行できることが分かる．このようにして発行可能であると判定された命令は，対応する演算ユニットが使用可能であるときに発行される．演算ユニットへ発行された各命令は，実行を開始するときにデスティネーションレジスタに対応するスコアボードのエントリへ not ready を書き込む．また，実行が終了してデスティネーションレジスタへ書き込む際に，そのレジスタに対応するスコアボードのエントリへ ready を書き込む．

InO プロセッサにおいて正確な例外を保証する方法は 2 つある．1 つ目の方法は，OoO プロセッサと同様に Reorder Buffer (ROB) を用いる方法である．もう一つは，全ての命令で，例外を検知するまでのレイテンシを同一に揃える方法である（ステージング [1]）．InO プロセッサは命令を InO に発行するため，ステージングを行うことで例外の発覚する順番も InO になり，例外への対処を簡単に行うことができる．すなわち，例外を検知したタイミングで，例外を起こした命令より前のステージにある命令をすべてパイプラインからフラッシュし，例外処理へ移ればよい．

### 4.2 提案手法の概観

提案手法は，ロード命令が OoO 発行可能であるタイミングを予測して OoO に先行発行し，その結果を後続命令が使えるようにする．これにより，このロード命令に依存した命令の発行を早め，プログラム実行のクリティカルパスを縮めることを狙う．投機発行されない命令は通常の InO プロセッサと同様の機構で InO 発行するため，従来通り軽量なハードウェアで実現可能である．

図 4 に，提案手法のマイクロアーキテクチャを図示する．提案手法は，ベースラインの InO プロセッサに対して以下の機構を追加する．

- 投機バッファ: 投機発行されたロードの結果などを書き込むバッファと，そのフリーリスト
- 投機命令キュー：未発行の投機ロードを格納する FIFO な命令キュー
- SVW: 投機ミス（発行が早すぎて不正なデータを受け取った可能性）を検出する機構
- 発行可能性予測器: 投機発行可能なロード命令と，その命令が投機発行可能なタイミングを予測する予測器

なお，投機バッファの各エントリは，そのバッファに書き込む命令が完了したかを示す 1bit のフィールド (ready bit) と，投機ロードにより得られた値を書くフィールド，投機実行で例外が発生したときの例外コードや例外の対処に必要なデータを書くフィールド（4.8 節参照）をもつ．また，発行する命令を選択するイシュー機構は，投機命令キューからも発行できるように拡張する (4.5 節参照)．さらに，スコアボードの各エントリには，投機バッファ番号を書き込むフィールドを追加する．このフィールドは，投機実行結果のフォワーディングを行うために用いる（4.6 節参照）．

これら機構を用いて，以下の (1)~(5) の手順で投機発行とその結果のフォワーディングが行われる (図 5~ 図 8)．

(1) 命令がフェッチされた時に，その命令が投機発行可能か，可能ならどのタイミングなのかを予測する（図 5）．
(2) 予測されたタイミングで，命令キューから投機命令キューに命令をコピーする（図 6）．
(3) 発行幅が余っている，つまり非投機な命令発行だけで発行幅を使い切れなかったとき，投機命令キューの先頭から命令を発行する（図 7）．
(4) 投機命令キューにコピーされた元命令が命令キューの先頭に来た時，必要であれば投機実行の完了を待ち，その後投機検証ユニットに発行する．投機検証ユニットに発行する際，スコアボードに not ready を書き込まず，代わりに投機バッファ番号を書き込む．これに

より，後続の命令は投機バッファからのフォワーディングを前提とした発行可能性判定を行うことができる（図 8）．

以下，それぞれの手順について詳しく説明してゆく．

### 4.3 手順 (1)：発行可能性予測

投機発行可能な命令がどれであるか，および投機発行可能になるタイミングは，投機発行可能性予測器を用いて予測する。この予測は，フロントエンドでフェッチやデコードと並行して行う．予測器は，予測対象の命令の PC をインデクスとしてアクセスすると，予測値を返すテーブルである．予測値には，以下の 3 種類がある．

- 投機発行可能ではない（以下、「投機不可」と呼ぶ）
- フロントエンドから命令キューに挿入した時点で既に発行可能（以下、「ディスパッチ時発行可」と呼ぶ）
- 命令キューの中央に来た時点以降なら発行可能（以下、「キュー中央で発行可」と呼ぶ）

