修士論文

スーパスカラ型自動メモ化プロセッサの提案および再利用予測による高速化

Implementation of Auto-Memoization Superscalar Processor with Reuse Prediction

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学大学院 工学研究科
修士課程 創成シミュレーション工学専攻
平成 25 年度入学　25413539 番

柴田 裕貴

平成 27 年 2 月 4 日

スーパスカラ型自動メモ化プロセッサの提案
および再利用予測による高速化

柴田 裕貴

内容梗概

ゲート遅延に対する配線遅延の相対的な増大から，微細化による高クロック化だけではマイクロプロセッサの性能向上を実現することは困難になってきた．こうした中で，SIMD やスーパスカラなどの命令間の並列性に着目した高速化手法が注目されるようになった．しかし，プログラム中に存在する命令間の並列性には限りがあることから，これらの手法にも限界がある．よって現在では，1 つの CPU に複数のコアを搭載したマルチコアプロセッサが主流となっている．このマルチコアプロセッサを有効活用するために，スレッドレベル並列性に着目した手法が多く研究されているが，そもそも並列性を持たずスレッドレベル並列性を抽出することが難しいプログラムがあるなど，様々な問題が存在している．そこで，計算再利用をハードウェアにより動的に適用する自動メモ化プロセッサに関する研究が行われている．

自動メモ化プロセッサは，関数およびループを計算再利用の対象区間と見なし，実行時にその入力と出力の組を再利用表に記憶しておく．そして，再び同一命令区間を同一入力を用いて実行しようとした際に，記憶しておいた出力を利用することで，その実行自体を省略する．並列化が処理全体の総量は変化させず複数の処理を同時実行することにより高速化を図る手法であるのに対し，計算再利用は処理自体を省略することで高速化を図る手法であり，その着眼点は根本的に異なっている．計算再利用は並列化とは直交する概念であるため，並列化が有効でないプログラムでも効果が得られる可能性があり，また並列化とも併用可能であるという利点がある．

さて，これまで自動メモ化プロセッサには，単命令発行でパイプライン化されていない単純なベースアーキテクチャが採用され，研究されてきた．しかし，現在市場に流通しているプロセッサにはこのような単純なアーキテクチャではなく，ハードウェア構成が複雑なスーパスカラアーキテクチャが広く採用されているため，これまで研究されてきた単命令発行型自動メモ化プロセッサの評価結果は必ずしも実用性を実証できていない可能性がある．そこで本論文では，自動メモ化プロセッサの実用性を確認するために，スーパスカラプロセッサ上に自動メモ化機構を実装する場合に発生しうる問題を検討し，そのような問題を回避するスーパスカラ型自動メモ化プロセッサの設計方式を提案する．例えば，再利用を適用する際に発生するオーバヘッドが引き起こすパイプラインストールを防ぐため

に，そのオーバヘッドと命令のパイプライン実行をオーバラップさせるように設計する．しかし，単純にこれらをオーバラップさせるだけでは，計算再利用適用時に無駄な命令を実行してしまうという新たな問題が発生する．そこで本論文では，設計したスーパスカラ型自動メモ化プロセッサに対する高速化手法として，計算再利用が適用可能か否かを予測することで，計算再利用適用時の無駄な命令実行を削減し，再利用による高速化の効果をより得られるようにする手法を併せて提案する．

スーパスカラ型自動メモ化プロセッサとそれに対する高速化手法の有効性を検証するために，SPEC CPU95 ベンチマークを用いてシミュレーションにより評価した．その結果，メモ化を行わず命令を実行する場合と比較して，既存の単命令発行型自動メモ化プロセッサのサイクル数削減率が，最大 17.4%，平均 5.1%であったのに対し，スーパスカラ型自動メモ化プロセッサのサイクル数削減率は，最大 13.1%，平均 2.4%となった．この結果から，スーパスカラ型自動メモ化プロセッサは既存の単命令発行型自動メモ化プロセッサと同様に，計算再利用による性能向上を達成できることがわかり，自動メモ化プロセッサの実用性を確認できた．また，高速化手法を適用したスーパスカラ型自動メモ化プロセッサのサイクル数削減率は，最大 13.6%，平均 2.7%となり，高速化手法の有効性を確認できた．

スーパスカラ型自動メモ化プロセッサの提案
および再利用予測による高速化

目次

1 はじめに

これまで，さまざまなプロセッサ高速化手法が提案されてきた．ゲート遅延が支配的であった時代には，集積回路の微細化によるクロック周波数の向上により，マイクロプロセッサの高速化を実現できた．しかし，ゲート遅延に対する配線遅延の相対的な増大や，集積回路の微細化に伴う消費電力および発熱量の増大といった問題から，マイクロプロセッサのクロック周波数の向上は困難になってきている．こうした中で，SIMD やスーパスカラなどの命令間の並列性に基づく高速化手法が注目されてきた．しかし，一般にプログラム中から命令間の並列性を抽出することは難しく，また，プログラム中に存在する命令間の並列性にも限りがあることから，これらの手法にも限界がある．よって現在では，消費電力や発熱量の問題を解決しつつプロセッサあたりの処理能力を向上させるため，SPARC T5 [1] や Opteron [2] などの，1 つの CPU に複数のコアを搭載したマルチコアプロセッサが主流となっている．また，ARM [3] などの，汎用プロセッサと比較して消費電力や発熱量による制約がさらに厳しい組み込み向けプロセッサにおいても同様にマルチコア化が進んでいる．そして，100 コア構成の TILE-Gx [4] が予定されているように，今後集積度の向上に伴って，1 つの CPU に搭載するコア数をさらに増大させたメニーコアプロセッサが一般化していくと予想されている．このため，複数のコアを用いてプログラムを高速化する一般的な手法として，スレッドレベル並列性（TLP: Thread Level Parallelism）に着目した様々な手法が研究されている．これらの手法は，プログラムを複数スレッドで処理できるように分割し，それらをそれぞれのコアに割り当てることで高速化を図っている場合が多い．例えば，並列処理ライブラリを用いてプログラマが明示的にプログラムを並列化したり，自動並列化コンパイラ [5,6] によりプログラムを複数のコアに対して自動的に割り当てるような手法が挙げられる．しかし，そもそも並列性を持たず TLP を抽出することが難しいプログラムが存在することや，プログラマが明示的に並列処理プログラムを記述することは困難である場合が多いなど，様々な問題が存在している．

これら既存のプロセッサ高速化手法は，粒度の違いはあれど，いずれもプログラムの持つ並列性に着目したものである．これに対し，計算再利用技術に基づいた高速化手法である自動メモ化プロセッサ [7] が提案されている．自動メモ化プロセッサは，関数およびループを計算再利用の対象区間と見なし，実行時にその入力と出力の組を再利用表に記憶しておく．そして，再び同一命令区間を同一入力を用いて実行しようとした際に，記憶しておいた出力を利用することで，その実行自体を省略する．並列化は処理全体の総量は変化させず複数の処理を同時実行することにより高速化を図る手法である．その一方で，

計算再利用は処理自体を省略することで高速化を図る手法であり，その着眼点は根本的に異なっている．計算再利用は並列化とは直交する概念であるため，並列化が有効でないプログラムでも効果が得られる可能性があり，また並列化とも併用可能であるという利点がある．

さて，これまで自動メモ化プロセッサには，単命令発行でパイプライン化されていない単純なSPARC V8アーキテクチャがベースアーキテクチャとして採用され，研究されてきた．しかし，現在市場に流通しているプロセッサにはこのような単純なアーキテクチャではなく，ハードウェア構成が複雑なスーパスカラアーキテクチャが広く採用されていることから，これまで研究されてきた単命令発行型自動メモ化プロセッサの評価結果は必ずしも実用性を実証できていない可能性がある．そのため，市場のプロセッサ事情に即した，より実用的な環境における自動メモ化プロセッサの性能を評価するために，スーパスカラプロセッサ上に自動メモ化機構を実装し，その性能を評価する必要がある．また，既存の単命令発行型自動メモ化プロセッサはSPARCアーキテクチャに強く依存した実装になっているため，SPARC以外のアーキテクチャで，単命令発行型自動メモ化プロセッサと同様のハードウェアを用いたメモ化を行うことが可能かどうかを確認する必要がある．そこで，SPARCに代わる自動メモ化プロセッサのベースアーキテクチャとしてARMアーキテクチャを採用する．ARMは，SPARCに比べ複雑な命令を多数持ち，近年，モバイル用のプロセッサだけでなくサーバ用のプロセッサにも用いられつつある．本論文では，このARMをベースとしたスーパスカラプロセッサ上に自動メモ化機構を実装する場合に発生しうる問題を検討し，そのような問題を回避するスーパスカラ型自動メモ化プロセッサの設計方式を提案する．例えば，特定の命令を単純に監視するだけでは再利用対象区間である関数を検出することができないという問題を解決するために，関数呼び出しや関数復帰を監視するデコーダを実装する．また，再利用対象命令区間のある入力がリタイアステージよりも前で失われてしまい，関数の入力を全て再利用表に登録することができないという問題を解決するために，パイプラインレジスタを拡張する．さらに，再利用を適用する際に発生するオーバヘッドが引き起こすパイプラインストールを防ぐために，そのオーバヘッドと命令のパイプライン実行をオーバラップさせる．これにより，パイプラインストールの発生を防ぐことはできるが，計算再利用適用時に無駄な命令を実行してしまうという新たな問題が発生する．そこで本論文では，設計するスーパスカラ型自動メモ化プロセッサに対する高速化手法として，計算再利用が適用可能か否かを予測することで，計算再利用適用時の無駄な命令実行を削減し，再利用による高速化の効果をより得られるようにする手法も併せて提案する．

以下，2 章では本研究で取り扱う自動メモ化プロセッサについて説明する．3 章では設計したスーパスカラ型自動メモ化プロセッサについて，その動作モデルや実装について説明し，4 章では，再利用予測によるスーパスカラ型自動メモ化プロセッサ高速化手法を提案する．そして，5 章でその高速化手法の実装方法について説明する．その後，6 章でスーパスカラ型自動メモ化プロセッサとそれに対する高速化手法を評価し，最後に 7 章で結論を述べる．

2　自動メモ化プロセッサ

本章では，本研究で取り扱う自動メモ化プロセッサの動作原理とその構成について概説する．

2.1　再利用機構の構成

計算再利用（Computation Reuse）とは，プログラムの関数やループなどの命令区間において，その入力の組（入力セット）と出力の組（出力セット）を記憶しておき，再び同じ入力セットによりその命令区間が実行されようとした場合に，過去の記憶された出力セットを利用することで命令区間の実行自体を省略し，高速化を図る手法である．また，この手法を命令区間に適用可能とすることを**メモ化（Memoization）**[8] と呼ぶ．メモ化は元来，高速化のためのプログラミングテクニックである．しかし，メモ化を適用するためには，プログラムを記述しなおす必要があり，既存のロードモジュールやバイナリをそのまま高速化することはできない．その上，ソフトウェアによるメモ化 [9] はオーバヘッドが大きく，フィボナッチ数を求めるプログラムなどの限られたプログラムでしか性能向上が得られない傾向がある．

そこで，ハードウェアを用いて動的にメモ化を行うプロセッサとして，**自動メモ化プロセッサ（Auto-Memoization Processor）**[7] が提案されている．自動メモ化プロセッサは，プログラムの実行時に動的に関数やループを検出しメモ化を適用することで，既存のバイナリを変更することなく高速に実行できる．なお，自動メモ化プロセッサは call 命令のターゲットから return 命令までの区間を関数，後方分岐命令のターゲットからその後方分岐命令までの区間をループとして検出する．自動メモ化プロセッサのハードウェア構成の概略を**図 1** に示す．自動メモ化プロセッサは，一般的なプロセッサと同様に，コアの内部に ALU，レジスタ（Reg），1 次データキャッシュ（D$1）等を持ち，コアの外部に 2 次データキャッシュ（D$2）を持つ．また，自動メモ化プロセッサ独自の機構として，命令区間およびその入出力セットを記憶しておく表である再利用表 **MemoTbl** とメモ化制御

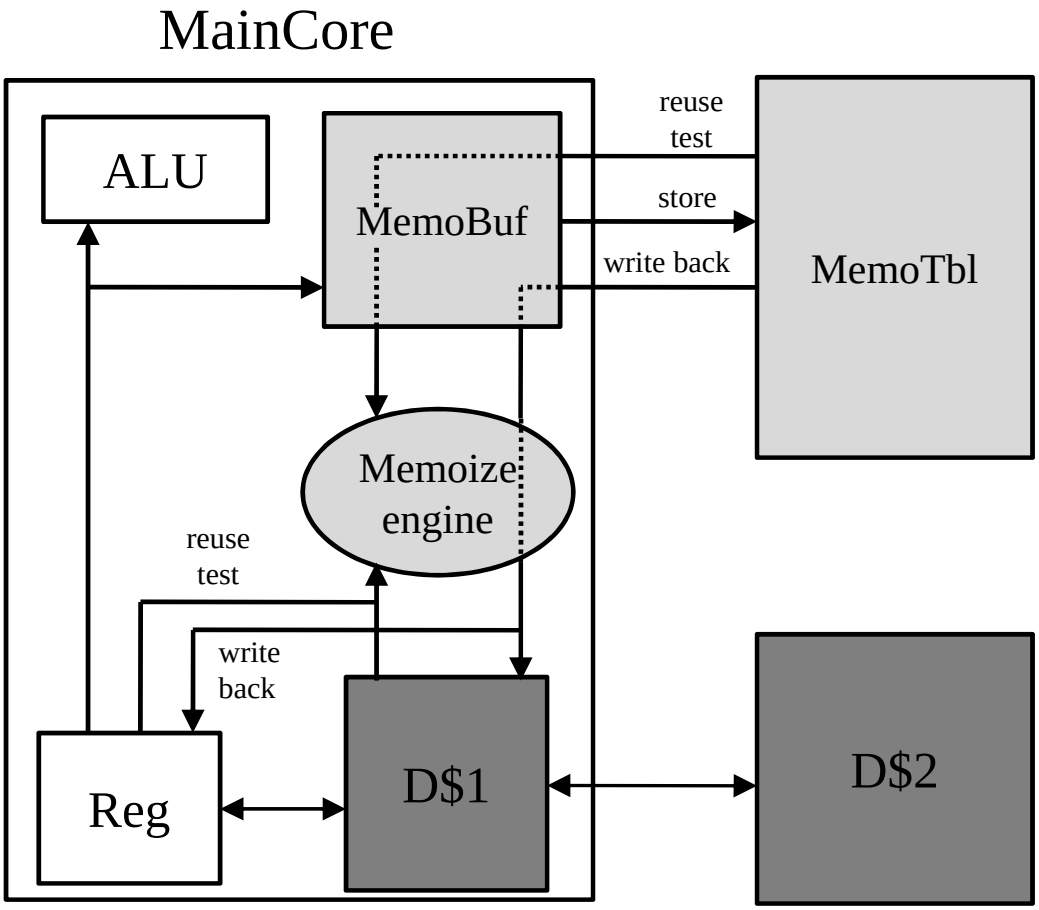

図 1: 自動メモ化プロセッサのハードウェア構成

機構（Memoize engine），および MemoTbl への書き込みバッファである **MemoBuf** を持つ．MemoTbl はサイズが大きく，コアからのアクセスコストが大きいため，入出力を検出する度に MemoTbl へアクセスするとオーバヘッドが大きくなる．そこで，このオーバヘッドを軽減するために，作業用の小さなバッファである MemoBuf をコアの内部に設けている．

自動メモ化プロセッサは再利用対象命令区間を検出すると，MemoTbl を参照し現在の入力セットと MemoTbl に記憶されている過去の入力セットを比較する．これを**再利用テスト**と呼ぶ．もし，現在の入力セットが MemoTbl 上に記憶されたいずれかの入力セットと一致する場合，メモ化制御機構はその入力セットに対応する出力セットをレジスタやキャッシュに書き戻し，命令区間の実行を省略する．一方，現在の入力セットが MemoTbl 上のいずれの入力セットとも一致しない場合，自動メモ化プロセッサは当該命令区間を通常実行しながら，その入出力を MemoBuf に登録し，実行終了時に MemoBuf の内容を

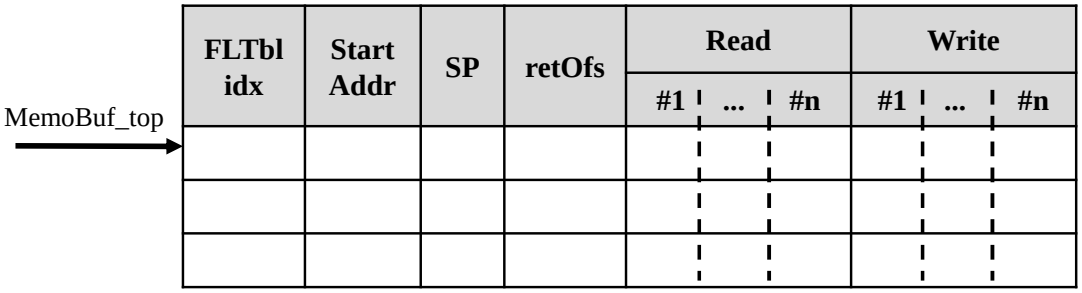

図 2: MemoBuf の構成

MemoTbl に登録することで将来の再利用に備える．

この MemoBuf の詳細な構成を**図 2** に示す．MemoBuf は複数のエントリを持ち，1 エントリが 1 入出力セットに対応する．各エントリは，どの命令区間に対応しているかを識別するための，後述する FLTbl のインデクス（FLTbl idx），その命令区間の開始アドレス（Start Addr），その命令区間の実行開始時のスタックポインタ（SP），関数の戻りアドレス，またはループの終端アドレス（retOfs），命令区間の入力セット（Read）および出力セット（Write）を持つ．なお，命令区間の各入出力には，複数の入出力値を保持できるようになっている．また，入れ子構造になった命令区間もメモ化対象とするために，MemoBuf は現在使用しているエントリをポインタ（MemoBuf_top）で指しており，命令区間の検出時にそのポインタをインクリメントし，命令区間の実行終了時にデクリメントすることで入れ子構造を保持している．

さて，一般に命令区間内では，複数の入力値が順に参照され使用される．しかし，同じ命令区間でも，その入力アドレスの列は分岐していく場合がある．例えば，条件分岐命令を実行すると，次に参照されるアドレスはその条件分岐命令の分岐結果によって変化してしまう．このように，ある命令区間の入力アドレスの列はその入力値によって分岐していく．そこで，自動メモ化プロセッサは，全入力パターンを木構造で管理し，MemoTbl に登録する．例えば，自動メモ化プロセッサが**図 3** に示すサンプルプログラムを実行する場合，関数 calc の全入力セットは**図 4** に示すような木構造で表現されることになる．なお，図 4 の木構造において，ノードは命令区間の入力値を，エッジは入力値と次に参照される入力値との対応関係を示しており，End はそれ以上の入力値が存在しないことを示す．つまりこの木構造上では，ルートノードから各リーフノードへのパスが，入力セットそれぞれに対応し，入力セット (i)，(ii)，(iii) はそれぞれサンプルプログラムの 12 行目，13 行目，14 行目における関数呼び出しに対応している．この例において，入力セット (i)

```
1  int a = 3, b = 4, c = 8;
2  int calc(x){
3    int tmp = x + 1;
4    tmp = tmp + a;
5    if(tmp < 7)
6      tmp = tmp + b;
7    else
8      tmp = tmp + c;
9    return(tmp);
10 }
11 int main(void){
12   calc(2); /* x = 2, a = 3, b = 4 */
13   b = 5; calc(2); /* x = 2, a = 3, b = 5 */
14   a = 5; calc(2); /* x = 2, a = 5, c = 8 */
15   a = 3; calc(2); /* x = 2, a = 3, b = 5 */
16   return(0);
17 }
```

図 3: サンプルコード

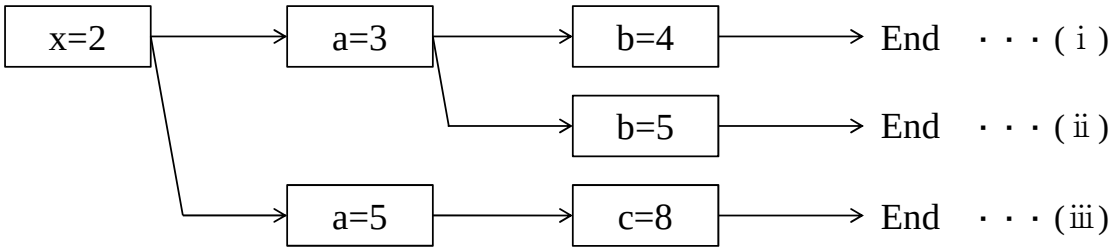

図 4: 入力パターンの木構造

と入力セット (ii) では，変数 b が 3 番目に参照されるのに対して，入力セット (iii) では，変数 c が 3 番目に参照される．これは，2 番目に参照される変数 a の値が異なることにより，プログラムの 5 行目における条件分岐の結果が変化し，次入力アドレスが変化したためである．

次に，この木構造で表現された全入力パターンを登録するための表である MemoTbl の詳細な構成を**図 5** に示す．MemoTbl は，命令区間を記憶する **FLTbl**，入力を記憶す

