卒業研究論文

パイプラインの使用効率向上による スーパスカラ型自動メモ化プロセッサの高速化

指導教員　　津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 23 年度入学　23115066 番

佐藤 裕貴

平成 27 年 2 月 9 日

パイプラインの使用効率向上による スーパスカラ型自動メモ化プロセッサの高速化

佐藤 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，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．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．

自動メモ化プロセッサは，単命令発行型とスーパスカラ型のそれぞれをベースとした実装が検討されている．なお，市場に流通しているプロセッサには，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一般的に採用されている．そのため本論文では，自動メモ化プロセッサのベースアーキテクチャとして，スーパスカラ型の ARM を採用した，スーパスカラ型自動メモ化プロセッサに着目する．スーパスカラ型自動メモ化プロセッサでは命令のパイプライン処理を，計算再利用のための入力値の一致比較とオーバラップして実行し，その際，入力値の一致比較を行う機構がキャッシュコントローラを独占的に使用している．そのため，入力値の一致比較中にロード命令が検出された場合，ロード命令はキャッシュアクセスを必要とするため，入力値の一致比較とオーバラップして実行できない．そこで本論文では，入力値の一致比較中に検出されたロード命令のロード先アドレス，ロードする値を登録するためのバッファを追加し，入力値の一致比較中にロード命令を実行可能にすることで，スーパスカラ型自動メモ化プロセッサの性能を向上させる手法を提案する．提案手法の有効性を検証するために，既存のスーパスカラ型自動メモ化プロセッサに提案手法を実装し，SPEC CPU95 ベンチマークを用いて評価した．その結果，通常実行時と比較して既存手法では最大 16.8%，平均 3.08%のサイクル数の削減であったのに対し，提案手法では最大 18.0%，平均 3.25%のサイクル数の削減率となり，提案手法の有効性を確認できた．

パイプラインの使用効率向上による
スーパスカラ型自動メモ化プロセッサの高速化

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．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 [1] 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可能な命令区間と見なし，実行時にその入出力を再利用表に記憶しておくことで，同一命令区間を同一入力を用いて再び実行しようとした際に，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．

これまで，自動メモ化プロセッサは，単命令発行でパイプライン化されていない単純な SPARC 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一般的に採用されている．そのため，ベースアーキテクチャをスーパスカラ型 ARM に変更した自動メモ化プロセッサ [2] が新たに提案されている．本論文では，このスーパスカラ型自動メモ化プロセッサの性能を向上させる手法について検討する．

このスーパスカラ型自動メモ化プロセッサには，ロード命令が検出された場合，ロード命令はキャッシュへのアクセスを必要とするため，計算再利用に必要となる入力値の一致比較とオーバラップして実行することができないという問題が存在する．そのため本論文では，入力値の一致比較中に発行されたロード命令の情報を登録するための専用バッファを新たに追加し，入力値の一致比較とオーバラップしてロード命令を実行できるようにする．このようにすることで，パイプラインの使用効率を向上させ，

スーパスカラ型自動メモ化プロセッサの性能を向上させる手法を提案する．

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 章では提案するパイプライン処理の高速化手法について説明し，4 章では，提案手法の実装について詳細に説明する．その後，5 章で本提案手法の評価を行い，最後に 6 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書き戻す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）**[3] 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，メモ化を行うためには，プログラムを記述し直す必要があり，既存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 [4] 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ことで，既存のバイナリを変更することなく高速実行可能なプロセッサである，**自動メモ化プロセッサ（Auto-Memoization Processor）**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メモ化可能な命令区間として検出し，その入出力を MemoTbl に記憶する．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力と出力の組（入出力