予測結果は命令キュー内のエントリに記録され，予測器の学習や命令キュー中央から投機命令キューへ挿入するときに参照される（4.4 節参照）．

予測器は，投機ミスの発生が判明した場合，より悲観的な予測に切り替えることで学習を行ってゆく．すなわち各ロード命令に対して「ディスパッチ時発行可」→「キュー中央で発行可」→「投機不可」の順で試してゆく．そして，投機ミスが起きなかった場合はその予測値を出力し続ける．具体的な予測器のテーブルの学習アルゴリズムは次の通りである．

- 予測器は，テーブルにエントリを持たない PC の命令については，「ディスパッチ時発行可」を予測する．
- 「ディスパッチ時発行可」と予測した命令が投機ミスした場合は，テーブルにエントリを確保し，「キュー中央で発行可」と書き込む．
- 「キュー中央で発行可」と予測した命令が投機ミスした場合は，（必要であればエントリを確保し，）対応するエントリへ「投機不可」と書き込む．
- 予測した命令が投機ミスを起こさなかった場合には，予測を変更しない（何も書き込まない）．
- エントリの置換に用いる LRU ビットは，予測時と学習時に更新する．投機ミスを起こさなかった場合の学習でも，LRU ビットだけは更新される．

### 4.4 手順 (2)：投機命令キューへの挿入

発行可能性予測器が予測したタイミングで，投機命令キューへ命令を挿入する．「ディスパッチ時発行可」と予測された命令は，フロントエンドから命令キューへ挿入する際に，同時に投機命令キューにも挿入すればよい．「キュー中央で発行可」と予測された命令については，フロントエンドから命令キューへ挿入する際，(1) 命令キューが半分