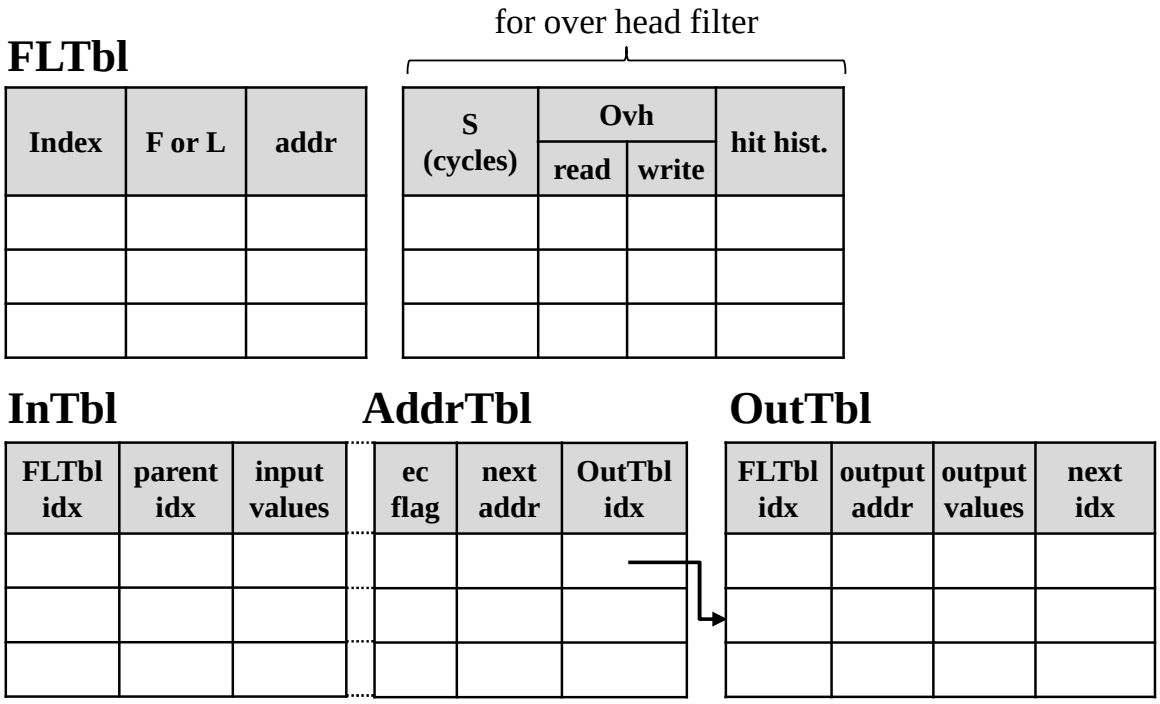

図 5: MemoTbl の構成

る **InTbl**，入力アドレスを記憶する **AddrTbl**，および出力を記憶する **OutTbl** の 4 つの表から構成される．これらのうち，InTbl が図 4 で示した木構造の各ノードを格納し，AddrTbl が各エッジを格納する．なお，FLTbl，AddrTbl，OutTbl は RAM での実装を想定しており，InTbl は高速な連想検索が可能な**汎用 3 値 CAM（Content Addressable Memory）**での実装を想定している．

FLTbl は 1 行が 1 命令区間に対応しており，その行番号（Index）を各命令区間の識別番号とする．また，各行にはメモ化のためのフィールドに加え，後述するオーバヘッドフィルタのためのフィールドが保持される．メモ化のためのフィールドには，関数とループを判別するフラグ（F or L）と，命令区間の開始アドレス（addr）が保持される．一方，オーバヘッドフィルタのためのフィールドには，当該命令区間のサイクル数（S），過去の再利用に要した入力検索および出力書き戻しオーバーヘッド（Ovh read/write），過去の再利用ヒット履歴（hit hist.）が保持される．

InTbl の各行は FLTbl の行番号 Index に対応する FLTbl idx を持ち，この値を用いてどの命令区間の入力値を記憶しているかを判別する．また，この入力値（input values）に加えて，命令区間の全入力パターンを木構造で管理するために，親エントリのインデクス（parent idx）を持つ．なお，input values は 1 キャッシュブロック分の入力値を記憶

し，比較する必要のないアドレスの入力値についてはドントケアで表現される．また，レジスタの値が入力値となる場合も同様に，複数レジスタの入力値をまとめて 1 つの入力エントリに記憶する．

AddrTbl は入力アドレスを記憶する表である，すなわち，AddrTbl は InTbl と同数のエントリを持ち，各エントリは 1 対 1 に対応する．AddrTbl の各行は入力値検索のために，次に参照すべきアドレス（next addr）を持つ．また，入力セットの終端エントリか否かを保持するフラグ（ec flag）を持ち，終端エントリである場合，出力を記憶している表である OutTbl のエントリを指すインデクス（OutTbl Index）も持つ．

OutTbl の各行は FLTbl idx に加えて，命令区間の出力先のアドレス（output addr），出力値（output values）を持つ．また，出力セットの各エントリをリスト構造で管理するため，次に参照すべきエントリのインデクス（next idx）を持つ．

2.2 再利用機構の動作例

計算再利用を適用するためには，現在の入力セットと過去に MemoTbl に記憶しておいた入力セットとを比較する必要がある．MemoTbl の検索手順を**図 6** に示す．ここで，図 6 は図 3 のサンプルプログラムが 14 行目まで実行された時の MemoTbl 中の状態を表している．また，図 6 では図を簡略化するため，FLTbl には命令区間に対応する Index，その命令区間が関数であるかどうか，およびその命令区間の開始アドレスのみを記述し，AddrTbl と OutTbl の FLTbl idx は InTbl のそれと同じ値であるため省略している．なお，図中の InTbl の input values における X はドントケアを表しており，そのアドレスに対応する値は検索時に比較されない．同様に図中の InTbl の parent idx における FF は親エントリが存在しないこと，すなわち自身が入力パターンの木構造の根に相当するエントリであることを表している．また，図 6 内の矢印は図 3 の 15 行目における関数呼び出し時の MemoTbl 検索フローを示している．

いま，再利用区間の実行開始アドレスが検出されると，まず，FLTbl が関数 calc の開始アドレスで検索される．そして，これにより得られた FLTbl idx を持ち，input values が現在のレジスタ上の入力値と一致し，かつ parent idx が FF であるようなルートエントリが検索される (a)．次に，該当するエントリがライン 00 で発見され，対応する AddrTbl の next addr が 0x200 番地を指しているため，そのアドレスに対応するキャッシュラインを参照する (b)．そして，得られた値を input values として持ち，parent idx が 00 であるエントリを InTbl から検索する (c)．以降，同様に検索を続ける (d)(e)．ここで，ライン 03 の ec flag が 1 であること，つまり入力セットの終端エントリに達したことが検出され

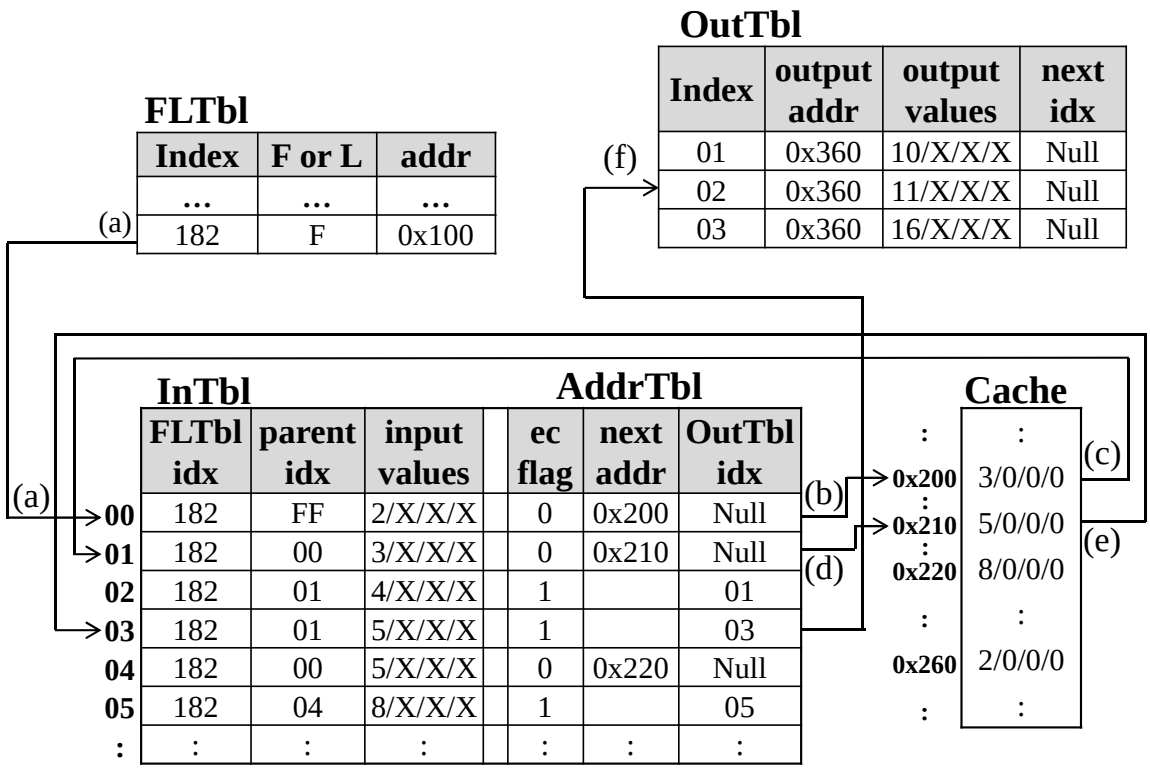

図 6: MemoTbl の検索手順

るため，自動メモ化プロセッサは再利用テストに成功する．このとき，検索の終点となる AddrTbl エントリの OutTbl idx に登録されているインデクスが指す OutTbl エントリを参照し (f)，読み出した出力値をレジスタやキャッシュに書き戻すことで命令区間の実行を省略することができる．

ところで，上述したように再利用テストを行うと，一致比較の回数は，キャッシュ参照の有無に関わらず入力となるキャッシュラインの数と同じ回数だけ必要となる．しかし，あるメモリアドレスに対応するエントリが再利用表に登録されてからそのメモリアドレスの値が一度も書き換わっていない場合，そのエントリに対する一致比較は本質的には不要である．例えば図 6 の例では，実際にキャッシュから値を読み出して比較すべき入力は 0x200 番地に対応するキャッシュラインのみであり，0x210 番地に対応するキャッシュラインを一致比較する必要はない．何故なら，図 2 のプログラムの 13 行目において再利用表にエントリが登録されてから，15 行目において再利用テストが開始されるまで，変数 b に対応する 0x210 番地の値は一度も書き換えられていないためである．それにも関わらず，キャッシュの値の比較は 2 回必要となっており，これは非効率的である．

そこで，自動メモ化プロセッサは，再利用テストの必要がないエントリに関して，その

一致比較自体を省略し，それ以外のエントリのみを一致比較する [10]．これを実現するため，自動メモ化プロセッサは AddrTbl の構成を拡張している．具体的には，AddrTbl の各エントリに，以下の 4 つのフィールドを追加している．

up: 親 AddrTbl エントリのインデクス

next addr2: 次に検索すべきアドレス

dn: 次に検索すべきインデクス

change flag: 当該 AddrTbl エントリの next addr に記憶されているアドレスに対し書き込みが発生したか否かを保持するフラグ

これにより，change flag がオンである AddrTbl エントリに関しては，これまでと同様に検索するが，change flag がオフである AddrTbl エントリに関しては，nextaddr2 および dn をキーとして次のエントリを検索する．その結果，自動メモ化プロセッサは再利用テストの必要がないエントリを迂回して検索を続けることができる．

なお，change flag がオフである AddrTbl エントリの next addr に対応するメモリアドレスに対し書き込みが発生した場合には，まず AddrTbl から当該 AddrTbl エントリを検索し change flag を立てる．次にそのエントリの up フィールドが示す AddrTbl のエントリに関し，nextaddr2 フィールドを書き込みが発生したメモリアドレスで，dn フィールドをそのメモリアドレスの登録されているエントリのインデクスで埋める．これにより，次回検索時には，書き込みが発生したメモリアドレスの内容に対する比較が行われるようになる．

2.3 オーバヘッドフィルタ機構

自動メモ化プロセッサは，計算再利用可能な命令区間の実行を省略することで高速化を図る手法であるが，その際には MemoTbl を検索するコスト，および入力が一致したエントリに対応する出力を MemoTbl からレジスタやメモリに書き戻すコストがオーバヘッドとして発生する．命令区間の中には，これらのオーバヘッドが大きく，再利用を適用することで却って性能が悪化してしまうものも存在する．そこで，FLTbl では，各命令区間に対し一定期間における再利用の成否状況をシフトレジスタ（図 5 中 hit hist.）を用いて記録し，それぞれの命令区間の再利用適応度を算出している．このシフトレジスタの構造を**図 7** に示す．シフトレジスタでは再利用テスト 1 試行分の成功，失敗の結果が 1 ビットで記憶され，再利用に成功した場合は 1 が，再利用に失敗した場合は 0 がセットされる．また，新たに再利用テストの結果を記憶する際には，シフトレジスタを右に 1 ビットシフトしてから，左端に新たな 1 ビットの情報をセットする．シフトレジスタはこのように動作

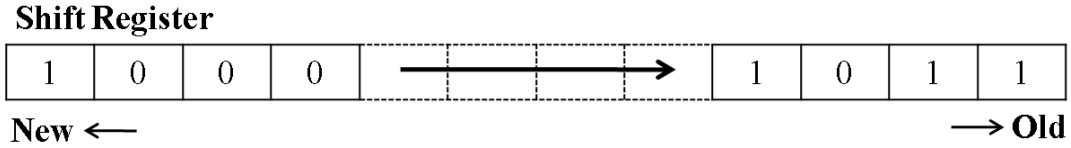

図 7: シフトレジスタの構造

することで，ビット幅と同じ回数分の過去の再利用の成否状況を記憶できる．

ある命令区間について，最近の一定回数 T の再利用テストのにおける再利用成功回数 M は，幅 T ビットの上記シフトレジスタから得られる．この値と，当該命令区間に対応する FLTbl エントリに記憶されている過去の省略サイクル数 S から，実際に削減できたサイクル数を，

$$M \cdot (S - Ovh^R - Ovh^W) \tag{1}$$

として計算する．なお Ovh^R，Ovh^W はそれぞれ，過去の履歴より概算した，当該命令区間の再利用表検索オーバヘッド，および再利用表からキャッシュ等への書き戻しオーバヘッドである．また，計算再利用を適用できなかった場合でも，再利用表の検索オーバヘッドは存在する．このオーバヘッドは，

$$(T - M) \cdot Ovh^R \tag{2}$$

として計算できる．

ここで，発生したオーバヘッド (2) よりも，削減できたサイクル数 (1) が大きいような命令区間は，再利用の効果が得られると考えられる．式 (1) から式 (2) を引いたものを ***Gain*** とすると，

$$Gain = M \cdot (S - Ovh^W) - T \cdot Ovh^R \tag{3}$$

となり，この $Gain$ が正値であれば，再利用の効果があると判断する．そして，そのように判断した命令区間に対してのみ再利用表への登録および再利用の適用を行う．

3　スーパスカラ型自動メモ化プロセッサ

本章では，既存の単命令発行型自動メモ化プロセッサの問題点と，設計したスーパスカラ型自動メモ化プロセッサの動作モデルおよび実装について述べる．

3.1 既存の単命令発行型自動メモ化プロセッサの問題点

これまで，自動メモ化プロセッサには，単命令発行でパイプライン化されていない単純な SPARC-V8 アーキテクチャ [11] がベースアーキテクチャとして採用され，研究されてきた．しかし，この既存の自動メモ化プロセッサには 2 つの問題点が存在する．

1 つは，自動メモ化プロセッサのベースアーキテクチャとして採用されている，この単命令発行のアーキテクチャが，現在市場に流通しているプロセッサのベースアーキテクチャとして広く採用されているスーパスカラアーキテクチャと，ハードウェア構成が大きく異なっている点である．そのため，これまで確認されてきた自動メモ化プロセッサの評価結果は必ずしも実用性を実証できていない可能性があり，市場のプロセッサ事情に即した，より実用的な環境における自動メモ化プロセッサの性能を評価する必要がある．

もう 1 つは，単命令発行型自動メモ化プロセッサが，SPARC アーキテクチャに強く依存した実装になっているという点である．例えば，SPARC の命令セットや ABI（Application Binary Interface）は，命令区間の検出が容易であるなど，メモ化を行う上で非常に都合が良い．つまり，SPARC 以外のアーキテクチャで，単命令発行型自動メモ化プロセッサと同様のハードウェアを用いたメモ化を行うことが可能かどうかは確認できていない．

そこで，SPARC に代わる自動メモ化プロセッサのベースアーキテクチャとして ARM アーキテクチャ [12] を採用する．ARM は，SPARC に比べ複雑な命令を多数持ち，近年，モバイル用のプロセッサだけでなくサーバ用のプロセッサにも用いられつつある [13]．本論文では，この ARM をベースとしたスーパスカラプロセッサ上に自動メモ化機構を実装し，そのスーパスカラ型自動メモ化プロセッサの性能を評価することで，自動メモ化プロセッサの実用性を確認する．

3.2 ベースアーキテクチャ

本論文では，自動メモ化機構を実装する ARM 型スーパスカラプロセッサとして OROCHI [14] を採用した．本節では，この OROCHI について詳しく説明する．OROCHI のハードウェア構成を**図 8** に示す．OROCHI は VLIW 命令と ARM 命令を同時実行可能なスーパスカラプロセッサである．ただし，自動メモ化機構を実装する上で，VLIW 命令用のユニット群は必要ないため無効化しており，図 8 でも省略している．OROCHI のパイプラインは全 9 段で構成される．このプロセッサは，1 次データキャッシュ，1 次命令キャッシュ，および共有 2 次キャッシュを持ち，フロントエンドでは命令フェッチ，デコード，リネームを行い，バックエンドではアウトオブオーダで命令を実行する．各パイプラインステージは以下のとおりである．

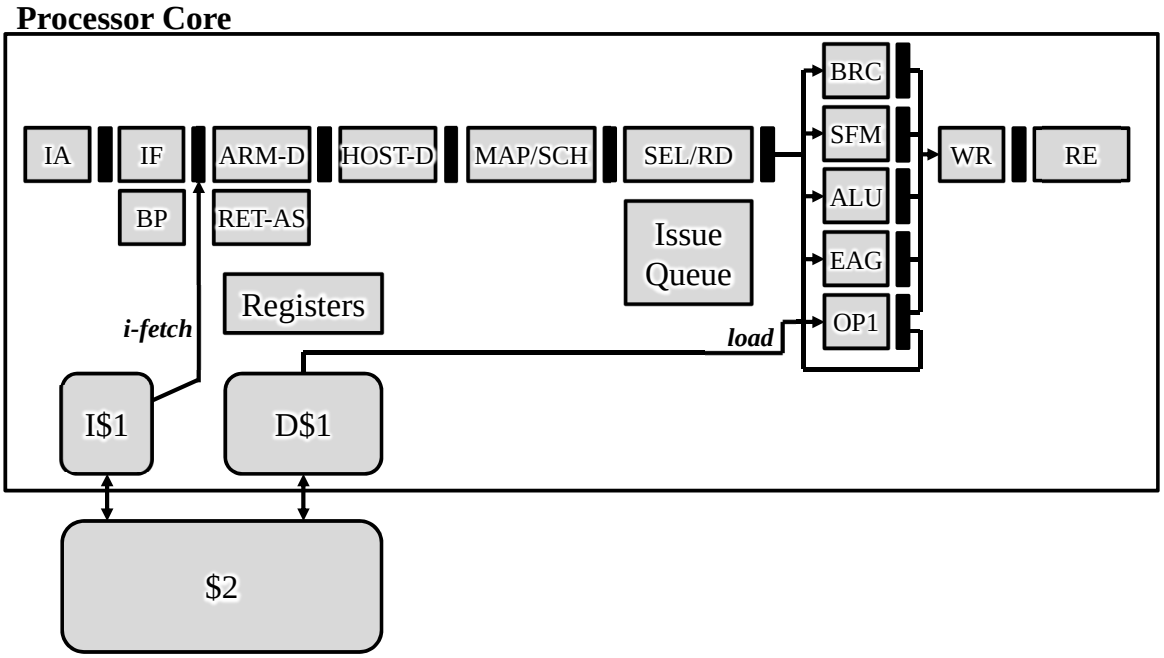

図 8: ARM 型スーパスカラプロセッサの構成

IA (Instruction Address)

次に実行すべき命令のアドレスを計算する．

IF (Instruction Fetch)

命令キャッシュから連続した複数命令をフェッチする．また，*g-share* [15] による分岐予測も行う．

ARM-D (Arm-Instruction Decode)

ARM 命令を高速実行可能な内部命令に分解する．例えば，マルチ LD 命令は，複数の 4bytes の LD 命令に，乗算命令などは，複数の加算命令とシフト命令に分解される．

HOST-D (Host-Instruction Decode)

条件実行命令を実行ステージで処理できるように分解する．なお，条件実行命令とは，命令の実行に条件を付加する ARM 独自の命令である．ARM のすべての命令は，実行時に条件フラグを参照し，フラグがセットされているときのみ実行するようにできる．OROCHI では，条件実行命令は 2 命令に分解される．1 つは，条件実行命令が実行される条件の成否をチェックする CSL 命令であり，もう 1 つは，その命令の条件が満たされていた場合に実行される命令である．実行条件が満たされていない場合，後者の命令は破棄される．

MAP/SCH (Register mapping/Schedule)

