修士論文

ツインテール・アーキテクチャにおける ハーフパンプFUアレイ

Half-pumped FU array of Twin-tail Architecture

指導教員　五島 正裕 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

学籍番号 66449

亘理 靖展

概要

本論文は，発行幅を増加させることなく演算器を追加し，同時実行可能な命令を増加させるツインテール・アーキテクチャの改良案を提案する．まず，ツインテール・アーキテクチャにおける命令の再実行方法を提案する．また，消費電力の低下を実現するための機構，ハーフパンプFUアレイを提案し，IPCの評価を行う．ハーフパンプFUアレイは，ツインテール・アーキテクチャの命令のスループットを保ちながらも，消費電力を抑えることを可能にする．シミュレーションによる評価では，ハーフパンプFUアレイを実装したツインテール・アーキテクチャは通常のツインテール・アーキテクチャと比べ，平均で2.4%程度の性能低下にとどまり，ベースモデルのスーパスカラ・プロセッサに対して，平均で10.7%の性能向上が得られた．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

スーパスカラ・プロセッサは命令をアウト・オブ・オーダに実行することで，同時実行可能な命令数を増やし，命令の並列性を抽出している．このとき，依存関係にある命令を全て back-to-back に実行することができれば，プログラムの実行時間は最短となる．しかし実際には，そのようなことは不可能であり，依存先の命令が何サイクルにもわたってスケジューリングされないことがある．

従来のプロセッサが依存関係にある命令を back-to-back に処理できない理由としては,「演算器・スケジューリング能力の不足」「フェッチ幅の不足」「分岐予測ミス」などが考えられる．

しかし，単純にバックエンドに演算器を追加したり，スケジューリング可能な命令数を増やすことによってこの問題を解消することは難しい．スーパスカラ・プロセッサに演算器を追加して同時実行可能な命令数を増やすためには発行幅を増やす必要がある．しかし研究によると，命令ウィンドウの回路規模は発行幅の 3 乗に比例して増大してしまう [4]．面積が増えるとその平方根に比例して配線長が長くなるので，発行幅の増加は配線長の増大につながる．

微細化されたプロセスで製造された LSI ではゲート遅延より配線遅延が支配的である．したがって，スーパスカラ・プロセッサの発行幅を増やすことは，命令ウィンドウ内の遅延が増大することにつながり，今後クロック速度の向上の足枷となる可能性がある．

そこで我々は，発行幅を増加させることなく演算器を追加し，同時実行可能な命令を増加させるツインテール・アーキテクチャを提案した [7][8]．

ツインテール・アーキテクチャは，リザベーション・ステーション/リオーダ・バッファ系のスーパスカラ・プロセッサをベースとしており，リザベーション・ステーションと並列に，専用の演算器群からなる実行系を追加する．詳細は次章以降で述べるが，追加された演算器群ではリザベーション・ステーションへディスパッチを行わずに命令を実行できるため，演算器を追加しても発行幅を増やす必要はない．また，ディスパッチをせずに命令が実行できるため，リザベーション・ステーション以降の実行系に比べ，早期に命令を実行することができる．さらに，追加された実行系で実行される命令をリザベーション・ステーションへディスパッチしないため，リザベーション・ステーション以降の実行系の資源を大幅に節約することができる．このような様々な性質により，

スーパスカラ・プロセッサの性能を向上させる．

1.2 目的

ツインテール・アーキテクチャでは，リザベーション・ステーションへ選択的にディスパッチを行うため，命令がミスしたときに通常のスーパスカラ・プロセッサと同様の方法では命令の再実行ができない場合がある．

そこで，本論文では，まず，ツインテール・アーキテクチャにおける命令の再実行方法を提案する．次に，性能を保ちながら，ツインテール・アーキテクチャで追加する演算器群の消費電力を低減させる，ハーフパンプ FU アレイという手法を提案し，IPC の評価をする．

1.3 構成

本論文の構成は以下の通りである．2 章でツインテール・アーキテクチャの構成を説明する．3 章ではツインテール・アーキテクチャによって性能向上が得られる原理について説明する．4 章で提案手法について説明する．5 章にはシミュレーションによる評価結果を掲げる．6 章では関連研究を挙げる．7 章で本研究の結論と今後の課題を述べる．

第2章　ツインテール・アーキテクチャの構成

本章ではツインテール・アーキテクチャの構成を説明する．

2.1　ツインテール・アーキテクチャ

ツインテール・アーキテクチャはリザベーション・ステーション/リオーダ・バッファ系のスーパスカラ・プロセッサをベースモデルとする．リザベーション・ステーション/リオーダ・バッファ系のスーパスカラ・プロセッサでは，フロントエンドでレジスタ読み出しを行い，レジスタ読み出しが終わった命令はリザベーション・ステーションへディスパッチされ，バックエンドで実行される．したがって，全ての命令はリザベーション・ステーションへディスパッチされたのちに，スケジューリング，発行のステージを経て，演算器で実行されることになる．

ツインテール・アーキテクチャでは，レジスタ読み出し直後に全てのソース・オペランドが揃って実行可能になっている命令に着目し，そのような命令を専用の演算器を追加して早期に実行を行う．

図 2.1 にツインテール・アーキテクチャのブロック図を示す．レジスタ読み出しの直下には，リザベーション・ステーションと並列に演算器群を追加し，リザベーション・ステーションへディスパッチを行う前にソース・オペランドが揃った命令を実行する．追加された演算器群からなる実行系を **In-Order tail**，リザベーション・ステーションとその下流の演算器による実行系を **Out-of-Order tail** と呼ぶ．以下，本稿では，In-Order/Out-of-Order tail をそれぞれ **IO tail**/**OoO tail** と略す．

図 2.1 に 2 ウェイのツインテール・アーキテクチャのブロック図を示す．レジスタ・ファイルの直下にある 4 つの演算器からなる実行系が IO tail であり，リザベーション・ステーションとそこから発行された命令を実行する 2 つの演算器からなる実行系が OoO tail である．

レジスタ読み出し後，IO tail での実行と OoO tail へのディスパッチは並列して行われる．IO tail には，ディスパッチ，スケジューリング，および発行のステージがないため，実行に至るまでのレイテンシが OoO tail に比べて大幅

に削減される．したがって，OoO tail に比べ，命令を早期に実行できる可能性がある．

OoO tail は通常のスーパスカラ・プロセッサと同様の構成をとるため，特に解説すべき点はない．本章では以降，IO tail の構成について詳しく述べる．

2.2 In-Order tail

2.2.1 演算器の配置

IO tail にはフェッチ幅と同数の演算器を多段に配置する．以降，多段に並べられた演算器群を FU **アレイ**(Function Unit Array) と呼ぶ．図 2.2 にフェッチ幅 2 × 2 段の IO tail のブロック図を示す．

スーパスカラ・プロセッサにおいて同時にフェッチされた命令群は**フェッチ・グループ**と呼ばれる．IO tail にはレジスタ読み出し後にフェッチ・グループ単位で命令が取り込まれる．IO tail は命令ウィンドウのような命令を溜めておく記憶領域を持たず，フェッチ・グループは毎サイクル，次段の演算器群に送られる．各フェッチ・グループの命令は一定のサイクルで IO tail の最終段に到達し，IO tail を抜ける．実行された命令は最終段を抜けた後，リタイア可能な状態になり，リオーダ・バッファでリタイアを待つ．実行されなかった命令も IO tail を通り抜けていき，最終段を抜けた時点で IO tail から破棄される．

2.2.2 In-Order tail における命令実行

IO tail に到着した時点でソース・オペランドが揃っている命令は 1 段目の演算器で実行できる．演算器を多段に配置しているのは，実行できる命令の数を増やすためである．IO tail では，次のようなバイパスを行うことで，IO tail で実行可能な命令を増やすことができる．

- 実行結果を次の段に送る

  同一フェッチ・グループに属する命令は，同時に次の段に移動する．したがって，実行された命令は実行結果を次の段に送ることで，同じフェッチ・グループ内のコンシューマ命令に対し，次のサイクルに次段で結果を受け渡すことができる．このようにすることで，フェッチ・グループ内の依存関係を持つ命令があった場合に，依存先の命令の実行ステージを 1 つ後段にずらし，実行していくことができるようになる．