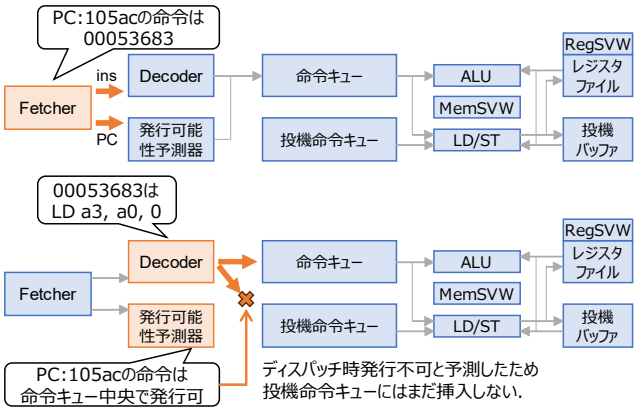


**図 5** 投機発行の手順 (1)：発行可能性予測

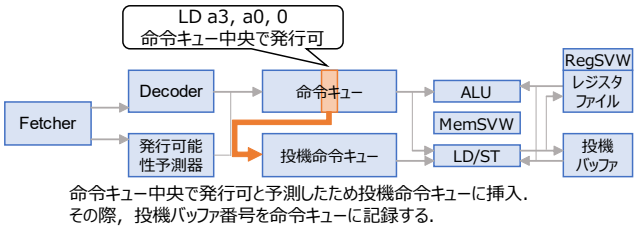


**図 6** 投機発行の手順 (2)：投機命令キューへの挿入

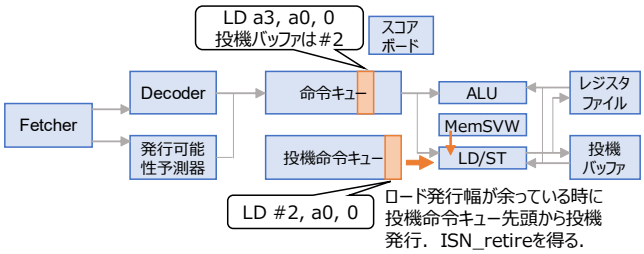


**図 7** 投機発行の手順 (3)：投機命令キュー先頭から投機発行

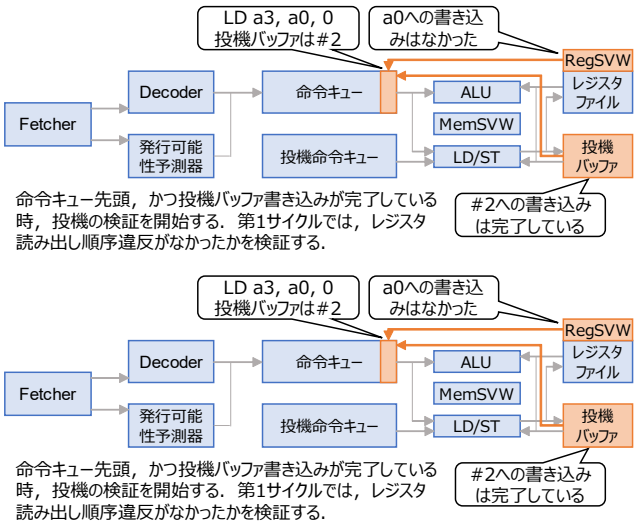


**図 8** 投機発行の手順 (4)：投機の検証とフォワーディング

未満しか埋まってなかったら，同時に投機命令キューにも挿入する，(2) そうでなければ，命令キュー中央を監視することでそこに到達したことを検出し，その際に投機命令

キューへ格納する．ここでいう命令キュー中央とは，命令キューサイズを $S$、最大発行幅を $I$ として，$S/2$ 番目から $S/2+I-1$ 番目のことである[*2]．このような手順で投機命令キューに命令を挿入するため，投機命令キュー内の命令はプログラム順で並んでいるとは限らない．

投機実行の結果を書き込む投機バッファは，投機命令キューへの挿入時に確保する．つまり，空いている投機バッファの番号を，フリーリストから取得する．投機バッファを取得できなかった場合，投機命令キューへ挿入するのを中止する．確保された投機バッファ番号は，命令キューと投機命令キューの双方に書き込む．

### 4.5 手順 (3): 投機命令キュー先頭からを投機発行

投機命令キューの先頭の命令は，非投機な命令発行でロード発行幅を使いきれなかったサイクルにのみ，発行される．つまり，投機ロードのための専用ユニットを追加するのではなく，既存のロードユニットを使用する．投機発行キュー先頭の命令は，スコアボードを読み出すことをせず，ロード発行幅が余っていれば無条件で発行される．したがって，投機発行をサポートするためにスコアボードの読み出しポートが増えたり，レジスタの読み出しポートが増えたりすることはない．

投機実行を開始するときには，投機バッファの ready bit を 0 (not ready) にする．また，投機実行結果が得られて，投機バッファへの書き込みが完了した時点で，投機バッファの ready bit を 1 (ready) にする．これらはいずれも，スコアボードの更新ではないことに注意する．

### 4.6 手順 (4): 投機の検証とフォワーディング

投機発行された命令が命令キュー内の先頭に来た場合，次の処理を行う．

(1) 投機実行が未完了の場合は完了するまで待つ．投機実行がまだ始まっていない場合も同様に待つ．
(2) 投機実行が完了したことを確認したのち，図 9 に示す流れで投機の検証を行う．
(3) 検証の開始と並行して，後続の命令への投機実行結果のフォワーディングも開始する．

投機実行が完了したかの確認は，投機実行結果が書き込まれる投機バッファの ready bit を毎サイクル読み出すことで行う．この命令の投機実行結果が書き込まれる投機バッファのインデクスは，手順 (2) で命令キューのエントリに書き込まれている（4.4 節参照）ため，それを使って投機バッファにアクセスすればよい．

投機の検証は，2 サイクルに分けて行う．第 1 サイクルには，投機実行時に読みだしたレジスタが，現在までに書き換わっている可能性がないかを，レジスタ SVW を用いて検証する．第 2 サイクルには，投機検証ユニットへ発行し，投機実行時に読みだしたアドレスのメモリが，現在までに書き換わっている可能性がないかを，メモリ SVW を用いて検証する．第 1 サイクルには，並行してスコアボードの読み出しも行う．このようにすることで，検証の結果レジスタを介した依存を守れていなかったと発覚した場合に通常の発行に切り替えることができ，パイプラインフラッシュを行わずに済む．

第 1 サイクルの検証が成功した場合，スコアボードに投機バッファ番号を書き込むことで，後続の非投機命令[*3]が投機結果を受け取り実行できるようにする．このフォワーディングは，以下の時点で終了する．終了時にはスコアボードから投機バッファ番号を消去し，投機バッファをフリーリストへ返却する．

