修士論文

繰り返し構造に着目した
動的なヘルパースレッディング

Dynamic Helper Threading for Repetition Structures

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

安藤 徹

概要

近年のプロセッサではパイプラインが深化する傾向にあり，分岐予測ミスペナルティが大きくなってきている．分岐予測ミスペナルティを削減する手法として分岐予測の研究が盛んに行われている．しかし，従来のように過去の分岐パターンや実行パス情報を用いる分岐予測では，分岐結果の規則性が低い場合には予測を行うことが困難となる．

そのような分岐命令に対して分岐予測を用いずに処理する分岐プレディシジョンという手法がある．分岐プレディシジョンでは，分岐命令の依存元の命令の実行終了時に分岐命令を早期実行し，その結果をフォワーディングすることによって，分岐予測を用いずに分岐先を決定することができる．しかし，フォワーディングを成功させるには分岐命令とその依存元の命令が十分離れていなければならないといった問題がある．

本研究では，プログラムの繰り返し構造に着目した，動的なヘルパースレッディングを提案する．ヘルパースレッドはメインスレッドの先行実行を行い，その実行結果を分岐プレディシジョンの補助として用いる．これによって分岐プレディシジョンではフォワーディングが間に合わない場面に対しても，分岐先を早期決定することができる．

目 次

第1章　はじめに **1**
1.1　背景 1
1.2　目的 2
1.3　構成 2

第2章　分岐予測 **3**
2.1　仮想関数 3
2.2　分岐予測ミスが性能に及ぼす影響 4
2.3　分岐予測の振る舞い 5

第3章　分岐プレディシジョン **7**
3.1　分岐予測ミス・ペナルティと遅延の関係 7
3.1.1　遅延による予測ミス・ペナルティの増加 7
3.1.2　コード上の距離を離す最適化 8
3.2　分岐結果のフォワーディング 9
3.3　早期リカバリ 10
3.4　実装と動作 11
3.4.1　学習とフォワーディング 12

第4章　ヘルパースレッディング **15**
4.1　概要 15
4.2　ヘルパースレッディングの従来研究 16
4.2.1　Simultaneous Subordinate Microthreading 16
4.2.2　Slipstream Processor 16
4.2.3　Speculative Data-Driven Multithreading 18
4.2.4　Dynamic Speculative Precomputation 20
4.2.5　Master/Slave Speculative Parallelization 21
4.3　従来研究のまとめ 21
4.3.1　先行実行命令列の作成 22
4.3.2　ヘルパースレッドの実行開始タイミング 22
4.3.3　ヘルパースレッドの実行結果の利用 22
4.4　ヘルパースレッドを用いない先行実行手法 23

4.4.1 Runahead Execution . . . 23
4.4.2 物理レジスタ 2 段階解放 . . . 24

第 5 章 繰り返し構造に着目したヘルパースレッディング 26
5.1 ヘルパースレッドの作成と実行 . . . 26
5.1.1 先行実行命令列の作成 . . . 26
5.1.2 ヘルパースレッドの実行開始タイミング . . . 29
5.2 分岐プレディシジョンの補助 . . . 29

第 6 章 評価結果 30
6.1 評価方法 . . . 30
6.2 評価モデル . . . 30
6.3 予備評価 . . . 30
6.4 ベンチマーク . . . 32
6.5 評価結果 . . . 32

第 7 章 おわりに 35
7.1 本研究のまとめ . . . 35
7.2 今後の課題 . . . 35

参考文献 36

発表文献 38

図 目 次

2.1 仮想関数の使用例 4
2.2 予測ミスがなくなった場合の性能向上率 (%) 6

3.1 add 命令を実行した場合のパイプライン 8
3.2 分岐予測が外れた場合のパイプライン 9
3.3 分岐予測が外れた場合のパイプライン（最適化後） 10
3.4 分岐予測が外れた場合のパイプライン（より距離を離した場合） 11
3.5 フォワーディングを行った場合のパイプライン 12
3.6 早期リカバリ 13
3.7 学習の動作 14
3.8 フォワーディングの動作 14

4.1 Slipstream processor using a two-way chip multiprocessor [11] . . 17
4.2 Hardware Implementation Aspects of DDMT [7] 18
4.3 Pipeline organization of dynamic speculative precomputation [2] . 20
4.4 Processor model used for description and evaluation of runahead [4] 23

5.1 従来のヘルパースレッドの例 27
5.2 値予測を用いたヘルパースレッドの例 28

6.1 予備評価のテストプログラム 31
6.2 BTB の予測ヒット回数 33
6.3 Base に対する Proposed の相対 IPC 33
6.4 先行実行を行った分岐命令の分岐予測ヒット率 34

表目次

2.1 分岐ヒット予測率 6

3.1 パイプライン・チャート上の記号 7

4.1 Comparison of researches 21

6.1 プロセッサの構成 31

第1章　はじめに

1.1　背景

近年のプロセッサではパイプラインが深化する傾向にあり，分岐予測ミスペナルティが大きくなってきている．分岐予測ミスペナルティを削減する手法として分岐予測の研究が盛んに行われている．しかし，従来のように過去の分岐パターンや実行パス情報を用いる分岐予測では，分岐結果の規則性が低い場合には予測を行うことが困難となる．

そのような分岐命令に対して分岐予測を用いずに処理する分岐プレディシジョン [18] という手法がある．分岐プレディシジョンはレジスタ間接分岐ターゲットフォワーディング [17] の考え方を条件分岐にまで一般化した手法で，分岐命令の依存元の命令の実行終了時に分岐命令を早期実行し，その結果をフロントエンドにフォワーディングする．その結果を元にフェッチすることによって分岐予測を用いずに分岐先を決定することができる．しかし，フォワーディングを成功させるには分岐命令とその依存元の命令が十分離れていなければならないといった問題がある．

また，違うアプローチとしてヘルパースレッディングという技術が挙げられる．ヘルパースレッディングは Simultaneous Multithreading(SMT) やチップマルチプロセッサ (Chip Multiprocessor, CMP) のようなマルチスレッド実行を行うプロセッサにおいて，逐次の単一スレッドプログラムの実行を高速化する技術である．通常 SMT や CMP は複数のスレッドを持つ処理において，複数のスレッドを並列に処理ることで高い実行効率を得ている．単一のスレッドのプログラムを実行するような状況ではマルチスレッドの実行資源を有効に利用することはできない．ヘルパースレッディングは単一スレッドのプログラムからヘルパースレッドを作成し，メインスレッドと並列に実行することでマルチスレッドの実行資源を有効に活用する．ヘルパースレッディングでは元のプログラムの命令の一部を抜き出し，メインスレッドに対して先行して実行することによって得られる結果を用いてメインスレッドの実行を高速化することが一般的である．つまり，分岐命令の先行実行を行い，その結果を分岐予測に用いることで予測精度を向上させたり，分岐予測ミスを早期に発見することでペナルティが削減できる．ヘルパースレッディングでは分岐命令に対してだけではなく，ロード命令の先行実行を行うことでロードのプリフェッチをし，メモリアクセスのレイテンシを隠蔽することも行われている．

1.2　目的

本研究では，プログラムの繰り返し構造に着目し，値予測を用いることで早いタイミングでヘルパースレッドを実行開始できるヘルパースレッディングを提案する．提案手法ではヘルパースレッドの実行結果を，分岐プレディシジョンの補助として用いる．先行実行の結果を用いることで通常の分岐プレディシジョンではフォワーディングが間に合わない場面に対しても効果が得られると考えられる．

1.3　構成

．本稿の構成は以下の通りである．まず，第 2 章で分岐予測について説明し，第 3 章で分岐プレディシジョンの概要とその問題点について説明する．第 4 章ではヘルパースレッドを先行スレッドとして扱うヘルパースレッディングについて説明する．第 5 章で提案手法であるプログラムの繰り返し構造に着目したヘルパースレッディングについて説明し，第 6 章で評価を行い，第 7 章でまとめる．

第2章　分岐予測

ある分岐命令が分岐するか否か決まるのを待つのではなく，命令フェッチと同時に当該分岐命令の結果を予測することで，パイプラインに命令が入らない時間 (分岐遅延) を削減することができる．

当然，予測した分岐命令より後に実行する命令は予測が当たった場合しか意味を持たない．よって，予測が外れた際には投機的に実行したこれらの命令に対して，実行結果を巻き戻す必要が生じる．この巻き戻し処理にはそれなりのサイクルが必要となり，加えて，その間実行した命令も無効となるため，パイプラインの使用効率は下がり，プロセッサ性能にとっては大きなペナルティと成り得る．しかし，通常分岐命令が分岐するか否かには統計的な偏りがあり，多くの場合過去の分岐結果を調べることで正しい分岐の方向を予測することができる．例えば，ループ文ではループカウンタが溢れるまでは分岐は成立し続ける．反対に，エラー処理に入るための分岐などは，正常系では常に分岐することはない．

次に一般的な分岐予測器では予測が困難である仮想関数呼び出しについて述べる．

2.1　仮想関数

仮想関数はオブジェクト指向言語で多態性を実現するために用いられる．多態性はプログラムにおいて必ずしも必須の機能ではないが，似た性質を持つデータ群に対して共通の手順で処理することができるため，有効に活用することでプログラムの見通しを良くし，プログラマの負担を軽減することができる．多態性を用いたクラスとその呼び出しの例を図2.1 に示す．この呼び出し例において drawables の中には，基底クラス Drawable の派生クラスである Circle や Square のオブジェクトが格納されている．それぞれが仮想関数として宣言された Draw() をオーバーライドしているため，クラス Drawable のオブジェクトとして Draw() を呼び出したとしても，実際には Circle クラスや Dog クラスで実装された Draw() である．どちらの関数が呼ばれるかは，過去の制御フローのみに依存するため，一般的な分岐予測器で分岐先を予測することは困難である．

```
class Drawable {
 public:
  virtual void Draw(void) = NULL;
};

class Circle : public Drawable {
 public:
  void Draw(void) { … };
};

class Square : public Drawable {
 public:
  void Draw(void) { … };
};

Drawable **drawables;
…
for (int i = 0; NULL != drawables[i]; i++) {
  drawables[i]->Draw();
}
```

図 2.1: 仮想関数の使用例

2.2 分岐予測ミスが性能に及ぼす影響

ここでは分岐予測ミスがプロセッサの性能に及ぼす影響について考察する．

- ipc 予測が完全にヒットした場合の平均 IPC
- L 条件分岐間の平均ランレングス
- a' 分岐予測ミス率
- p 分岐予測ミス・ペナルティ

とすると，L 命令を処理するには式 2.2 のように平均 c_l サイクルかかる．

$$c_l = L/ipc + pa'(cycles) \tag{2.1}$$

したがってこの場合の IPC は式 2.2 のようになる．

$$IPC = L/c_l = L/(L/ipc + pa') \tag{2.2}$$

分岐予測ミスによる IPC の変化率 r は式 2.2 のように計算できる．

$$r = ipc/IPC = (L/ipc + pa')/(L/ipc) = 1 + (pa')/(L/ipc) \tag{2.3}$$

この式により，分岐予測以外の条件を理想化して，分岐予測ミスがなくなった場合の性能向上率を概算したものが図2.2 である．分岐予測ミスがなくなった場合に 2 倍以上の性能が得られるベンチマークも存在している．これから分かるように，分岐予測ミスがプロセッサの性能に及ぼす影響は大きい．