各命令のオペランドを，論理レジスタから，命令ウィンドウと物理レジスタを兼ねるリオーダバッファへマッピングする．また，同時に各命令のスケジューリングも行う．

SEL/RD (Select and Read)

命令が発行可能かどうかを調べ，依存関係のない命令を発行する．

IE (Instruction Execution)

ベースアーキテクチャは5並列の実行ステージを持つ．

BRC　分岐の taken/untaken を判定する．

SFM　シフト演算を処理する．また，積和補助演算も処理する．

ALU　加減算命令を処理する．

EAG　アドレス計算を処理する．また，積和補助演算も処理する．

OP1　ロード・ストア命令を処理する．

WR (WriteBack)

各実行ステージで処理された命令をリオーダバッファに登録する．

RE (Retire)

先行命令が全て完了した命令の実行結果をリオーダバッファから論理レジスタへ書き戻す．なお，分岐予測ミスなどの際に速やかに実行を再開するために，命令がコミットされる順番はプログラムの命令列と同じであることが保証されている．

上記の ARM-D ステージの説明で述べたように，OROCHI は命令分解機構を取り入れており，ARM 命令を簡単な処理を行う内部命令へと変換，分解することで，各内部命令の処理量を均等にし，それらが常に同じサイクル数で処理されるようにしている．これにより，OROCHI は命令の効率的なパイプライン実行を可能としている．

3.3　スーパスカラ型自動メモ化プロセッサのパイプライン動作

本節では，上述した OROCHI に対して自動メモ化機構を実装した際の動作モデルについて，メモ化を行わない通常実行時，再利用テスト成功時，再利用テスト失敗時の3つに分けて説明する．

3.3.1　メモ化を行わない通常実行時

まず，メモ化を行わない通常実行時のパイプライン実行の様子を**図 9** に示す．なお，以降の説明を簡単化するために，スーパスカラ型自動メモ化プロセッサのパイプラインのウェイ数は2とし，各パイプラインは Fe（フェッチ），De（デコード），Ex（命令実行），Re（リタイア）の4段構成をとり，各ステージは1サイクルで処理されると仮定する．ま

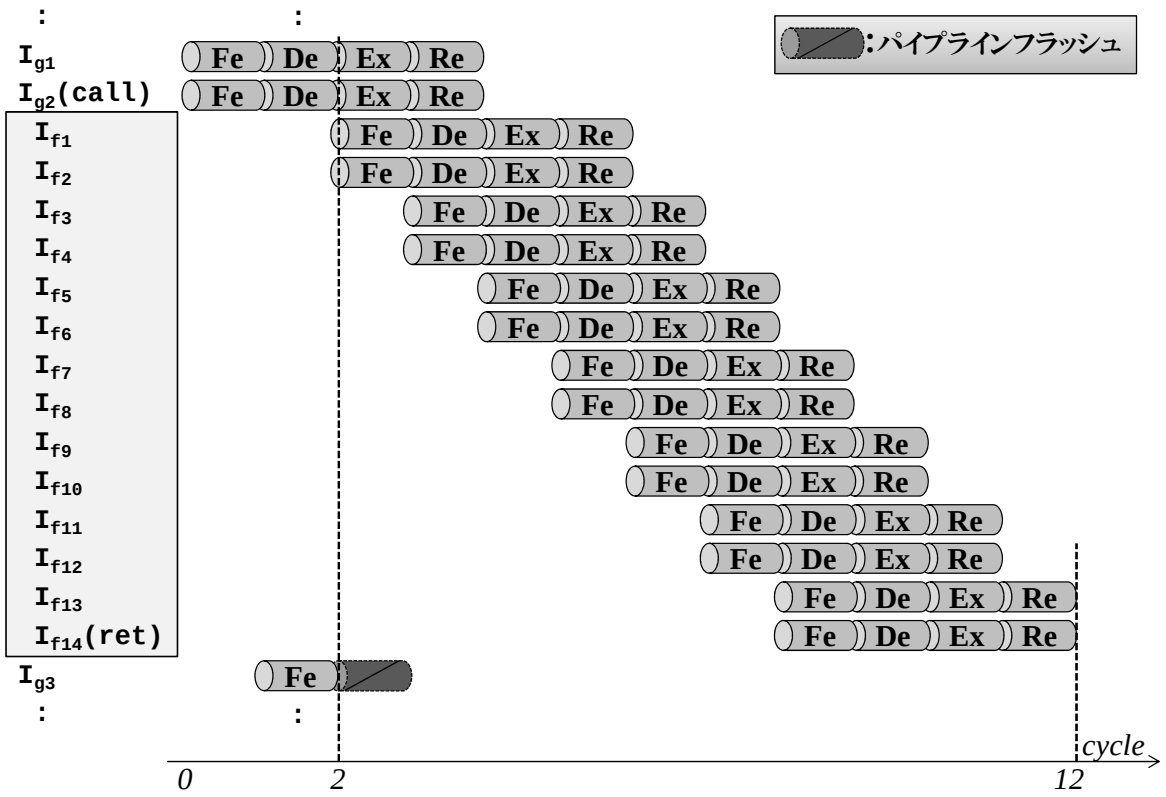

図 9: 通常実行時のパイプライン実行の様子

た，キャッシュミスなどによりパイプラインがストールすることなく，理想的な状態で各ステージでの処理が実行されるものとする．この図において，I_{g2} は関数呼び出しのための命令，I_{f14} は関数復帰のための命令であり，I_{f1} から I_{f14} までの区間が関数である．なお，関数の先頭命令をフェッチしてから，関数の復帰命令をリタイアするまでに要するサイクル数が，その関数の実行に要するサイクル数である．

まず，この図の命令列の実行が開始されると，2 サイクル目にて関数呼び出しのための命令がデコードされる．このとき，関数内部の命令の実行に移るために，関数復帰先以降の命令がパイプラインからフラッシュされ，関数の先頭アドレスでプログラムカウンタの値が上書きされる．その後，この関数はストールすることなく実行され，メモ化を行わず通常実行する場合，その実行には 10 サイクルを要する．

3.3.2 再利用テスト成功時

次に，再利用テスト成功時のパイプライン実行の様子を**図 10** に示す．スーパスカラ型自動メモ化プロセッサは関数呼び出しのための命令を検出すると，再利用を適用できるかどうかを確かめるために再利用テストを行う．この再利用テストを正しく行うためには，関数呼び出しのための命令より前の命令によるレジスタやキャッシュへの書き込みが完了

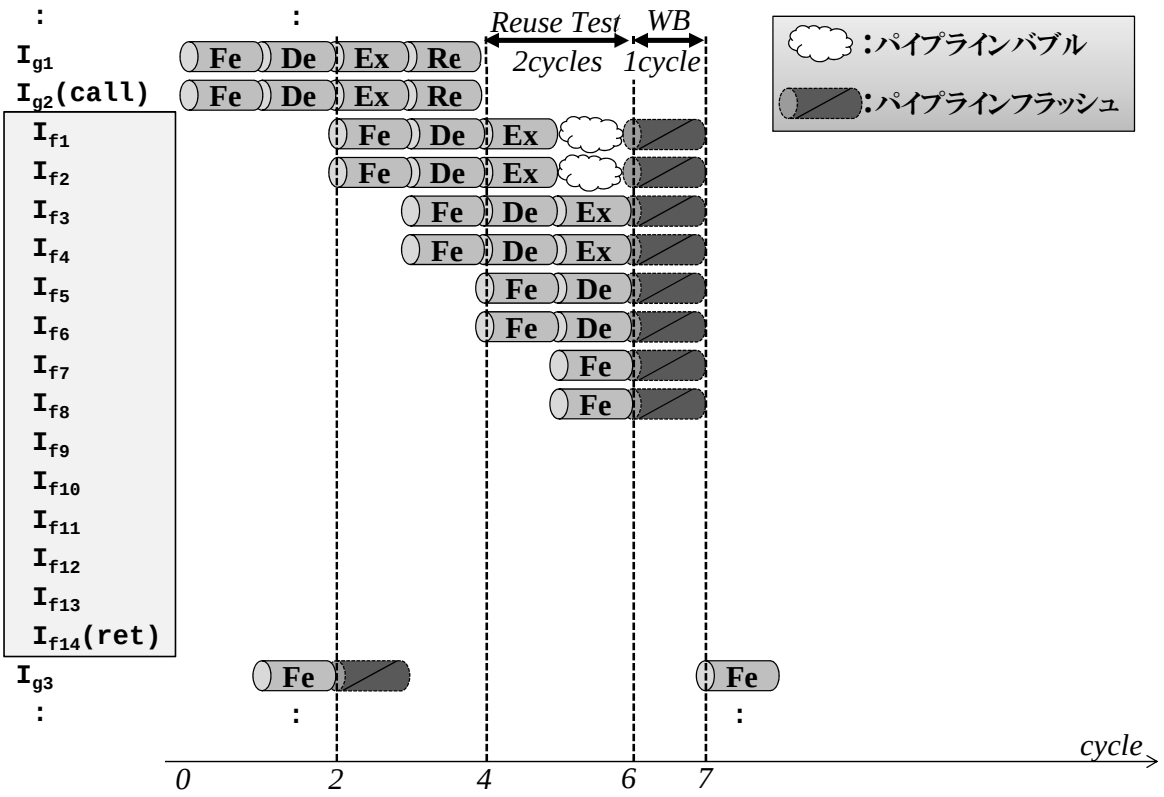

図 10: 再利用テスト成功時のパイプライン実行の様子

している必要がある．ここで，関数呼び出しのための命令がリタイアされていれば，それより以前の命令が終了しており，関数の入力となる値がレジスタやキャッシュに存在していることが保証される．そのため，再利用テストは関数呼び出しのための命令がリタイアされたタイミングである 4 サイクル目で行う．なお，再利用テストを行っている間はパイプラインをストールさせず，現在実行中の関数を，この再利用テストとオーバラップして引き続き実行させることによって，再利用テストのオーバヘッドを緩和する．ただし，再利用テスト中のパイプライン実行では，リタイアステージでの処理を禁止している．これは，関数内の命令がリタイアされてしまうと，一致比較対象の入力値が上書きされる可能性があり，その結果，過去と入力が異なるにも関わらず一致比較に成功してしまうことで不正な再利用が適用されてしまう可能性があるからである．

その後，6 サイクル目にて，この再利用テストに成功し，計算再利用を適用できることが判明したとすると，自動メモ化機構は過去の実行結果を再利用表からレジスタやキャッシュに書き戻すと同時に，パイプラインをフラッシュする．これは，既にパイプラインに投入されている命令は再利用対象区間である関数内の命令であり，実行する必要がないためである．図の例では，再利用テストに成功し，関数の実行を省略したことによって，2

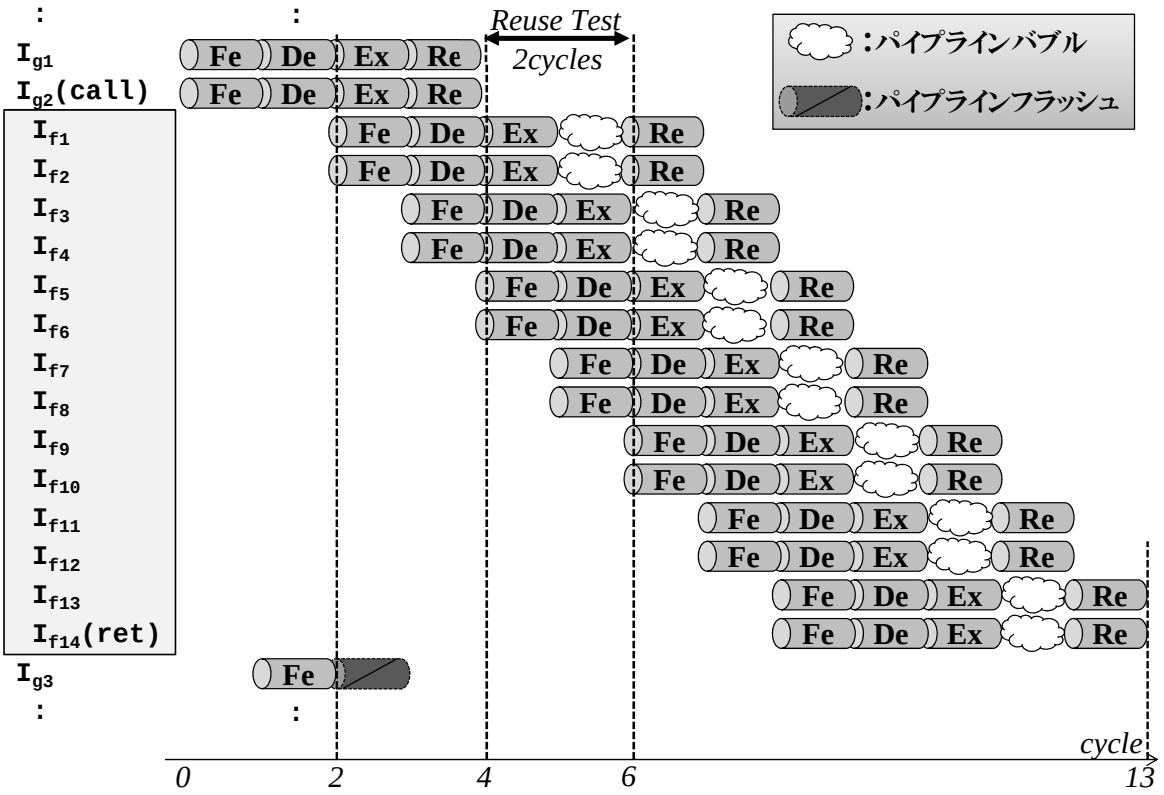

図 11: 再利用テスト失敗時のパイプライン実行の様子

サイクル目から 6 サイクル目までの関数内部の命令実行は全て無駄となる．そのため，この無駄な命令実行が計算再利用による性能向上を妨げてしまう可能性がある．なお，この問題を解決するための手法を 4 章で提案する．

その後，7 サイクル目にて，関数復帰先の命令（I_{g3}）をフェッチすることで，実行を継続する．再利用テストの成功により，図 10 に示す例の場合では，関数の実行に要したのは 5 サイクルのみであり，図 9 のメモ化を行わない場合と比べて，5 サイクル高速化できている．

3.3.3 再利用テスト失敗時

最後に，再利用テスト失敗時のパイプライン実行の様子を**図 11** に示す．再利用テスト成功時の場合と同様に，関数呼び出しのための命令がリタイアされたタイミングである 4 サイクル目にて，スーパスカラ型自動メモ化プロセッサは再利用テストを開始する．また同様に，再利用テストを行っている間，リタイアステージでの処理が禁止された状態で，現在実行中の関数が再利用テストとオーバラップして引き続き実行される．その後，6 サイクル目にて，この再利用テストに失敗し，この関数に計算再利用を適用できないことが判明すると，禁止していたリタイアステージでの処理が再開され，通常通り命令が実行さ

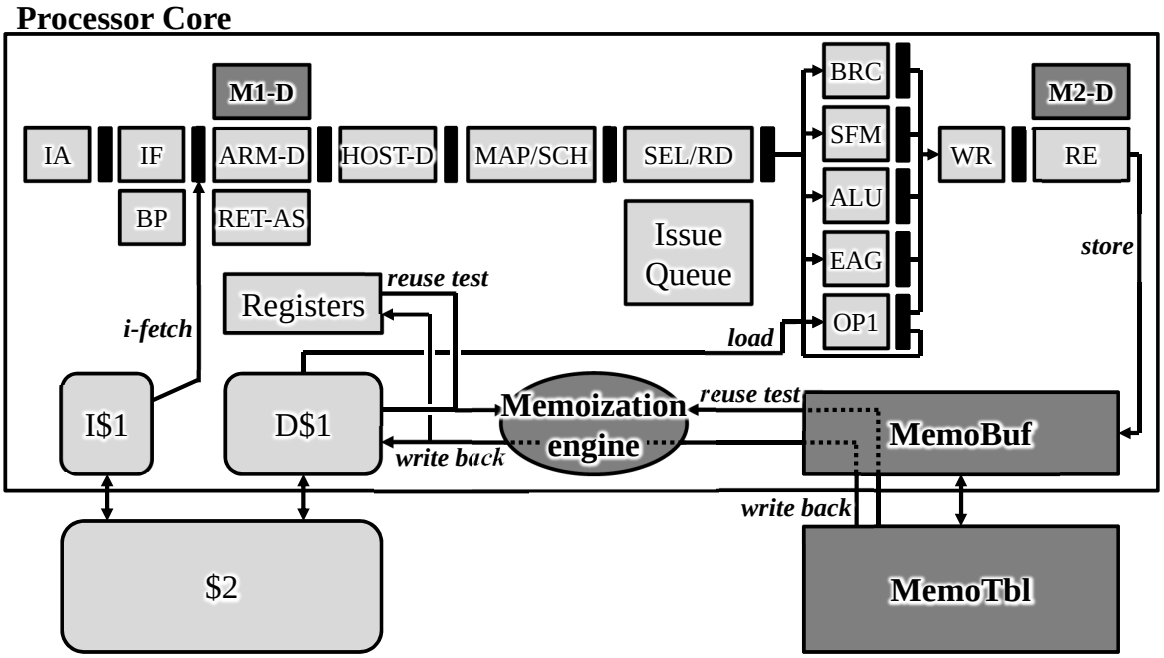

図 12: スーパスカラ型自動メモ化プロセッサの構成図

れる．この図の例の場合，関数の実行には 11 サイクルを要しており，図 9 のメモ化を行わない例の場合と比べて，1 サイクル増加している．このサイクル数の増加は，再利用テストのオーバヘッドが原因である．ただし，命令のパイプライン処理と再利用テストをオーバラップさせることによって，この図の例の場合，本来 2 サイクルを要する再利用テストのオーバヘッドを 1 サイクルに緩和することができている．

3.4 メモ化のためのハードウェア拡張

本節では，スーパスカラ型自動メモ化プロセッサの実装について説明する．スーパスカラ型自動メモ化プロセッサの汎用性が失われないようにするために，3.2 節で述べた OROCHI に対して，このプロセッサ独自の命令分解機構に依存することなく，自動メモ化機構を実装した．**図 12** に，設計したスーパスカラ型自動メモ化プロセッサの構成図を示す．自動メモ化機構については，2 章で述べた既存の自動メモ化プロセッサと同様に実装する．以降では，スーパスカラ型自動メモ化プロセッサを設計する上で必要な独自のハードウェア拡張について説明する．

3.4.1 関数検出デコーダ

再利用対象区間である関数を特定するためには，関数呼び出し，および関数復帰を検知する必要がある．SPARC の ISA（Instruction Set Architecture）では関数呼び出しの

(C-i)
```
bl  func
```

(C-ii)
```
mov lr,pc
mov pc,R0
```

(C-iii)
```
mov lr,pc
ld  pc,[R0]
```

図 13: 関数呼び出しコード

(R-i)
```
mov   pc, lr
```

(R-ii)
```
stmdb sp!,{r4,r5,r6,lr}
            :
ldmia sp!,{r4,r5,r6,pc}
```

図 14: 関数復帰コード

ための命令は `call`，関数復帰のための命令は `ret` と規定されている．そのため，既存の SPARC をベースとしている単命令発行型自動メモ化プロセッサは，これら二つの命令を監視することにより，関数を検出している．一方で，ARM の ISA では関数呼び出し，および関数復帰に特定の命令を使用すべきであるといった規定はない．ここで，**図 13** と**図 14** に，ARM バイナリにおける関数呼び出し，および関数復帰コードを示す．このように，ARM では関数呼び出し，および関数復帰コードの種類は多岐にわたるため，単命令発行型自動メモ化プロセッサと同様に特定の命令を単純に監視するだけでは，関数を検出することはできない．そこで，図 12 に示すように，関数呼び出し，および関数復帰を検出するための，関数検出デコーダ（M1-D，M2-D）をデコードステージとリタイアステージに追加する．

図 15 にこれらデコーダによる関数呼び出し，および関数復帰検出フローを示す．まず，M1-D はプログラムカウンタの値を上書きする命令を監視する (1)．検知したプログラムカウンタ上書き命令の種類によって，その後の関数呼び出し，および関数復帰検出フローが 3 つのパターンに分けられる．まず，その検知した命令が `bl` 命令である場合 (2)，スーパスカラ型自動メモ化プロセッサはその命令を関数呼び出しのための命令として検出する (C-i)．次に，検知した命令が `b` 命令，または `mov` 命令である場合 (3)，その命令のソースオペランドを調べる．ソースオペランドが関数復帰アドレスを保持するリンクレジスタ（`lr`）以外であるならば (4)，次は直前の命令を調べる．その直前の命令が関数復帰アドレスを退避する命令（`mov lr, pc` 等）である場合 (5)，検知した命令を関数呼び出しのための命令として検出する (C-ii)．一方で，ソースオペランドが関数復帰アドレスを保持するリンクレジスタであるならば (6)，検知した命令を関数復帰のための命令として検出する (R-i)．最後に，検知した命令が `ld` 命令である場合 (7)，直前の命令を調べる．その直