- 投機検証ユニットのパイプラインを抜け，投機ロードの結果が論理レジスタに書き戻されたとき．今後はその論理レジスタから読み出せるので，投機バッファを解放してよい．
- 第 2 サイクルの投機検証が失敗した時．このとき後続の命令をすべてフラッシュするため，フォワーディング結果を使う命令は存在しなくなる．

このようにすることで，ソースオペランドについてスコアボードから「ready かつ投機バッファ番号が $N$」という情報を得た命令は，レジスタ読み出しステージで論理レジスタの代わりに投機バッファ $N$ から読み出すことで，投機実行の結果を使うこととなる．

### 4.7 投機実行の検証の詳細

投機的に先行発行されたロードによって得られる結果は，次の二点の場合に誤ったものとなりうる．

- ロードストア順序違反： ロードを投機実行した時点では，ロードが依存しているストアがまだ実行されていなかった場合
- レジスタ読み出し順序違反： ロードを投機実行した時点では，ロードのソースレジスタの値を生成する命令

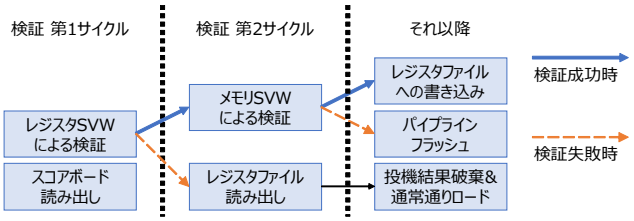


**図 9** 投機検証の流れ

[*2] 命令キューはリングバッファで実装され，有効な命令が入っている範囲は 2 つのポインタ head と tail で管理される．命令キュー中央の位置は head から一定の距離にあるため，バンク化すれば命令キューの多ポート化は不要である．

[*3] 投機ロードへのフォワーディングは行わない．投機バッファは論理レジスタではないため，レジスタ SVW を用いたレジスタ依存違反検出ができないためである．そもそも投機命令キューの命令はスコアボードを読みださないから，これは自然と実現される．

がまだ実行されていなかった場合

提案手法では，前者の検出をメモリ SVW で，後者をレジスタ SVW によって検出する．メモリ SVW は，Bloom-like SVW を用いる．レジスタ SVW は，SVW を用いる[*4]．

ロードストア順序違反の検出は，Bloom-like SVW を 3.2.2 節で説明した通りに動作させればよい．一方，レジスタ読み出し順序違反の検出は，SVW を動作させることにより行うが，一部の動作を変更する必要がある．これは，ストアは必ず InO に行われるためメモリが ready になるタイミングは InO であるのに対し，レジスタが ready となるタイミングは OoO だからである．以下では，その動作の詳細を示す．

まず，すべてのストア命令に一意な番号 ssn を振るのではなく，すべてのデスティネーションレジスタを持つ命令（以下，単に命令と呼ぶ）に一意な番号 isn (instruction sequence number) を振る．isn は ssn と同様，プログラム中の順番が早いほど，小さい番号が振られるようにする．RAM で作られたテーブル $svw$ やハッシュ関数 $hash$ を用意するのは SVW から変更がない．これに加え，レジスタを ready にした命令の持つ isn の最大値を保持する変数 ISN_retire を用意し，0 で初期化する．これら準備を行ったうえで，命令を実行するときには以下の工程を行う．

- 命令を実行したとき，$svw[hash(\text{デスティネーションレジスタ番号})]$ に，その命令の isn を書き込む．デスティネーションレジスタが複数ある場合は，それぞれのデスティネーションレジスタに対してこれを行う．また，この命令の isn が ISN_retire よりも大きい場合に限り，ISN_retire をこの命令の isn で更新する．
- ロード命令を投機実行するとき，レジスタ読み出しやフォワーディング受け取りのタイミングで ISN_retire を読み出し，記録しておく．
- 投機実行を検証するとき，ロード命令のソースレジスタそれぞれに対して，$svw[hash(\text{ソースレジスタ番号})]$ に書かれた isn を読む．そのどちらかまたは両方がロード命令を投機実行するときに記録した ISN_retire の値より大きければ，レジスタ読み出し順序違反が発生したと判定する．

### 4.8 正確な例外の保証

提案マイクロアーキテクチャは，次のようにして正確な例外を保証する．まず，投機発行した命令（説明の便宜上 A とする）が例外を起こした場合には，その例外コードや例外処理に必要な情報を投機ロードバッファへ書き込んでおく．そして，A の投機発行を SVW で検証して投機ミスが検出されなかった場合には，A の例外処理へ移る．この際に A に割り当てられた投機ロードバッファを参照して，例外コードや復帰に必要な情報を得る．