- 実行結果を自分自身および前段に送る

  後段の演算器で実行された結果を自分自身と前段の演算器に送る．これにより，フェッチ・グループ間をまたがる依存関係を持つ命令は，先行す

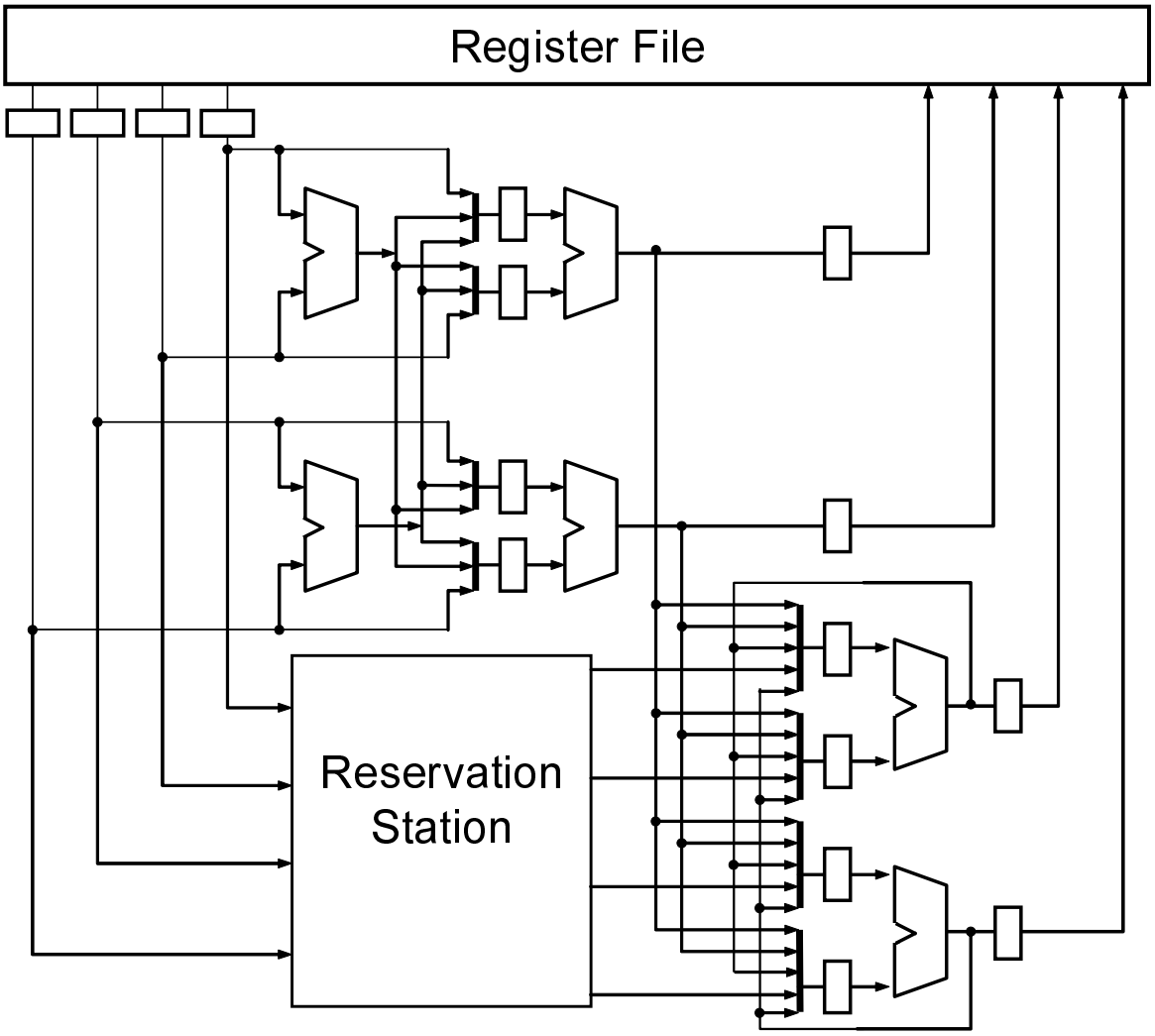


図 2.1: ツインテール・アーキテクチャのブロック図

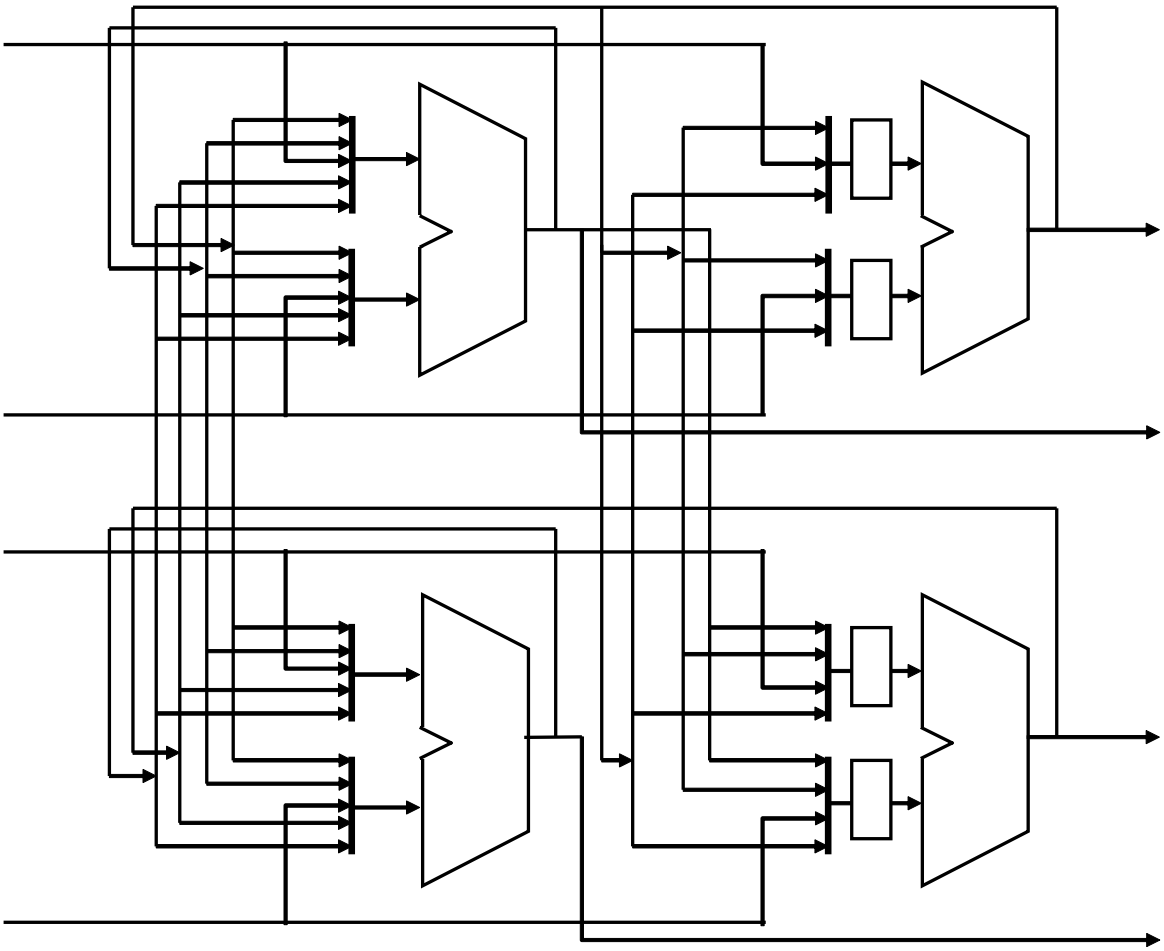


図 2.2: In-Order tail のブロック図

るプロデューサ命令の実行結果を受け取ることができるようになる．この場合はフェッチ・グループ内の依存関係がある場合と異なり，依存先の命令の実行ステージは後段にずれずに，依存元の命令と同じ段か，それより前の段で実行が可能になる．

結果を受け取った命令はソース・オペランドが揃った段で実行できる．このようにIO tailの演算器間にバイパスのパスを設けることで，IO tailで実行される命令を増やすことができる．

なお，命令がIO tailを抜けるまでにソース・オペランドが揃わなければ，その命令はIO tail内では実行されない．

In-Order tailで実行する命令

IO tailでは，次の3種類の命令を実行することを想定している．

- 整数演算命令
- 分岐命令
- ロード命令

Out-of-Order tailへの実行結果のバイパス

IO tailで実行された命令の結果はOoO tailにバイパスする．IO tailに多段に配置された演算器の全てからOoO tailへのバイパスを行うと，OoO tailのバイパス・ネットワークが複雑になる．したがって，バイパス・ネットワークの複雑化を避けるため，OoO tailへのバイパスは，命令がIO tailを抜ける時に最終段の演算器から行う．また，OoO tailからIO tailへはオペランドのバイパスを行わない．

分岐予測ミスの回復

IO tailで実行された分岐命令の結果，分岐予測ミスが発覚した場合，その結果を用いて分岐予測ミスの回復を行う．また，回復と同時に分岐予測器のアップデートも行う．

ロード命令の実行結果のバイパス

IO tailでアドレス計算を行ったロード命令は，次段の演算器から1次キャッシュへのアクセスを開始する．キャッシュはIO tailとOoO tail両方からの要求を処理するため，調停回路を必要とする．IO tailで実行したロード命令の結果はIO tailではなく，OoO tailへ直接送る．この結果はOoO tailからキャッシュ・アクセスを行った場合と同じパスを使ってバイパス，書き込みを行う．また，キャッシュへのリード・ポートはIO tailの各段に対し，1つずつを想定している．

なお，ISA で分離ロード/ストアが規定されている場合には，IO tail の 1 段目でアドレス計算を行い，2 段目以降からキャッシュ・アクセスを開始する．この場合には，IO tail の 1 段目で実行可能な，ロード命令は存在しないので，2 段目以降にキャッシュへのリード・ポートを設置すればよい．

In-Order tail で実行できない命令

IO tail 内で実行できない命令は以下に分類される．

- IO tail 内に演算器がない命令

  IO tail 内に演算器を設けない命令は実行しない．
  演算器が大きいもの，命令の実行に複数サイクルかかるものがこれにあたる．Float 命令や，ストア命令，掛け算，除算などが該当する．

- ソース・オペランドが揃わない命令

  - フェッチ・グループ内の依存が複数ある命令

    2.2.2 節で述べたように，フェッチ・グループ内に依存がある場合，依存先の命令の実行は依存元の命令より 1 サイクル遅れ，1 段後段で実行されることになる．たとえば，3 段の IO tail の場合には，フェッチ・グループ内で依存している命令が 2 つまでならば，依存先の命令は 3 段目で実行できる．しかし，フェッチ・グループ内で 3 つ目の依存がある場合には，一番最後の依存先の命令は IO tail の最終段までにソース・オペランドが揃わず，IO tail 内ではその命令は実行不可能になる．

  - ロード命令に依存する命令

    先ほど述べたように，本論文では，ロード命令の実行結果は OoO tail へ直接届き，IO tail には届かないとしている．したがって，ロード命令に依存する命令が IO tail でロード命令の実行結果を得られるのは，ロード命令の実行結果のレジスタ書き込みが終わった後である．1 次キャッシュのリード・レイテンシが 3 サイクル，レジスタ書き込み/読み出しがそれぞれ 2 サイクルかかるとすると，1 次キャッシュにヒットした場合でも，ロード命令の実行から 7 サイクル間は，ロード命令の実行結果を IO tail で使うことができない．ロード命令の実行はアドレス計算が終わった後に開始されるため，早くて IO tail の 2 段目が実行開始される．したがって，この場合，最低でもロード命令の後ろ 8 フェッチ・グループにあるロード命令に依存する命令は IO tail で実行されない．

- アドレス一致/不一致予測器によって，一致と予測されたロード命令

アドレス一致/不一致予測器によって，一致と予測されたロード命令は，先行のストア命令の実行が終了してからでないと，メモリ・アクセスを開始しない．ストア命令はIO tailで実行しないため，実行はOoO tailで行われる．OoO tailでの実行レイテンシはIO tailでの実行レイテンシよりも長く，ロード命令より先行のストア命令の実行が先に終わることは少ない．したがって，先行のストア命令とアドレスが一致すると予測されたロード命令はIO tailで実行できない可能性が高い．

2.3 命令のステアリング

IO tailで実行可能と予測される命令はOoO tailへディスパッチする必要はない．実際にそうすることによって，IO tailで実行される命令の分，OoO tailの資源を節約できる．

一方，IO tailでは，命令は実行可能/不可能に関わらず一定のサイクルで同じ演算器列を通り抜ける．IO tail内では，実行に必要な演算器がそれぞれの命令の実行可能/不可能に関わらず，1つずつ割り当てられていることなる．つまり，実行不可能な命令をIO tailへ投入することを止めても，実行可能な命令の使えるハードウェア資源は増加しない．

したがって，IO tailには実行不可能な命令も含め，全ての命令を投入してよいが，OoO tailにはIO tailで実行できると予測される命令はディスパッチする必要がない．IO tailで命令が実行できるかどうかをディスパッチ前に予測することで，命令をOoO tailへ選択的にディスパッチする．

In-Order tailでの実行可能性の予測

IO tailでの実行可能性を予測するために，レジスタ・リネームと並行して以下の動作を行う．

まず，レジスタ・リネーム時にソース・オペランドが揃っている命令は，そのデスティネーション・レジスタ番号をテーブルに書き込む．そして，後続のフェッチ・グループの命令はそのテーブルを参照することで，自分のソース・オペランドがIO tail内で供給されるか判別する．また，同一フェッチ・グループ内の依存関係を調べるため，先行する命令のデスティネーション・レジスタ番号と後続の命令のソース・レジスタ番号の比較を行う．

このIO tailでの実行可能性の予測には，ソース・オペランドのレディネスの情報が必要になるが，エントリの中身までは知らなくてもよい．レジスタのエントリの中身と，レディネスの情報を格納する領域は分離できる．その場合，レディネスを調べるための遅延は，エントリを読み出すための遅延よりずっと小さくすることが可能である．したがって，IO tailにおける命令の実行可能性の予測はレジスタ読み出しと並行して行える．これにより，ディスパッチ前

に前もって，IO tail での実行可能性を判断することができ，OoO tail への選択的なディスパッチが可能になる．

In-Order tail での実行可能性の予測ミス

レイテンシ予測ミスや，アドレス一致/不一致予測ミスによって，IO tail で実行できると予測された命令が実行されない場合がある．IO tail で実行できると予測され，OoO tail へディスパッチされなかった命令は，IO tail が唯一の実行ステージとなる．したがって，IO tail での実行予測がミスして，IO tail 内でソース・オペランドが揃わなければ，実行の機会を失い，その命令を実行できなくなる．そこで，命令が IO tail を抜ける時に実行可能性の予測ミスを検出し，予測ミスした命令は再実行しなければならない．4.1 節ではこの予測ミス時の再実行方法を提案する．