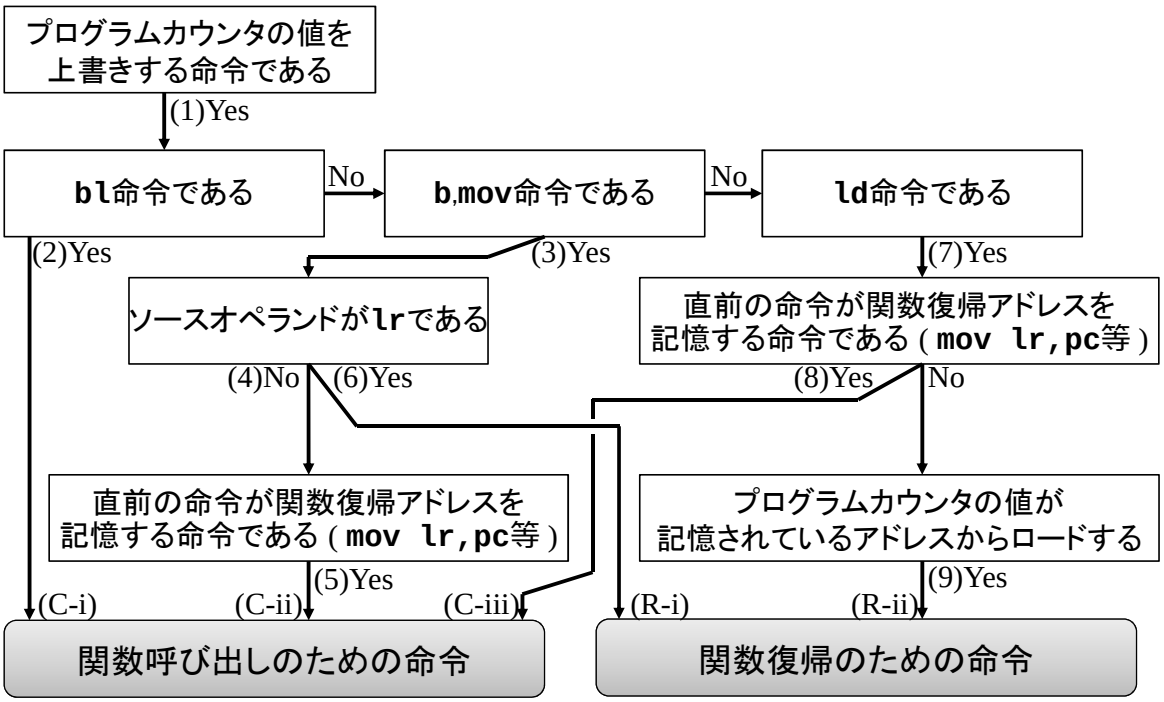

図 15: 関数呼び出し，および関数復帰の検出フロー

前の命令が関数復帰アドレスを退避する命令であるならば (8)，検知した命令を関数呼び出しのための命令として検出する (C-iii)．なお，関数プロローグにおいてスタックに退避された関数復帰アドレスが，関数エピローグにおいてロード命令により直接プログラムカウンタに上書きされ，関数復帰が行われる場合がある．このような場合にもそのロード命令を関数復帰のための命令として検出するために，M2-D は関数プロローグでプログラムカウンタの値が退避されたアドレスを記憶する．そして，そのアドレスから値を読み出すロード命令を検知したとき (9)，そのロード命令を関数復帰のための命令として検出する (R-ii)．以上のように関数呼び出し，および関数復帰のための命令を検出することで，再利用対象区間である関数を検出することができる．

3.4.2 パイプラインレジスタの拡張

2 章で述べたように，自動メモ化プロセッサは計算再利用を関数に適用するために，MemoBuf や MemoTbl に各関数の入出力を登録する必要がある．ここで，関数の入出力とは，関数内で読み書きされた値とアドレスの組のことである．スーパスカラ型自動メモ化プロセッサでは，関数の入出力を登録するタイミングはリタイアステージを想定している．これは，リタイアステージよりも前のステージで関数の入出力を登録しようとすると，実行された命令が分岐予測ミスなどによりフラッシュされた場合に，フラッシュされ

た命令に対応するMemoBufのエントリを無効化する必要があり，その無効化のための操作が大きなオーバヘッドとなるからである．しかし，リタイアステージで関数の入出力を登録する場合にも問題はある．それは，関数の入力となるロード命令のソースオペランドの値がリタイアステージよりも前で失われてしまうために，リタイアステージで関数の全ての入力をMemoBufに登録することができないという問題である．

ここで，関数の入力となるロード命令のソースオペランドの値がリタイアステージよりも前で失われてしまう例を，`ld R0, [R1]`という命令が実行される場合で示し，その解決方法を説明する．`ld R0, [R1]`というロード命令が実行される場合，関数の入力となる対象は`R0`にロードされる値と`R1`が指しているアドレスである．さて，この命令が実行ステージのOP1ユニットで処理されるとき，論理レジスタ`R0`に対応する物理レジスタに対し`R1`が指しているアドレスに格納されている値がロードされる．その後，この命令がリタイアされるとき，その物理レジスタの値が論理レジスタ`R0`に書き込まれる．このとき，このロード命令のリタイア処理には，ソースオペランドである`R1`に関する情報は必要ないため，リオーダバッファからその情報が失われてしまっている．そのため，このままでは`R1`が指しているアドレスを入力としてMemoBufに登録することができない．そこで，入力となるロード命令のソースオペランドの値をリタイアステージまで伝播させるために，実行ステージ以降のパイプラインレジスタでも，ソースオペランドを保持し，次段に渡すように，パイプラインレジスタを拡張する．これにより，実行ステージ以降で失われてしまうソースオペランドの値をこのパイプラインレジスタを通して後段まで伝播させることで，関数の入出力をリタイアステージでMemoBufに登録できるようにする．

4　再利用予測による高速化

本章では，3章で述べたスーパスカラ型自動メモ化プロセッサにおいて，再利用テスト成功時に発生する無駄な命令実行を削減し，計算再利用による高速化の効果をより得られるようにする手法を提案する．

4.1　再利用テスト成功時の問題点

3章で述べたように，スーパスカラ型自動メモ化プロセッサは，再利用テストが開始されると，関数内部の命令のパイプライン処理を再利用テストとオーバラップさせることで，再利用テストのオーバヘッドを緩和している．ここで，再利用テストは，その成功，失敗に応じて次に実行される命令が切り替わるという条件分岐のような性質を持っている．そのため，3章で説明したような命令のパイプライン処理と再利用テストを単純にオーバ

ラップさせる方法では，再利用テストのオーバヘッドを緩和できる場合とできない場合がある．

例えば，再利用テストに失敗する場合，再利用テスト中に実行した命令は，計算再利用を適用できなかった関数内部の命令となり，再利用テスト失敗後もその関数内部の命令を実行することになる．そのため，再利用テストと命令のパイプライン処理のオーバラップが，再利用テストのオーバヘッドの緩和に繋がる．実際に，図 11 の再利用テスト失敗時の例を見てみると，2 サイクルを要した再利用テストのオーバヘッドが，1 サイクルに緩和されていることが分かる．一方で，再利用テストに成功する場合，再利用テスト中に実行した命令は，計算再利用の適用により実行が省略される関数内部の命令となる．そのため，再利用テスト中に実行した命令は全て無駄となり，この無駄な命令実行が計算再利用による高速化の効果を妨げてしてしまう可能性がある．実際に，図 10 の再利用テスト成功時の例を見てみると，2 サイクル目から 6 サイクル目までの関数内部の命令実行が無駄となっていることが分かる．

このように，3 章で設計した再利用テストと命令のパイプライン処理のオーバラップ方法では，オーバラップした命令のパイプライン処理が再利用テスト成功時には全て無駄となってしまう．そこで，この再利用テスト成功時の無駄な命令実行を削減し，計算再利用による高速化の効果をより得られるようにする手法を提案する．なお本論文では，再利用テスト中の無駄な実行サイクル（図 10 の例の場合，4 サイクル目から 6 サイクル目）を**検索バブル**，再利用テスト外の無駄な実行サイクル（図 10 の例の場合，2 サイクル目から 4 サイクル目）を**再利用バブル**と呼ぶ．以降では，これら 2 つのバブルを削減するための手法について，順に述べる．

4.2 再利用テスト中のバブルの削減

本節では，再利用テスト成功時における再利用テスト中の無駄な実行サイクル，すなわち，検索バブルを削減するための手法について述べる．

4.2.1 再利用予測機構

前節でも述べたように，再利用テストは条件分岐のような性質を持っている．再利用テストに失敗したときは，次に実行される命令が切り替わることなく，関数内部の命令が実行されるのに対し，再利用テストに成功したときは，次に実行される命令が，関数内部の命令から関数復帰先命令に切り替わる．そこで，分岐予測機構のように，再利用テストの成功，失敗を予測する再利用予測機構を提案する．この機構が再利用テストに失敗すると予測したときは，再利用テスト失敗後に実行することとなる関数内部の命令を再利用テス

ト中に投機的に実行する．一方で，再利用テストに成功すると予測したときは，再利用テスト成功後に実行することとなる関数復帰先以降の命令を再利用テスト中に投機的に実行する．このようにすることで，予測が正しかった場合，再利用テスト終了後に実行することとなる命令を再利用テスト中に投機的に実行することができ，検索バブルを削減することができる．なお，再利用予測機構の予測方法として，本論文では，過去の再利用テストの履歴に基づいた予測方法を採用する．この予測方法の詳細については，4.2.3 項で述べる．

ところで，自動メモ化プロセッサには 2.3 節で述べたようにオーバヘッドフィルタと呼ばれる機構が備わっている．このオーバヘッドフィルタ機構は，ある命令区間に対して再利用の効果があるか否かを判断し，再利用の効果があると判断した命令区間に対してのみ再利用表へのエントリの登録および再利用の適用を行う．そのため，ある関数に対して再利用予測を行なった後に，このオーバヘッドフィルタ機構がその関数に対して再利用の効果がないと判断した場合は，再利用テストが行われず，その関数に対して行った再利用予測のための処理が無駄となってしまう．また，オーバヘッドフィルタ機構がある関数に対して再利用の効果がないと判断した後に，その関数に対して再利用予測を行った場合も同様に，再利用テストが行われないため，その関数に対して行った再利用予測のための処理が無駄となってしまう．そこで，上記のような無駄な再利用予測を行わないようにするために，オーバヘッドフィルタ機構が再利用の効果があると判断した場合にのみ，続けて再利用予測を行うようにする．

4.2.2　再利用予測機構を用いた場合の動作モデル

本項では，再利用予測機構を用いた場合の動作モデルについて説明する．この動作モデルのパターンは，再利用テストに成功すると予測するか，失敗すると予測するかの 2 通りの各予測に対して，実際に行われた再利用テストが成功するか，失敗するかの 2 通りの結果が生じるため，全部で以下の 4 通りに分類することができる．

(SS) 再利用テストに成功すると予測して，実際に成功するパターン
(FS) 再利用テストに失敗すると予測したが，実際には成功するパターン
(FF) 再利用テストに失敗すると予測して，実際に失敗するパターン
(SF) 再利用テストに成功すると予測したが，実際には失敗するパターン

まずはじめに，図 9 で用いた命令列を実行する場合に，(SS) のパターンに該当するパイプライン実行の様子を**図 16** に示す．プログラムの実行が開始され，4 サイクル目にて関数呼び出しのための命令（$\mathtt{I}_{g2}$）がリタイアされると，再利用予測機構はこれから行う再利用テストの成功，失敗を予測する．このとき，再利用テストに成功すると再利用予測機

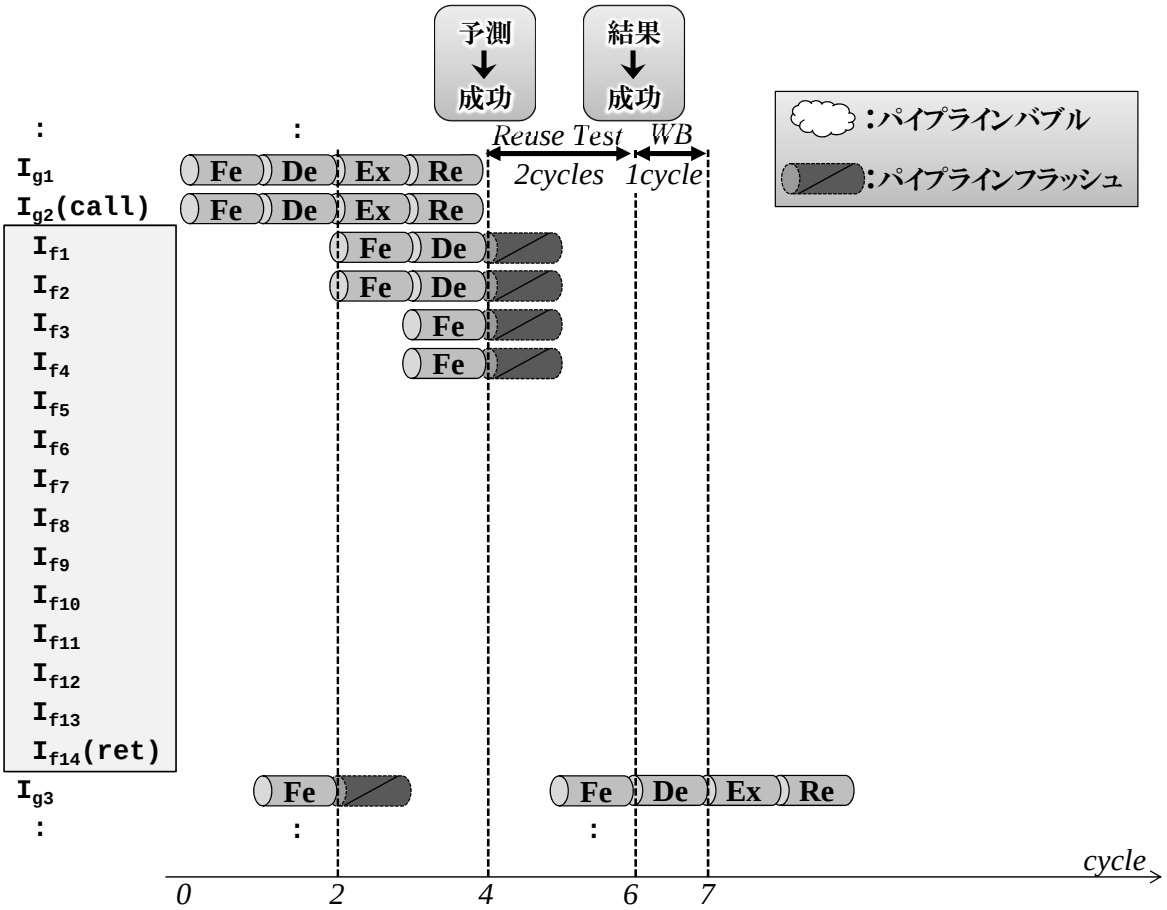

図 16: (SS) のパターンに該当するパイプライン実行の様子

構が予測したとすると，現在パイプラインに投入されている命令をフラッシュし，再利用テストの間，関数復帰先以降の命令を投機的に実行させる．ただし，この間は実行ステージでの処理を禁止している．これは，関数内部の命令を実行していないために，関数復帰先以降の命令のソースオペランドに関するデータ依存が解決されていない可能性があるからである．その後 6 サイクル目にて，予測のとおりに再利用テストに成功したすると，過去の実行結果を再利用表からレジスタやキャッシュに書き戻すことで，関数の実行を省略し，関数復帰先以降の命令の通常実行に戻る．このように再利用予測機構による予測結果を利用することで，再利用テスト成功後に実行する予定の命令を，再利用テスト中に実行することができる．その結果，関数の実行に要したサイクル数は，図 10 の例の場合の 5 サイクルと変わらないが，この例では，検索バブルが発生せず，再利用テスト中に関数復帰先以降の命令を実行できた分，実行サイクル数の削減が期待できる．

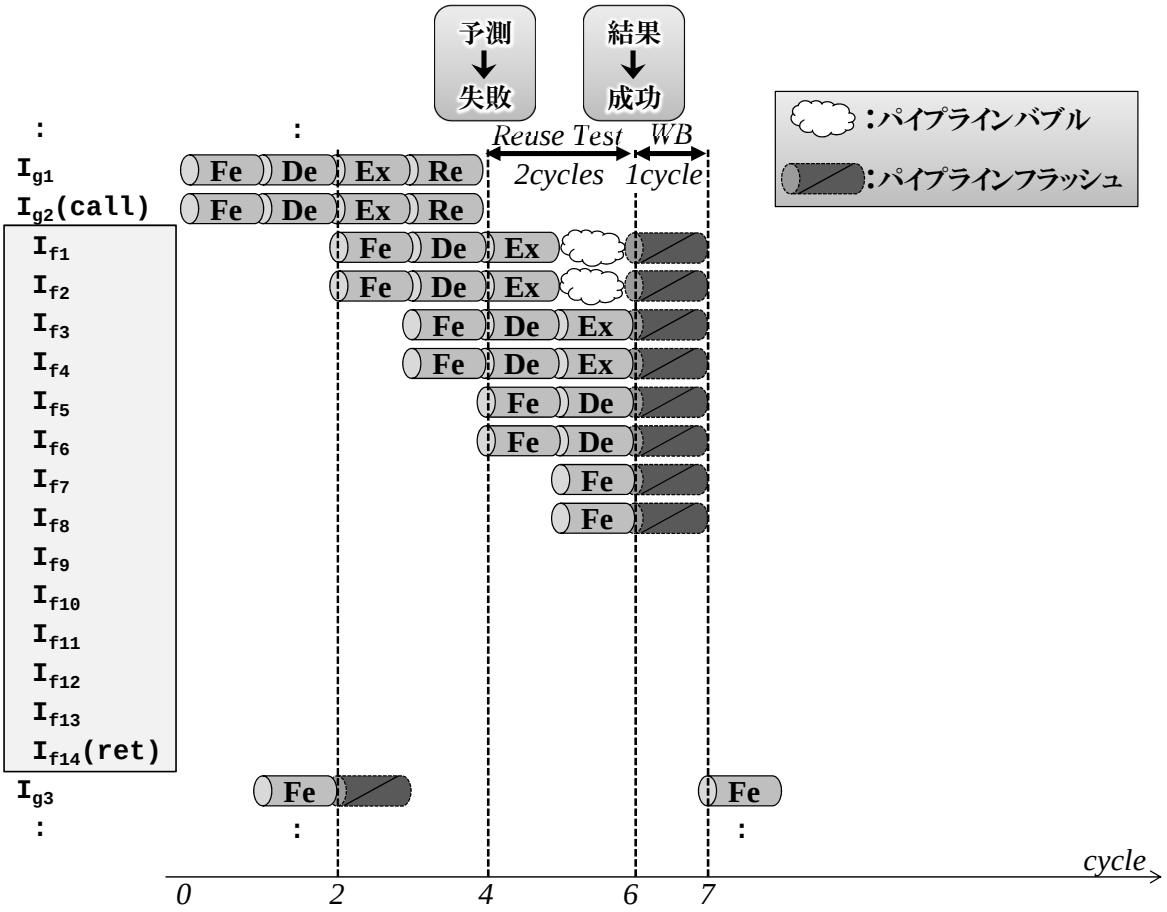

図 17: (FS) のパターンに該当するパイプライン実行の様子

次に，**図 17** に (FS) のパターンに該当するパイプライン実行の様子を示す．4 サイクル目にて，再利用テストに失敗すると再利用予測機構が予測したとすると，再利用テスト中は関数内部の命令を投機的に実行させる．その後 6 サイクル目にて，予測に反して再利用テストに成功したとすると，過去の実行結果を再利用表からレジスタやキャッシュに書き戻すと同時にパイプラインをフラッシュし，関数復帰先以降の命令の通常実行を開始する．この例では，再利用テストに失敗するという予測が実際の結果と異なってしまったため，関数復帰先以降の命令の投機実行に失敗したことになり，検索バブルが発生してしまう．

次に，**図 18** に (FF) のパターンに該当するパイプライン実行の様子を示す．4 サイクル目にて，再利用テストに失敗すると再利用予測機構が予測したとすると，再利用テスト中は関数内部の命令を投機的に実行させる．その後 6 サイクル目にて，予測のとおりに再利用テストに失敗したとすると，禁止していたリタイアステージでの処理を再開し，当該関数の命令の通常実行に戻る．この例では，再利用テストに失敗するという予測が実際の結