[*4] レジスタ読み出し順序違反の検出に Bloom-like SVW を用いなかったのは，レジスタ番号に対する適切な複数のハッシュ関数を現時点では発見できていないためである．

## 5. 評価

サイクル精度シミュレータ Sniper [9] に提案手法を実装し評価した．Sniper はトレースベースなシミュレータであり，その入力トレースは Intel Pin [15] で取得した x86-64 のものを用いた．シミュレーションに用いたパラメータを表 1 に示す．この表に示すベースラインとして用いた InO プロセッサは，ARM 社の最新の InO コアである Cortex A510 [3] に基づいている．

ベンチマークには SPEC CPU 2017 Speed [10] を用い，入力データセットには ref を用いた．ベンチマークのコンパイルは Intel Core i7-10700 上で gcc 9.4.0 を用いて行った．コンパイルオプションには “-O3 -march=native -fno-unsafe-math-optimizations -fno-tree-loop-vectorize -fno-strict-aliasing” を用いた．ベンチマークの先頭 $10^{11}$ (100G) 命令をスキップし，$10^8$ (100M) 命令をシミュレーションして評価した．これは Sniper シミュレータが実機より数桁遅いため，実機での動作を前提としたベンチマークを全実行するには途方もない時間がかかるためである．

提案手法においてメモリ順序違反を検出する Bloom-like SVW のハッシュ関数の数は 2 とした．これは，ハッシュ関数の数を増やしていったとき 1 → 2 では大幅に偽陽性率が下がるが，2 より大きくしていっても偽陽性率にあまり変化がないことが報告されているためである [14]．

ベースラインとなる InO プロセッサによるモデルに加え，提案手法を実装した以下のモデルを評価した：

(1) **tail+mid**：提案手法．ディスパッチ時 (tail) とキューの中央 (mid) の双方で命令が投機発行キューに挿入されうるモデル．

(2) **tail**：ディスパッチ時 (tail) のみで命令が投機発行キューに挿入されうるモデル．

**表 1** プロセッサのパラメータ

| | InO | 提案手法 |
|---|---|---|
| 発行幅 | 3 | |
| フロントエンドレイテンシ | 4 | |
| 発行レイテンシ | 1 | |
| 演算器数 | Int 1, Int/fADD 1, Int/fMUL/fDIV 1<br>iMUL/iDIV 1, Load 1, Load/Store 1 | |
| 分岐予測器 | Two-level local branch predictor | |
| L1 キャッシュ | 命令 64 KiB, データ 64 KiB<br>8-way, 64 B line, 4 cycles | |
| L2 キャッシュ | 512 KiB, 16-way, 64 B line, 8 cycles | |
| プリフェッチャ | Stride prefetcher | |
| 命令キュー | 36 エントリ | |
| 投機命令キュー | - | 18 エントリ |
| 投機バッファ | - | 18 エントリ |
| SVW | - | レジスタ用 16 エントリ<br>メモリ用 16 エントリ× 2 |
| 発行可能性予測器 | - | 32 エントリ, 2-way, 60-bit tag |

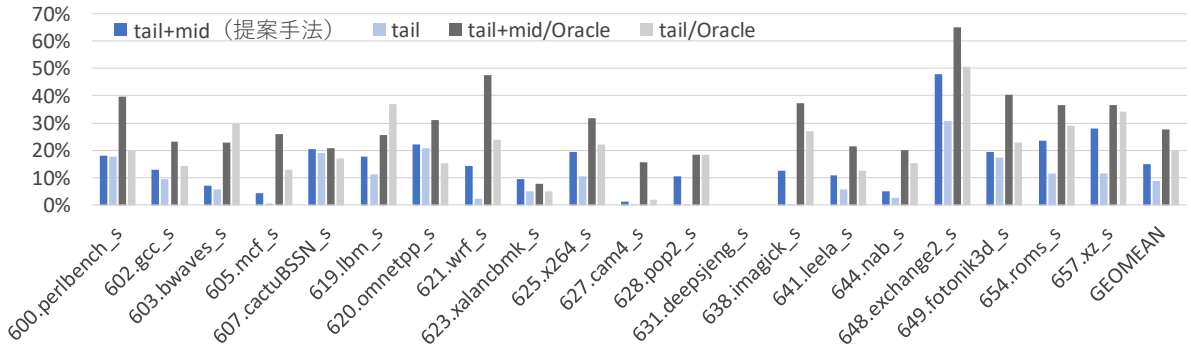


**図 10** ベースラインの InO プロセッサに対する性能向上

(3) **tail+mid/Oracle**：tail+mid に対し，発行可能性予測と投機ミスの検証が理想的に行われた場合のモデル．このモデルでは，投機ミスの検証は理想的に行われ，偽陽性が生じない．