2.3 分岐予測の振る舞い

表 2.1[18] は SPEC CPU2000[9] と SPEC CPU2006[10] に含まれる全アプリケーションのうち，分岐予測ヒット率が 95%以下であったアプリケーションの結果を取りだしたものである．各列の意味は以下の通りである．

- **全分岐命令—予測ヒット率** 実行された全ての分岐命令における，分岐予測ヒット率を意味する．
- **ミスの 8 割を起こす分岐命令—命令数** 発生した分岐予測ミス回数のうち 8 割のミスを起こした命令について取り上げたものを意味する．たとえば 435.gromacs の場合，分岐予測ミスの 8 割が 1 個の分岐命令によって引き起こされており，それらの命令の分岐予測ヒット率の平均は 54.04%であることを意味する．

上記の結果より，SPEC CPU2000 と SPEC CPU2006 に含まれる分岐予測ヒット率の低いアプリケーションの多くでは，数個から数十個と少数の分岐命令により分岐予測ミスの 8 割以上が引き起こされていることが分かる．また，これらの命令における分岐予測ヒット率は，全体の予測ヒット率と比較して大幅に低くなっている．

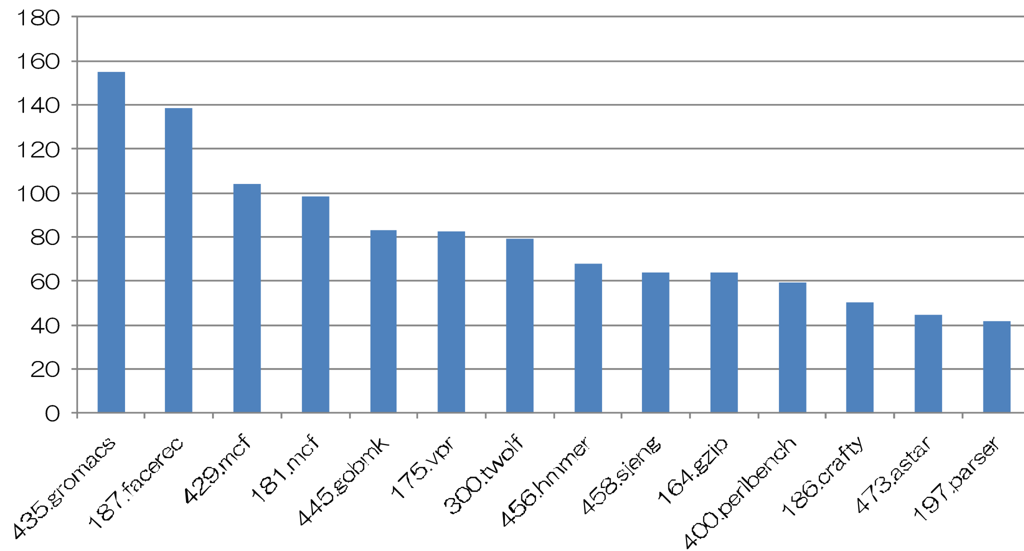

図 2.2: 予測ミスがなくなった場合の性能向上率 (%)

表 2.1: 分岐ヒット予測率

ベンチマーク	全分岐命令 予測ヒット率	ミスの 8 割を起こす分岐命令 予測ヒット率	 命令数
435.gromacs	80.60	54.04	1
187.facerec	82.67	51.67	1
429.mcf	86.96	76.75	21
181.mcf	87.68	77.81	22
445.gobmk	89.62	83.57	298
175.vpr	89.74	76.12	25
300.twolf	90.08	83.72	14
456.hmmer	91.53	73.32	3
458.sjeng	91.98	84.01	63
164.gzip	91.99	67.33	7
400.perlbench	92.57	16.72	4
186.crafty	93.71	87.09	93
473.astar	94.39	86.60	2
197.parser	94.77	90.72	267

第3章　分岐プレディシジョン

3.1　分岐予測ミス・ペナルティと遅延の関係

レジスタ間接分岐のうち，複数の分岐先を取り得るものについては正確な予測を行うことが困難である．このレジスタ間接分岐を，予測を用いずに処理する手法として，豊島らはレジスタ間接分岐ターゲット・フォワーディング [17] を提案している．レジスタ間接分岐ターゲット・フォワーディングでは，主にオブジェクト指向言語における仮想関数呼び出しによるレジスタ間接分岐をターゲットとしている．

このレジスタ間接分岐ターゲット・フォワーディングを条件分岐にまで一般化したものが分岐プレディシジョンである．この両者の基本的な考え方はほぼ同じものであるが，共に分岐命令の依存元の命令の遅延が Out-of-Order 実行ではうまく隠ぺいできないことによる．以下では，この点について説明を行う．なお，以下では説明のため，依存関係にある命令のうち，依存元の命令を Producer，依存先の命令を Consumer と呼ぶことにする．

3.1.1　遅延による予測ミス・ペナルティの増加

通常の命令の場合　一般に，Out-of-Order 実行を行うプロセッサでは，Producer と Consumer のコード上の距離は性能に大きな影響を与えない．例として図 3.1 に，2 回の連続した load(ld) とそれに依存する add 命令の実行の様子を示す．パイプライン上の各ステージの意味は，表 3.1 に示す通りである．load 命令によって生じる遅延は，Out-of-Order 実行により，依存関係にない後続命令を実行することで隠ぺいされる．

表 3.1: パイプライン・チャート上の記号

記号	ステージ
IF	命令フェッチ
SC	命令スケジューリング
EX	実行
AC	アドレス計算
L1	L1 キャッシュ・アクセス

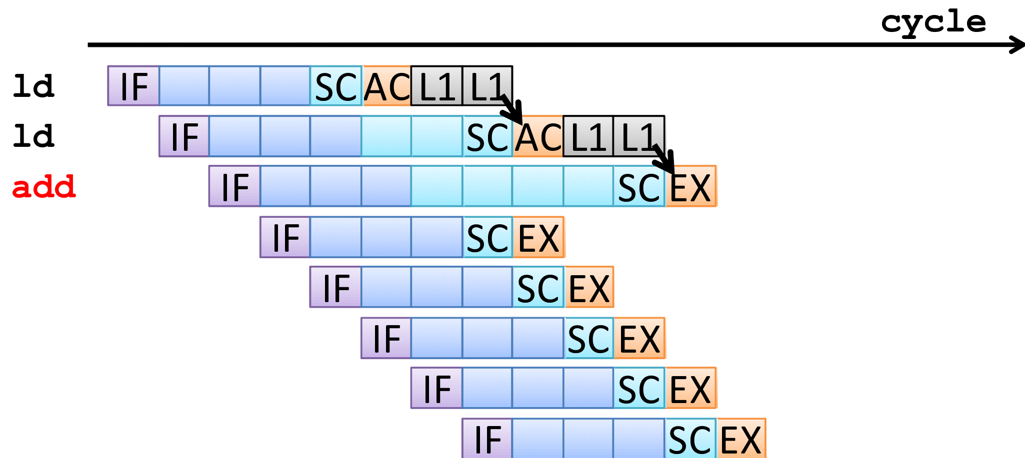

図 3.1: add 命令を実行した場合のパイプライン

分岐命令の場合　これに対し，Consumer が分岐命令であった場合，その Producer との距離は，Out-of-Order 実行では隠ぺいできない．図 3.2 に，2 回の連続した load と，それに依存する分岐命令の実行の様子を示す．同図の分岐命令では，分岐予測が外れたものとする．

分岐命令の後続命令は，分岐先が確定するまでの間，分岐予測によって実行先を決定する．このため，分岐命令がフェッチされてから，実行されるまでの遅延が分岐予測ミス・ペナルティとなる．図 3.2 のように，load 命令 (ld) の遅延に合わせて分岐命令 (br) の実行を遅らせた場合，その分だけ分岐予測ミス・ペナルティが増加する．

3.1.2　コード上の距離を離す最適化

上記の予測ミス・ペナルティの増加は，分岐命令とその Producer のコード上の距離をコンパイラなどによって静的に離すことにより軽減することができる．これを行った場合の実行の様子を図 3.3 に示す．あらかじめ ld をコード上で br から離しておくことにより，br はフェッチされた後，最速のタイミングで実行することが可能になる．これにより，分岐予測によってフェッチを行う期間—すなわち分岐予測ミス・ペナルティを短縮することができる．

なお，この最適化を図 3.3 に示す距離以上に行っても，効果を得ることはできない．図 3.3 からさらに命令間の距離を離したものを，図 3.4 に示す．図 3.3 と同様に，br の実行は最速のタイミングで行われるため，その予測ミス・ペナルティは，図 3.3 の場合と変わらない．

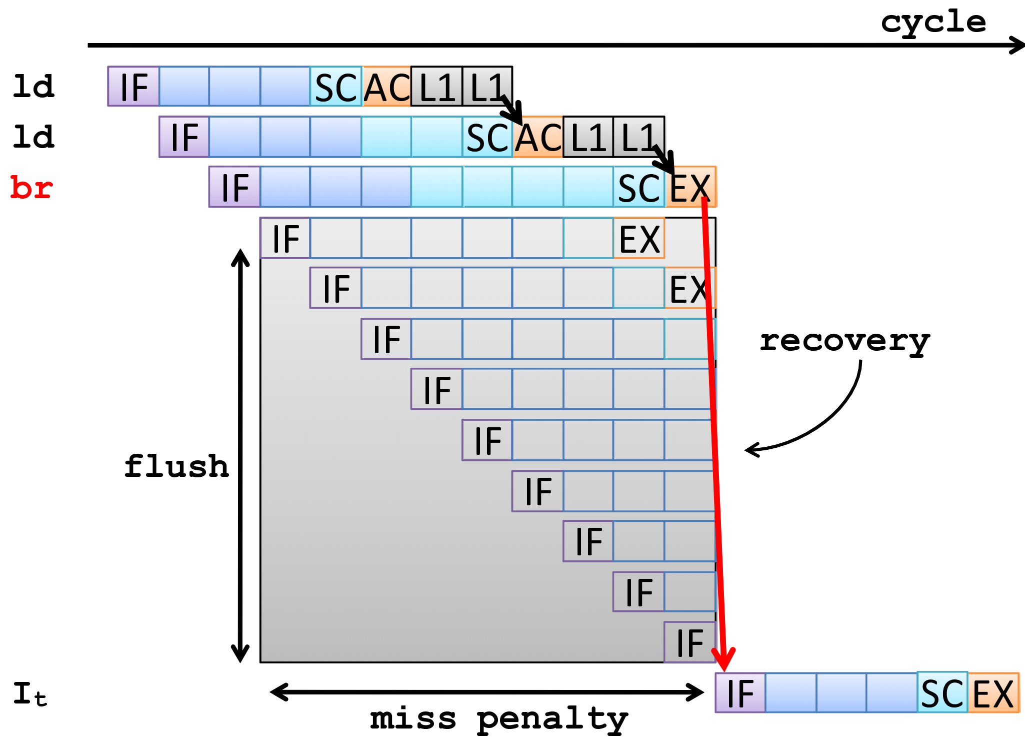

図 3.2: 分岐予測が外れた場合のパイプライン

3.2　分岐結果のフォワーディング

先に述べた図 3.4 では，分岐命令のフェッチ時には既にその Producer(ld) の実行は終了している．レジスタ間接分岐ターゲット・フォワーディングや分岐プレディシジョンの基本的なアイデアは，このように分岐命令のフェッチ時に Producer の実行が終了している場合，Producer の実行結果をフロントエンドへフォワーディングすることにある．