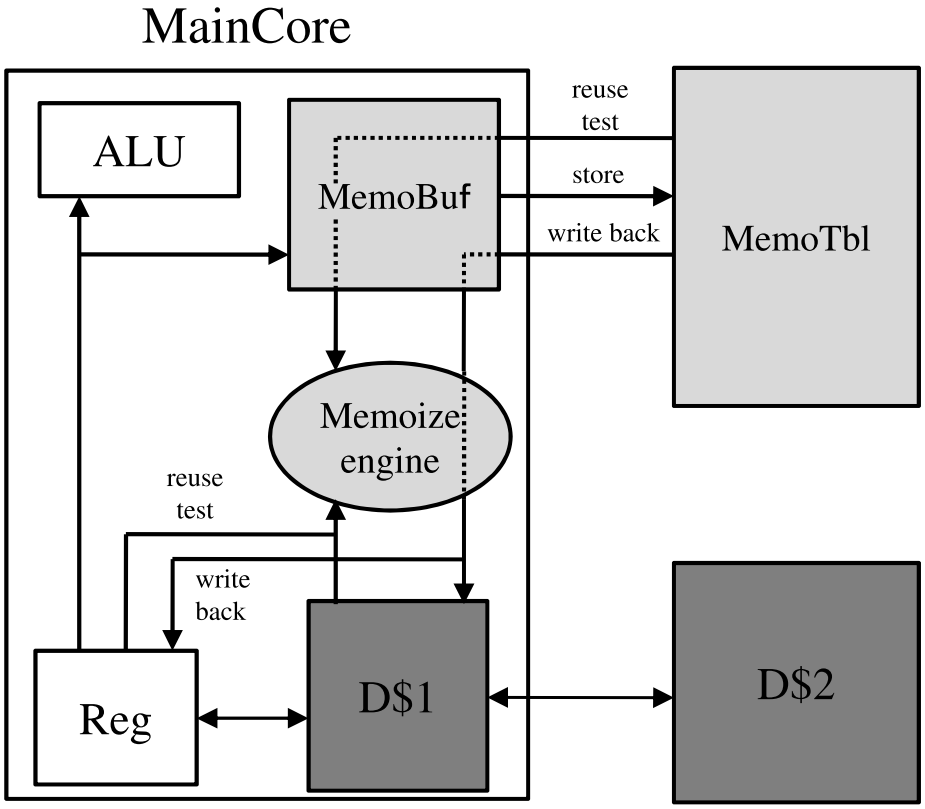

図 1: 自動メモ化プロセッサのハードウェア構成

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．

自動メモ化プロセッサは再利用対象区間を検出すると，MemoTbl を参照し現在の入力セットと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，入力セットに対応する出力セットはレジスタやキャッシュに書き戻され，自動メモ化プロセッサは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 に格納し，実行終了時に MemoBuf の内容を 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），