(4) **tail/Oracle**：tail に対し，予測と検証が理想的に行われた場合のモデル．

図 10 に，ベースラインとなる InO プロセッサからの提案手法の各モデルによる性能向上を示す．tail+mid は幾何平均でベースラインと比較して 15% の性能向上を示した．一方，ディスパッチ時発行のみを行う tail による性能向上は 9% であり，キューの中央で発行を行うことにより大きな性能向上が実現できることがわかる．また，理想的な予測と検証を行う tail+mid/Oracle と tail/Oracle による性能向上は 28% と 20% の性能向上が得られた．これらの結果より，予測と検証アルゴリズムの改善により，大きな性能向上の余地があることがわかる．

## 6. おわりに

InO プロセッサは消費電力や回路面積が重視される環境で広く用いられているものの，その一方で性能の低さが課題となる．我々は InO プロセッサにおけるロード命令の投機的な先行発行を提案した．提案手法では発行可能性を予測してロード命令をあらかじめ投機的に発行しておくことにより，そのコンシューマ命令のストールを効果的に取り除く．これにより，提案手法は InO プロセッサの単純さを維持しつつ，その性能を改善することができる．評価の結果，提案手法は SPEC CPU 2017 [10] Speed ベンチマークにおいて，既存の InO プロセッサと比べて 15% 性能が向上した．また，理想的な予測器や検証機構を用いた場合には 28% 性能が向上することが確かめられ，予測器などの改良によるさらなる性能向上の余地が示された．今後は提案手法の回路面積や消費電力などを含むより詳細な評価や，予測/検証アルゴリズムの改良を行う予定である．

## 参考文献

[1] De Kruijf, M. and Sankaralingam, K.: Idempotent processor architecture, *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 140–151 (2011).

[2] Greenhalgh, P.: "Big. Little processing with arm cortex-a15 & cortex-a7", ARM White paper (2011).

[3] ARM: Cortex A510 Processor, https://www.arm.com/ja/products/silicon-ip-cpu/cortex-a/cortex-a510.

[4] Andes Technology: Andes 45-Series Processor, http://www.andestech.com/en/products-solutions/andescore-processors/.

[5] CHIPS Alliance: VeeR EH1 Core, https://github.com/chipsalliance/Cores-VeeR-EH1.

[6] Wolff, S. R. and Barnes, R. D.: Revisiting Using the Results of Pre-Executed Instructions in Runahead Processors, *IEEE Computer Architecture Letters*, Vol. 13, No. 2, pp. 97–100 (2013).

[7] McFarling, S. and Hennesey, J.: Reducing the Cost of Branches, Vol. 14, No. 2 (1986).

[8] Chang, P. P., Chen, W. Y., Mahlke, S. A. and Hwu, W.-m. W.: Comparing static and dynamic code scheduling for multiple-instruction-issue processors, *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 25–33 (1991).

[9] Sniper Simulator, `https://github.com/snipersim/snipersim`.

[10] SPEC CPU(R) 2017, `https://www.spec.org/cpu2017/`.

[11] Jeong, I., Park, S., Lee, C. and Ro, W. W.: CASINO core microarchitecture: Generating out-of-order schedules using cascaded in-order scheduling windows, *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 383–396 (2020).

[12] Carlson, T. E., Heirman, W., Allam, O., Kaxiras, S. and Eeckhout, L.: The load slice core microarchitecture, *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 272–284 (2015).

[13] Roth, A.: Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization, *32nd International Symposium on Computer Architecture (ISCA'05)*, IEEE, pp. 458–468 (2005).

[14] 西川 卓, 塩谷 亮太, 入江 英嗣:Bloom-like SVW の評価: 電子情報通信学会技術研究報告: Vol. 115, No. 243, pp. 27–34 (2015).

[15] Intel Pin, `https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html`.