図 3.5 にこのフォワーディングを行った場合の動作を示す．図のように，Producer(ld) と Consumer(br) が十分にコード上で離れている場合，Producer の結果をフロントエンドへフォワーディングすることにより，後続命令のフェッチ先を決定する．これにより，Consumer の後続の命令は分岐予測を用いてフェッチ先を決定する必要がなくなり，分岐予測ミスの発生をなくすことができる．

条件分岐への一般化　分岐プレディシジョンがレジスタ間接分岐ターゲット・フォワーディングと異なるのは，Consumer が条件分岐である点である．条件分岐の場合，レジスタ間接分岐とは異なり，Producer の結果をフォワーディングしただけでは分岐先を決めることはできない．そこで，Producer の実行終了時に対となる Consumer の条件判定を早期実行し，その結果をフォワーディングする．この早期実行は，例えば分岐命令が branch equal であった場合，依存

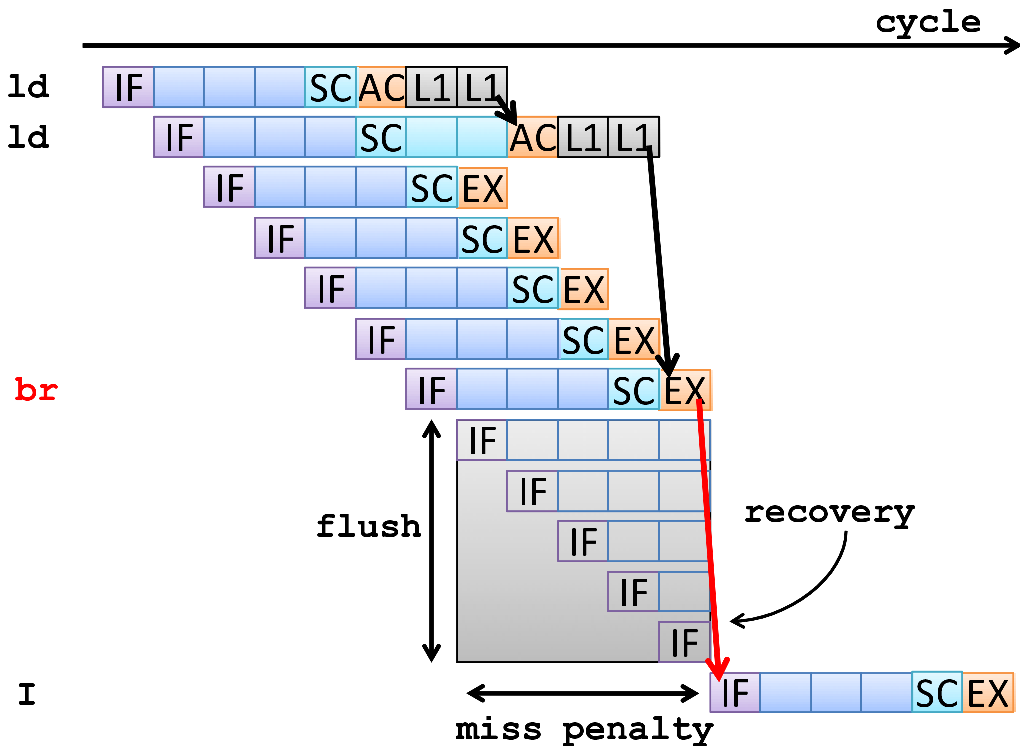

図 3.3: 分岐予測が外れた場合のパイプライン（最適化後）

元である Producer のディスティネーションのゼロ判定を Producer の実行時に行うということである．これにより，Consumer の後続の命令は分岐予測ではなく，フォワーディングされた結果を用いてフェッチ先を決定することができる．

3.3 早期リカバリ

Producer と Consumer のコード上の距離が十分ではない場合，Producer の実行終了が Consumer のフェッチ開始に間に合わないため，フォワーディングを行うことはできない．フォワーディングを行うために必要なコード上の距離は，おおよそパイプライン段数とフェッチ幅の積によって決定されるが，通常 20 から 40 命令程度となる．一般に，コンパイラによって依存関係にある特定の命令をこのように大きく離してスケジューリングすることは困難であり，レジスタ間接分岐ターゲット・フォワーディングではこの点が問題となった．

そこで，Producer と Consumer が十分に離れておらず，フォワーディングを行えない場合，分岐予測ミスからの早期リカバリを行う．図 3.6 に早期リカバリを行った場合の実行の様子を示す．図 3.6 では，Producer(cmp) と Consumer(br) が十分に離れていないため，Producer の実行終了が Consumerno フェッチに間に合わない．このため，Consumer の後続の命令は，従来と同様に分岐予測に

図 3.4: 分岐予測が外れた場合のパイプライン（より距離を離した場合）

よってフェッチを行う．Producer の実行が終了した場合，対となる Consumer の条件判定を早期実行する．この結果が Consumer の分岐予測結果と異なった場合，その時点でリカバリを行う．これにより，Consumer の実行が終了まで待つことなく，予測ミスからのリカバリを行うことが可能となる．

3.4　実装と動作

以下では，レジスタ間接分岐ターゲット・フォワーディングや分岐プレディシジョンを実現するための実装について述べる．レジスタ間接分岐ターゲット・フォワーディングや分岐プレディシジョンは，おおよそ以下の 3 つのテーブルを組み合わせて用いることによって実現される．

- **RPT:Register Producer Table**　各論理レジスタを最後に更新した命令のアドレスを保持する．

- **BCT:Branch Consumer Table**　BCT は Producer に対応する Consumer のアドレスと条件判定の種類を保持する．レジスタ間接分岐ターゲット・フォワーディングの場合は，条件判定を行わないため，Consumer のアドレスのみを保持する．

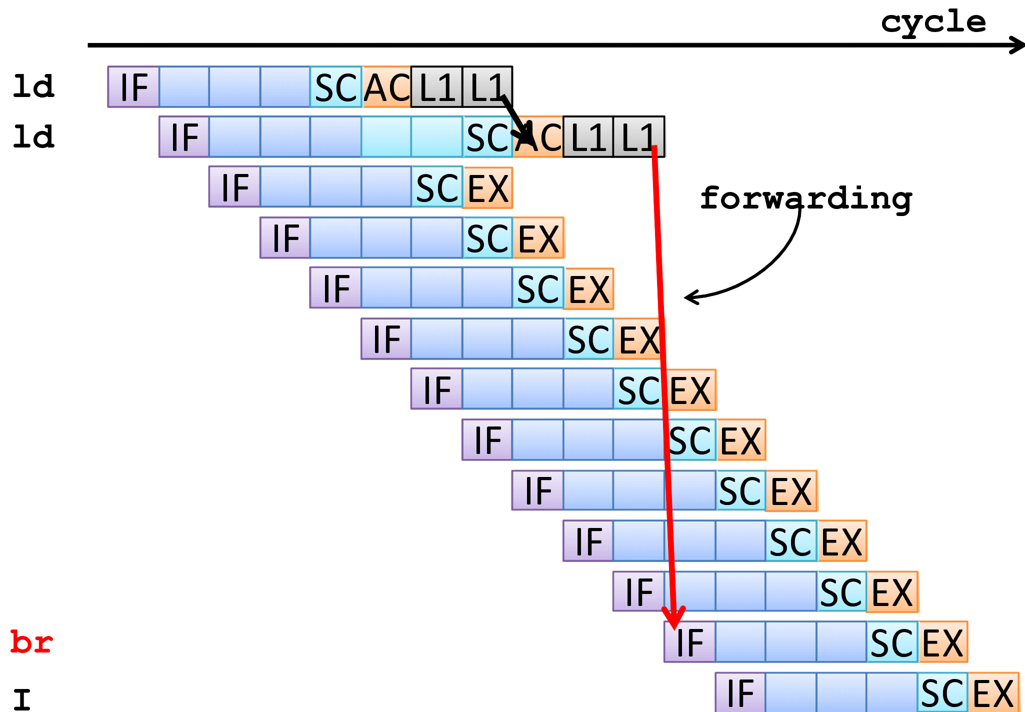

図 3.5: フォワーディングを行った場合のパイプライン

- **BFB:Branch Forwarding Buffer**　BFB はフォワーディングを行うための分岐命令の結果を保持する．レジスタ間接分岐ターゲット・フォワーディングの場合は分岐先ターゲットのアドレスが，分岐プレディションの場合は分岐の成立/不成立が格納される．

3.4.1　学習とフォワーディング

以下では，上記のテーブルやバッファを用いた動作を，フォワーディングを行うための学習とフォワーディングそのものに分けて説明を行う．

Producer と Consumer の組み合わせの学習　学習を行う際の動作を，図 3.7 に示す．同図では，アドレス 0010 にある load 命令が Producer であり，アドレス 0020 にある br 命令が Consumer である．この学習は，以下のように RPT の更新と，BCT の更新の 2 つを経て行われる．

1. **RPT の更新**　Procuder のデコード後，そのディスティネーション・レジスタ番号 r1 に対応するエントリに，Producer の命令アドレスを書き込む．

2. **BCT の更新**　Consumer のデコード後，そのソース・レジスタ番号 r1 を用いて RPT を引き，対応する Producer の命令アドレスを得る．得られた

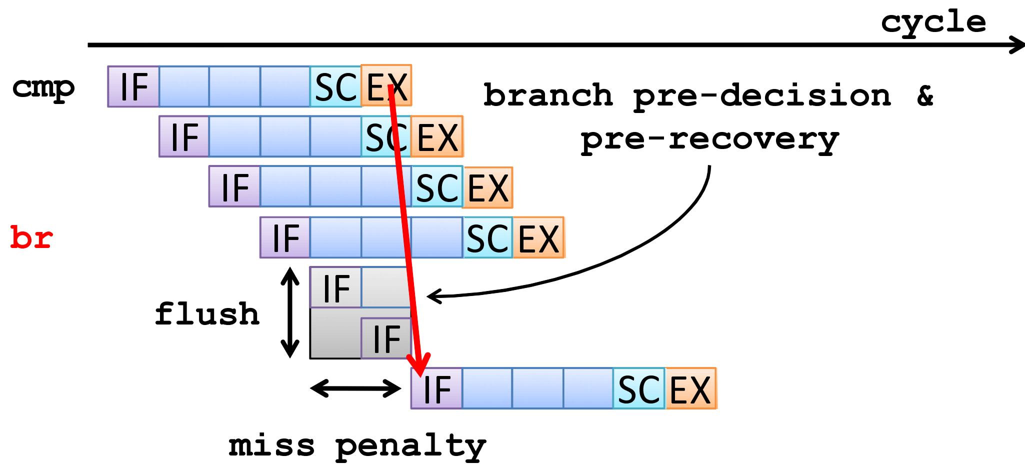

図 3.6: 早期リカバリ

Producer の命令アドレスに対応するエントリに，Consumer の命令アドレスを書き込む．分岐プレディシジョンの場合は，さらに条件判定の種類も BCT に書き込む．

分岐先ターゲットのフォワーディング　フォワーディングを行う際の動作を図 3.8 に示す．フォワーディングは，以下のように BCT と，BFB を用いて行われる．

1. **Consuemr 情報の取得**　BCT を Producer の命令アドレスを用いて引く．対応するエントリがテーブルに存在し，ヒットした場合には Consumer のアドレスや条件判定の種類を得る．
2. **分岐結果の書き込み**　Producer の実行時に得られた分岐結果を，先に得られた Consumer のアドレスを用いて BFB に書き込む．
3. **分岐先ターゲットの取得**　Consumer のフェッチ時に，その命令アドレスを用いて，BFB を引く．これにより，Producer によって生成された分岐結果が得られる．この時得られるのは，レジスタ間接分岐ターゲット・フォワーディングであるならば分岐先ターゲットのアドレス，分岐プレディシジョンなら分岐先方向である．