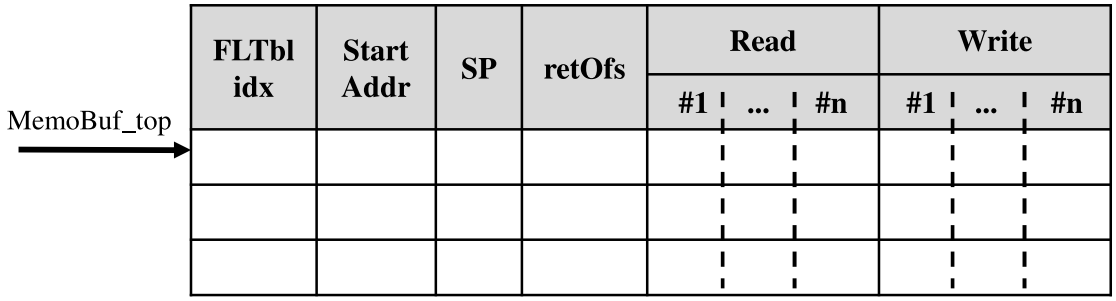

図 2: MemoBuf の構成

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する．この例において，入力セット (i) 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

```
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

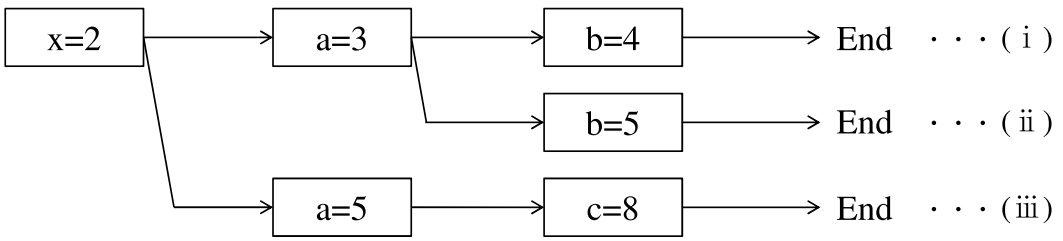

図 4: 入力パターンの木構造

する **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

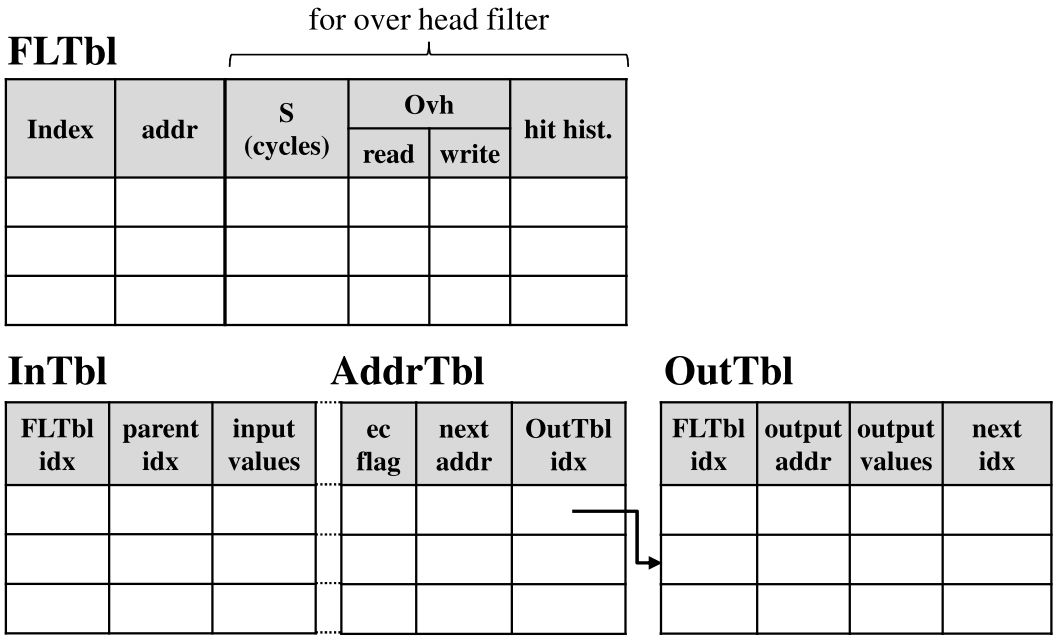

図 5: MemoTbl の構成

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付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異なる場合も同様に，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，次に参照するべきアドレス（next addr）を持つ．また，入力セットの終端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 idx）を持つ．

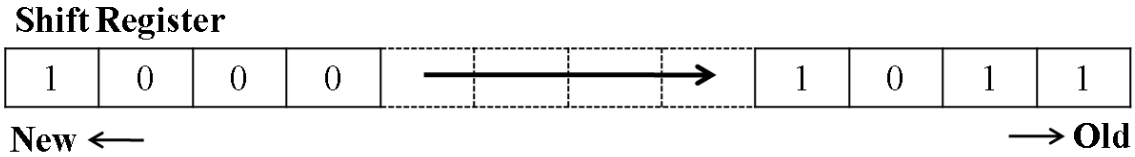

図 6: シフトレジスタの構造

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）および出力値（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 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 6** に示す．シフトレジスタでは再利用 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．

例えば，ある命令区間について，最近の一定回数 T の再利用試行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，再利用テストに失敗した場合でも，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．

2.3 ベースアーキテクチャ