次章では，ツインテール・アーキテクチャの効果について説明する．

第3章　ツインテール・アーキテクチャの効果

ツインテール・アーキテクチャでは主に以下の 2 つの効果により，スーパスカラ・プロセッサのスループットを向上させる．

- 命令が早期実行される効果
- IO tail で実行される命令分，OoO tail の資源が節約される効果

本章ではこの 2 つの効果について説明する．

3.1　命令の早期実行による効果

IO tail はスケジューリング・ロジックを持たないため，OoO tail よりも命令の実行までのレイテンシが短くなる．図 3.1 の上にツインテール・アーキテクチャのパイプライン，下はスーパスカラ・プロセッサのパイプラインをそれぞれ示す．IF はフェッチ，RF はレジスタ読み出し，disp はディスパッチ，sched はスケジューリング，issue は発行，exec は実行，WB は書き戻しの，それぞれのパイプライン・ステージを表している．

図 3.1 に示したのは命令 i1 の結果を命令 i2 が使用するようなプログラムの実行の様子である．今，命令 i1 が IO tail で，命令 i2 が OoO tail で実行される状況を考える．命令 i1 が IO tail で実行され早期に結果が得られたことで，命令 i2 は命令ウィンドウに入り次第，発行される．一方，通常のスーパスカラ・プロセッサでは，命令 i1 の実行終了と共にジャスト・イン・タイム でその結果を利用できるように，命令 i2 は 1 サイクル遅れてウェイクアップされる．

このように通常のスーパスカラ・プロセッサならば命令を即座に発行できない状況でも，ツインテール・アーキテクチャでは命令を発行できる可能性がある．ここに示した例では，ツインテール・アーキテクチャは命令列全体の実行を 1 サイクル短縮している．

また，特にロード命令と分岐命令は早期実行の効果が大きいため，以下にロード命令と分岐命令が早期実行された時の効果を示す．

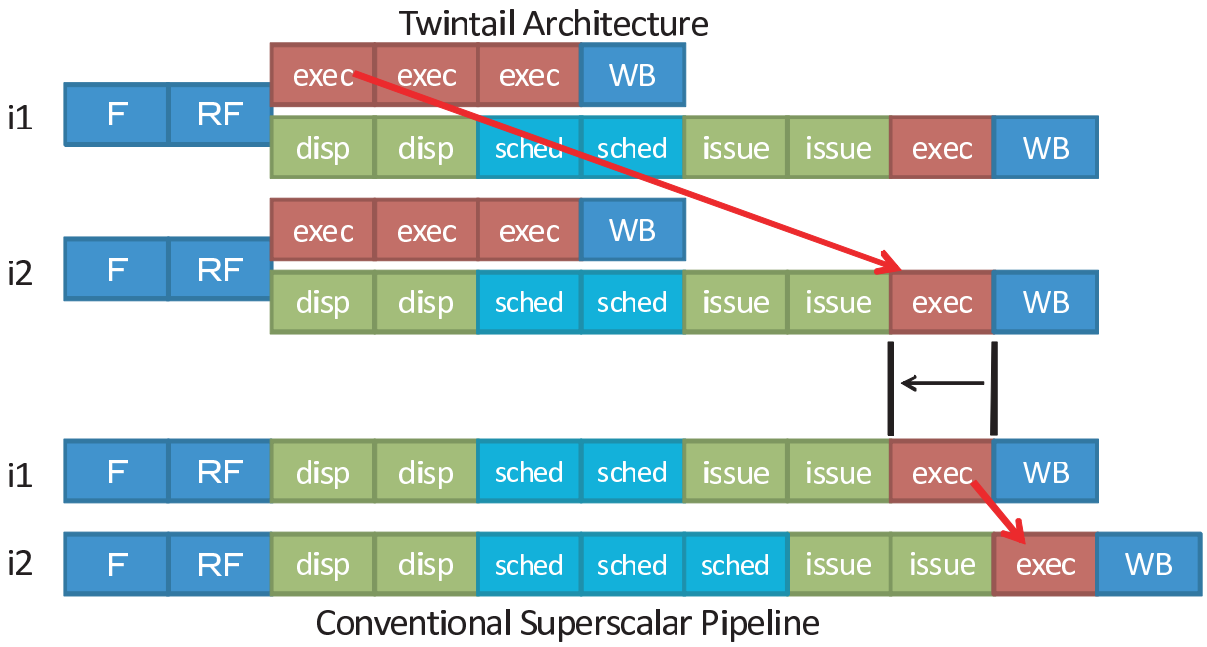


図 3.1: 早期実行のパイプライン・チャート

3.1.1 ロード命令が早期実行される効果

キャッシュ・アクセスのレイテンシは長いため，IO tail の実行タイミングで開始されると，レイテンシを隠蔽できる効果が高い．

図 3.2 に IO tail 内でロード命令が実行できた場合のパイプライン・チャートを示す．図 3.2 に示した例では，IO tail でキャッシュ・アクセスを開始できたことにより，1 次キャッシュへのアクセス・レイテンシが隠ぺいされ，OoO tail の依存先の命令は，即座にロード命令の実行結果を利用することができている．ここに示した例では，ツインテール・アーキテクチャは命令列全体の実行を，1 次キャッシュのアクセス・レイテンシの 3 サイクル分短縮している．

3.1.2 分岐命令が早期実行される効果

IO tail で分岐命令の結果が得られると，分岐予測ミスの発覚が早くなるため，分岐予測ミス・ペナルティが軽減される．

以上のようにツインテール・アーキテクチャでは早期実行によって，スーパスカラ・プロセッサが引き出せていない命令の並列性を抽出して性能向上を得ているといえる．以下では，スーパスカラ・プロセッサが抽出できていない並列性についての説明を行う．

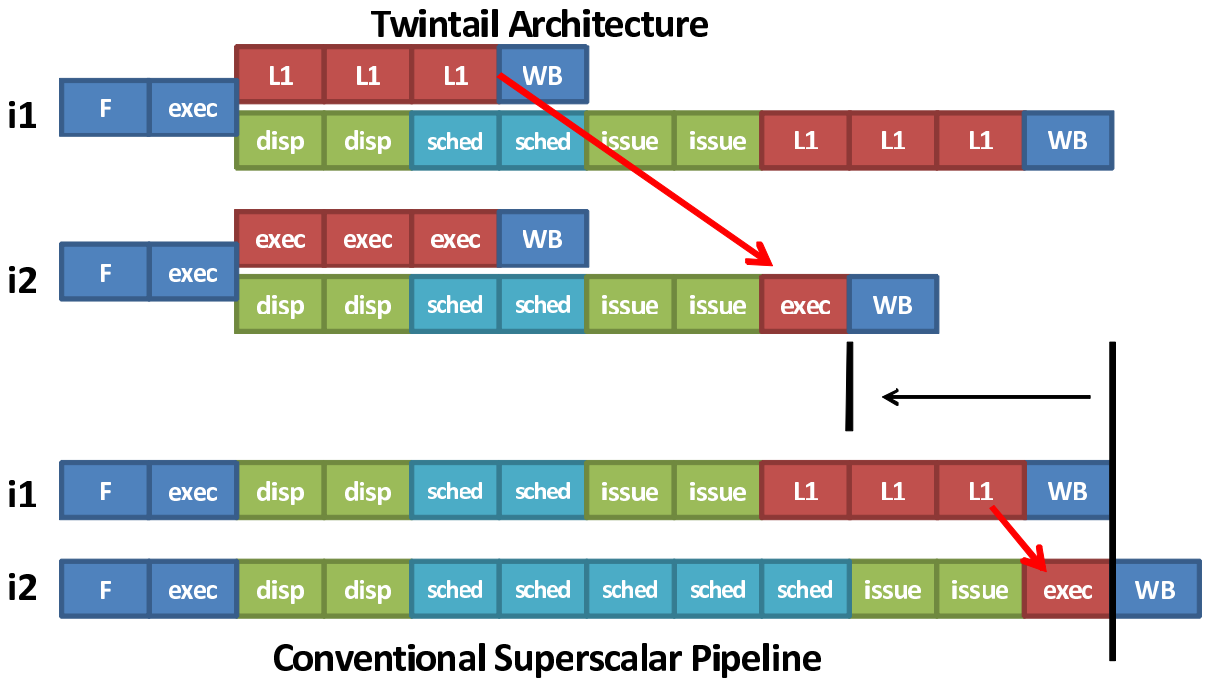


図 3.2: ロード命令の早期実行の効果

スーパスカラ・プロセッサの抽出する命令の並列性　依存関係を持つ命令同士を矢印で結んでいったグラフをデータフロー・グラフと呼ぶ．データフロー・グラフ上の全ての依存関係にある命令をこのグラフの通りに back-to-back に実行することができれば，プログラムの実行時間は最短となる．図 3.3 にデータフロー・グラフ上の命令を back-to-back に実行できた場合を示す．縦軸 t は時間を表しており，この図は命令 A〜E および 1〜5 がどのタイミングで実行されたかを表している．

現実に存在するプロセッサには様々な制約があるため，当然のことながらこれを常に実現することは不可能である．そのため，実際には依存元の命令が実行されてから依存先の命令が実行されるまでに何サイクルもかかってしまう場合が多い．図 3.3 に back-to-back に命令が実行できず，プログラムの実行時間が伸びる場合のデータフロー・グラフを示す．

従来のプロセッサが依存関係にある命令を back-to-back に処理できない理由として,「演算器・スケジューリング能力の不足」「フェッチ幅の不足」「分岐予測ミス」などが考えられる．

演算器・スケジューリング能力の不足

図 3.4 に示すように，演算器・スケジューリング能力の不足から実行時間に遅れが出てしまう場合がある．図において従来のプロセッサでは本来は命令 3 を優先して実行しなければならないサイクルでスケジューリング 能力の不足

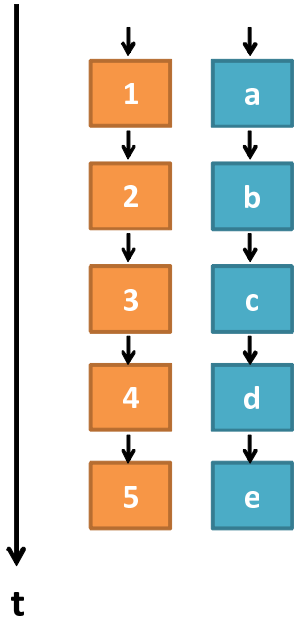


図 3.3: back-to-back に命令を実行できる時のデータフロー・グラフの例

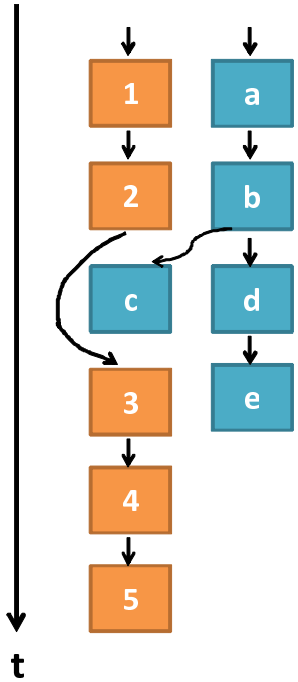


図 3.4: back-to-back に命令を実行できない時のデータフロー・グラフの例

からCとDを優先して実行してしまっている．それにより，このサイクルで3を実行した場合に比べ，実行時間が1サイクル遅れてしまうのである．

これに対して，ツインテール・アーキテクチャで図3.5のように3をIO tailで実行することができたとすると，OoO tailでは問題のサイクルの次のサイクルで4の実行を即座に開始することができる．これにより，実行時間の遅れが解消されていることが分かる．