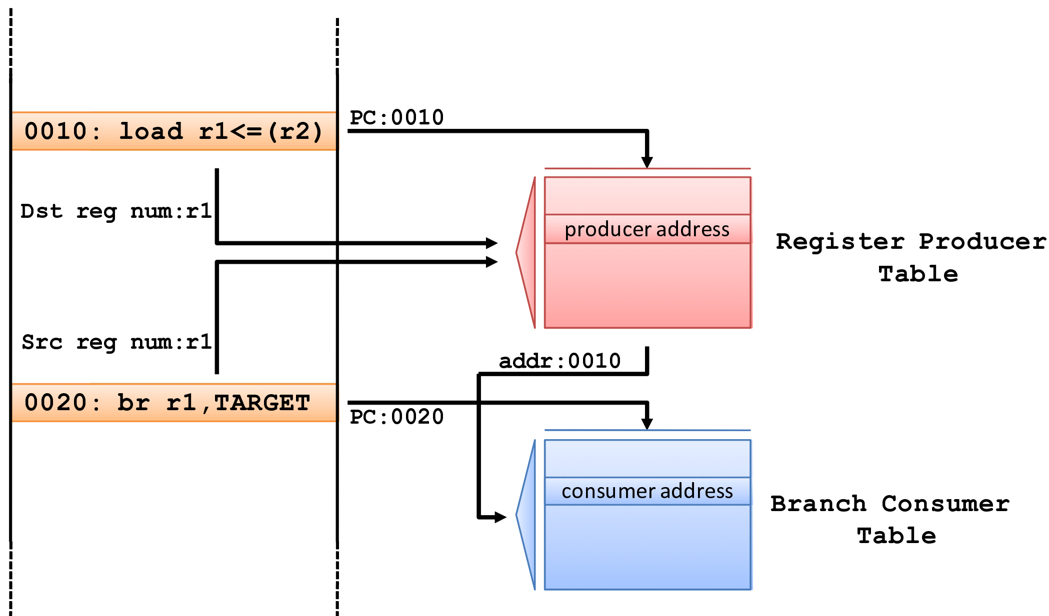

図 3.7: 学習の動作

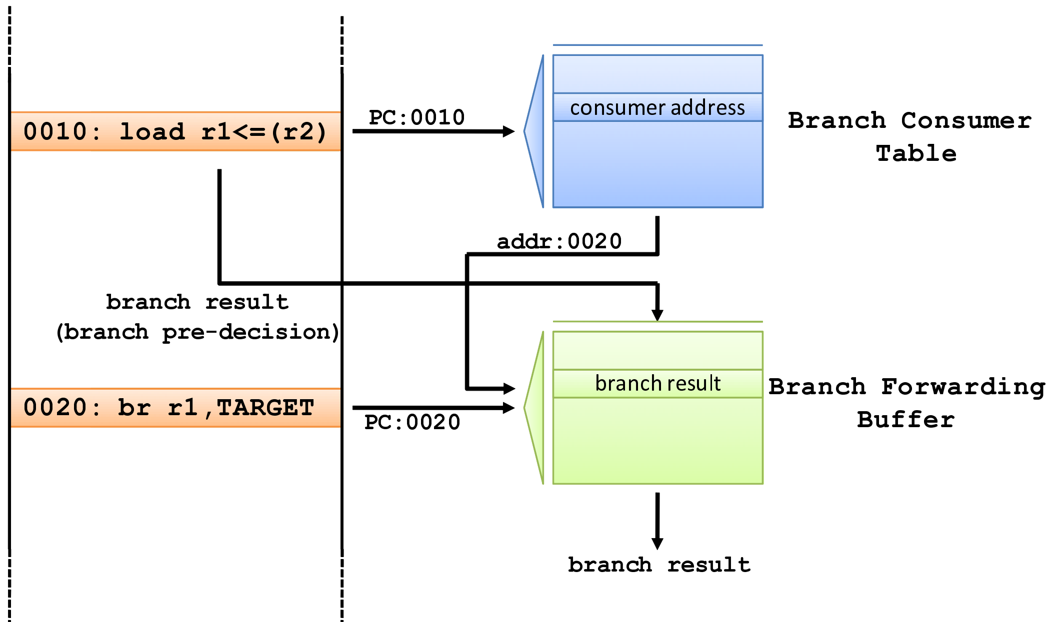

図 3.8: フォワーディングの動作

第4章　ヘルパースレッディング

従来からマルチスレッドプロセッサで単一スレッドのプログラムの実行を高速化する手法としてヘルパースレッディングの研究が行われている．ヘルパースレッディングの研究例の多くはメインスレッドの命令の一部をヘルパースレッドとしてメインスレッドに先行して実行することで高速化を実現している．本研究が提案するヘルパースレッディングもヘルパースレッドをメインスレッドの先行スレッドとして用いる．

本章ではヘルパースレッドをメインスレッドに対する先行スレッドとして用いるヘルパースレッディングについて述べる．

4.1　概要

先行実行命令列の作成　ヘルパースレッドの実行はメインスレッドの実行より先行させるためには，ヘルパースレッドで先行実行する命令はメインスレッドで実行される命令よりも少なくしなければならない．そのため，ヘルパースレッドで実行を行う先行実行命令列の作成がヘルパースレッディングの性能を決める重要な点の一つとなる．

先行実行を行う命令の数は，メインスレッドの実行を妨げないという点では少なければ少ないほどよく，理想的には先行実行の結果を用いることでメインスレッドの実行が高速化される命令のみ先行実行を行えられればよい．しかし，対象の命令だけを先行実行することは不可能で，メインスレッドの実行に対してほとんど先行させることはできないため，対象の命令の依存元の命令を加えて複数の命令を先行実行することが必要となる．先行実行命令列の作成ではメインスレッドに対する先行度と先行実行命令の数との兼ね合いが問題になってくる．

また，作成するタイミングとしては実行前にコンパイラなどを用いて作成しておく方法と実行時の情報を用いて動的に作成する方法が考えられる．前者ではプロファイリング情報を基に複雑な処理や解析が行えるため効率的な命令列の選択ができると考えられるが，再コンパイルの必要があるといった欠点がある．後者はあまり複雑な処理を行うことはできないが，入力データなど実行時の情報を用いることができるという利点がある．

ヘルパースレッドの実行開始タイミング　ヘルパースレッドの実行開始のタイミングは先行実行命令列の作成と合わせてヘルパースレッドの先行度に大きく影響してくる．一般にはヘルパースレッドの実行に必要なデータを与える命令をトリガとして実行開始されることが多い．

実行結果の利用　ヘルパースレッドの実行結果の利用法としては分岐予測器の補助，ロードのプリフェッチが一般的である．

分岐予測は過去の分岐パターンや実行パス情報を用いて行われるため，それらに相関のない分岐命令は分岐予測が困難である．そこでヘルパースレッドで実際に実行した結果を用いて予測することで予測精度の向上を図っている．

ロードのプリフェッチでは実際に実行を行うことで，従来のプリフェッチャでは困難であった不規則パターンのメモリアクセスに対応できると考えられる．

いずれにしてもヘルパースレッドの実行はメインスレッドの実行にたいして十分前に終了している必要がある．そのため前述したように先行実行命令列の作成方法とヘルパースレッドの実行開始タイミングが重要となってくる．

4.2　ヘルパースレッディングの従来研究

本節では，ヘルパースレッディングの従来研究を紹介する．

4.2.1　Simultaneous Subordinate Microthreading

Chappell らの提案している Simultaneous Subordinate Microthreading (SSMT) [1] ではヘルパースレッドで実行される先行実行命令列をコンパイラによって作成する．コンパイラによって挿入される SPAWN 命令によってヘルパースレッドの実行が開始される．先行スレッドによって分岐予測の精度向上，ロードのプリフェッチ，キャッシュ管理の効果が得られると述べられており，その中で分岐に対する評価が行われている．プロファイリング情報によってコンパイル時に gshare 分岐予測器では予測精度が低くなってしまう，他の分岐命令との相関がない分岐命令に対してヘルパースレッドを作成する．ヘルパースレッドの実行によって得られた分岐履歴を利用し，分岐予測を行うことで分岐予測精度の向上を図っている．

4.2.2　Slipstream Processor

Slipstream Processor [5] ではマルチスレッド環境を利用して 2 つの実行ストリームを並列に実行する．一つはプログラムの進行に必要ない計算を動的にスキップして投機的に実行される．この実行ストリームを Advanced stream

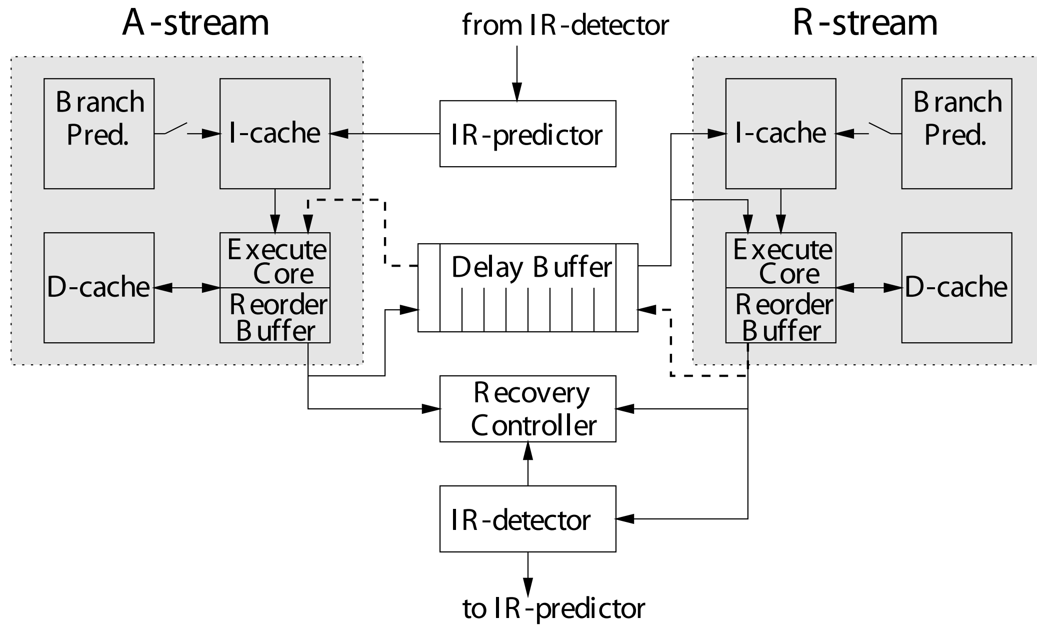

図 4.1: Slipstream processor using a two-way chip multiprocessor [11]

(A-stream) と呼ぶ．もう一方はプログラムをそのまま実行するストリームで Redundant stream (R-stream) と呼ぶ．つまり，A-stream が先行実行ストリームで R-stream が本実行ストリームとなる．R-stream は A-stream の実行のチェッカーとしての働きを持つ．A-stream は投機的に必要ないと予測される命令を取り除いて実行されるので，R-stream による予測ミスのチェックが必要である．一方，A-stream は R-stream にとってプログラムに基づく予測となる．A-stream での分岐予測ミスや実行結果を値予測として用いることで，R-stream が効率的に実行される．