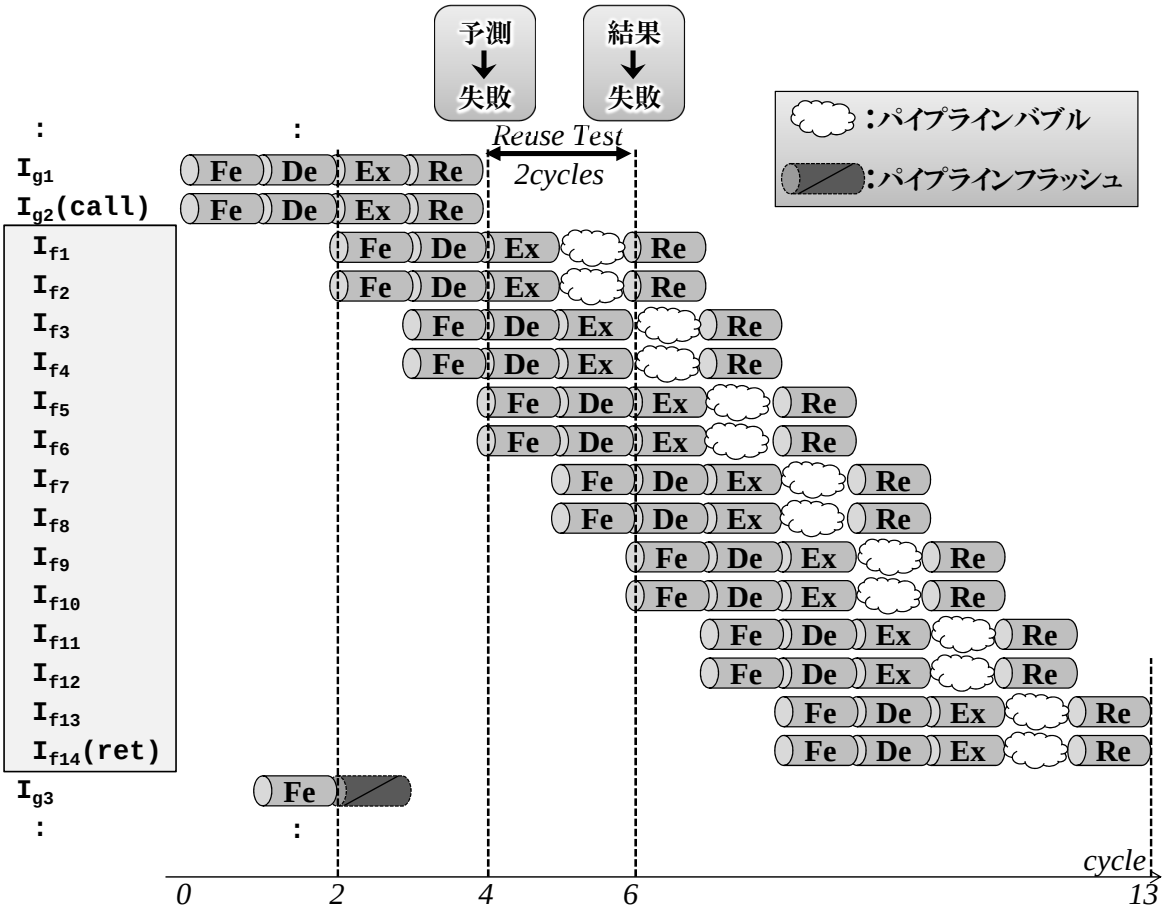

図 18: (FF) のパターンに該当するパイプライン実行の様子

果と一致していたため，関数内部の命令の投機実行に成功したことになり，再利用テストのオーバヘッドを緩和することができている．

最後に，**図 19** に (SF) のパターンに該当するパイプライン実行の様子を示す．4 サイクル目にて，再利用テストに成功すると再利用予測機構が予測したとすると，現在パイプラインに投入されている命令をフラッシュし，再利用テストの間，関数復帰先以降の命令を投機的に実行させる．その後 6 サイクル目にて，予測に反して再利用テストに失敗したとすると，再利用を適用できなかった関数を通常実行するために，現在パイプラインに投入されている関数復帰先以降の命令をフラッシュし，関数の先頭命令から実行を開始する．この例では，再利用テストに成功するという予測が実際の結果と異なっていたため，関数復帰先以降の命令の投機実行に失敗したことになり，再利用テスト終了後に，関数の先頭命令から実行を再開しなければならない．その結果，この関数の実行に要したサイクル数は 17 サイクルとなり，図 9 のメモ化を行わない例と比べて，7 サイクルも増加してしまう．

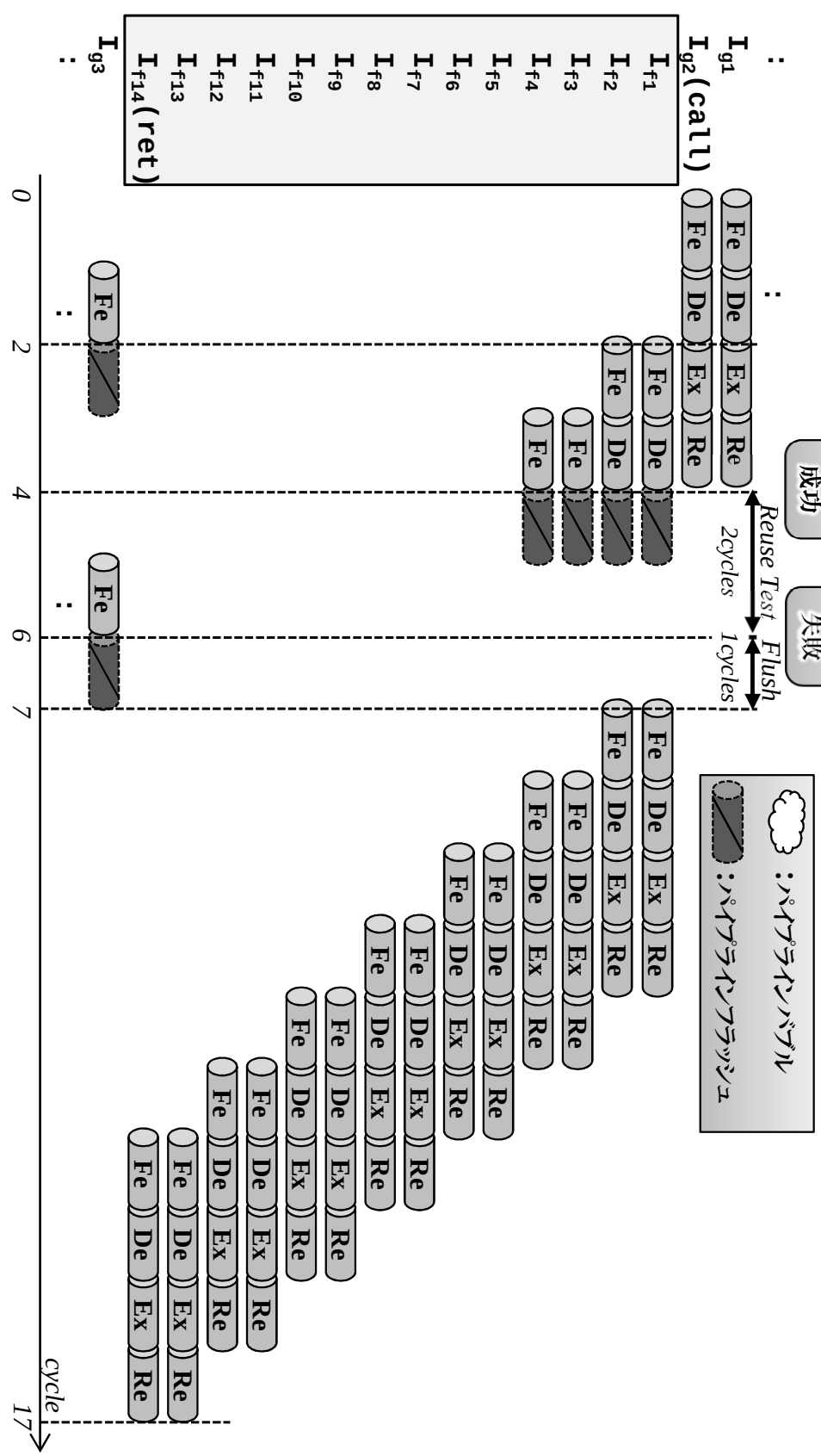

図 19: (SF) のパターンに該当するパイプライン実行の様子

以上で説明したように，(SS) や (FF) のパターンように再利用テストの予測と結果が一致していれば，再利用テスト中の命令の投機実行に成功し，再利用テストのオーバヘッドが緩和できる．しかし，(FS) や (SF) のパターンように再利用テストの予測と結果が異なってしまうと，再利用テスト中の命令の投機実行に失敗してしまい，再利用テストのオーバヘッドが緩和できない．特に，(SF) のパターンに該当する場合では，再利用テストに失敗するもうひとつの動作パターンである (FF) と比べて大幅に性能が低下してしまうと考えられるため，再利用予測機構の予測精度が重要となる．

4.2.3 再利用テストの予測方法

前項で述べたように，検索バブル削減手法の有効性は，再利用予測機構の予測精度に依存する．再利用テストの予測方法については様々なものが考えられるが，本論文では，過去の再利用テストの履歴に基づいた予測方法を採用する．これは，再利用を適用できる命令区間は，その後も頻繁に再利用を適用できる可能性が高く，また，そのような命令区間に対して，再利用テストに成功すると予測できれば，大幅に検索バブルを削減することができると考えられるためである．ここで，再利用テストに失敗すると予測して，実際には成功する場合は，検索バブルが発生するが，再利用テストには成功しているため，予測と結果が異なることのデメリットは小さい．しかし，前項で述べたように，再利用テストに成功すると予測して，実際には失敗してしまう場合は，大幅に実行が遅れてしまうため，予測と結果が異なることのデメリットは大きい．つまり，再利用テストに成功すると予測する条件は，失敗すると予測する条件よりも厳しくしなければならない．そこで，再利用予測機構は過去の再利用テストの履歴から，再利用テストの成功確率を求め，その成功確率がある一定の閾値を超えるときのみ，次の再利用テストに成功すると予測するようにする．そして，この閾値を高く設定することにより，再利用テストに成功すると予測して実際には失敗してしまう場合の，大きなデメリットを抑制する．

4.3 再利用テスト外のバブルの削減

本節では，再利用テスト成功時における再利用テスト外の無駄な実行サイクル，すなわち，再利用バブルを削減するための手法について述べる．例えば，図 16 において，2 サイクル目から 4 サイクル目が再利用バブルとなっている．これは，関数呼び出しのための命令がリタイアされて初めて再利用予測が行われることに起因している．つまり，リタイアステージよりも前に，再利用予測を行うことができれば，再利用テストを行う前から，関数復帰先以降の命令を先行して実行することができ，再利用バブルを削減できると考えられる．ここで，関数呼び出しのための命令はデコードステージで最初に検知可能である．

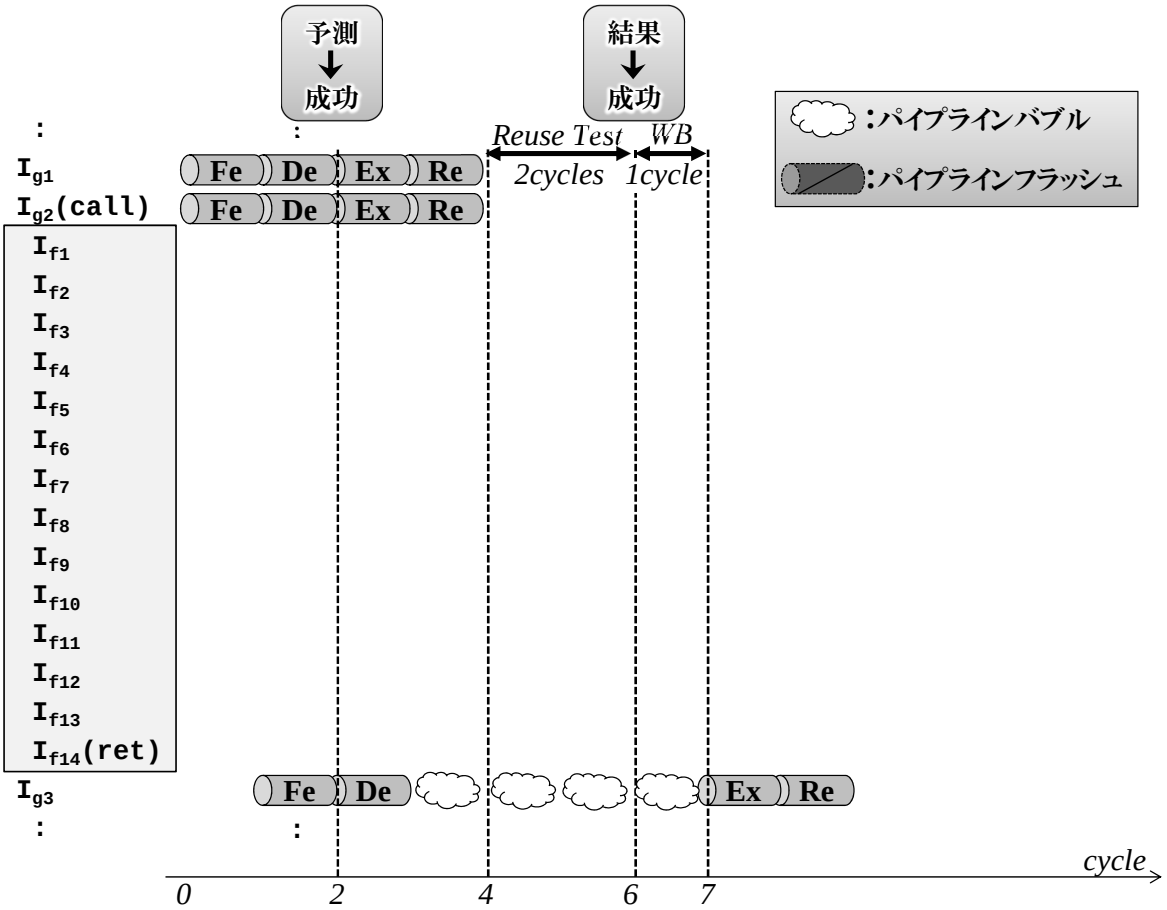

図 20: SS パターンに該当するパイプライン実行の様子

そこで，関数呼び出しのための命令がデコードステージで処理された際に，再利用予測を行うようにする手法を提案する．

4.3.1 デコードステージで再利用予測を行う場合の動作モデル

本項では，再利用バブル削減手法を適用した動作モデルについて説明する．再利用バブル削減手法を適用した動作モデルのパターンは，4.2.2 項で述べた動作モデルのパターンと同じように，(SS)，(FS)，(FF)，(SF) の 4 通りに分類することができる．

まず，(SS) のパターンに該当するパイプライン実行の様子を**図 20** に示す．図の命令列の実行が開始され，2 サイクル目にて関数呼び出しのための命令（I_{g2}）がデコードされると，再利用予測機構は，この関数呼び出しのための命令がリタイアされた時に行われる再利用テストの成功，失敗を予測する．このとき，再利用テストに成功すると予測したとすると，当該再利用テストが終了するまで関数復帰先以降の命令を投機的に実行させる．ただしこの間の実行では，4.2.2 項で述べたように，関数復帰先以降の命令のソースオペ

ランドに関するデータ依存が解決されていない可能性があるため，関数復帰先以降の命令のみ，実行ステージで処理することを禁止している．その後 4 サイクル目にて再利用テストが開始され，6 サイクル目にて再利用テストに成功したとすると，過去の実行結果を再利用表からレジスタやキャッシュに書き戻し，禁止していた実行ステージでの処理を再開することで，関数復帰先以降の命令の通常実行に戻る．このようにすることで，再利用テスト外（2 サイクル目から 4 サイクル目）に無駄な命令実行をすることがなくなり，再利用バブルを削減することができる．ただしこの図では，検索バブル削減手法のみを適用した場合におけるパイプライン実行の様子を表している図 16 と比べて，パイプラインのスループットは変わっていないようにみえる．これは，図 20 では説明の簡単化のために，パイプラインを 4 段としており，発行ステージ，および発行キューを省略しているからである．本来プロセッサは，アウトオブオーダで命令を実行するために発行キューを保持しており，デコードした命令を発行キューに一度蓄え，発行ステージで，この発行キュー内に蓄えられた命令の中から，実行可能な命令を実行ステージに発行する．そのため，発行キュー内に多くの命令が蓄えられているほど，発行ステージでバブルが発生してしまうことが少なくなり，パイプラインのスループットは向上する．ここで，図 20 と図 16 に示す例において，デコードステージで処理された命令が発行キューに蓄えられるとした場合，再利用テスト成功後の 7 サイクル目にこの発行キュー内に蓄えられている命令数は，図 20 に示す例では 10 命令，図 16 に示す例では 2 命令と計算できる．この計算結果から，図 20 に示す再利用バブル削減手法を適用した例のほうが多くの命令を発行キュー内に蓄えているとわかる．そのため，7 サイクル目以降の関数復帰先以降の命令のパイプライン処理で，図 16 に示す例よりもパイプラインのスループットは向上する．

次に，(FS)，(FF) のパターンは，再利用予測を行うタイミングがデコードステージに変更されるのみで，パイプライン実行の様子は図 17，図 18 に示す例と全く同じになる．また，(SF) のパターンは，再利用テストが終了するまで，図 20 に示す，(SS) のパターンに該当するパイプライン実行例と全く同じになり，そして，再利用テストが失敗してからは，図 19 に示す，(SF) のパターンに該当するパイプライン実行例と全く同じになる．

4.3.2 再利用予測を行う対象の関数

前項で述べたように，再利用バブル削減手法では，再利用予測のタイミングと再利用テストのタイミングが異なる．そのため，ある関数呼び出しのための命令をデコードステージで処理してから，その命令がリタイアされ再利用テストが行われるまでに，別の関数呼び出しのための命令をデコードステージで処理する場合が存在する．この場合，その別の関数に対して再利用予測を行うべきか否かを考える必要がある． ここで，そのような場

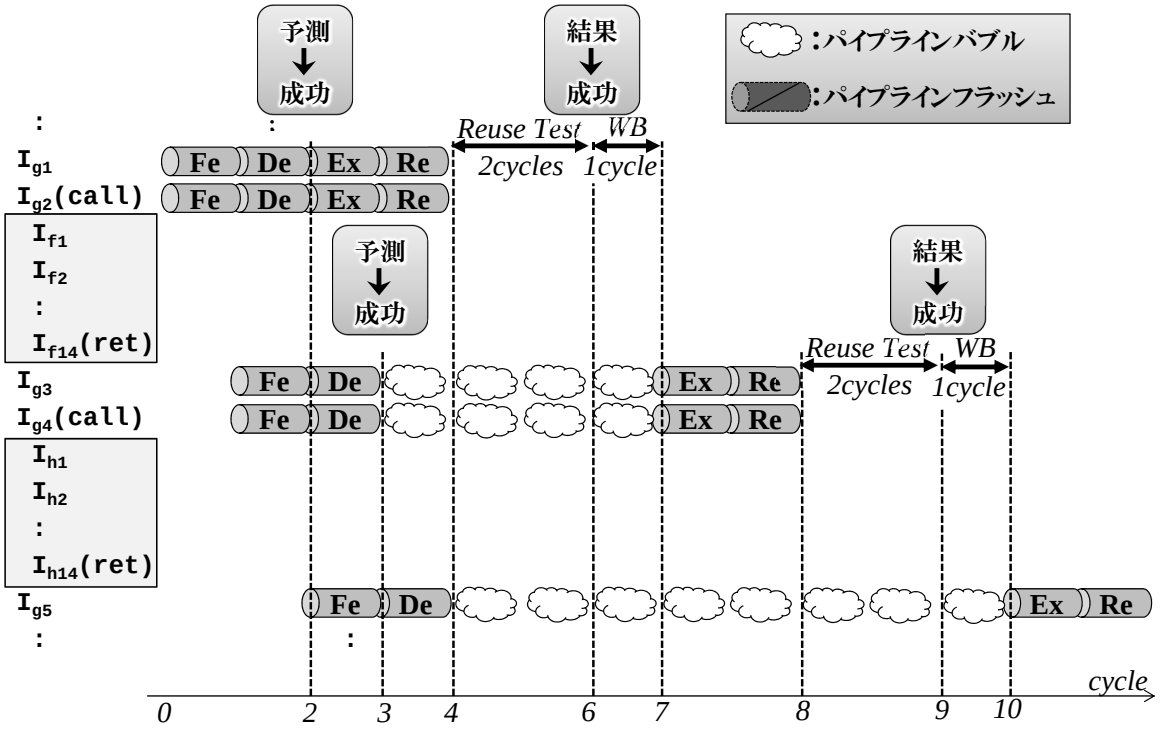

図 21: 再利用予測を行う場合のパイプライン実行の様子

合に再利用予測を行った場合のパイプライン処理の様子を**図 21** に，行わない場合のパイプライン処理の様子を**図 22** に示す．これらの図において，I_{g2} と I_{g4} は関数呼び出しのための命令，I_{f14} と I_{h14} は関数復帰のための命令であり，I_{f1} から I_{f14} までの区間と I_{h1} から I_{h14} までの区間が関数である．ここで，I_{f1} から I_{f14} までの区間の関数のことを関数 f，I_{h1} から I_{h14} までの区間の関数のことを関数 h と呼ぶこととする．

まず，図 21 に示す，再利用予測を行う場合では，2 サイクル目にて関数 f を呼び出すための命令がデコードされた時，再利用予測機構は，その関数 f に対する再利用予測を行う．このとき，再利用テストに成功すると予測したとすると，関数 f に対する再利用テストが終了するまで，関数 f の復帰先以降の命令を実行する．その後，3 サイクル目にて，別の関数 h を呼び出すための命令がデコードされた時，この図に示す例では，その関数 h に対する再利用予測を行っている．このとき，再利用テストに成功すると予測したとすると，関数 h に対する再利用テストが終了するまで，関数 h の復帰先以降の命令を実行する．その後，関数 f に対する再利用テスト，および関数 h に対する再利用テストが成功したとすると，これらの命令実行は全て無駄とはならない．

一方，図 22 に示す，再利用予測を行わない場合では，図 21 に示した例と同様に，2 サイクル目にて，関数 f に対する再利用予測を行う．このとき，再利用テストに成功すると予