フェッチ幅の不足

依存関係にある命令をback-to-backに実行できない他の理由としてフェッチ幅の不足が挙げられる．従来のプロセッサではフェッチ幅が十分でないために，プロセッサに内在する並列性を十分に引き出すことができていないのである．において仮に演算器がCとDに占有されていなかったとしてもそのサイクルに3がバックエンド実行ステージまで到達できなければ，2と3をback-to-backに実行することはできない．つまり，依存関係にある命令をback-to-backに実行するためには，依存元の命令がフェッチされた次のサイクルには依存先の命令がフェッチされていなければならないのである．依存先の命令のフェッチが遅れた場合，その分だけ命令の実行も遅れてしまう．

これに対して，ツインテール・アーキテクチャではフェッチからIO tailでの実行までのステージ数が短いため，依存先の命令をIO tailで実行することができれば，この遅れが軽減されるのである．

分岐予測ミス

分岐予測ミスも依存関係にある命令をback-to-backに実行できない理由の一つである．分岐予測ミスが起こるとそれ以降の命令はフラッシュされ，その後に正しい分岐先の命令のフェッチが開始される．このときの従来のプロセッサとツインテール・アーキテクチャでの振る舞いが図3.1のようになることがある．図のように分岐予測ミス後の命令がIO tailで実行できた場合には従来のプロセッサに比べて実行時間が短縮されることが分かる．

このように，ツインテール・アーキテクチャは従来のプロセッサではback-to-backに実行を行うことができていない場面において，依存先の命令をIO tailで実行してしまうことによってデータフロー・グラフ上からこの命令を取り除くことができる．データフロー・グラフ上でクリティカリティの高い命令が含まれるパス上から命令を取り除くことができれば，プログラムの実行時間は短縮される．

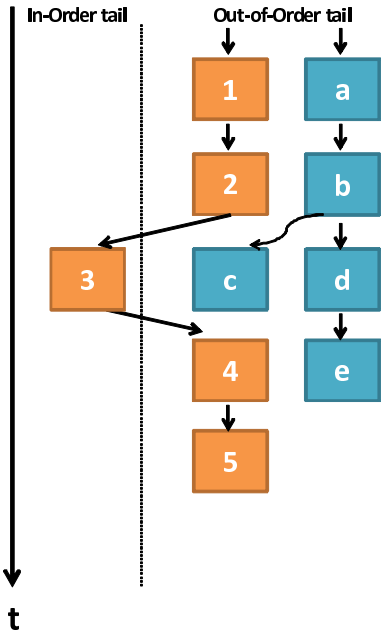


図 3.5: ツインテール・アーキテクチャのデータフロー・グラフ例

3.2 Out-of-Order tailの資源が節約される効果

2.3 節で述べたように，IO tail で実行可能と予測した命令は OoO tail にディスパッチしない．そのため，OoO tail の資源が節約できる．

OoO tail の資源の中でも，特に命令ウィンドウ・サイズは性能に大きな影響を与える．以下ではその原理について説明する．

命令ウィンドウ・サイズが性能に与える効果

命令ウィンドウ・サイズは，キャッシュ・ミスや L2 キャッシュへのアクセスが頻繁に発生するプログラムのパフォーマンスを大きく改善する．

ロード命令がキャッシュ・ミスを起こすと，そのロード命令がリタイアするまで，後続の命令がリタイアできないので，命令ウィンドウがいっぱいになった後，フロントエンドからの命令の供給が止まる．

図 3.6，図 3.7 にキャッシュ・ミスを起こすロード命令が近くに配置されたプログラムの実行の様子を示す．図 3.6 の上は，命令ウィンドウ・サイズに余裕がある場合で，図 3.7 は余裕がない場合である．

命令ウィンドウ・サイズに余裕がある場合には，1 つ目のロード命令のメモリ・アクセスと並列に，2 つ目のロード命令のメモリ・アクセスを開始できる．しかしながら，命令ウィンドウ・サイズに余裕がない場合，1 つ目のロード命令がリタイアした後に，後続のロード命令が命令ウィンドウに入り，メモリ・アクセスを開始する．つまり，メモリ・アクセスが並列に行えなくなる．

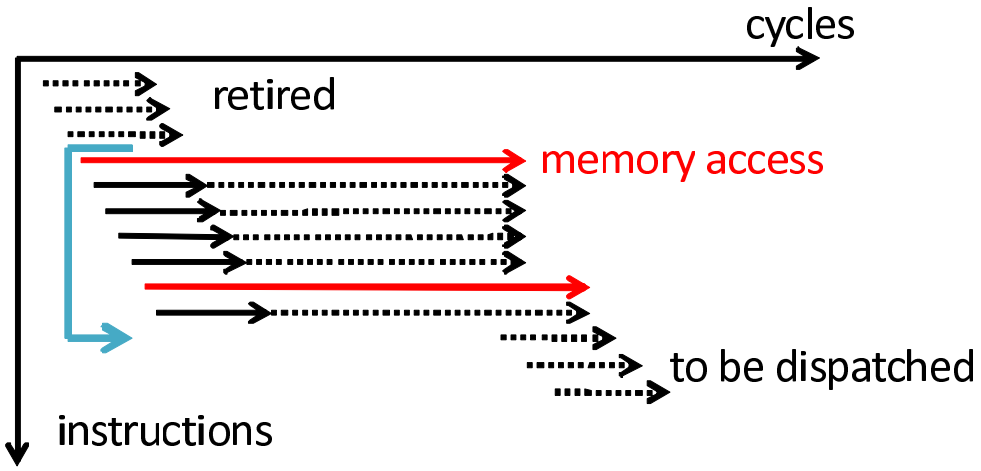


図 3.6: 命令ウィンドウ・サイズに余裕がある時のメモリ・アクセスの例

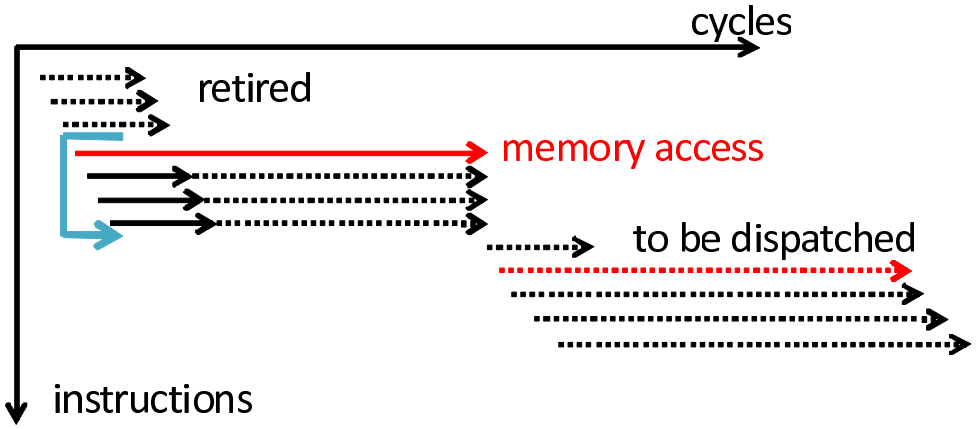


図 3.7: 命令ウィンドウ・サイズに余裕がない時のメモリ・アクセスの例

ツインテール・アーキテクチャではIO tailで実行できると予測した命令をOoO tailへディスパッチしないことで，命令ウィンドウ・サイズを増加させた場合と同様の効果が得られる．

第4章　提案手法

本章では，提案手法について説明する．まず，ツインテール・アーキテクチャにおける命令の再実行方法について述べ，次に，IO tail の消費電力を減らすための手法，ハーフパンプ FU アレイについて説明をする．

4.1　命令の再実行方法

IO tail で実行可能と予測した命令は OoO tail へディスパッチを行わない．しかし，IO tail 内で実行可能と予測して，依存元の命令がミスしたために再実行を行う必要がある場合や，予測をミスしソース・オペランドが揃わなかった命令は実行の機会を失うので再実行する必要があるが，リザベーション・ステーションから再発行して再実行することはできない．したがって，通常のスーパスカラ・プロセッサの再スケジュールとは異なった方法で命令の再実行をする．

本節では，まず一般的なスーパスカラ・プロセッサにおける再実行の方法について述べた後に，ツインテール・アーキテクチャにおける命令の再実行方法について述べる．

4.1.1　スーパスカラ・プロセッサにおける再実行方法

ある命令の実行結果が誤っていることが判明した場合，その結果に依存する後続命令をキャンセルし，再実行しなければならない．どの命令を再実行しなければならないかは，ミスの種類により異なる．

1. 制御投機ミス時

   実行し直さなければならないパスの命令はまだフェッチされていない．したがって，正しいパスの命令をフェッチして実行をやり直す必要がある．

2. データ投機ミス時

   実行し直す命令はすでにフェッチされている．したがって，再フェッチの必要はなく，命令を再スケジュールして再発行することで実行をやり直す．

通常のスーパスカラ・プロセッサでは命令ウィンドウや再スケジュール用のバッファに発行後の命令を溜めておくことにより，データ投機ミス時の再スケジュールが可能となる．

なお，プロセッサの構成によっては，データ投機ミスに対しても，命令を再スケジュールではなく，フェッチしてやり直す場合がある．この場合には，再スケジュールとは異なり，リタイアまで命令を命令ウィンドウや再スケジュール用のバッファに取っておく必要がなくなる．

4.1.2 ツインテール・アーキテクチャにおける再実行方法

ツインテール・アーキテクチャではIO tailで実行できると予測した命令はOoO tailへディスパッチしない．したがって，データ投機ミス時に再実行する命令の中に，IO tailで実行できると予測した命令がある場合，その命令を再発行することはできない．

さらに，IO tailで実行できると予測した命令は，IO tailが唯一の実行ステージとなる．しかし，データ投機ミスにより，IO tailで実行できると予測された命令が実行できなくなる場合がある．したがって，IO tailで実行できると予測した命令を実行できなかった場合にも，命令を再実行する必要がある．

そこで，ツインテール・アーキテクチャでは，次のようなポリシーによって再実行を行う．

1. 制御投機ミス時

   通常のスーパスカラ・プロセッサと同様，正しいパスの命令をフェッチしてやり直す．

2. データ投機ミス時

   ミスを起こした命令より後方の命令を全てキャンセルし，ミスした命令から再フェッチしてやり直す．

3. ステアリング・ミス時

   IO tail内で実行できると予測され，実行できなかった命令は再実行する必要がある．この場合にはデータ投機ミス時と同様に，実行されなかった命令より後方の命令をキャンセルし，ミスした命令から再フェッチする．

つまり，OoO tailにディスパッチしなかった命令を再実行する場合，ミスを起こした命令以降を再フェッチすることで再実行をする．再フェッチは再スケジュールに比べ，ミス・ペナルティが大きい．しかし，高性能なアドレス一致/不一致予測器やレイテンシ予測器を利用することで，ミスの頻度を小さくできるため，再フェッチによる性能低下の影響を低く抑えることができる．

また,「ミスを起こした命令以降を再フェッチすることで再実行する」と述べたが，フェッチしてやり直す場合，やり直す命令のフェッチ時にチェックポイントを設ける必要がある．