Chip MultiProcessor [3] を利用した場合の Slipstream processor は Fig 4.1 のようになる．影のかかった範囲は通常のマルチプロセッサを構成する要素で，Slipstream processor は新たに 4 つの回路を必要とする．

1. Instruction-removal predicor (IR-predictor) は従来の分岐予測器に変更を加えたもので，インデックスには PC とグローバル分岐履歴との XOR を用いる．各エントリはタグと 2 ビットカウンタと確信度カウンタで構成され，1 つの基本ブロックの情報を保持する．タグは基本ブロックの始め命令の PC で IR-predictor を引いたときにそのエントリが現在の基本ブロックの情報であることを確認するために用いられる．2 ビットカウンタは基本ブロックの最後の分岐命令の分岐方向を予測するために用いる．確信度カウンタは基本ブロックの命令数だけあり，それぞれの命令が取り除いてもよい命令だと判断されたときにインクリメントされ，カウンタ

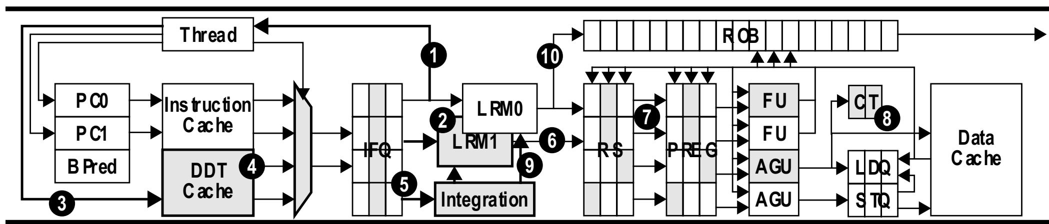

図 4.2: Hardware Implementation Aspects of DDMT [7]

が飽和している場合にはA-streamにおいて当該命令を取り除いて実行される．IR-detectorで取り除いてもよいと判断される命令は分岐予測器で正確に予測できる分岐命令，変更を起こさない命令，実行結果が参照されない命令がある．

2. Instruction-removal detector (IR-detector) はR-streamでリタイアされた命令とそのアドレスと実行結果を保持しておき，依存関係などからプログラムから取り除いてもかまわない命令を見つける．命令が取り除いてもよいと判断された場合にはIR-predictorの確信度カウンタをインクリメントする．

3. Delay BufferはA-streamからR-streamへ実行結果を渡すのに用いられる．

4. Recovery ControllerはA-streamにおいて取り除くべきではない命令を取り除いてしまった場合にA-streamの状態を復帰する機構である．

シミュレータによる評価では平均12%の性能向上が得られている．性能向上が得られているプログラムではA-streamでは50%程度の命令が取り除かれている．一方，性能が下がってしまっているプログラムでは10%程度しか命令が取り除かれていない．

Slipstream processorでは元のプログラムのコピーを用意して，実行していく中で，動的に制御フローに関係のない命令を取り除いていくことで先行実行する命令列を構築していく．したがってプログラムの初期の段階では先行実行する必要のない，無駄な命令を2重に実行していたことになる．また，先行実行は常に行われていることからも，先行実行の効率は悪いと考えられる．先行実行で得られたデータを予測としては使うが，値そのものを用いることはなく，先行実行は投機的に行われる．

4.2.3 Speculative Data-Driven Multithreading

RothらはSpeculative Data-Driven Multithreading (DDMT) [7] という先行実行手法を提案している．予測を失敗をする分岐命令とキャッシュ・ミスするロー

ド命令をクリティカルな命令と呼ぶ．DDMT ではクリティカルな命令のために独立したスレッド (Data-Driven Thread: DDT) を生成し，本スレッド (Control-Driven Thread: CDT) と並行して実行する．DDT はクリティカルな命令とその依存関係にある命令のみによって構成されているので CDT より早く結果を得ることができる．マルチスレッド環境としては Simultaneous Multithreading を採用しており，Fig 4.2 のような構成である．

DDT の作成はプロファイリング情報を利用する．まずはじめにプロファイリング情報を元にクリティカルな命令を DDT に加える．次に依存関係にある命令を加えていく．間に別のクリティカルな命令があった場合はその命令も DDT に加えられる．命令の追加は現在の DDT の最初の命令と依存関係にある命令からはじめに加えたクリティカルな命令までの実行完了までのサイクル数を DDT と CDT について計算し，その差が十分大きくなったところで終える．このサイクル数の計算には DDT,CDT それぞれのフェッチ幅も考慮に入れる．このため DDT は CDT に比べて十分早く実行を完了することが保証される．

DDT は，DDT の最初の命令のソースオペランドを生成する命令がリネームされた時点をトリガとしてフォークされる．大量の DDT がフォークされないように，トリガとなれる命令は CDT の命令だけになっている．つまり，DDT の命令は他の DDT のトリガ命令にはなれない．DDT がフォークされると，CDT のレジスタマップテーブル (RMT) がコピーされ，DDT からのフェッチがスケジューリングされる．DDT はプログラムの状態を変更しないため，リオーダ・バッファ(ROB) やロード・ストア・キュー (LSQ) のエントリは割り当てられない．

DDT の実行結果はレジスタ統合 [6] を利用することによって，CDT や他の DDT から利用できるようになっている．命令はリネーム時に Integration Table(IT) と呼ばれる表を引いて，他の DDT または CDT がその命令を発行したかどうかを調べ，発行済みであればその命令は発行せず実行終了する．

DDMT は分岐予測ミスしそうな分岐命令やキャッシュ・ミスしそうな命令を元に，それらの命令が依存する命令を追加していくことで先行実行する命令列を構築する．DDT に含むことができる最大サイズは 32 命令としてあり，数十から百命令程度先の命令を先行実行できると考えられる．つまりロード命令は 1 次キャッシュ・ミスによる性能低下を隠蔽できる．DDMT の問題点としては DDT の実行結果の統合率が 3 割以下と低いことがあげられる．これは DDT のトリガ命令をリネームした時点ではその先のパスが予測できないために，無効な命令を実行してしまうためであると考えられる．またレジスタ統合によって時間短縮ゲインは得られているが，そのために DDT の実行結果を保持しておく大量の物理レジスタが必要であることも欠点としてあげられる．シミュレーションでは物理レジスタは 512 エントリとなっている．

図 4.3: Pipeline organization of dynamic speculative precomputation [2]

4.2.4 Dynamic Speculative Precomputation

Collins らの提案している Dynamic Speculative Precomputation (DSP) [2] は動的にループ内で頻繁にキャッシュ・ミスをするロード命令を見つけ，Precomputation slice (P-slice) という命令列を抜き出し，Simultaneous Multithreading [13, 12] を利用して先行実行し，データのプリフェッチを行う．Fig 4.3 は DSP プロセッサのモデルを示す．グレイになっている要素が DSP の機能に必要な構成要素である．破線で示されている Slice Cache は必ずしも必要ではないが，性能を向上させる．

Delinquent Load Identification Table (DLIT) は実行時に頻繁にキャッシュ・ミスをするロード命令を明らかにするハードウェアである．2 次キャッシュをミスしたロード命令が DLIT に追加される．DLIT に追加された命令の中で少なくとも 100 回以上実行され，平均で 4 サイクル以上そのスレッド内で最も古い命令となっていたロード命令を頻繁にキャッシュ・ミスをするロード命令とする．それらに対して P-slice を作成し，先行実行をする．P-slice の作成には Retired Instruction Buffer (RIB) を用意する．RIB は大きいトレース・キャッシュのようにコミットされた命令を保持する．それぞれの命令について PC とソースとディスティネーションの論理レジスタ番号を保存する．また P-slice に追加された命令であることを示すための “marked bit” を持つ．RIB の動作には 3 つの状態がある．一つ目の状態はアイドル状態で，多くのアプリケーションでは 99%以上 RIB はこの状態であり，それほどパワーを必要としないハードウェアである．DLIT で頻繁にキャッシュ・ミスをするロード命令と判断された命令がコミットされたときに RIB は命令収集モードに入る．このモードではその命令のスレッドがコミットする命令を保存していく．そして，そのロード命令の 2 つ目のインスタンスがコミットされたときに RIB は slice 作成モードに入り，対象のロード命令の依存先の命令の marked bit をセットすることでその命令を P-slice に加えることで P-slice を作成する．作成された P-slice は Slice Information Table (SIT) で扱われる．SIT には毎サイクルデコードされた命令のアドレスが伝えられ，P-slice の実行のトリガとなる命令がデコードされたら，次のサイクルで P-slice の実行が開始される．

P-slice の実行のトリガとなる命令は，簡単には，一つ前のイタレーション

表 4.1: Comparison of researches

	The timing of selecting instructions	Trigger of helper thread	The use of helper thread
SSMT	static	trigger instruction	branch pred.
Slipstream	dynamic	always	branch pred. and value pred.
DDMT	static	trigger instruction	integration and data prefetching
DSP	dynamic	trigger instruction	data prefetching
Master/Slave	static	trigger instruction	check for distilled program

の P-slice が対象としているロード命令にすることができる．しかし，これではキャッシュ・ミスの一部しか隠蔽できないことが多い．そこで 3 つの改善手法が提案されている．一つ目は，トリガとなる命令を一つ前のイタレーションの命令で，P-slice の最初の命令のソースを与える命令にする手法である．これにより，数命令分だけ P-slice の実行を早めることができる．2 つ目はループを展開して数個前のイタレーションの命令をトリガとする手法である．この場合 P-slice にはループを展開した分だけ，P-slice 中の命令が依存する命令が追加される．3 つ目は P-slice の連鎖で，P-slice の実行の終わりを次のイタレーションの P-slice のトリガとする手法である．

DSP では，2 次キャッシュ・ミスをしたロード命令を元にその依存する命令を追加していくことで P-slice を作成する．P-slice はあるトリガ命令によって起動される．また，先行実行によって得られたデータは使われることはなく，プリフェッチの効果だけを利用する．

4.2.5 Master/Slave Speculative Parallelization

Zille らはプログラムを省略して実行し，別スレッドで並列して検証を行うことで実行を高速化する手法を提案した [15]．Master と複数の Slave と呼ばれるプロセッサを用意し，Master では distilled program と呼ばれる，省略されたプログラムを実行する．そして checkpoint ごとに，本来のプログラムを Slave で実行する．Master と Slave の実行結果を検証し，違っていれば状態を回復し，合っていれば本来のプログラムよりも早く実行が進む.

4.3 従来研究のまとめ

上で述べた従来研究の比較を表 4.1 に示す.