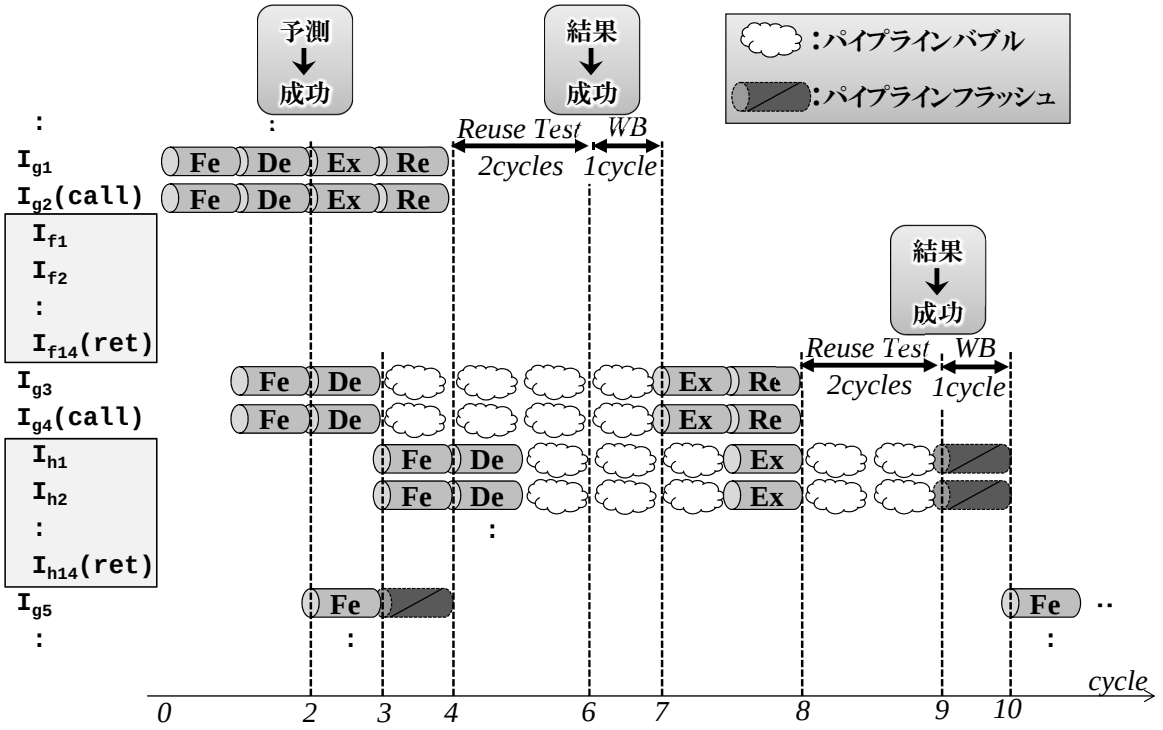

図 22: 再利用予測を行わない場合のパイプライン実行の様子

測したとすると，関数 f に対する再利用テストが終了するまで，関数 f の復帰先以降の命令を実行する．その後，3 サイクル目にて，別の関数 h を呼び出すための命令をデコードした時，この図に示す例では，関数 f に対する再利用テストがまだ終了していないため，再利用予測機構は関数 h に対する再利用予測を行わない．そのためプロセッサは，関数 h の復帰先以降の命令をパイプラインからフラッシュし，関数 h の先頭アドレスでプログラムカウンタの値を上書きすることで，関数 h 内部の命令の実行に移る．その後，図 21 で示した例と同様に，関数 f に対する再利用テスト，および関数 h に対する再利用テストが成功したとすると，パイプラインに投入されている関数 h 内部の命令を実行する必要がないことが判明するため，9 サイクル目にてパイプラインをフラッシュしなければならず，無駄な実行サイクルが発生してしまう．

ここで，以上で述べた 2 つの例で示したそれぞれの方式のメリット，およびデメリットについて考える．まず，図 21 で示した例では，関数 h に対しても再利用予測を行うため，図 21 で示したように再利用テストの予測と結果が一致していれば，命令の実行中にパイプラインをフラッシュすることがなく，無駄な命令サイクルが発生しないというメリットがある．しかし，ある再利用テストが，どの関数の再利用予測に対応するものであるかを，その再利用テストが行われる関数を呼び出した命令と対応付けて管理し，パイプ

ライン実行を制御する必要があり，実装コストが高いというデメリットがある．一方で，図 22 で示した例では，再利用予測は行われたものの，まだ再利用テストが終了していないような関数が存在するか否かを知ることさえできれば，パイプライン実行を簡単に制御することができる，つまり実装コストが低いというメリットがある．しかし，関数 h に対して再利用予測を行わないため，図 22 で示したようにその関数に対する再利用テストが成功すると，無駄な命令実行をしてしまい，計算再利用による性能向上を妨げてしまうデメリットがある．以上の 2 つの例のうち，図 21 で用いている，別の関数にも再利用予測を行う方式のほうが，性能向上がより得られると考えられるが，本論文では実装コストの低さから，図 22 で用いている，別の関数には再利用予測を行わない方式を採用した．

4.4 オーバヘッドフィルタの改良

これまで説明してきた検索バブル削減手法と再利用バブル削減手法の適用により，(SS) のパターンにおける再利用テストのオーバヘッドが緩和されるため，再利用による効果が得られると考えられる命令区間の判断基準を見直す必要がある．そこで，2.3 節で述べたオーバヘッドフィルタを再定義する．ここで，最近の一定回数 T の再利用テストにおける (SS) のパターンの発生回数を M_{SS}，(FS) のパターンの発生回数を M_{FS} とする．また，再利用テストに成功すると予測した場合に，再利用テストの間に実行できるステージ数を N とする．なお，各ステージは 1 サイクルで処理されるものと仮定する．まず，最近の一定回数 T の再利用テストにおいて，再利用の適用により削減できたサイクル数を表す式 (1) を，

$$M_{SS} \cdot (S - Ovh^R - Ovh^W + N) + M_{FS} \cdot (S - Ovh^R - Ovh^W) \tag{4}$$

として再定義する．(SS) のパターンでは，関数復帰先以降の命令を N ステージ分だけ実行できるため，この式 (4) では，そのサイクル数を再利用の適用により削減できるサイクル数に含めている．次に，再利用テストに失敗した場合のオーバヘッドを表す式 (2) を，

$$(T - M_{SS} - M_{FS}) \cdot Ovh^R \tag{5}$$

として再定義する．そして，式 (4) から式 (5) を引くことによって，式 (3) で定義した $Gain$ を，

$$Gain = M_{SS} \cdot (S - Ovh^W + N) + M_{FS} \cdot (S - Ovh^W) - T \cdot Ovh^R \tag{6}$$

として再定義する．この $Gain$ が正値であれば，再利用の効果があると判断し，このように判断した命令区間に対してのみ再利用表への登録および再利用の適用を行うように既存

のオーバヘッドフィルタ機構を改良する．

ところで，4.2 節で述べたように，ある関数に対して再利用の効果があるとオーバヘッドフィルタ機構が判断した場合にのみ，その関数に対して続けて再利用予測が行われる．ここで，再利用バブル削減手法の適用により，再利用予測を行うタイミングが，関数呼び出しのための命令がリタイアされた時から，デコードされた時に変更されている．そのためこのオーバヘッドフィルタ機構が，ある関数に対して再利用の効果があるか否かを判断するタイミングも，関数呼び出しのための命令がリタイアされた時から，デコードされた時に変更する．このようにすることで，ある関数に対して再利用の効果があるとオーバヘッドフィルタ機構が判断した場合にのみ，その関数に対して続けて再利用予測が行われるようにする．

5　再利用予測による高速化手法の実装

本章では，再利用予測による高速化手法を実現するための実装について述べる．

5.1　再利用テスト中のバブルを削減するための拡張

本節では，再利用テスト成功時における再利用テスト中の無駄な実行サイクル，すなわち，検索バブルを削減するための手法の実装について述べる．4.2.3 項で述べたように，再利用予測機構は過去の再利用テストの履歴に基づいて再利用テストの結果を予測する．この予測方法を実現するためには，過去の再利用テストの結果を記憶しておくためのハードウェアが必要である．なお，2.3 節で述べたオーバヘッドフィルタのために実装されている幅 T ビットのシフトレジスタ（hit hist.）を利用すれば，この予測方法は実現可能であるため，検索バブル削減手法のために新たにハードウェアを追加する必要はない．再利用予測機構は，再利用予測時にこのシフトレジスタを参照することで，過去 T 回行った再利用テストの成功率を算出する．そして，その成功率がある一定の閾値を超えていたら再利用テストに成功すると再利用予測機構は予測する．

ところで，この予測方法によって得られた予測結果は再利用予測時に必要となるだけでなく，再利用テスト終了時にも必要となる．例えば，(SS) の場合は，再利用テスト成功時に過去の実行結果を書き戻すのみで，関数復帰先以降の命令の通常実行に戻ることができる．一方で，(FS) の場合は，再利用テスト成功時に過去の実行結果を書き戻すと同時に，パイプラインをフラッシュすることで，関数復帰先以降の命令を実行する必要がある．また，(FF) の場合は，再利用テスト失敗時に禁止していたリタイアステージでの処理を再開するのみで，関数内部の命令の通常実行に戻ることができる．一方で，(SF) の場合は，

再利用テスト失敗時にパイプラインフラッシュを行い，プログラムカウンタの値を関数の先頭アドレスで書き換えることで，関数内部の命令の通常実行に切り替える必要がある．このように，再利用テストの予測と結果に応じて，再利用テスト終了時にパイプライン実行を制御しなければならない．そこで，再利用予測の結果を記憶しておくための 1 ビットのレジスタを再利用予測機構に設ける．再利用予測機構は，再利用テスト終了時まで，再利用予測の結果をこのレジスタに保持しておくことで，このレジスタの値と再利用テストの結果に応じて，上記のように再利用テスト終了時にパイプライン実行を制御することができる．

5.2 再利用テスト外のバブルを削減するための拡張

本節では，再利用テスト成功時における再利用テスト外の無駄な実行サイクル，すなわち，再利用バブルを削減するための手法の実装について述べる．再利用バブル削減手法を適用した場合，ある関数に対して再利用予測を行なってから，その関数に対する再利用テストが開始されるまで，パイプライン中に実行ステージで処理しても良い命令と処理してはならない命令が混在する場合が存在する．例えば，(SS) のパターンに該当するパイプライン実行の様子を表している図 20 の，3 サイクル目のパイプラインの状態を考える．このときパイプライン中には実行ステージで処理しても良い命令として，関数呼び出しのための命令以前の命令（$\mathtt{I}_{g1}$ と $\mathtt{I}_{g2}$）が，処理してはならない命令として，関数復帰先以降の命令（$\mathtt{I}_{g3}$ 以降の命令）が存在することになる．そこで，これら 2 種類の命令を区別するためのフラグ（実行制限フラグ）を発行ステージ SEL/RD までのパイプラインレジスタと発行キューに追加する．実行制限フラグは，再利用テストに成功すると予測した後に ARM-D ステージで処理される関数復帰先以降の命令に対してセットされる．そして，このフラグがセットされた命令を SEL/RD ステージで発行しないようにすることで，実行ステージでの処理の禁止を実現する．なお，実行制限フラグは再利用テストが終了した際に，すべてクリアされる．また，分岐予測ミスなどにより，デコードした関数呼び出しのための命令がフラッシュされ，再利用テストが行われない場合にも実行制限フラグはクリアされる．このように実行制限フラグを用いることで，ある関数に対して再利用予測を行なってから，その関数に対する再利用テストが開始されるまで，パイプライン中に混在する実行ステージで処理しても良い命令と処理してはならない命令を区別することができる．

さて本論文では，4.3.2 項で述べたように，ある関数呼び出しのための命令をデコードステージで処理してから，その関数に対する再利用テストが行われるまでに，別の関数呼び

出しの命令をデコードステージで処理する場合に，その別の関数に対して再利用予測を行わない方式を採用した．この方式を実現するためには，再利用予測は行われたものの，まだ再利用テストが終了していないような関数が存在するかどうかを知る必要がある．そこで，そのような関数の数を記憶しておくためのカウンタ（関数カウンタ）を再利用予測機構に設ける．再利用予測機構は，関数呼び出しのための命令をデコードステージで検知したときに関数カウンタの値をインクリメントし，再利用テストが終了したときにデクリメントする．なお，分岐予測ミスや再利用予測ミスなどにより，パイプラインがフラッシュされる場合には，関数カウンタの値をクリアする．以上のように操作することで，関数カウンタの値が 0 のときは，再利用予測は行われたものの，まだ再利用テストが終了していない関数は存在しないとわかるため，再利用予測を行う．一方で，関数カウンタの値が 1 以上のときは，そのような関数が存在しているとわかるため，再利用予測を行わない．なお，再利用予測が行われない場合，パイプライン実行の制御はベースプロセッサ側に移るため，その動作は，再利用テストに失敗すると予測した場合と同じとなる．

5.3 実行モデル

本節では，検索バブル削減手法と再利用バブル削減手法を実装したスーパスカラ型自動メモ化プロセッサの実行モデルについて説明する．図 22 で用いた命令列を，スーパスカラ型自動メモ化プロセッサが実際に持つ 9 段のパイプラインで実行した場合における命令実行の様子を**図 23** に示す．この図において，F1 は IA ステージ，F2 は IF ステージ，D1 は ARM-D ステージ，D2 は HOST-D ステージ，MAP は MAP/SCH ステージ，SEL は SEL/RD ステージ，EX は BRC，SFM，ALU，EAG，OP1 ステージのいずれか，WR は WR ステージ，RE は RE ステージを表している．また図中では，5.2 節で述べた実行制限フラグの状態を旗で表し，関数カウンタの値を最下段に示している．なお，説明の簡単化のために，パイプラインのウェイ数は 2 で，命令は分解されないものとし．各ステージは 1 サイクルで処理されると仮定する．また，キャッシュミスなどによりパイプラインがストールすることなく，理想的な状態で実行が進むと仮定する．

まず，プログラムの実行が開始され，3 サイクル目にて関数呼び出しのための命令（$\mathtt{I}_{g2}$）が D1 ステージで処理されると，再利用予測機構は，過去の再利用テストの履歴が記憶されているシフトレジスタを参照することで，再利用テストの成功，失敗を予測する．このとき，過去の再利用テストの成功率が，設定したある一定の閾値を超えていたとすると，再利用テストに成功すると予測する．そして，次に実行される命令が，関数呼び出しのための命令（$\mathtt{I}_{g2}$）によって関数内部の命令に切り替わることを，再利用予測機構は中止する

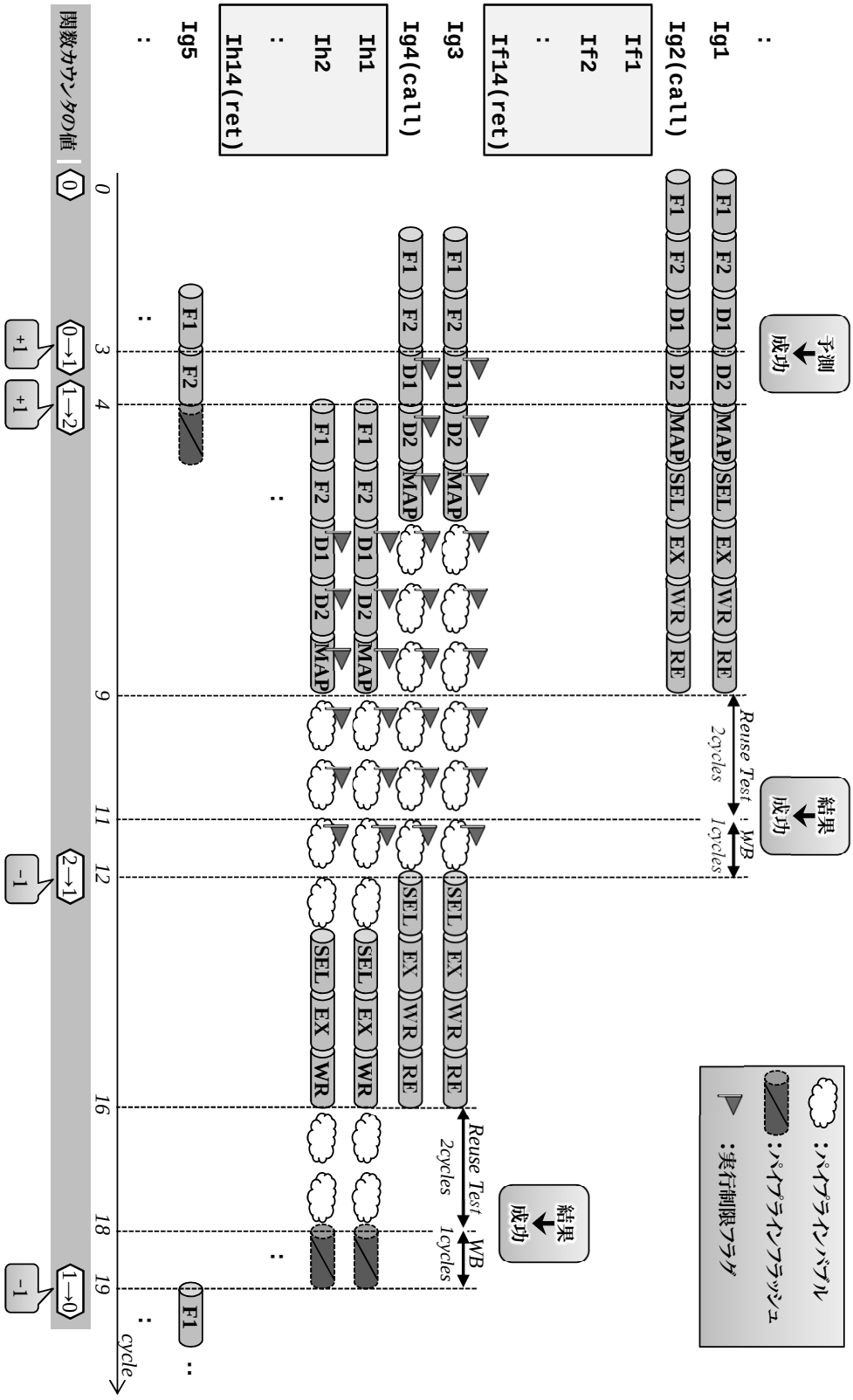

図 23: パイプライン実行の様子

ことで関数復帰先以降の命令を実行させる．ただし，関数復帰先以降の命令は，ソースオペランドのデータ依存が解決されていない可能性があるため，SEL ステージでの処理を禁止しなければならない．そこで，これ以降 D1 ステージで処理する命令に対して実行制限フラグをセットする．またこのとき，関数カウンタの値をインクリメントすることで，この関数に対する再利用テストが終了するまで，他の関数に対して再利用予測を行わないようにする．その後，4 サイクル目にて別の関数呼び出しのための命令（$\mathtt{I}_{g4}$）が D1 ステージで処理されると，再利用予測機構は関数カウンタの値を確認する．このとき，関数カウンタの値が 1 となっているため，再利用予測を行わず，関数カウンタの値をインクリメントする．再利用予測を行わない場合，その後のパイプライン実行はベースプロセッサ側が制御することになる．この場合，再利用テストに失敗すると予測した場合と全く同じ動作となるため，再利用予測機構は予測結果を記憶するためのレジスタに失敗を示す値を記憶する．その後，9 サイクル目にて関数呼び出しのための命令（$\mathtt{I}_{g2}$）が RE ステージで処理されると，再利用テストが開始される．そして，11 サイクル目にて再利用テストが終了し，その再利用テストに成功したとすると，過去の実行結果を再利用表からレジスタやキャッシュに書き戻し，関数カウンタの値をデクリメントする．またこのとき，関数に再利用が適用されたことにより，関数復帰先以降の命令のソースオペランドに関するデータ依存が解決されることになる．そのため，全ての命令に対する実行制限フラグをクリアする．再利用が適用された関数の復帰先以降の命令は，実行制限フラグがクリアされたことにより，通常実行される．その後，16 サイクル目にて関数呼び出しのための命令（$\mathtt{I}_{g4}$）が RE ステージで処理されると，再利用テストが開始される．なお，この再利用テスト中の命令実行では，RE ステージでの処理のみが禁止される．これは，3.3.2 項でも述べたように，関数内の命令がリタイアされてしまうと，一致比較対象の入力値が上書きされる可能性があり，その結果，過去と入力が異なるにも関わらず一致比較に成功してしまうことで不正な再利用が適用されてしまう可能性があるからである．その後，18 サイクル目にて再利用テストが終了し，その再利用テストに成功したとすると，過去の実行結果を再利用表からレジスタやキャッシュに書き戻し，関数カウンタの値をデクリメントする．このとき，再利用テストの予測結果を記憶するためのレジスタには失敗を示す値が記憶されており，再利用予測と再利用テストの関係が (FS) のパターンに該当するとわかるため，パイプラインをフラッシュし，関数復帰先以降の命令の実行に切り替える．以上のように，追加したハードウェアを用いてスーパスカラ型自動メモ化プロセッサの実行を制御する．

6 評価

3 章で述べたスーパスカラ型自動メモ化プロセッサを実現するために必要な機構と，4 章と 5 章で述べた 2 つの高速化手法を ARM 型スーパスカラプロセッサのシミュレータに実装した．また，ベンチマークプログラムを用いてその性能を評価した．

6.1 評価環境

スーパスカラ型自動メモ化プロセッサの評価に用いたシミュレータの仕様を**表 1** に，既存の単命令発行型自動メモ化プロセッサの評価に用いた SPARC V8 シミュレータの仕様を**表 2** に示す．MemoTbl 内の InTbl に用いる CAM の構成は MOSAID 社の DC18288 [16] を参考にし，既存の単命令発行型自動メモ化プロセッサでは，32Bytes 幅 ×4K 行の 128KBytes とした．また，スーパスカラ型自動メモ化プロセッサでは，既存の単命令発行型自動メモ化プロセッサと同じエントリ数を記憶できるようにするために，64Bytes 幅 ×4K 行の 256KBytes とした．なお，両プロセッサのクロック周波数は各々の CAM のクロック周波数の 10 倍と仮定して検索オーバヘッドを見積もっている．また，オーバヘッドフィルタ機構と再利用予測機構で利用する hit hist. は，幅 64 ビットとし，再利用履歴を十分に保持しておけるようにした．ところで，スーパスカラ型自動メモ化プロセッサのシミュレータにはループに計算再利用を適用するための機構が未実装であるため，両プロセッサとも関数を対象とした計算再利用のみで評価した．