リザベーション・ステーション/リオーダ・バッファ系のスーパスカラ・プロセッサでは，物理レジスタ系のスーパスカラ・プロセッサに比べ，レジスタ・マッピングテーブルのチェックポイントを取る必要がないため，チェックポイントのハードウェア・コストは小さくなる．しかしながら，ツインテール・アーキテクチャでは全ての命令がステアリング・ミスを起こしうるため，その全ての命令について，チェックポイントを取るのは得策ではない．したがって，ステアリング・ミスをした命令があった場合には，その命令の最も近辺で，チェックポイントを取った命令までさかのぼり，その命令から再フェッチをすることにする．

5 章では，ツインテール・アーキテクチャのシミュレーションを行い，再実行方法の妥当性に関する評価をする．

4.1.3 再実行方法に伴う変更

スーパスカラ・プロセッサでは，命令はディスパッチ時にロード・ストアキューへイン・オーダに入る．ロード・ストアキューへ入った命令は，実行可能になったものからアウト・オブ・オーダに演算器へ発行され実行される．実行完了した命令から順にロード・ストアキューをチェックし，アドレス一致/不一致予測ミスの検出が行われる．

ツインテール・アーキテクチャでは，OoO tail にディスパッチしない命令に対しても，データ投機ミスを検出する必要がある．つまり，ロード・ストアキューに入れない命令のアドレス一致/不一致予測ミスを検出しなければならない．

そのため，ロード・ストアキューからアドレス一致/不一致予測ミスの検出バッファを分離し，OoO tail へディスパッチしない命令も，アドレス一致/不一致予測ミスの検出用のバッファには入れる．

アドレス一致/不一致予測ミスの検出バッファは複雑なスケジューリング・ロジックを持たないので容量を増やすことは容易である．複雑なスケジューリング・ロジックを持つロード・ストアキューには，IO tail で実行される命令を入れないことで容量を節約する．

このようにすることで，OoO tail の資源を節約するという利点を保ったまま，データ投機ミスの検出をすることが可能になる．

4.2 ハーフパンプFUアレイ

ツインテール・アーキテクチャでは，多くの命令をIO tailで実行することができ，実行した命令はOoO tailへディスパッチしないため，OoO tailの資源を節約できる．したがって，IO tailのコスト・パフォーマンスを上げれば，性能を保ったまま，OoO tailを縮小することができると考えられる．

そこで，我々はIO tailのFUアレイの消費電力を下げるための機構，ハーフパンプFUアレイを提案する．ハーフパンプFUアレイではIO tailをOoO tailの周波数の半分，ウェイ数を倍にして動作させる．電力消費は周波数の3乗に比例するので，IO tailのスループットを保ったまま消費電力を削減することができる．

4.2.1 ハーフパンプFUアレイの構成

ハーフパンプFUアレイでは，各段の演算器はウェイ数と同じだけ用意する．また，キャッシュのポートは通常のものと同じで，各段に1つとする．

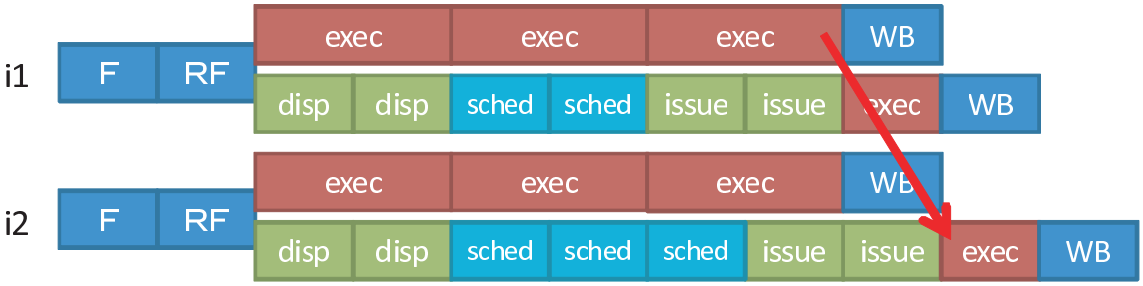


図 4.1: ハーフパンプFUアレイのパイプライン・チャート

3段のIO tailにハーフパンプFUアレイを用いた場合の動作を図4.1に示す．

各段の実行に2サイクルかかるので，IO tailを抜けるのに6サイクルかかる．命令は，連続した2サイクル分のフェッチ・グループを合わせ，2サイクルに一度IO tailに入れられる．また，OoO tailへは通常通り毎サイクル，各フェッチ・グループをディスパッチする．

4.2.2 ハーフパンプFUアレイのデメリット

ハーフパンプFUアレイを用いた場合，いくつかのデメリットが生じる．次にハーフパンプFUアレイを用いた場合のデメリットについてまとめる．

1. 早期実行が遅れる

(a) Out-of-Order tail へ結果のバイパスが遅れる

OoO tail へのバイパスは命令が IO tail を抜ける時に行われる．したがって，ハーフパンプ FU アレイにすると，命令が IO tail を抜けるまでにかかるサイクル数が倍になるため，IO tail の段数によっては，OoO tail へのバイパスが遅れることがある．図 4.1 では，3 段の IO tail であるため，OoO tail にバイパスが届くのが通常のツインテール・アーキテクチャに比べ，1 サイクル遅くなっている．

(b) In-order tail 内での実行が遅れる

ハーフパンプ FU アレイを用いると IO tail 内での実行レイテンシが伸びる．したがって，分岐命令やロード命令の実行が遅れるため，3.1 節で述べたような，分岐予測ミス・ペナルティが軽減される効果や，ロード命令のキャッシュ・アクセス・レイテンシの隠蔽による性能向上の効果が減ることになる．

2. IO tail 内で実行される命令数が減る

2.2.2 節で述べたように，同一フェッチ・グループ内に依存関係があると，コンシューマ命令の実行はプロデューサ命令の実行より 1 段遅れる．ハーフパンプ FU アレイではフェッチ・グループが統合され，同一フェッチ・グループ内の依存が増えるため，IO tail を抜けるまでに実行不可能な命令が増加する．

次章で，ハーフパンプ FU アレイがツインテール・アーキテクチャの性能に与える影響について評価を行う．

第5章　評価

5.1　評価方法

当研究室で開発したシミュレータ「鬼斬弐」に対して提案手法を実装し，評価を行った．実行するプログラムには，SPECint2000，SPECfp2000 のプログラムを用い [2]，最初の 1G 命令をスキップした後，100M 命令を実行した．入力には train を用いた．

以下のモデルについて評価をした：

- **BASEMODEL**

  表 5.1 に示す構成の，スーパスカラ・プロセッサのベース・モデルである．

- **TWINTAIL**

  フェッチ幅 4 × 3 段の IO tail を用いたツインテール・アーキテクチャのモデルである．

- **HALFPUMP**

  IO tail に 8 × 3 段のハーフパンプ FU アレイを用いたツインテール・アーキテクチャのモデルである．

なお，評価対象のツインテール・アーキテクチャでは IO tail 内のバイパスは 3 段目から 1 段目へのバイパスに 1 サイクルかかり，他は後続の命令が即座に利用可能とした．

5.2　IPC の評価

ベース・モデルおよび，4 × 3 段の IO tail と，8 × 3 段のハーフパンプ FU アレイを実装した IO tail を用いたツインテール・アーキテクチャの IPC を測定した．

図 5.1 に結果を示す．グラフは，ベース・モデルの IPC を 1 として正規化してある．

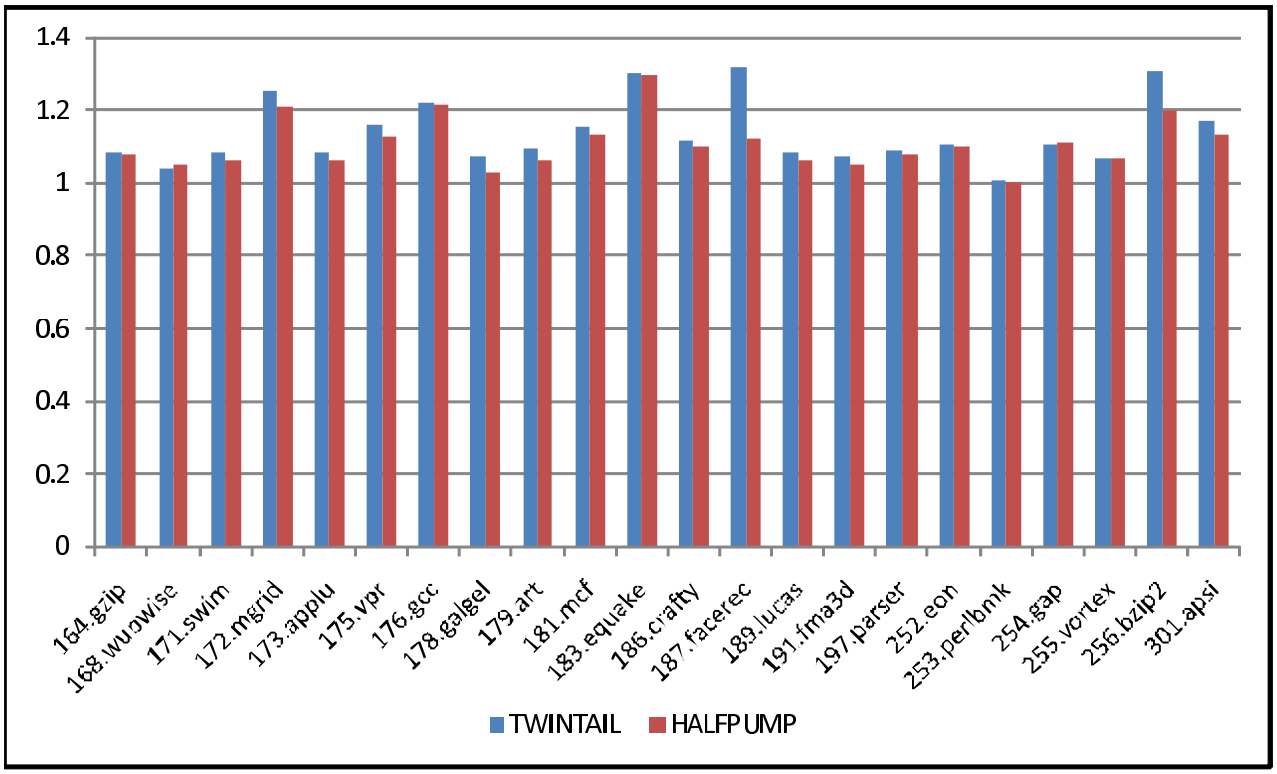


図 5.1: ツインテール・アーキテクチャの性能

5.2.1 ツインテール・アーキテクチャの性能向上

BASEMODEL に対して，TWINTAIL では最大で 31.7%，平均で 13.6%の性能向上が得られた．これは，4.1 節で述べたように，命令の再実行方法を再フェッチにしても，アドレス・一致/不一致予測やレイテンシ予測の精度が高いことによって，ミスの頻度自体が小さくなるため，ミス・ペナルティを十分小さくできることに起因する．

また，平均で 13.6%と大きな性能向上が得られた理由は，3.2 節で述べたように IO tail で多くの命令を実行でき，実行された分，OoO tail の資源に余裕ができたためである．

5.3 節では実際に IO tail で実行できた命令の割合を挙げる．

5.2.2 ハーフパンプ FU アレイの性能低下

HALFPUMP では，BASEMODEL に対し最大で 29.7%，平均で 10.7%の性能向上が得られた．また，TWINTAIL に対する性能低下は，平均で 2.4%にとどまった．