4.3.1　先行実行命令列の作成

SSMT，DDMT，Master/Slave はプログラムの実行前にコンパイラやプロファイリング情報を利用してヘルパースレッドで実行する先行実行命令列を作成する．一方，Slipstream Processor，Dynamic Speculative Precomputation では実行時にヘルパースレッドを作成する．

先行実行命令列の作成方法はSSMT，DDMT，Dynamic Speculative Precomputation が同じような形をとっており，まず先行実行するべき命令と特定し，その命令の依存元の命令を加えていくことで作成する．この方法はどこまでの命令を先行実行命令列に加えるかが重要な点になる．

Slipstream Processor は他の手法とは違っていて，始めは全く同じプログラムを実行して，実行中に片方のスレッドから命令を取り除いていく．この方法では取り除く命令に限界があり，ヘルパースレッドで実行する命令の数が多くなってしまうという問題点がある．

4.3.2　ヘルパースレッドの実行開始タイミング

従来の研究例ではコンパイラよって実行開始命令が挿入されているか，先行実行命令列の最初の命令のソースオペランドの値を与えるメインスレッドの命令がトリガとなって実行を開始する．そのため，ヘルパースレッドの実行開始のタイミングは限定されており，メインスレッドに対して十分先行して実行完了するように先行実行命令列を作成しなければならない．メインスレッドに対してより先行させようとすると，先行実行命令の数を多くしなければならなくなり，ヘルパースレッドの実行に資源をより割かなければならなくなるといった欠点が考えられる．

4.3.3　ヘルパースレッドの実行結果の利用

SSMT と Slipstream Processor ではヘルパースレッドでの先行実行の結果を分岐予測に用いることで分岐予測の精度向上を図っている．Dynamic Speculative Precomputation ではロードのプリフェッチによってメモリアクセスのレイテンシを隠蔽している．DDMT は分岐命令に対しては分岐予測ミスを早期に発見することで分岐予測ミスペナルティを削減している．ロード命令に対してはプリフェッチの効果がある．また，ヘルパースレッドで実行された命令の結果は他のスレッドで再利用することで同じ命令を再実行しない．しかし，このためにハードウェアが複雑になってしまっている．

図 4.4: Processor model used for description and evaluation of runahead [4]

4.4 ヘルパースレッドを用いない先行実行手法

本節ではヘルパースレッドを用いずに先行実行を行い，ヘルパースレッディングと同様の効果を得る従来研究を紹介する．マルチスレッド環境を利用するヘルパースレッディングに対して，別スレッドを生成しないため，コンテキストを消費しないという利点がある．

4.4.1 Runahead Execution

Mutlu らは Runahead 実行 [4] という先行実行手法を提案している．この手法では先行実行と本実行が同時に実行されることはない．先行実行を行うときは Runahead モード，本実行は Normal モードと呼ぶ．命令ウィンドウが長いレイテンシの命令によってブロックされたときに，状態をチェックポイントし，プロセッサは Runahead モードに入る．Runahead モードに入ると，その原因となった命令の後の命令のフェッチ，実行，そして見せかけのリタイアが行われる．原因となった命令が完了したときには状態をチェックポイントしたときに戻し，Normal モードに入る．この Runahead 実行により正確なデータのプリフェッチを行う．

Runahead 実行を行うプロセッサの構造を Fig 4.4 に示す．影のかかった部分が Runahead 実行のために新たに必要となる構成要素である．まず Runahead

モードに入るときの処理について説明する．データ・キャッシュ・ミス，命令キャッシュ・ミスなどをトリガとしてプロセッサはRunaheadモードに入る．このとき，Runaheadモードを抜けるときのためにレジスタの状態のチェックポイントを作成する．

次にRunaheadモードでの命令の実行について説明する．それぞれの物理レジスタにはその値が無効であるかどうかを判断するためにinvalid(INV) bitを追加する．無効なソース・オペランドを持つ命令は無効な命令とする．INV bitは偽のプリフェッチや偽のデータを使っての分岐の決定を防ぐのに使われる．無効なデータを生み出す最初の命令はRunaheadモードに入る原因となった命令である．無効な命令のディスティネーションレジスタはINV bitをセットし，無効でない命令のディスティネーションレジスタはINV bitをリセットする．このように命令を無効な命令とそうでない命令に分け無効でない命令を実行していく．分岐命令が無効な命令となった場合は，分岐予測によって投機的に分岐先を決定する．

Runaheadモードからの復帰は分岐予測ミスからの復帰と同様の方法で行われる．プロセッサ内にある全ての命令をフラッシュし，物理レジスタの状態をチェックポイント時に戻す．

この手法では，ある命令がキャッシュ・ミスをしている間に，無効でないデータを用意できる命令だけを先行実行するため数100命令以上先の命令の先行実行を行うと考えられる．そのため2次キャッシュ・ミスに効果が得られる．Runaheadモードに入るときに，状態をチェックポイントする等，オーバーヘッドが大きいため，1次キャッシュ・ミスのように比較的低レイテンシであるキャッシュ・ミスにおいては有効性が低いと考えられる．Runahead実行の先行実行は実行できる命令はすべて実行する．したがって先行実行してもあまり効果のない命令まで実行していると考えられる．

4.4.2　物理レジスタ2段階解放

Yamamotoらは物理レジスタを2段階に分けて解放することによって先行実行を行い，プリフェッチやメモリ曖昧性を早期に除去する手法[14]を提案している．この手法は別スレッドを生成せず，単一スレッドで先行実行を実現している．

1段階目の物理レジスタの解放はリネーム・ステージで行われる．解放は従来のマップ表とフリー・リストに加えて解放表(DAT: Deallocation Table)と呼ばれる表で管理する．この表の各エントリは物理レジスタに対応し，当該物理レジスタを解放する命令が登録されるリオーダ・バッファ(ROB)のエントリ番号を保持する．

動作はまず，命令の論理デスティネーション・レジスタに現在割り当てられている物理レジスタを解放し，フリー・リストに追加する．この時，解放する

物理レジスタに対応するDATのエントリに，自身が割り当てられたROBのエントリ番号を書き込む．また，従来と同じく，フリー・リストより空き物理レジスタを得，論理デスティネーション・レジスタに割り当てる．この時，DATを参照し，当該物理レジスタを解放するROBエントリ番号ROBPを得る．ROBPは2段階目のレジスタの解放タイミングを知るための命令のタグとなる．

2段階目の解放は，命令のコミット時に行われる．この時解放される物理レジスタは，従来と同じく，コミットされる命令の論理レジスタに以前に割り当てられていた物理レジスタである．従来と異なるところは，解放の際に，ROBのエントリ番号ROBPを命令ウィンドウに放送する点である．放送されたROBPが，命令ウィンドウで待ち合わせている命令のROBPタグと一致したなら，その命令は実行結果の書き込みを許可される．

命令は，命令ウィンドウにおいて，自身のデスティネーション物理レジスタの2段階目の解放を待ち合わせる．しかし，書き込み許可を得る前に，ソースオペランドが利用可能となったとき一度だけ実行することによって本実行に先立つ先行実行ストリームを生成する．先行実行によって得られた結果は物理レジスタに書き込むことはできないのでバイパス論理経由で依存命令に渡すことになる．これによってソースオペランドが揃った命令も先行実行され連鎖的に先行実行は進んでいく．先行実行は物理レジスタが十分にあるときの命令レベル並列性を利用して進むため，本実行よりも速く進む．

ロード命令の先行実行により，データ・キャッシュへデータがプリフェッチされる．これによって本実行時のデータ・キャッシュ・ミスを回避することができる．このプリフェッチは実際に命令を実行して行われるので，不規則なアクセス・パターンにも対応できる．また，メモリ命令のアドレス計算命令の先行実行によりアドレスをロード・ストア・キュー書き込むことで，ロードの本実行時には，先行するストア命令のアドレスが判明しており，メモリ依存の曖昧性を早期に除去できる．これによってもロード命令を早期に発行することができる．

この手法では命令ウィンドウ内のソースオペランドが揃った命令について先行実行を行うので，本実行に対して最大でも命令ウィンドウのエントリ数だけ先の命令の先行実行しか行えない．そのため主に1次キャッシュ・ミスに対して効果が得られると考えられる．先行実行を行う命令はRunahead実行と同じように実行できる命令すべてである．したがってRunahead実行と同じ欠点があるといえる．

第5章　繰り返し構造に着目したヘルパースレッディング

本研究で提案するヘルパースレッディングは，プログラムの繰り返し構造に着目し，値予測を利用してヘルパースレッドの作成と実行を行う．ヘルパースレッドは実行時に動的に作成，実行開始され，キャッシュミスをするロード命令と分岐予測ミスをする分岐命令をメインスレッドに対して先行して実行する．ロード命令の先行実行では関連研究と同様にプリフェッチの効果があると考えられる．分岐命令の先行実行では実行結果を分岐プレディシジョンの補助として使うことにより，分岐予測ミスペナルティを削減する．

本章では，値予測を用いたヘルパースレッドの作成と実行と，分岐プレディシジョンの補助について説明する．

5.1　ヘルパースレッドの作成と実行

5.1.1　先行実行命令列の作成

先行実行命令列の作成は分岐予測ミスをする分岐命令やキャッシュミスをするロード命令を検出するところから始まる．これは実行時に分岐予測ミスとキャッシュミスをカウントする表を用意することで実現する．

先行実行を行うべき命令が検出した後，従来研究と同様にその命令の依存元の命令を先行実行命令列に加えていくことで作成していく．命令を加えていく時にその命令が値予測可能かどうかを調べ，可能であればその命令の依存元の命令を加えないという点が従来と違っている．具体的な手順は以下のようになる．

1. 分岐予測ミスする分岐命令，またはキャッシュ・ミスをするロード命令を検出する．

2. (1) で検出した命令のソースレジスタを探索するソースレジスタ群に加える．

3. 検出した命令から上に見ていき，探索するソースレジスタ群をデスティネーションとする命令を探す．

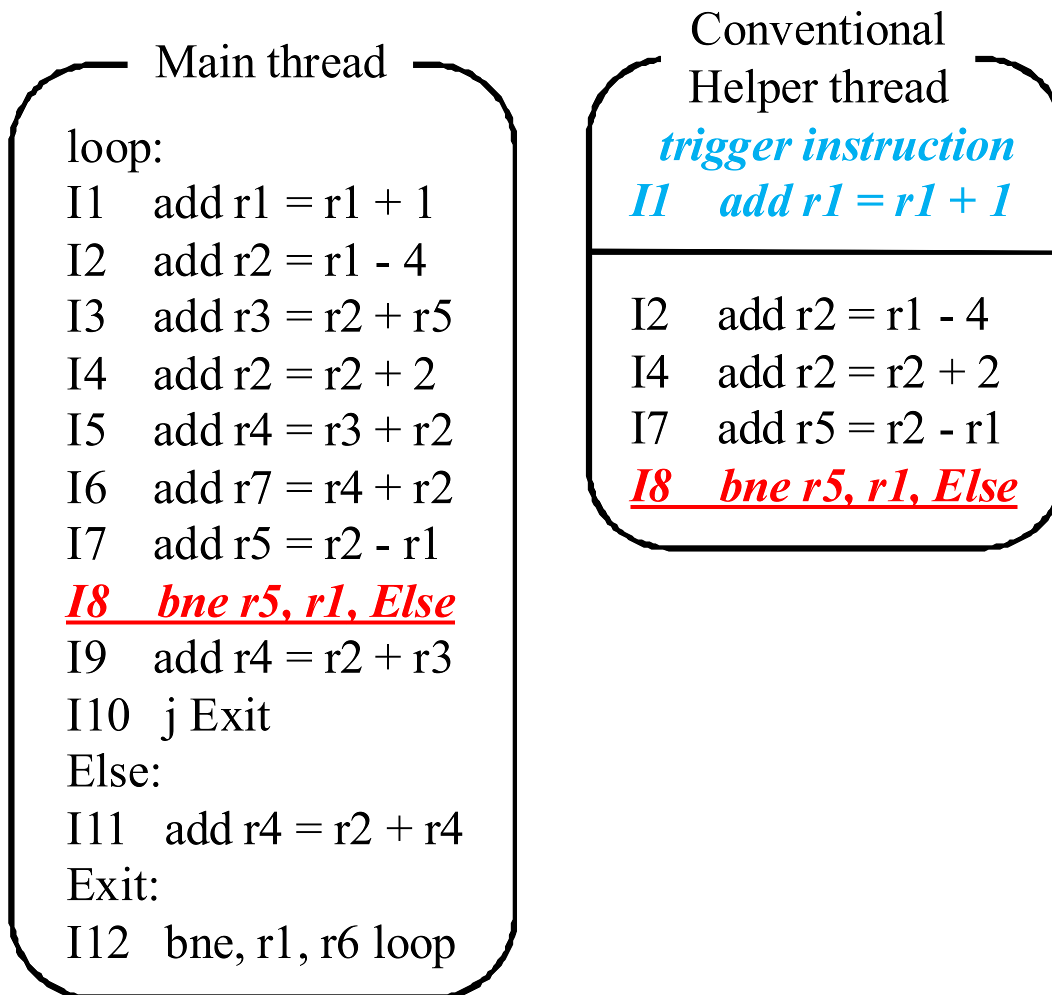

図 5.1: 従来のヘルパースレッドの例

4. (3) で見つけた命令のデスティネーションレジスタを探索するソースレジスタ群から外す
5. (3) で見つけた命令が値予測可能でなければ，そのソースレジスタを探索するソースレジスタ群に加える．
6. 探索するソースレジスタ群の数が 0 になるまで，3-5 を繰り返す．