6.2 評価結果

まず，3 章で述べたスーパスカラ型自動メモ化プロセッサが，既存の単命令発行型自動メモ化プロセッサと同様に計算再利用による性能向上を得られるかどうかについて検証するため，以下の 4 つのプロセッサを評価した．

(Ns) 自動メモ化機構を実装していない SPARC ベースの単命令発行型プロセッサ

(Ms) 単命令発行型自動メモ化プロセッサ

(Na) 自動メモ化機構を実装していない ARM ベースのスーパスカラプロセッサ

(Ma) スーパスカラ型自動メモ化プロセッサ

ベンチマークプログラムには，SPEC CPU95 INT (train) のプログラムを用いた．SPARC ベースのプロセッサで評価する場合は，これらのプログラムを gcc-3.0.2 (-O2 -msupersparc) によりコンパイルし，スタティックリンクにより生成したロードモジュールを用いた．一方で，ARM ベースのプロセッサで評価する場合は，gcc-4.1.1 (-O2 -msoft-float -march=armv4)

表 1: スーパスカラ型自動メモ化プロセッサのシミュレータ諸元

MemoBuf	128 KBytes
MemoTbl CAM	256 KBytes
Comparison (register and CAM)	9 cycles / 64 Bytes
Comparison (Cache and CAM)	10 cycles / 64 Bytes
Writeback (MemoTbl to Reg. / Cache)	1 cycle / 64 Bytes
L1 I-cache	16 KBytes
line size	64 Bytes
ways	4 ways
miss penalty	8 cycles
L1 D-cache	32 KBytes
line size	64 Bytes
ways	4 ways
miss penalty	8 cycles
L2 cache	2 MBytes
line size	64 Bytes
ways	4 ways
miss penalty	40 cycles
pipeline stage	
IA (Instruction Address)	1 insn / cycle
IF (Instruction Fetch)	2 insns / cycle
ARM-D (ARM-Instruction Decode)	4 μop. / cycle
HOST-D (Host-Instruction Decode)	4 μop. / cycle
MAP/SCH (Register Mapping/Schedule)	4 μop. / cycle
SEL/RD (Select and Read)	4 μop. / cycle
IE (Instruction Execution)	1 μop. / cycle
WR (Writeback)	1 μop. / cycle
RE (Retire)	4 μop. / cycle
Reorder Buffer	32 entries

表 2: 単命令発行型自動メモ化プロセッサのシミュレータ諸元

MemoBuf	64 KBytes
MemoTbl CAM	128 KBytes
Comparison (register and CAM)	9 cycles / 32 Bytes
Comparison (Cache and CAM)	10 cycles / 32 Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle / 32 Bytes
D1 cache	32 KBytes
line size	32 Bytes
ways	4 ways
latency	2 cycles
miss penalty	10 cycles
D2 cache	2 MBytes
line size	32 Bytes
ways	4 ways
latency	10 cycles
miss penalty	100 cycles
Register windows	4 sets
miss penalty	20 cycles / set

表 3: 実行命令数の削減率

	099. go	124. m88ksim	129. compress	130. li	132. ijpeg	134. perl	147. vortex
(Ma)	3.3%	16.5%	1.2%	5.6%	1.1%	2.5%	24.8%
(Ms)	1.7%	22.3%	<1%	5.5%	<1%	1.4%	28.3%

によりコンパイルし，スタティックリンクにより生成したロードモジュールを用いた．

まず，スーパスカラ型自動メモ化プロセッサ (Ma) と単命令発行型自動メモ化プロセッサ (Ms) における実行命令数の削減率を**表 3** に示す．なお，表の (Ma) の行には (Na) の実行命令数に対する削減率を，(Ms) の行には (Ns) の実行命令数に対する削減率を示している．この結果から，スーパスカラ型自動メモ化プロセッサ (Ma) では，計算再利用により，関数の実行を省略できたことによって，多くのベンチマークプログラムで，実行命令

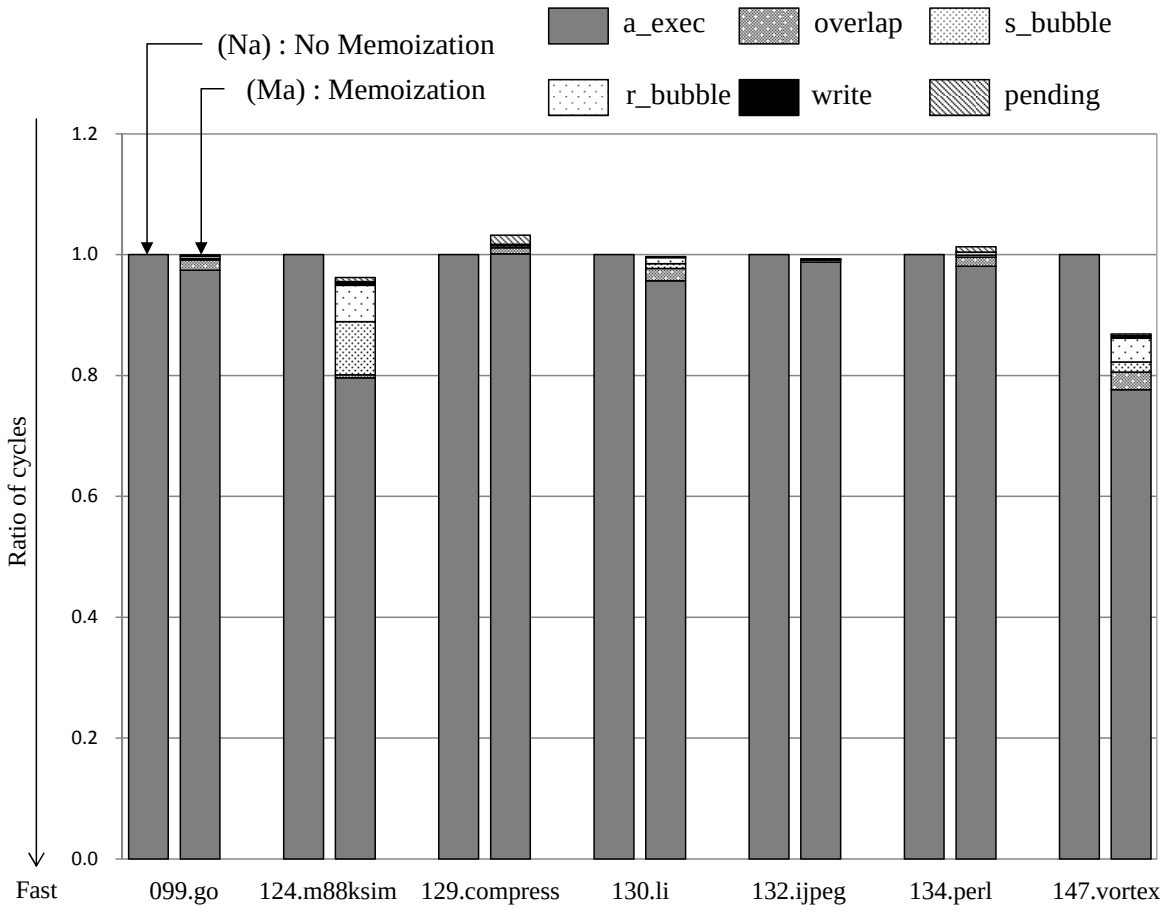

図 24: 実行サイクル数比（スーパスカラ型自動メモ化プロセッサ）

数が削減されていることがわかる．また，その削減率を単命令発行型自動メモ化プロセッサ (Ms) と比較すると，ほぼ同等の結果となっていることがわかる．特に，単命令発行型自動メモ化プロセッサ (Ms) で大きな削減率を達成している，124.m88ksim や 147.vortex でも，スーパスカラ型自動メモ化プロセッサ (Ma) の削減率は，(Ms) の削減率に近い数字となっている．これらのことから，自動メモ化プロセッサはそのベースアーキテクチャに依存することなく，関数に計算再利用を適用し，実行命令数を削減できることが確認できた．プログラム全体では，スーパスカラ型自動メモ化プロセッサ (Ma) の実行命令数の削減率は平均で 7.5%となり，実行サイクル数の削減や，IPC の向上が期待できる．

次に，上述した 4 つのプロセッサの実行サイクル数を評価した． その結果を，**図 24**，および**図 25** に示す．これらの図は，各ベンチマークプログラムの結果を 2 本のグラフで示しており，図 24 は自動メモ化機構を実装していない ARM ベースのスーパスカラプロセッサ (Na)，およびスーパスカラ型自動メモ化プロセッサ (Ma) が要したサイクル数

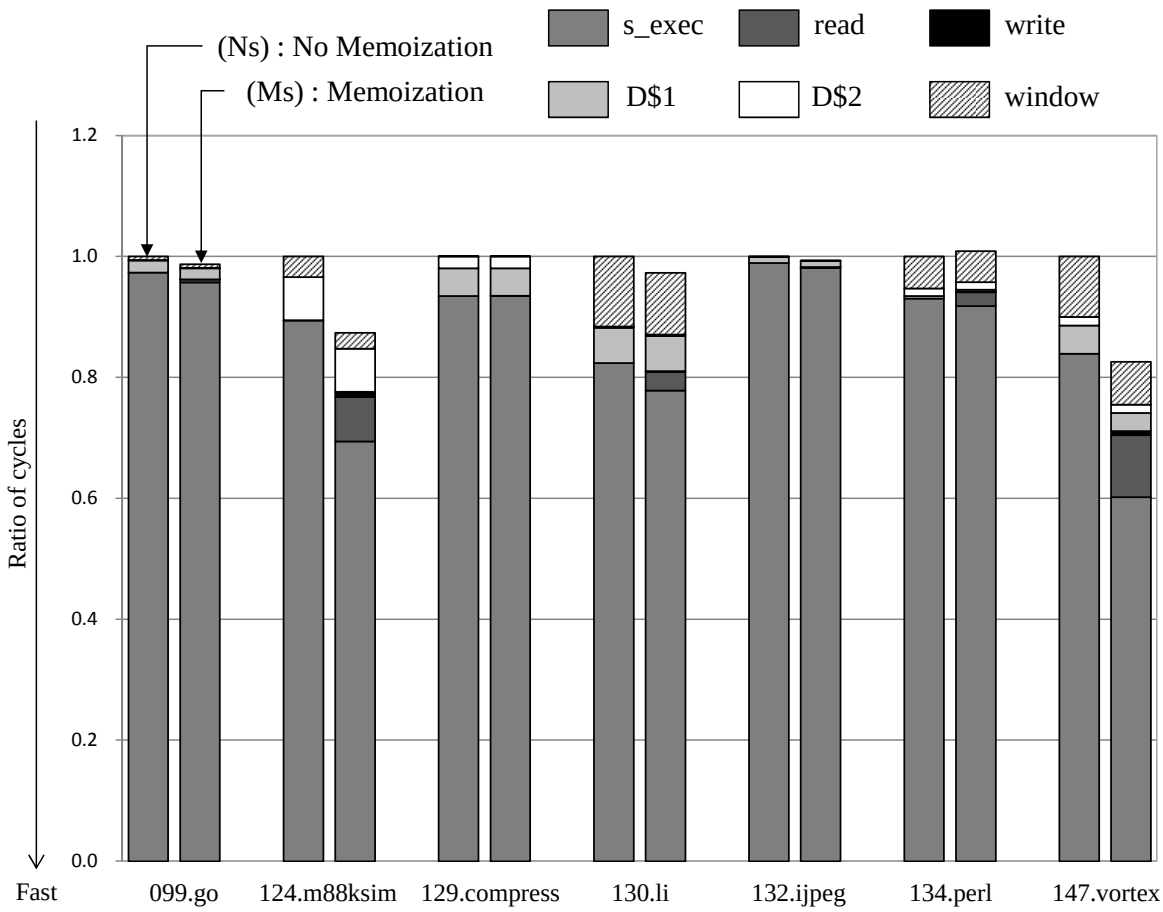

図 25: 実行サイクル数比（単命令発行型自動メモ化プロセッサ）

を，図 25 は自動メモ化機構を実装していない SPARC ベースの単命令発行型プロセッサ (Ns)，および単命令発行型自動メモ化プロセッサ (Ms) が要したサイクル数をそれぞれ表している．なお，図 24 の各グラフは (Na) の実行サイクル数を 1 として，図 25 の各グラフは (Ns) の実行サイクル数を 1 として正規化している．凡例はサイクル数の内訳を示しており，図 24 の a_exec は 1 次および 2 次キャッシュミスペナルティを含んだ命令実行サイクル数，overlap は再利用テスト中において，命令の実行が無駄とならなかったオーバラップサイクル数，s_bubble は検索バブル，すなわち，再利用テスト中の無駄な実行サイクル数，r_bubble は再利用バブル，すなわち，再利用テスト外の無駄な実行サイクル数，write は MemoTbl の出力をレジスタやメモリに書き込む際に要したサイクル数（書き戻しオーバヘッド），pending は再利用テストを開始するためにキャッシュコントローラがビジー状態からアイドル状態に遷移するまで待機していたサイクル数である．一方で，図 25 の s_exec は命令実行サイクル数，read は現在の入力と MemoTbl 内の入力との比較に

要したサイクル数（検索オーバヘッド），write は MemoTbl の出力をレジスタやメモリに書き込む際に要したサイクル数（書き戻しオーバヘッド），D$1 および D$2 は 1 次と 2 次データキャッシュミスペナルティ，window はレジスタウインドウミスペナルティである．

各ベンチマークプログラムについて見てみると，前述した表 3 の場合と同様に，そのサイクル数削減率はスーパスカラ型自動メモ化プロセッサ (Ma) と単命令発行型自動メモ化プロセッサ (Ms) で同じ傾向を示していることが確認できる．プログラム全体では，スーパスカラ型自動メモ化プロセッサ (Ma) のサイクル数削減率が，平均 2.1%，最大 13.1%，単命令発行型自動メモ化プロセッサ (Ms) のサイクル数削減率が，平均 5.1%，最大 17.4%となった．これらの結果から，スーパスカラ型自動メモ化プロセッサ (Ma) のサイクル数削減率は，単命令発行型自動メモ化プロセッサ (Ms) と同様の傾向を示すものの，全体としてはその削減率が小さくなってしまっていることがわかる．この理由として，単命令発行型自動メモ化プロセッサ (Ms) で大きく性能向上している 124.m88ksim や 147.vortex で，スーパスカラ型自動メモ化プロセッサ (Ma) が，(Ms) ほど性能向上していないことが挙げられる．そこで，図 24 の 124.m88ksim，および 147.vortex について詳しく見てみると，147.vortex では，無駄な実行サイクルである s_bubble や r_bubble が総実行サイクル数の約 6%，124.m88ksim では約 15%も占めていることが確認できる．つまり，この無駄な実行サイクルによって，スーパスカラ型自動メモ化プロセッサ (Ma) のサイクル数削減率が単命令発行型自動メモ化プロセッサ (Ms) よりも小さくなってしまったと考えられる．

最後に，自動メモ化機構を実装していない ARM ベースのスーパスカラプロセッサ (Na) とスーパスカラ型自動メモ化プロセッサ (Ma) の IPC（Instructions Per Cycle）を評価した．その結果を，**図 26** に示す．図では各ベンチマークプログラムの結果を 2 本のグラフで示しており，左は自動メモ化機構を実装していない ARM ベースのスーパスカラプロセッサ (Na) の IPC を，右はスーパスカラ型自動メモ化プロセッサ (Ma) の IPC を示している．なお，多数の内部命令に分解される複雑な命令と，単純な加算命令のような分解されない命令を同じ 1 命令として評価することは適切ではないため，分解後の内部命令の IPC を評価した．また，自動メモ化プロセッサは命令の実行自体を省略しつつ，メモ化を行わない場合と同じ結果を出力するため，実際に実行された命令数で IPC を算出してしまうと，計算再利用による性能向上が正しく評価できない．そこで本評価では，実行が省略された命令も含め，本来実行すべきであった命令数を用いて，IPC を算出した．

この図より，124.m88ksim，132.ijpeg，147.vortex のプログラムでは，計算再利用により IPC が向上していることがわかる．しかし，129.compress や 134.perl では IPC が低下してしまっている．これは，129.compress や 134.perl では，表 3 や図 24 を見てわかるよ

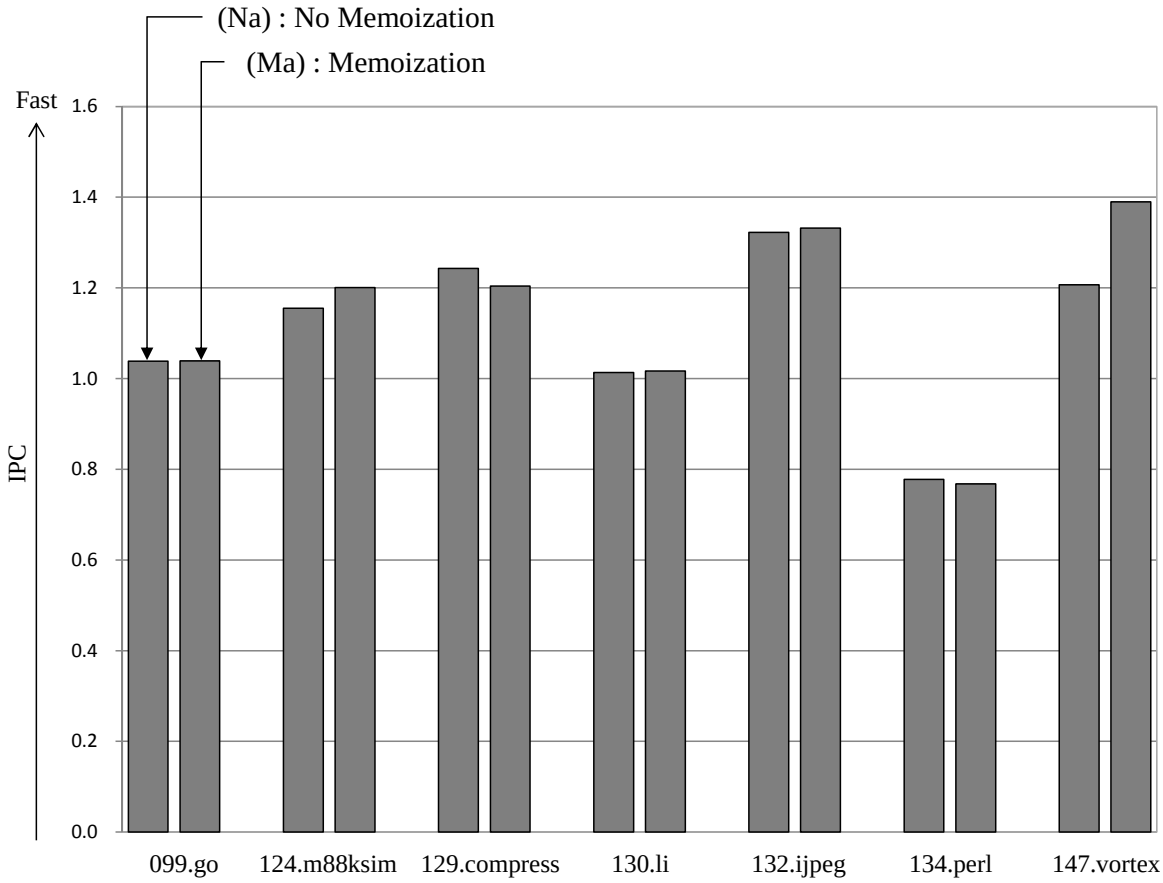

図 26: IPC（スーパスカラ型自動メモ化プロセッサ）

うに，計算再利用を適用できた関数がほとんど存在しておらず，また，再利用テストのオーバヘッドによって実行サイクル数が増加してしまったからである．結果をまとめると，自動メモ化機構を実装していない ARM ベースのスーパスカラプロセッサ (Na) の IPC は平均 1.09，最大 1.32 となったのに対し，そのプロセッサに自動メモ化機構を実装したスーパスカラ型自動メモ化プロセッサ (Ma) の IPC は平均 1.12，最大 1.39 となり，スーパスカラ型自動メモ化プロセッサの有効性を確認できた．

以上の 3 つの評価結果より，スーパスカラ型自動メモ化プロセッサは既存の単命令発行型自動メモ化プロセッサと同様に計算再利用による性能向上を得られることがわかり，自動メモ化プロセッサの実用性を確認できた．しかし，その性能向上率は，無駄な実行サイクルである検索バブルと再利用バブルの発生により，単命令発行型自動メモ化プロセッサよりも小さくなってしまうことがわかった．ところで，本論文ではスーパスカラ型自動メモ化プロセッサを設計するだけでなく，これらのバブルを削減するための高速化手法も提