TWINTAIL に対する性能低下は，187.facerec が 14.8%と最も大きかった．このベンチマークは，TWINTAIL で最もベース・モデルに対する性能向上が大きい．そのため，ハーフパンプ FU アレイを導入することによって，IO tail で早期実行できなくなる命令数も多いため，性能低下が大きくなる．

また，若干ではあるが，一部のベンチマークで TWINTAIL よりも HALFPUMP の IPC の方が高くなっている．これは，次のような理由による．ロード命令がレイテンシ予測ミスをして再実行される時，次のアクセス時には 1 次キャッシュにヒットするとして，レイテンシ予測器を更新する．しかし，IO tail で早期実行をすると，1 次キャッシュにデータが来ていない状態でヒットと予測されてキャッシュ・アクセスをし，連続してレイテンシ予測ミスを起こすことがある．この現象の頻度は 1 度目から 2 度目のキャッシュ・アクセスまでのタイミングにより変化し，TWINTAIL より HALFPUMP の方が性能がよくなることがある．

ハーフパンプ FU アレイの性能低下が平均して，2.4%と小さかった理由を考察するために，次に IO tail で実行できると予測される命令の割合および，実際に実行される命令の割合を挙げる．

5.3 In-Order tail で実行される命令の割合

図 5.2 に，全命令のうちステアリング時に IO tail で実行可能と予測された命令の割合を，図 5.3 には，実行可能なクラスの命令のうち，実行可能と予測された命令の割合を示す．また，図 5.2 には IO tail で実行されてリタイアし

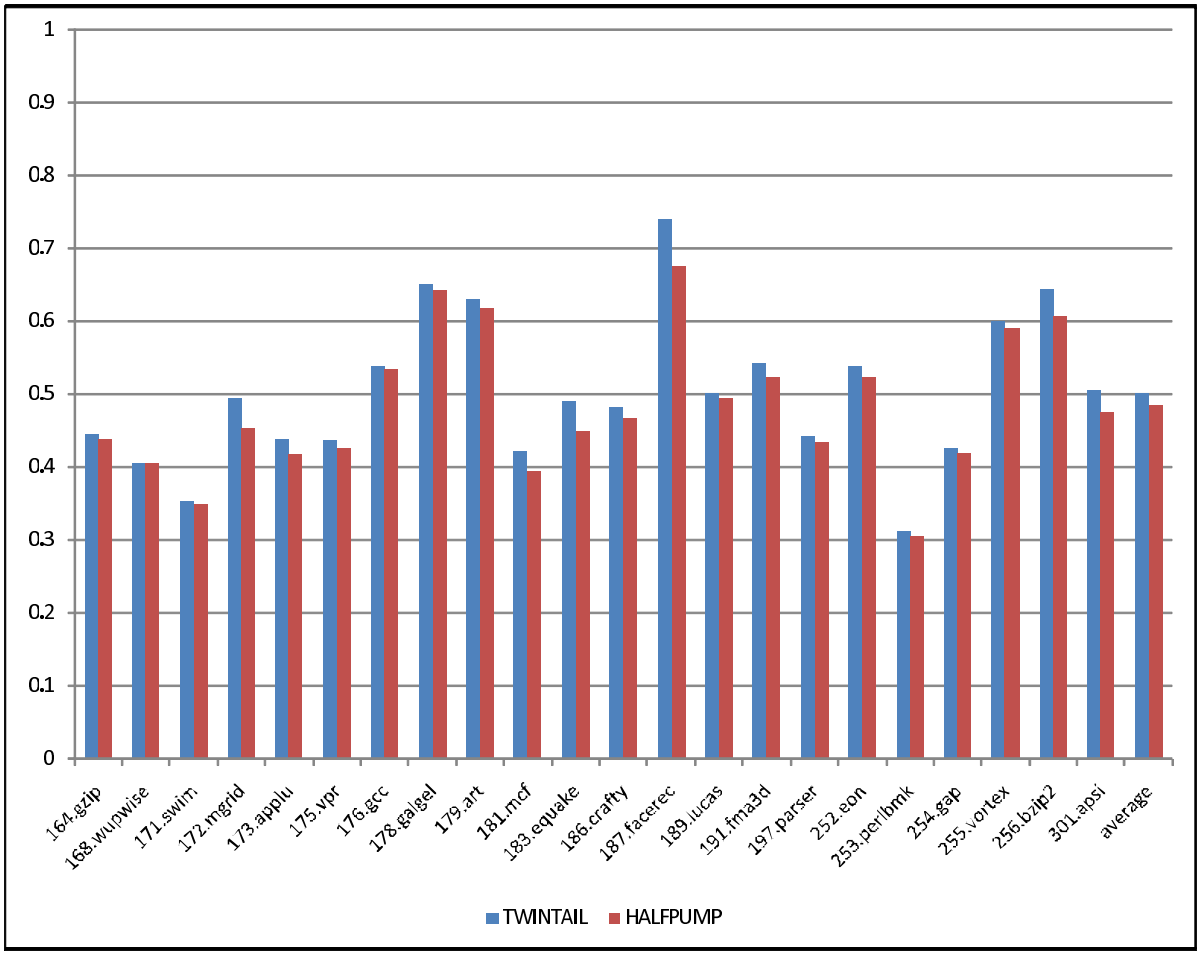


図 5.2: ステアリング時における In-Order tail での実行予測割合

た命令の割合を示す．

全てのベンチマークでHALFPUMPの方がTWINTAILに比べ，ステアリング時にIO tailで実行されると予測された命令の割合も，IO tailで実行された命令がリタイアした割合も双方ともに低くなっている．

これは，4.2節で述べたように，ハーフパンプFUアレイではIO tailで同じ段にある命令間の依存が増えることに伴い，IO tailで実行できない命令が増加するからである．ハーフパンプFUアレイでは平均で，IO tailで実行できる命令が5%程度減少しているが，リタイアした命令の40%以上はIO tailで実行されている．

図 5.1では，ハーフパンプFUアレイを導入したツインテール・アーキテクチャにおいても，平均10%以上の性能向上が得られている．このことから，ハーフパンプFUアレイを用いたIO tailでも，OoO tailの資源節約効果を引き出し，性能を高めるのに十分な量の命令を実行できているといえる．

5.4 パフォーマンス・クリティカルなループの実行の改善

BASEMODELとTWINTAILによるプログラムの実行の様子をタイミング・チャート に表した．シミュレータを用いて実際のプログラムにおける命令のパイプライン・ステージ の遷移を記録し，本研究室で開発した視覚化アプリケーション「Kanata」を用いてタイミング・チャートを描いた．実行したプログラムはベンチマーク256.bzip2である．

図 5.5と図 5.6のタイミング・チャートは縦に命令をフェッチ・オーダに表示し，横にパイプライン・ステージの遷移をフェッチ開始からリタイアまで表示している．したがって，タイミング・チャートをズームアウトすると，プログラムのIPCが命令毎のタイミング・チャートに塗りつぶされた領域の傾きとして現れる．ここでは傾きの違いが判るように，ツインテール・アーキテクチャによる実行の様子と通常のスーパスカラ・プロセッサによる実行の様子を重ねて表示している．

パイプライン・ステージの塗り分けはフェッチからレジスタ読み出しまでを青，ディスパッチから発行までを緑，実行を赤としている．また図 5.6の(a)は図 5.5をズームアウトして眺めた図であり，(b)は(a)をさらにズームアウトして眺めた図である．

図 5.5を見るとツインテール・アーキテクチャとベース・モデルでは少しずつフェッチのタイミングがずれている．これは命令ウィンドウのエントリがより早く解放されるため，ツインテール・アーキテクチャの方がフェッチがストールしにくいためである．命令ウィンドウのエントリがより早く解放されるのはツインテール・アーキテクチャの方が命令が早く実行され，早くリタイア可能

になるためで，スループットの違いとして現れている．

ここに実行の様子を示した命令列は，長さ 100 命令程度の関数 fullGtU が繰り返し呼び出されているループである．fullGtU はベンチマーク 256.bzip2 の最初の 5G 命令の実行において実行命令数の 70%近くを占める．従ってこのループの効率的な実行が 256.bzip2 のパフォーマンスにとって最もクリティカルである．なお fullGtU は ShellSort を実装した関数である．

タイミング・チャートをズームアウトして眺めると IPC の改善がはっきりと判る．256.bzip2 はツインテール・アーキテクチャによって 30%程度の性能向上が得られたベンチマークである．タイミング・チャートに現れる傾きの違いもこの性能向上と同程度であり，ツインテール・アーキテクチャがパフォーマンス・クリティカルなループの実行を改善したことにより IPC の向上に寄与したことが判る．

表 5.1: ベース・モデルの主要なパラメータ

Fetch Width	4
Issue Width	int: 2, float: 2
Integer Units	ALU × 4, MUL × 1, DIV × 1, 32entries instruction queue(used universally by all integer instructions)
Floating Point Units	ADD/MUL/DIV × 1, 16entries instruction queue(used universally by all FP instructions)
Memory Instructions Queue	16entries
L1 Cache	32KB, 4ways, 3cycles to access
L2 Cache	4MB, 8ways, 10cycles to access
Memory	200cycles to access
Physical Registers	int: 96 float:64
Re-order Buffer Entries	84
Pipeline Depth	Fetch-3, Read-2, Dispatch-2, Schedule-2, Issue-2, Write Back-2