(3) においてループを一回りして，手順 1 で特定した命令まで達したときに探索するソースレジスタ群の数が 0 にならなければ，ヘルパースレッドの生成が失敗したとする．

従来との違いを図 5.1，図 5.2 を例にして説明する．どちらの図も左側の命令列がメインスレッドで実行される命令で，右側の命令列がヘルパースレッドで実行される命令である．まず，このループで I8 の分岐命令が頻繁に分岐予

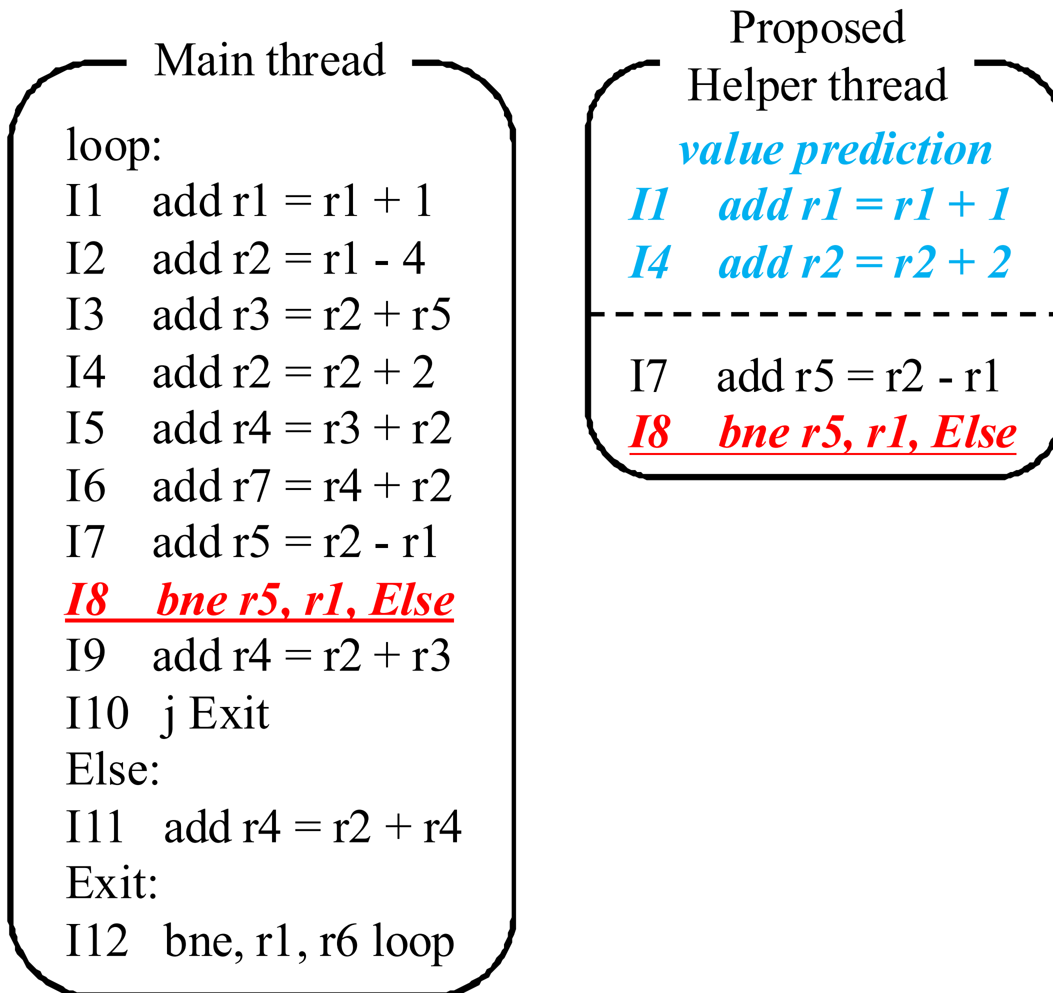

図 5.2: 値予測を用いたヘルパースレッドの例

測を失敗する命令であると分かったとする．従来のヘルパースレッドではこの分岐命令の依存元の命令を見ていくとI7が見つかる．さらに上に見ていくとI4，I2，I1という順に依存元の命令が見つかる．ここではI2までを先行実行命令列に加えてI1をヘルパースレッドの実行のトリガとなる命令とする．

値予測を用いた手法においてもI8の分岐命令の依存元の命令を探索していく．ここではI1とI4の命令が値予測可能であるとすると依存元の命令はI7，I4と順に見つかるが，I4は値予測可能であるためI4の依存元の命令であるI2は先行実行命令列に加えない．

この例においては従来のヘルパースレッドではメインスレッドがI2からI8まで7命令のところ，ヘルパースレッドは4命令となり3命令分先行して実行を行うことが期待できる．一方，値予測を用いた場合には値予測を用いて好きなタイミングで実行を開始できるため，従来よりも先行して実行を行うことができると考えられる．

5.1.2 ヘルパースレッドの実行開始タイミング

従来研究では先行実行命令列のはじめの命令が依存している命令がトリガとなってヘルパースレッドが実行開始されることが多い．そのため，メインスレッドに対してどの程度先行できるかはある程度限定されてしまう．

本研究のヘルパースレッドは 5.1.1 で述べたように値予測を前提として先行実行命令列を作成するので，メインスレッドのある特定の命令をトリガとする必要がない．先行実行命令列を作成した時点で対象の分岐命令やロード命令はメインスレッドで近い将来に再度実行されると予想して，ヘルパースレッドの実行を開始する．これによってヘルパースレッドの実行はメインスレッドに対して十分先行して完了することが期待できる．

5.2 分岐プレディシジョンの補助

??で述べたように分岐プレディシジョンでは，分岐命令とそれが依存している命令が十分離れていないと効果が薄い．そこでヘルパースレッドの実行結果を分岐プレディシジョンの補助として利用することを考える．5.1 で説明したように提案するヘルパースレッディングではヘルパースレッドの実行に値予測を用いることによって，メインスレッドと独立に実行開始できる．したがって，ヘルパースレッドの分岐命令の実行がメインスレッドの分岐命令のフェッチよりも前に完了するようにヘルパースレッドの実行を開始することが可能である．メインスレッドではヘルパースレッドの実行結果を利用することによって，より通る確率の高いパスをフェッチすることが期待できる．

分岐プレディシジョンでは実際の実行結果をフォワーディングするため，対象の分岐命令のフェッチに間に合えば確定したパスをフェッチすることができる．一方，ヘルパースレッドの結果を利用する場合にはヘルパースレッドの実行に値予測を用いているため確定したパスをフェッチすることにはならない．そのため，従来の分岐予測器による予測精度よりも，その分岐命令が依存している命令の値予測の予測精度が高い場合に効果があると考えられる．

第6章　評価結果

6.1　評価方法

研究室で開発したシミュレータである鬼斬弐[16] に対し，提案手法を実装して評価を行った．鬼斬弐はプロセッサのサイクル・レベル・シミュレータであり，一般にプロセッサ・アーキテクチャの研究で広く用いられている SimpleScalar ツールセット [8] と比較して，より精度の高いシミュレーションを行うことが可能である．

6.2　評価モデル

評価したプロセッサのパラメータを表 6.1 に示す．評価は以下の 2 つのモデルを実装して行った．

- **Base**　分岐予測器に GShare 予測器を用いた標準的なモデル．
- **Proposed**　BASE の分岐予測に先行実行の結果を用いた予測を加えたモデル．同時に実行するスレッドはメインスレッド 1 つ，ヘルパースレッド 1 つの合計 2 スレッドとした．また，先行実行は分岐予測がよく外れる分岐命令に対して行い，ロード命令に関しては対象としていない．

6.3　予備評価

図 6.1 のテストプログラムを用いて予備評価を行った．このテストプログラムの実行では，関数ポインタによって 2 つの関数が交互に呼び出されるため，通常の BTB による予測が行えない．一方，提案手法ではループ変数の i の値予測が容易であるため，ヘルパースレッドによる先行実行で呼び出す関数を確定できるため，効率的に実行できる．

図 6.2 はそれぞれのモデルにおいての BTB のヒット/ミス回数を表す．Base では 33%程度ミスをしているのに対して，Proposed ではほとんどミスをしていない．これによって，プログラムにこのような構造があれば，性能向上を図ることが期待できる．テストプログラムの IPC については呼び出される関数の長さによって向上率が変わり，予備評価の本質とは関係がないため割愛する．

表 6.1: プロセッサの構成

パラメータ	値
ISA	Alpha
fetch width	6 inst.
execution unit	int : 2, fp : 2, mem : 2.
instruction window	int : 32, fp : 16, mem : 16
main register file	int : 128, fp : 96
branch prediction	8KB g-share
BTB	2K entry, 4-way
RAS	8 entry
L1C	32KB, 4-way, 64B/line, 3 cycle
L2C	4MB, 8-way, 64B/line, 15 cycle
main memory	200 cycle

```
for(i = 0; i < N; i++)
{
    sum += func_tbl[i%2]();
}
```

図 6.1: 予備評価のテストプログラム

6.4　ベンチマーク

評価には，SPEC CPU2000[9] ベンチマーク・セットの中から，20 本のアプリケーションを使用した．アプリケーションのコンパイルには gcc 4.2.2 を用い，コンパイルオプションについては，SPEC CPU2000 に含まれる環境設定スクリプトが指定するものに従った．各ベンチマークの入力データ・セットには train を用い，最初の 1G 命令をスキップして，続く 100M 命令を実行した．

6.5　評価結果

Base に対する Proposed の相対 IPC を図 6.3 に示す．このグラフを見ると，Proposed ではほとんど性能向上が見られず，一部のベンチマークでは若干性能低下が見られる．この原因を以下で考察する．

図 6.4 は Proposed においてヘルパースレッドによる先行実行を行った分岐命令の分岐予測ヒット率を表している．この結果から性能向上が得られなかった理由が 2 つ考えられる．1 つは Base において十分に分岐予測が当たっている分岐命令に対しても先行実行をしてしまっている点である．一部のベンチマークでは Proposed において先行実行した分岐命令の，Base での分岐予測率がほとんど 1 に近くなっている．そのため，先行実行を行う命令数の分だけ性能低下してしまう．今回の実装では単純に分岐予測ミスの数だけをカウントして，先行実行を行う分岐命令を決めているため，実行回数が多い分岐命令が数% ミスした場合にも先行実行を行ってしまっている．これを解決するには，一定間隔でカウントをクリアするか，ヒット回数もカウントする方法が考えられる．