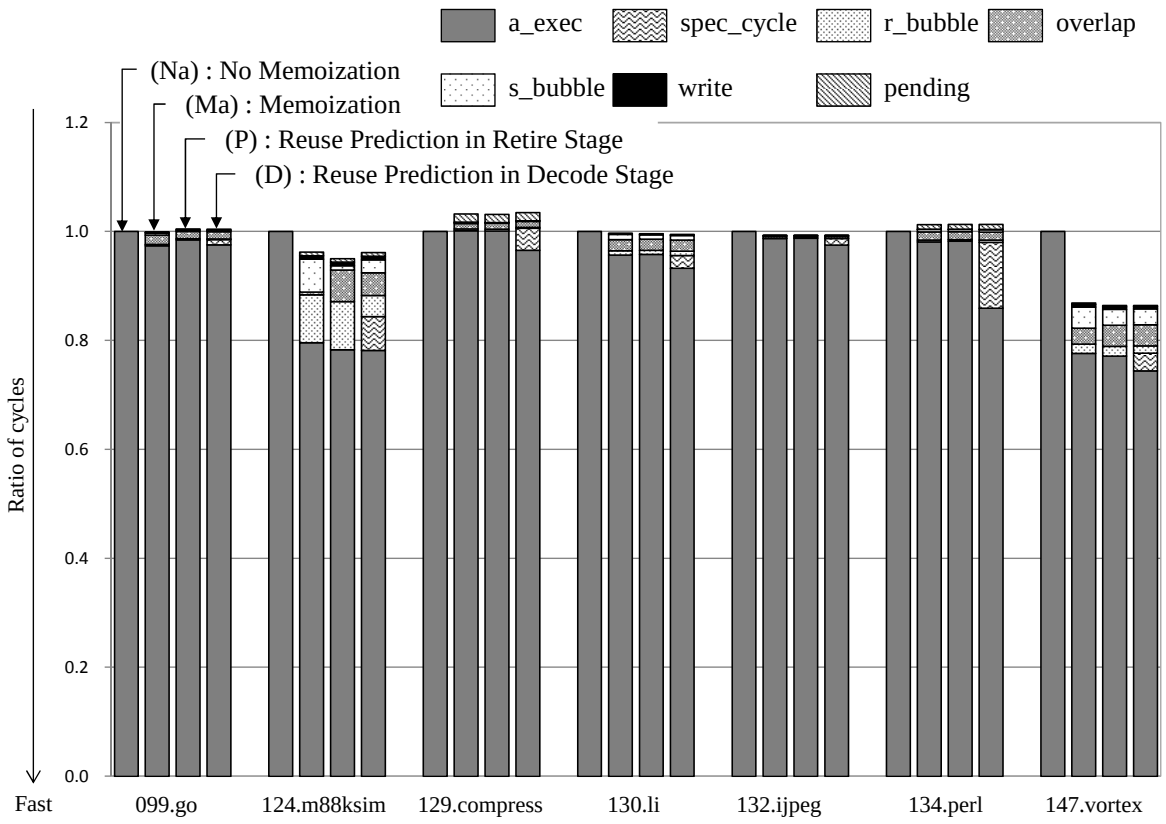

図 27: 実行サイクル数比

案した．この高速化手法が効果的に作用すれば，この性能向上率の差を埋めることができると考えられる．

そこで，本論文で提案した高速化手法である，検索バブル削減手法と再利用バブル削減手法の有効性を検証するために，これまでに評価したスーパスカラ型自動プロセッサに加え，以下の 2 つのプロセッサの実行サイクル数を評価した．

(P) (Ma) に検索バブル削減手法を適用したスーパスカラ型自動メモ化プロセッサ

(D) (P) に再利用バブル削減手法を適用したスーパスカラ型自動メモ化プロセッサ

その結果を，**図 27** に示す．この図は，各ベンチマークプログラムの結果を 4 本のグラフで示しており，左から，自動メモ化機構を実装していない ARM ベースのスーパスカラプロセッサ (Na)，スーパスカラ型自動メモ化プロセッサ (Ma)，検索バブル削減手法を適用したスーパスカラ型自動メモ化プロセッサ (P)，再利用バブル削減手法を適用したスーパスカラ型自動メモ化プロセッサ (D) が要したサイクル数を表しており，(Na) の実行サイクル数を 1 として正規化している．また，凡例の spec_cycle は，再利用バブル削減手法を適用したスーパスカラ型自動メモ化プロセッサ (D) において，再利用予測を行なってから，

再利用テストを開始するまでに無駄とならなかった命令実行サイクル数を表している．なお，(P) と (D) において，4.2.3 項で述べたように，再利用テストに成功すると予測して，実際には失敗してしまう場合は，大幅に実行が遅れてしまうため，再利用テストに成功すると予測する条件を厳しくし，再利用テストの成功回数が過去 64 回の 9 割以上に相当する 58 回以上に達したとき，次の再利用テストに成功すると予測するようにパラメータを設定した．

まず，検索バブル削減手法を適用したスーパスカラ型自動メモ化プロセッサ (P) について見てみると，124.m88ksim や 147.vortex で，検索バブルを表す s_bubble が削減されることで性能向上していることがわかる．しかし，s_bubble の削減率と比べて，(P) の性能向上率はわずかであることが確認できる．これは，再利用テストに成功すると予測した場合，再利用テスト中は，関数復帰先以降の命令は MAP/SCH ステージまでしか実行することができず，SEL/RD ステージで待機していることが多いためだと考えられる．そのため今後は，再利用テストに成功すると予測した場合，再利用テスト中に SEL/RD ステージで処理しても良い命令，すなわち，ソースオペランドが関数内部の命令に依存していない関数復帰先以降の命令を検出し，そのような命令を実行ステージに発行する機構を実装していく必要がある．

次に，再利用バブル削減手法を適用したスーパスカラ型自動メモ化プロセッサ (D) について見てみると，124.m88ksim では，(P) よりも性能が低下していることがわかる．4.3.2 項で述べたように再利用バブル削減手法では，ある関数呼び出しのための命令をデコードステージで処理してから，その関数に対する再利用テストが行われるまでに，別の関数呼び出しのための命令をデコードステージで処理する場合に，その別の関数に対して再利用予測を行わないが，124.m88ksim の性能低下はこのことに起因している．このように，その別の関数に対して再利用予測を行わない場合，その関数に対する再利用テストが成功した場合には，検索バブルが発生してしまう．実際に，その別の関数に対して再利用予測を行わない場合のパイプライン実行の様子を示している図 22 を見てみると，関数 h に対する再利用テスト中（8 サイクル目から 9 サイクル目）に無駄な命令実行をしており，検索バブルが発生していることがわかる．一方で，検索バブル削減手法では，再利用予測のタイミングと再利用テストのタイミングが同じであるため，その別の関数に対しても再利用予測を行うことができ，再利用テストの予測と結果が一致した場合には，検索バブルは発生しない．ここで，図 22 中で用いている命令列を，検索バブル削減手法のみを適用したスーパスカラ型自動メモ化プロセッサが実行した場合におけるパイプライン実行の様子を**図 28** に示す．この図では，8 サイクル目にて，関数 h に対して再利用予測が行われるた

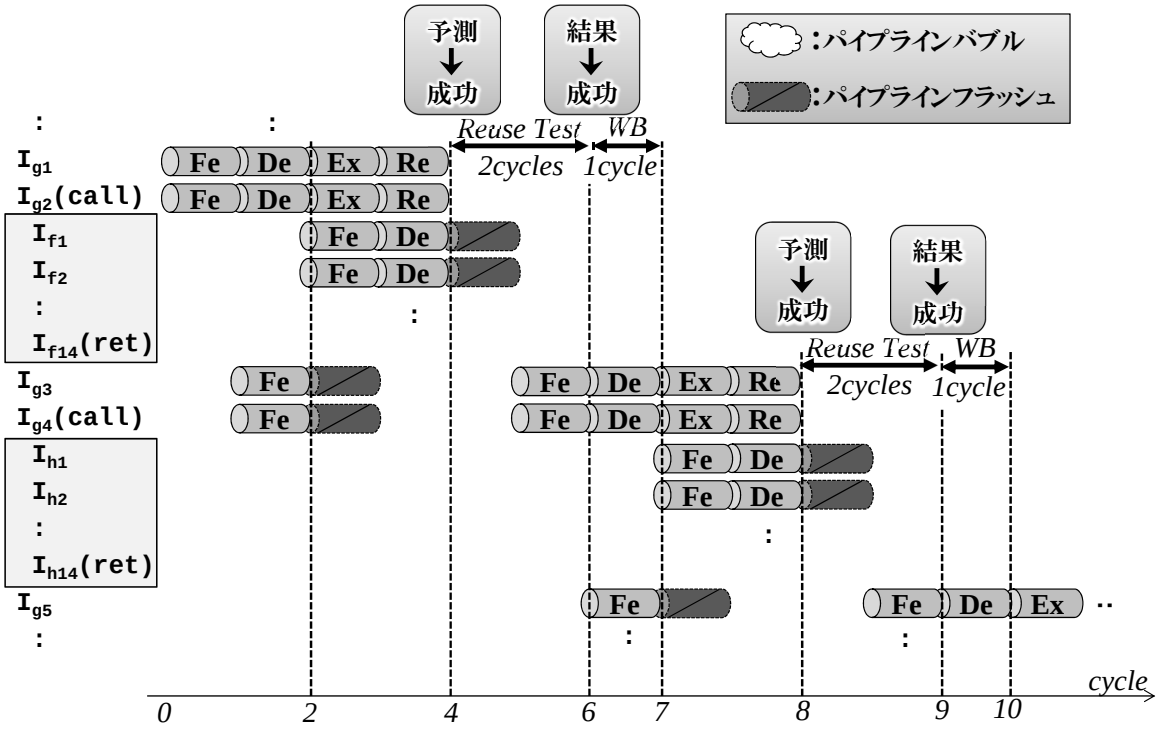

図 28: 検索バブル削減手法のみを適用した場合のパイプライン実行の様子

め，再利用テスト中に，再利用テスト成功後に実行する関数復帰先以降の命令を投機的に実行することができ，検索バブルが発生しない．その結果，図 22 と図 28 を比較すると，図 22 に示す，再利用バブル削減手法を適用した場合のほうが，パイプラインのスループットは低下していることがわかる．124.m88ksim では，再利用バブル削減手法を適用したことによる r_bubble の削減サイクル数よりも，上述した理由による s_bubble の増加サイクル数のほうが上回ってしまい，(P) よりも性能低下した．そのため今後は，ある関数呼び出しのための命令をデコードステージで処理してから，その関数に対する再利用テストが行われるまでに，別の関数呼び出しのための命令をデコードステージで処理する場合，その別の関数に対して再利用予測を行うように再利用バブル削減手法を改良する必要がある．

結果をまとめると，(Ma) ではサイクル数の削減率が最大 13.1%，平均 2.4%だったのに対し，(P) では最大 13.6%，平均 2.7%，(D) では最大 13.6%，平均 2.5%となった．この結果から，本論文で提案した高速化手法では，スーパスカラ型自動メモ化プロセッサの性能向上率を単命令発行型自動メモ化プロセッサの性能向上率に近づけることはできたが，その差を完全に埋めることはできなかった．そのため今後は，提案した高速化手法を改良することで，スーパスカラ型自動メモ化プロセッサの性能をさらに向上させる手法を検討

表 4: (SS)，(FS)，(FF)，(SF) の各実行パターンの発生回数

	099. go	124. m88ksim	129. compress	130. li	132. ijpeg	134. perl	147. vortex
(SS)	2,672	979,199	8,422	34,915	10,008	3	2,583,811
(FS)	97,945	90,432	8,893	150,474	178,557	26,147	3,834,596
(FF)	595,515	1,685,258	1,103,645	4,146,642	3,181,871	800,985	39,806,689
(SF)	468	509	185	1,409	889	2	91,048
正答率	98.4%	96.7%	99.2%	96.5%	94.7%	96.8%	91.5%

する必要がある．

6.3 考察

本節では，検索バブル削減手法を適用したスーパスカラ型自動メモ化プロセッサ (P) における再利用予測の精度について詳しく考察する．(P) において，(SS)，(FS)，(FF)，(SF) の各実行パターンの発生回数を各ベンチマークプログラムごとに調査した．その結果を，**表 4** に示す．この表に正答率として示してある数値は，(SS) と (FF) の発生回数の和を，全実行パターンの発生回数の総和で除算し，その結果を 100 分率で表したものである．つまり，再利用テストの予測と結果が一致していた確率を表している．この結果から，全てのベンチマークプログラムで，正答率が約 90%以上となっており，過去の再利用テストの履歴に基づいた予測方法の有効性を確認できた．また，(SF) のパターンがほとんど発生していないことから，再利用テストの成功回数が過去 64 回の 9 割以上に相当する 58 回以上に達した時，次の再利用テストに成功すると予測するようにしたパラメータの設定が適切に作用していることも確認できた．

しかしながら，(Ma) と比べて (P) の性能が向上しているベンチマークプログラムである 147.vortex に着目してみると，(FS) の実行パターンの発生回数が (SS) の実行パターンの発生回数を上回っていることがわかる．そのため，予測方法の改良により，さらなる性能向上を狙える可能性がある．そこで，この 147.vortex について，各関数の再利用テストの履歴を詳細に調査し，各関数ごとにその特徴をまとめた．その結果を，**図 5** に示す．なお，関数の数は膨大であるため，再利用テストに成功した回数が多い関数のみ示した．各関数について，開始アドレス（addr），再利用テストの成功回数（reuse），(SS) のパターンの発生回数（count），関数の特徴（characteristic）を示している．この表より，開始ア

表 5: 再利用テストの履歴から得られた各関数の特徴 (147.vortex)

addr	reuse	count	characteristic
0x818e4	1,266,896	1,205,719	頻繁に再利用される
0x52514	878,096	0	2 回に 1 回再利用される
0x721d8	869,468	0	2 回に 1 回再利用される
0x37e5c	798,002	596,066	頻繁に再利用される

ドレスが 0x818e4 と 0x37e5c の関数は，再利用テストに成功する多くの場合に，再利用テストに成功すると予測できていることが分かる．これは，これらの関数が頻繁に再利用されることから，前節で設定した，再利用テストの予測方法のパラメータ設定で適切な予測ができたからである．一方で，開始アドレスが 0x52514 と 0x721d8 の関数は，(SS) のパターンの発生回数が 0 回であり，一度も (SS) のパターンで実行されることがなかったことがわかる．これらの関数は，2 回に 1 回再利用テストに成功する，すなわち，再利用テストに成功する，失敗する，成功する，失敗する，が繰り返される実行フェーズが多く，再利用テストに成功すると予測する条件を厳しく設定した予測方法では，これらの関数に対して再利用テストに成功すると予測することができなかったからである．そのため，各関数の再利用履歴の特徴に応じて，再利用テストに成功すると予測する条件を動的に変更することで，開始アドレスが 0x52514 と 0x721d8 などの関数に対しても，適切な再利用予測が行えるようになれば，147.vortex でさらに性能向上すると考えられる．

7 おわりに

本論文では，自動メモ化プロセッサの実用的なパフォーマンスを評価するために，ARM アーキテクチャベースのスーパスカラプロセッサ上に自動メモ化機構を実装する場合に発生しうる問題を検討し，スーパスカラ型自動メモ化プロセッサを設計した．また，設計したスーパスカラ型自動メモ化プロセッサに対する高速化手法として，再利用テストの結果を予測することで，計算再利用適用時の無駄な実行サイクルを削減し，再利用による高速化の効果をより得られるようにする手法を提案した．

設計したスーパスカラ型自動メモ化プロセッサとそれに対する高速化手法の有効性を検証するために，SPEC CPU95 ベンチマークを用いてシミュレーションにより評価した．評価の結果，スーパスカラ型自動メモ化プロセッサは，既存の単命令発行型自動メモ化プロセッサと同様に，計算再利用による性能向上を得られることが確認できた．また，高速

化手法を適用したスーパスカラ型自動メモ化プロセッサでは，再利用テストの結果を正しく予測できたことで，計算再利用適用時の無駄な実行サイクルが削減できることを確認した．その結果，通常通り命令を実行する場合と比較して，高速化手法を適用していないスーパスカラ型自動メモ化プロセッサでは，最大 13.1%，平均 2.4%のサイクル数の削減であったのに対し，高速化手法を適用したスーパスカラ型自動メモ化プロセッサでは，最大 13.6%，平均 2.7%のサイクル数の削減率となり，高速化手法の有効性を確認した．

本研究の今後の課題として，検索バブル削減手法をより効果的にするために，再利用テストに成功すると予測した場合，再利用テスト中に SEL/RD ステージで処理しても良い命令，すなわち，ソースオペランドが関数内部の命令に依存していない関数復帰先以降の命令を検出し，そのような命令を実行ステージに発行する機構を実装することが挙げられる．また，再利用バブル削減手法をより効果的にするために，ある関数呼び出しのための命令をデコードステージで処理してから，その関数に対する再利用テストが行われるまでに，別の関数呼び出しのための命令をデコードステージで処理する場合に，その別の関数に対して再利用予測を行えるようにする機構を実装することも今後の課題として挙げられる．さらに，スーパスカラ型自動メモ化プロセッサには関数に計算再利用を適用するための機構しか現在実装できていないため，ループに計算再利用を適用するための機構を実装することも今後の課題である．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の津邑公暁准教授，松尾啓志教授，齋藤彰一准教授，松井俊浩准教授，梶岡慎輔助教，川島龍太助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた津邑研究室，松尾研究室，齋藤研究室および松井研究室の方々に感謝致します．特に，研究に関して貴重な意見を下さった山田龍寛氏，小田遼亮氏，神村和敬氏，井手上慶氏，橋本高志良氏，松永拓也氏，津村高範氏，佐藤裕貴氏に深く感謝致します．

著者発表論文

論文

1. Takanori TSUMURA, Yuuki SHIBATA, Kazutaka KAMIMURA, Tomoaki TSUMURA, Yasuhiko NAKASHIMA: “Hinting for Auto-Memoization Processor based on Static Binary Analysis”, Proc. 2nd Int’l Workshop on Computer Systems and Architectures (CSA’14), held in conjunction with CANDAR’14, pp.426–432 (Dec. 2014)

2. Yuuki SHIBATA, Takanori TSUMURA, Tomoaki TSUMURA, Yasuhiko NAKASHIMA: “An Implementation of Auto-Memoization Mechanism on ARM-based Superscalar Processor”, Proc. Int’l. Symp. on System-on-Chip 2014 (SoC2014), 8 pages (Oct. 2014)
3. Yuuki SHIBATA, Kazutaka KAMIMURA, Tomoaki TSUMURA, Hiroshi MATSUO, Yasuhiko NAKASHIMA: “CAM Size Reduction Method for Auto-Memoization Processor by considering Characteristics of Loops”, Proc. 1st Int’l Workshop on Computer Systems and Architectures (CSA’13), held in conjunction with CANDAR’13, pp.378–384 (Dec. 2013)

報文

1. 柴田 裕貴, 津村 高範, 津邑 公暁, 中島 康彦: “ARM 型スーパスカラプロセッサにおける自動メモ化機構の実装と評価”, 情処研報, Vol.2014-ARC-212, No.11, pp.1–9 (Oct. 2014)
2. 津村 高範, 柴田 裕貴, 神村 和敬, 津邑 公暁, 中島 康彦: “実行バイナリの静的解析による自動メモ化プロセッサの高速化”, 情処研報 (SWoPP2014), Vol.2014-ARC-211, No.13, pp.1–9 (Jul. 2014)
3. 柴田 裕貴, 神村 和敬, 津邑 公暁, 松尾 啓志, 中島 康彦: “再利用表パージアルゴリズムの改良による自動メモ化プロセッサのハードウェア削減手法”, 情処研報 (SWoPP2013), Vol.2013-ARC-206, No.15, pp.1–9 (Jul. 2013).

参考文献

[1] Feehrer, J., Jairath, S., Loewenstein, P., Sivaramakrishnan, R., Smentek, D., Turullols, S. and Vahidsafa, A.: The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets, *IEEE Micro*, Vol. 33, No. 2, pp. 48–57 (2013).

[2] Conway, P. and Hughes, B.: The AMD Opteron Northbridge Architecture, *IEEE Micro*, Vol. 27, No. 2, pp. 10–21 (2007).

[3] ARM Ltd: *The ARM Cortex-A9 Processors* (2007).

[4] Tilera Corporation: *TILE-Gx Processor Family Product Brief* (2009).

[5] Intel Corporation: *Product Brief: Intel C++ Compiler 11.0* (2008).

[6] SunMicrosystems: Sun Studio: C, C++ & Fortran Compilers and Tools, http://developers.sun.com/sunstudio/ (2009).

[7] Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp. 245–250 (2007).

[8] Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).

[9] Huang, J. and Lilja, D. J.: Exploiting Basic Block Value Locality with Block Reuse, *Proc. 5th Int'l Symp. on High-Performance Computer Architecture (HPCA-5)*, pp. 106–114 (1999).

[10] 津邑公暁, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 並列事前実行機構における主記憶値テストの高速化, 情報処理学会論文誌：コンピューティングシステム, Vol. 45, No. SIG 1(ACS 4), pp. 31–42 (2004).

[11] Sun Microsystems: *UltraSPARC III Cu User's Manual* (2002).

[12] ARM Limited: *"ARM Architecture Reference Manual"ARM DDI 0100E* (2000).

[13] AMD Corporation: *AMD Opteron A1100 Processor* (2014).

[14] KOJIMA, T. and NAKASHIMA, Y.: Design of a Centralized Instruction Window Superscalar for Evaluation of OROCHI, *IPSJ SIG Notes*, Vol. 2006, No. 127, pp. 61–66 (2006).

[15] McFarling, S.: Combining Branch Predictors, *Technical report, WRL Technical Note TN-36* (1993).

[16] MOSAID Technologies Inc.: *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edition (2003).