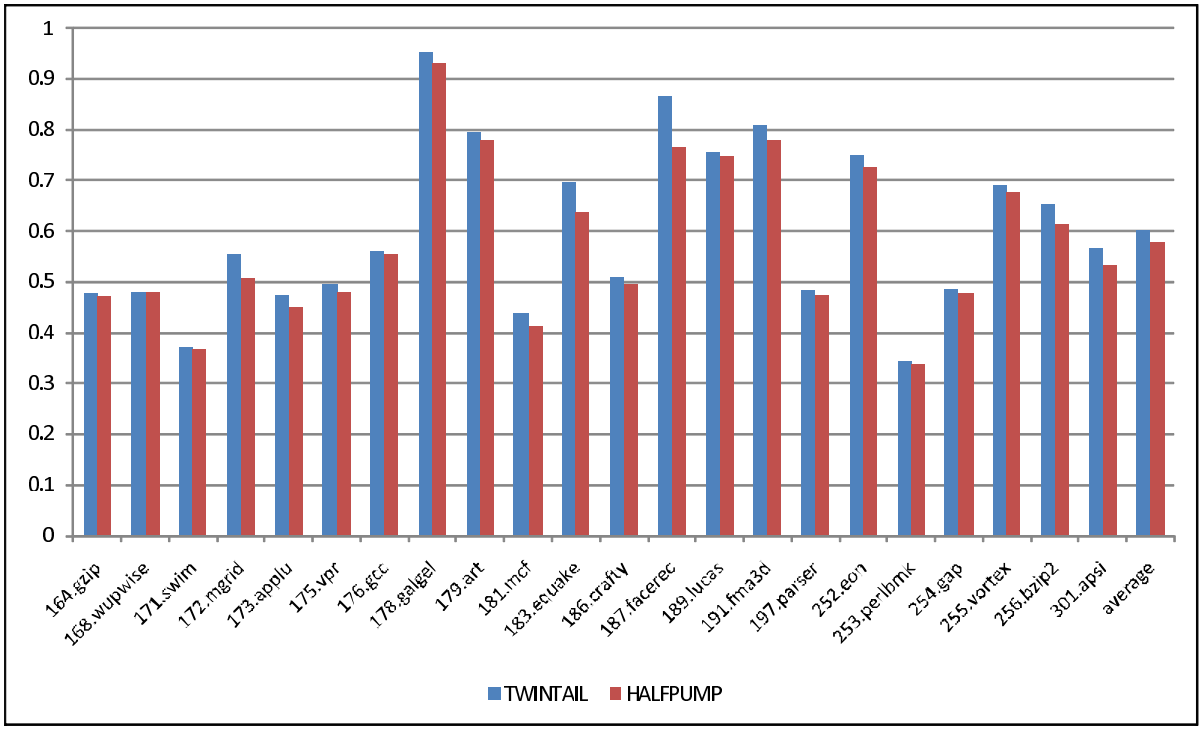


図 5.3: 実行可能なクラスの命令に対する，In-Order tail での実行予測割合

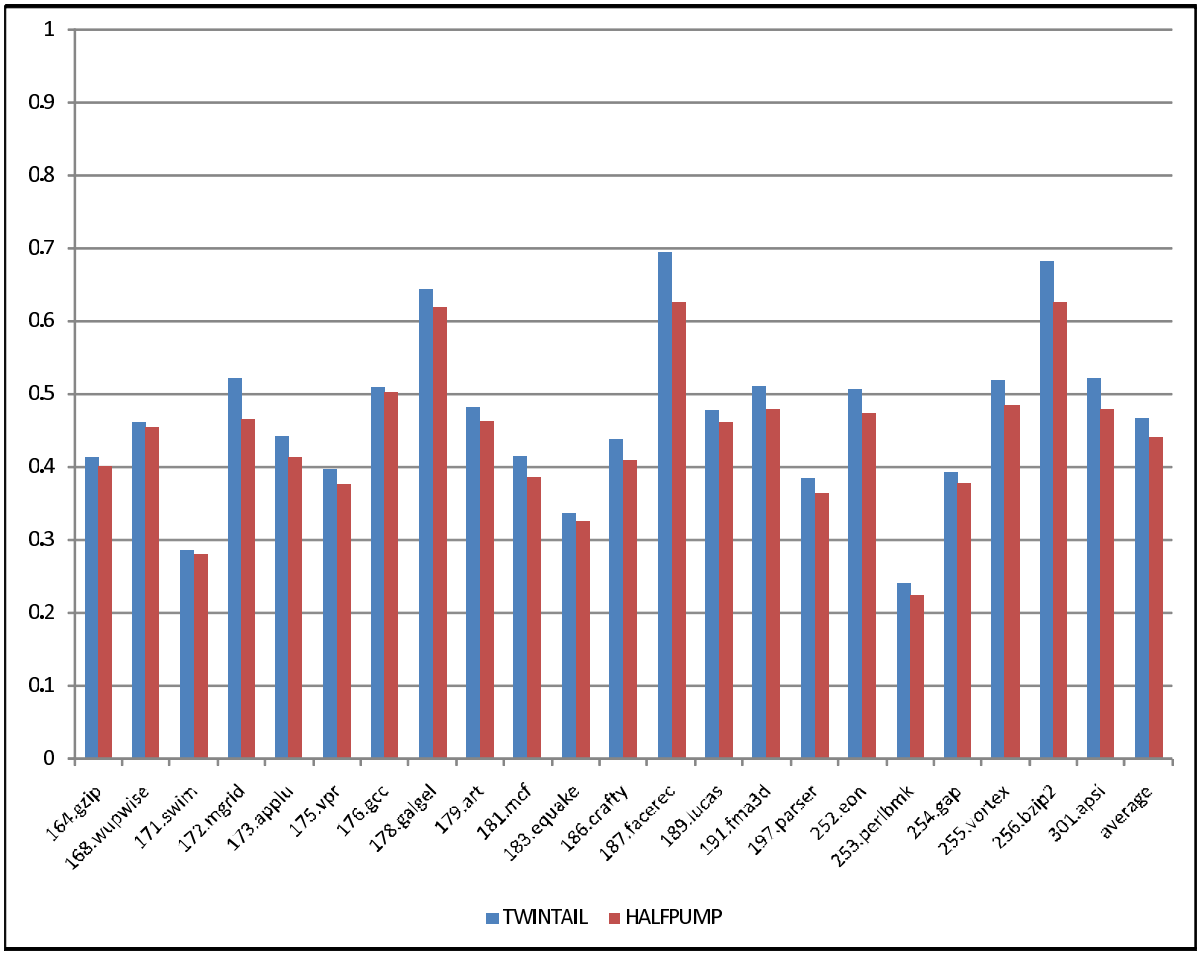


図 5.4: In-Order tail での命令実行割合

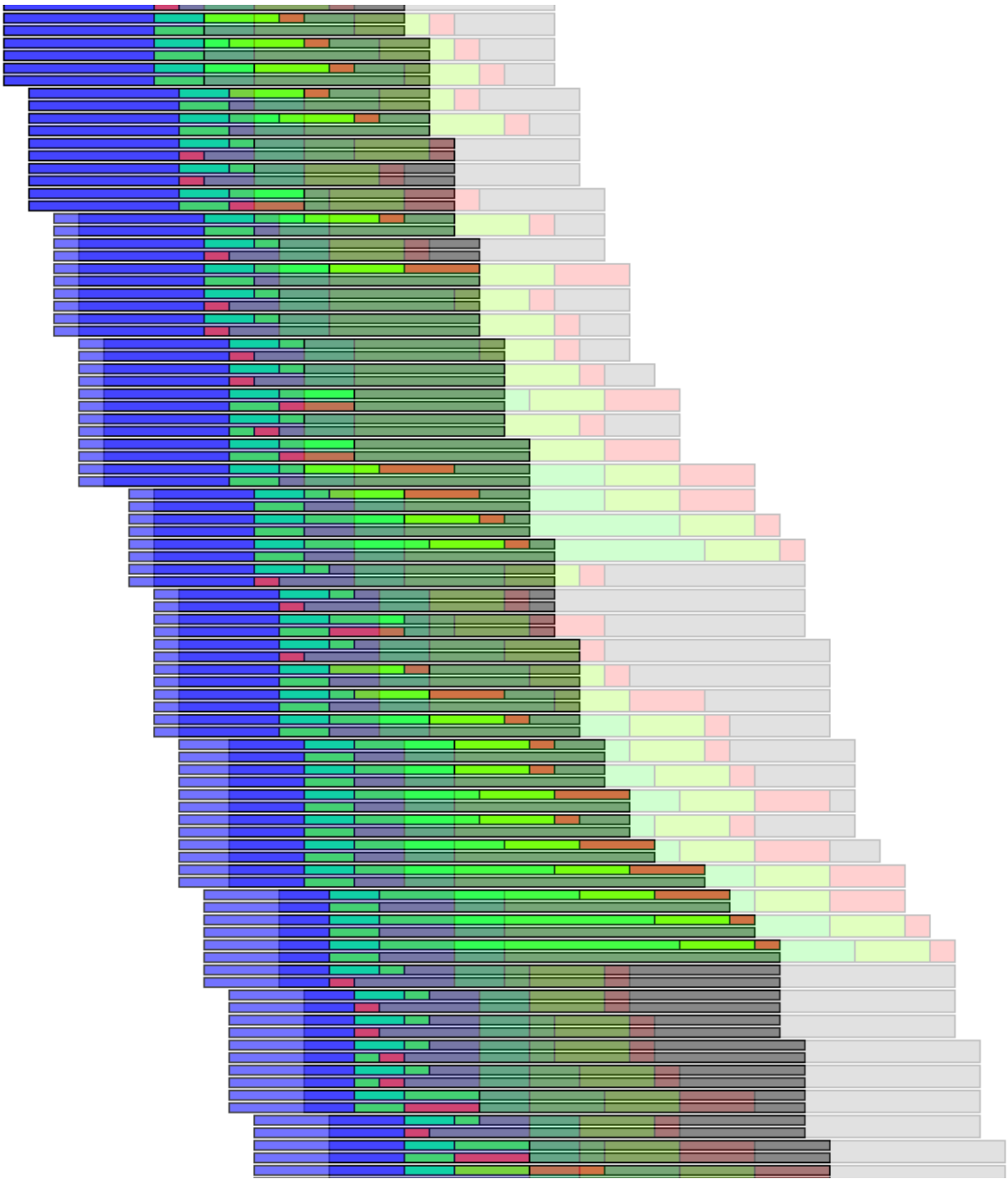


図 5.5: パフォーマンス・クリティカルなループの実行の様子 (拡大版)

(a) (b)

図 5.6: パフォーマンス・クリティカルなループの実行の様子

第6章　関連研究

本章では，本論文に関する関連研究を挙げる．

6.1　命令の再実行方法

6.1.1　POWER4

POWER4では，アドレス一致/不一致予測ミスに対しては再フェッチを，レイテンシ予測ミスに対しては再発行することで，命令の再実行を行う [1].

ロード命令の実行結果に依存する命令は，一旦ロード命令が1次キャッシュにヒットするとして投機的に命令キューから解放される．もしロード命令が1次キャッシュ・ミスを起こした場合は，ロード命令に依存する命令は命令キューに戻され，データが正しく1次キャッシュにロードされるのを待ってから再発行されて，実行される．また，アドレス一致/不一致予測ミスは起こらないとして，投機的に命令キューから命令を解放し，ミスを起こした場合には，ミスした命令から再フェッチする．

6.2　先行実行に関する関連研究

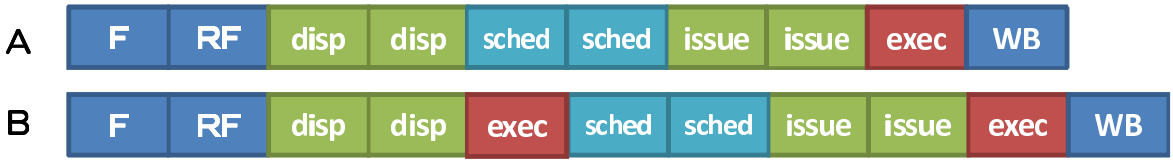
本節では，プロセッサの資源制約を解消することで，先行実行を行う手法について述べる．

6.2.1　フロントエンド実行

小西らはフロントエンドに演算器を追加し，バックエンドに加えてフロントエンドでも命令の実行を行う手法、フロントエンド実行を提案している [6].

図 6.1 に従来のプロセッサのパイプラインとフロントエンド実行を行う場合のパイプラインを示す.