2 つ目の理由は，先行実行の結果の精度である．正しく先行実行ができていれば，先行実行の結果を用いた分岐予測は必ずヒットするはずであるが，そうはなっておらず，Base よりも分岐予測ヒット率が下がっているベンチマークもある．これは値予測の精度が悪いために，先行実行の結果が間違ってしまうといったことが考えられる．今回の実装では前回の値と，前回と前々回の値の差を保存して今回の値を予測し，合っていたら信頼度を上げるといった単純な値予測を用いた．この値予測の場合，例えば命令の結果が，0 が続いて不規則なタイミングで 1 になる場合には値予測を失敗することがよく起きてしまう．

別の理由としては，先行実行を行っている分岐命令の数が全体の命令数に対して少ないということが挙げられる．これについては，同時に実行するヘルパースレッドを複数にすることで改善できると考えられる．

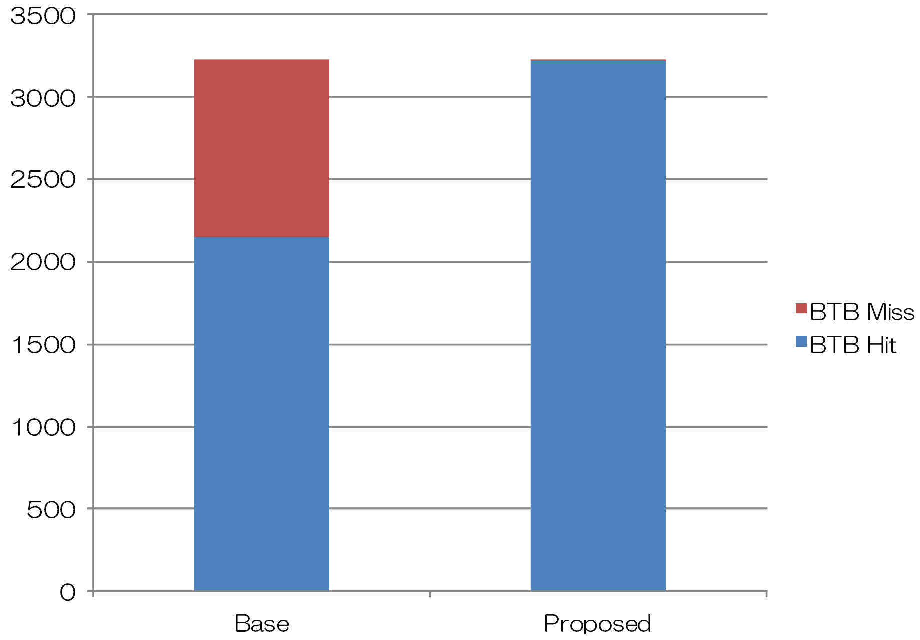

図 6.2: BTB の予測ヒット回数

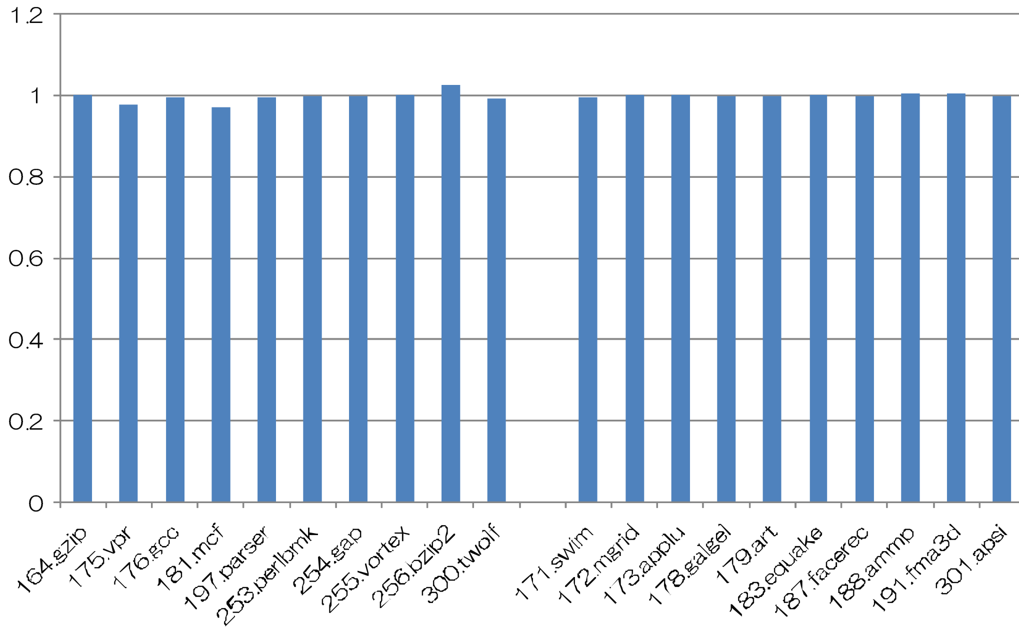

図 6.3: Base に対する Proposed の相対 IPC

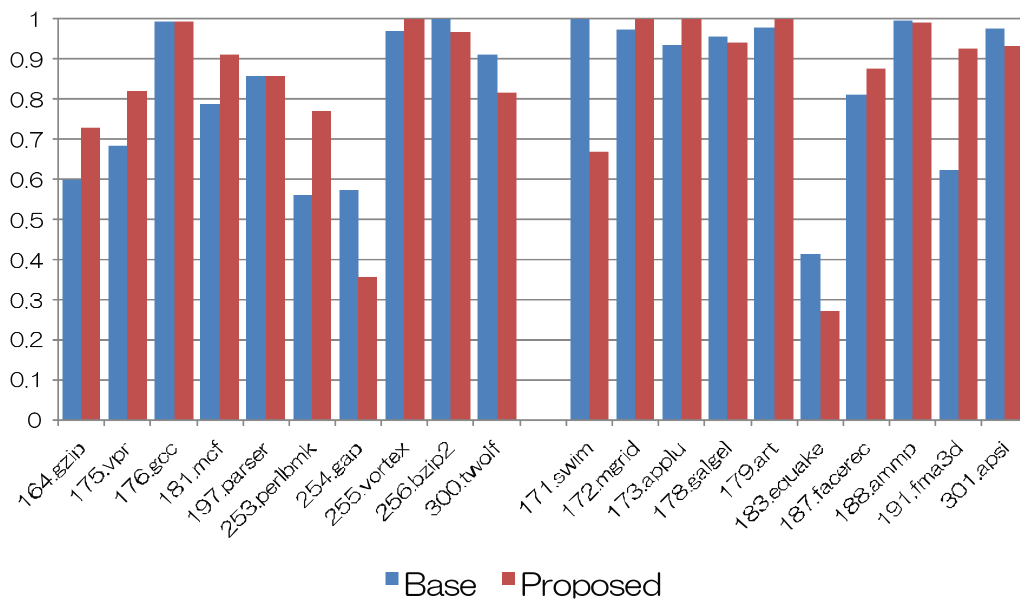

図 6.4: 先行実行を行った分岐命令の分岐予測ヒット率

第7章　おわりに

7.1　本研究のまとめ

本研究ではプログラムの繰り返し構造に着目した，動的なヘルパースレッディングを提案した．ヘルパースレッドはループ中の分岐予測ミスをする分岐命令とキャッシュ・ミスをするロード命令に対して生成され，値予測を利用して実行開始される．それによって従来のヘルパースレッディングと異なり，ヘルパースレッドがメインスレッドと独立に実行を開始することができる．そして，ヘルパースレッドの先行実行結果を分岐プレディシジョンの補助として用いることによって分岐先を早期に決定することができる．実際に予備評価ではそのことが示されている．

7.2　今後の課題

予備評価では性能向上が見られたが，ベンチマーク・プログラムを用いた評価では性能向上が見られなかった．この主な原因としてヘルパースレッドによって先行実行を行う分岐命令の選択手法と値予測の精度の 2 点が考えられる．

したがって，今後の課題として，分岐予測ミスをする分岐命令を選択する精度と値予測の精度を上げる手法を確立することが挙げられる．

参考文献

[1] R. Chappell, J. Stark, S. Reinhard S. Kim, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 186–195, May 1999.

[2] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pp. 306–317, December 2001.

[3] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, Vol. 30, No. 9, September 1997.

[4] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pp. 129–140, February 2003.

[5] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000.

[6] A. Roth and G. Sohi. Register integration: A simple and efficient implementation of squash re-use. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000.

[7] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceeding of the 7th International Symposium on High-Performance Computer Architecture*, January 2001.

[8] SimpleScalar LLC. *SimpleScalar version 3.0. http://www.simplescalar.com/*.

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

[10] The Standard Performance Evaluation Corporation. *SPEC CPU2006 suite http://www.spec.org/cpu2006/*.

[11] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of Architecture Support for Programming Language and Operating Systems*, November 2000.

[12] D. Tullsen, S. Eggers, J. Emer, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, pp. 392–403, June 1995.

[13] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementabel simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[14] A. Yamamoto, Y. Tanaka, H. Ando, and T. Shimada. Data prefetching and address pre-calculation through instruction pre-execution with two-step physical register deallocation. In *Proceedings of the 8th Workshop on Memory Performance: Dealing with Applications, Systems and Architectures*, pp. 41–48, September 2007.

[15] C. Ziles and Gurindar Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pp. 85–96, November 2002.

[16] 渡辺憲一, 一林宏憲, 五島正裕, 坂井修一. プロセッサ・シミュレータ「鬼斬」の設計. 先進的計算基盤システムシンポジウム SACSIS 2007 (ポスター), pp. 194–195, 2007.

[17] 豊島隆志, 入江英嗣, 五島正裕, 坂井修一. レジスタ間接分岐ターゲットフォワーディング. 先進的計算基盤システムシンポジウム (SACSIS), 第 2006 巻, pp. 325–332, 2006.

[18] 塩谷亮太, 五島正裕, 坂井修一. 分岐プレディシジョン. 情報処理学会報告 2008-ARC-179, pp. 67–72, 2008.

発表文献

主著論文

1. グラフィクスプロセッサを用いた 3 次元空間情報の実時間符号化手法の検討
   安藤 徹, 田口 裕一, 苗村 健
   3 次元画像コンファレンス 2007, 4-3, pp. 67–70, July 2007.

2. GPU-Oriented Light Field Compression for Real-Time Streaming
   Toru Ando, Yuichi Taguchi, Takeshi Naemura
   ACM SIGGRAPH 2007 Posters, no. 70, San Diego, CA, Aug. 2007.

3. プログラムの繰り返し構造に着目した動的なヘルパースレッディング
   安藤 徹, 塩谷 亮太, 五島 正裕, 坂井 修一
   情報処理学会報告 2008-ARC-179, pp. 139–144, Aug. 2008.

謝辞

非常に多くの方々のご協力のもと，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．

塩谷亮太氏には，研究内容や方針に関して，実に多くのアドバイスを頂きました．

八木原春水さん，伊世知代さん，長谷部環さん，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．

最後に，研究生活全般でお世話になった坂井・五島研究室の皆様，そして筆者を支えてくださった家族・友人の皆様に深く感謝いたします．ありがとうございました．