フロントエンド実行ではそのステージとして1〜2サイクルを割り当てる．フロントエンド実行に1サイクルを割り当てる際にはシングル・サイクルで実行を完了することができる ALU 演算やシフト演算のみを行う．これを行うためにはフェッチ幅と同数の演算器をフロントエンドに用意してやるだけでよい．



A：スーパスカラ・プロセッサのパイプライン
B：フロントエンド実行のパイプライン

図 6.1: フロントエンド実行のパイプライン・チャート

また，フロントエンド実行に 2 サイクルを割り当てる際には、依存関係にある 2 つのシングル・サイクル演算を行うことに加えて，ロード命令を行うことも提案されている．この場合には 1 ステージ目でアドレスの計算を行い，2 ステージ目にキャッシュのアクセスを行う．アクセスするキャッシュとしては 1 ステージでアクセスすることが可能な 1KB 程度の 0 次キャッシュをフロントエンドに用意することが考えられている．

このようにフロントエンド実行を行うためにはフェッチ幅と同数の演算器および小容量のキャッシュを用意すれば十分である．現在のスーパスカラ・プロセッサにとってこれらは大きなハードウェアではなく，したがってフロントエンド実行に必要なハードウェア・コストは小さいと言ってよい．

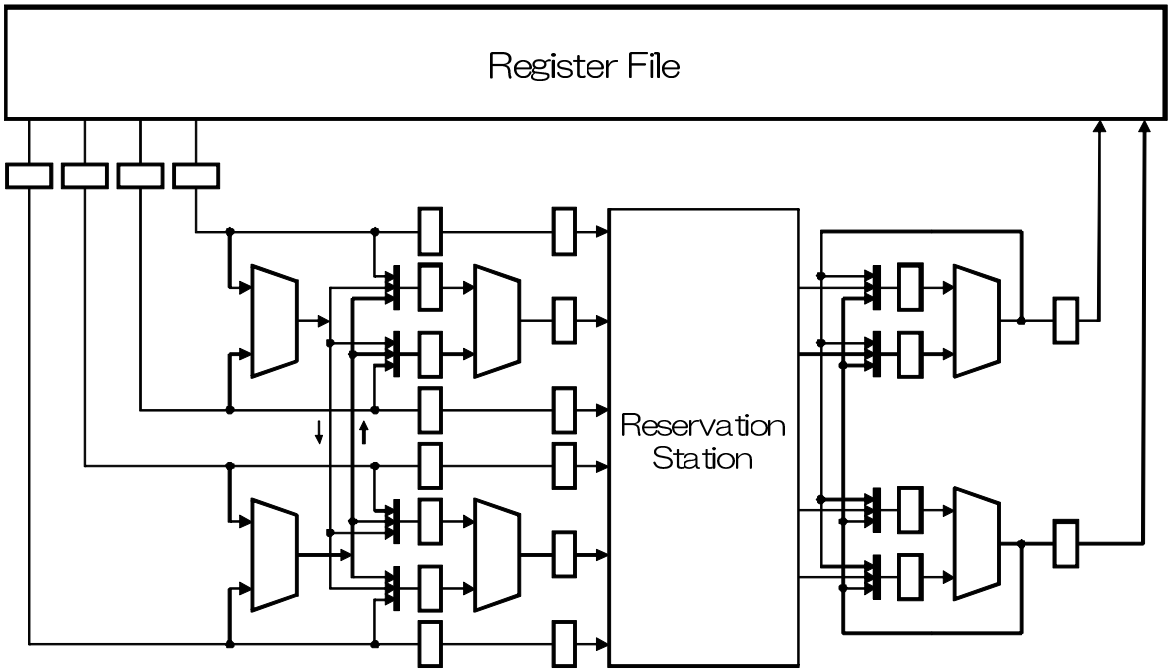


図 6.2: フロントエンド実行のブロック図

図 6.2 に具体的なフロントエンド実行機構を示す．図 6.2 は 2 ウェイのスーパスカラ・プロセッサにおいて，2 段のフロントエンド実行を行う場合の実行機構である．

フェッチされた命令はまずフロントエンドでレジスタ読み出しを行い，この時点で演算可能ならば直下に配置された演算器で直ちに演算を行う．この時点で演算可能でない命令は，フロントエンド演算器の最終段まで進んだ後に従来のスーパスカラ・プロセッサと同様に命令ウィンドウにディスパッチされ，演算可能となったものから発行されてバックエンドの演算器で実行されることになる．

図 6.2 のようにフロントエンド実行を多段にする場合には，基本的には全対全の接続が必要となる．図 6.2 のフロントエンド実行ユニットでは自分自身やフロントエンド実行の 2 段目から 1 段目へのバイパスを設けることはお粉ていないが，これらのバイパスを設けることも考えられ，そうすることにより，ツインテール・アーキテクチャ同様にフロントエンド実行できる命令数を増加させることができる．

ツインテール・アーキテクチャはフロントエンド実行ステージによってパイプライン段数が増加してしまうことを改善した手法である．

フロントエンド実行では従来のパイプラインにフロントエンド実行ステージを追加していたために全体のパイプライン段数が増加してしまっていた．それに対して，ツインテール・アーキテクチャでは新たに加える実行ステージを従来のパイプラインである OoO tail から切り離し，全く別のパイプラインである IO tail として独立させている．

したがって，IO tail のパイプライン段数をいくら増加させたとしても OoO tail のパイプラインが増加することはない．この点でツインテール・アーキテクチャはフロントエンド実行のデメリットを改善している．

6.2.2 先行実行を利用したロード命令のレイテンシ削減および正確なスケジューリング手法

山本らは物理レジスタをリネーム・ステージと命令ウィンドウとの 2 段階で解放することによって，物理レジスタの不足によるリネーム・ステージでのストールを解消することを提案している [5].

さらに，命令ウィンドウで 2 段階目の解放を待ちうけている命令のソース・オペランドが揃い，実行可能であるならば，先行実行を行い，そのことによってロード・データのキャッシュへのプリフェッチを実現している．

先行実行された命令の結果をその命令の結果に依存する命令にフォワーディングすることで，先行実行を連鎖して行うことができる．先行実行された命令の中にロード命令が存在すれば，そのロード命令によるプリフェッチが行われ

るのである．

この手法はロード命令を先行実行することでキャッシュ・アクセス・レイテンシを減少させることができる点で本研究と関連がある研究であるといえる．

6.2.3 RENO

Vlad らはリネーミングを工夫することで，一部の命令をリネーミング･ステージで取り除く手法として RENO(RENaming Optimizer) を提案している [3]. RENO はマッピング・テーブルの上手な利用と物理レジスタの共有構成を用いて命令を取り除き，依存関係を構成し直す手法である．

RENO は

1. 取り除かれた命令をデータフロー・グラフから取り除ける
2. 取り除かれた命令は以降でハードウェア資源を使用しない

といった利点を持っており．これにより，8〜13%の性能向上が見込めることが示されている．

これはフロントエンドで命令実行と等価な処理を行うことで，その命令を取り除く手法であるという点で本研究と関連があるといえる．

第7章　おわりに

7.1　まとめ

本論文では，スーパスカラ・プロセッサが抽出できていない命令の並列性を抽出することで，性能向上を得ることができるツインテール・アーキテクチャの改良手法を提案した．ツインテール・アーキテクチャは，命令ウィンドウへのディスパッチと並列に命令を実行することで，命令の早期実行を行う．さらに，早期実行できる命令をディスパッチしないことで，資源を節約することができ，バックエンドで処理可能な命令を増加させる．

本論文では，まずツインテール・アーキテクチャにおける，命令の再実行方法について提案した．また，追加された実行系のスループットを保ったまま，半分の周波数で動作させることで消費電力の低減を図る，ハーフパンプFUアレイを提案・評価した．

シミュレーションによるツインテール・アーキテクチャの評価では，命令ウィンドウへディスパッチを行わない命令を再フェッチによって再実行しても，全てのベンチマークでベースモデルのスーパスカラ・プロセッサより性能が良くなり，平均で13.6%の性能向上が得られることがわかった．これは，予測器の精度が十分に高いため，データ投機ミスがほとんど起きていないことによる．

さらに，ハーフパンプFUアレイの評価では，ハーフパンプFUアレイを利用しないツインテール・アーキテクチャに対して平均2.4%の性能低下にとどまり，ベース・モデルのスーパスカラ・プロセッサに対し，平均10.7%の性能向上が得られることが分かった．IO tailの演算器をハーフパンプFUアレイにすることで，IO tailで実行できなくなる命令は5%程度であり，40%程度の命令はIO tailで実行され，OoO tailへディスパッチていないため，十分にOoO tailの資源を節約できているといえる．

7.2　今後の課題

今後の課題としては，IO tailでは平均40%の命令が実行できるため，命令ウィンドウ・サイズや発行幅を縮小し，プロセッサ内でのIO tailの役割を高めることが挙げられる．今回の評価ではロード命令の実行結果がレジスタにライト・バックされるまでは，IO tailでロード命令の結果を利用できないとして

いる．したがって，ロード命令の直後にあるコンシューマ命令は IO tail で実行できていない．しかし，IO tail にハーフパンプ FU アレイを用いることで，IO tail での命令の実行時間に余裕が生まれるため，0 次キャッシュを設けることができるようになる．これにより，ロード命令に依存する命令も IO tail で実行することで，IO tail で実行できる命令数を増やし，OoO tail の資源を更に節約することができるようになると考えられる．

より現実的な改良を施していくことで，魅力的な選択肢としてツインテール・アーキテクチャを提案していきたい．

参考文献

[1] J. S. Fields Jr. H. Le J. M. Tendler, J. S. Dodson and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, Vol. 46, pp. 5–25, 2002.

[2] The Standard Performance Evaluation Corporation. *SPEC CPU2000 suite http://www.spec.org/cpu2000/*.

[3] Amir Roth Vlad Petric, Tingting Sha. RENO: A Rename-Based Instruction Optimizer. In *International Symposium on Computer Architecture. Proceedings of the 32nd annual international symposium on Computer Architecture.*, pp. 98–109, 2005.

[4] 五島正裕. Out-of-order ILP プロセッサにおける命令スケジューリングの高速化の研究. 京都大学 (博士論文), 2004.

[5] 山本哲弘, 安藤秀樹, 島田俊夫. 先行実行を利用したロード命令のレイテンシ削減および正確なスケジューリング手法. 先進的計算基盤システムシンポジウム SACSIS, pp. 403–410, 2006.

[6] 小西将人, 福田匡則, 五島正裕, 中島康彦, 森眞一郎, 富田眞治. フロントエンド実行によるプリロードの提案. In *2004-ARC-159*, pp. 31–36, 2004.

[7] 平井遥, 入江英嗣, 五島正裕, 坂井修一. ツインテール・アーキテクチャ. In *2006-ARC-169*, pp. 43–48, 2006.

[8] 堀尾一生, 平井遥, 入江英嗣, 五島正裕, 坂井修一. ツインテール・アーキテクチャ. 先進的計算基盤システムシンポジウム SACSIS, pp. 303–311, 2007.

発表文献

主著論文

- ツインテール・アーキテクチャの改良
  亘理 靖展, 堀尾 一生, 入江 英嗣, 五島 正裕, 坂井 修一
  情報処理学会 研究会報告 ARC 2007-ARC-174(Aug. 2007) pp.7-12

- ツインテール・アーキテクチャにおけるハーフパンプ FU アレイ
  亘理 靖展, 堀尾 一生, 渡辺 憲一, 塩谷 亮太, 入江 英嗣, 五島 正裕, 坂井 修一
  先進的計算基盤システムシンポジウム SACSIS2008(June. 2008) (投稿中)

- ツインテール・アーキテクチャにおけるハーフパンプ FU アレイ
  亘理 靖展, 堀尾 一生, 渡辺 憲一, 塩谷 亮太, 入江 英嗣, 五島 正裕, 坂井 修一
  情報処理学会論文誌 コンピューティングシステム (投稿中)

謝辞

非常に多くの方々から多大なご指導，ご協力，励ましを頂き，本論文を完成させることができました．この場を借りて，感謝の意を表したいと思います．本研究を進めるにあたり，坂井修一教授，五島正裕准教授から，大変多くのご指導を頂きました．ここに深く感謝の意を表します．

入江英嗣博士には，相談会等で様々な形でアドバイスを頂きました．一林宏憲氏，塩谷亮太氏をはじめ，原理系グループのメンバーの皆様には，ミーティングにおける議論を通して，貴重なご意見を頂きました．特に渡辺憲一氏には，シミュレータ「鬼斬弐」の実装など，本論文の製作に不可欠な多くの仕事を行っていただきました．また，堀尾一生氏には，研究を進めるにあたって多くのご指摘を頂きました．

八木原春水さん，月村美和さんには，研究室における設備の導入や各種事務手続きなど，研究室で過ごすための様々なご支援を頂きました．