自動メモ化プロセッサは，単命令発行型である SPARC-V8 アーキテクチャとスーパスカラ型である ARM アーキテクチャ[5] のそれぞれをベースとした実装が検討されている．ここで，現在市場に流通しているプロセッサのアーキテクチャとして，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一般的に採用されている．そのため，本論文で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，スーパスカラ型である ARM アーキテクチャを採用した，スーパスカラ型自動メモ化プロセッサに着目する．

スーパスカラ型自動メモ化プロセッサの構成を**図 7** に示す．スーパスカラ型自動メモ化プロセッサでは，ベースプロセッサとして 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 OROCHI [6] を採用している．なお，スーパスカラ型自動メモ化プロセッサでは，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 7 でも省略している．また，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．

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* [7] による分

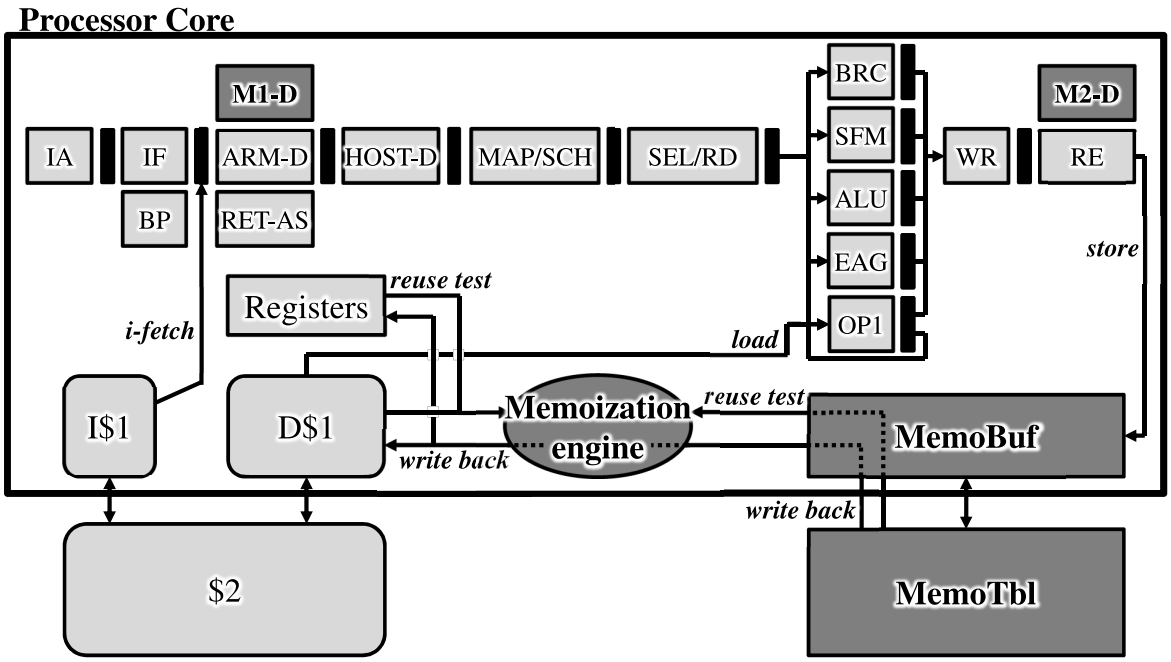

図 7: スーパスカラ型自動メモ化プロセッサの構成

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，命令の効率的なパイプライン実行を可能としている．

2.4 スーパスカラ型自動メモ化プロセッサの動作モデル

本節では，スーパスカラ型自動メモ化プロセッサの実行の様子を，再利用テスト成功時と，再利用テスト失敗時に分けて説明する．

2.4.1 再利用テスト成功時

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．メモ化を行わない通常実行時，および再利用テスト成功時のパイプライン実行の様子を**図 8** に示す．

図 8 において，$\mathtt{I}_{\mathtt{g2}}$ は関数呼び出しのための命令，$\mathtt{I}_{\mathtt{f14}}$ は関数復帰のための命令であり，$\mathtt{I}_{\mathtt{f1}}$ から $\mathtt{I}_{\mathtt{f14}}$ までの区間が関数である．ここで，関数の先頭命令をフェッチしてか

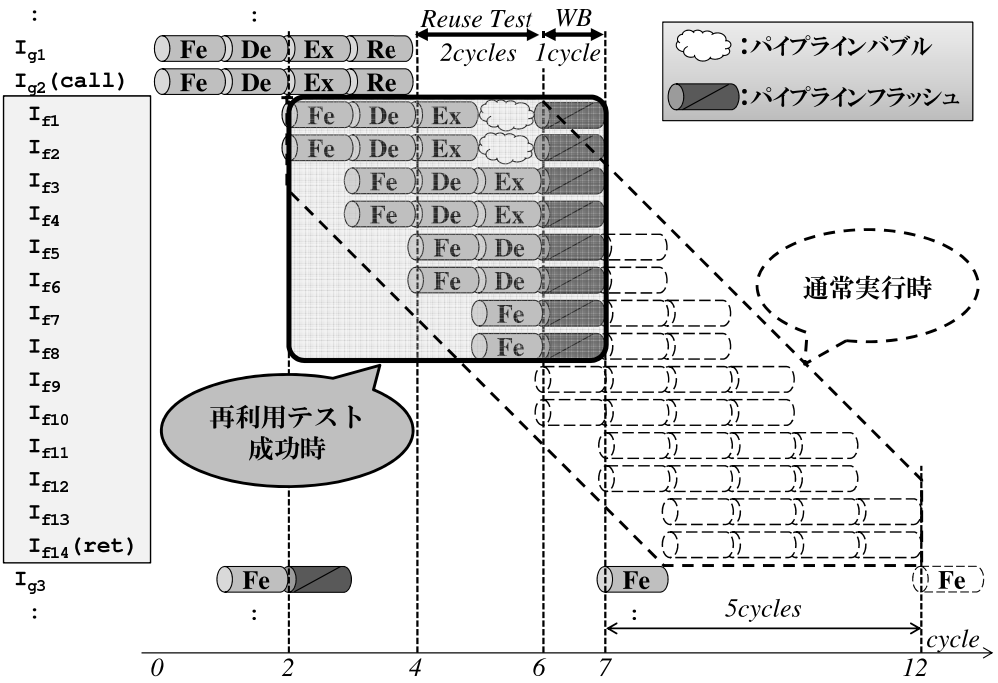

図 8: 再利用テスト成功時のパイプライン実行の様子

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．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は，図 8 中の点線で囲んだ部分で表しており，この関数の実行には 10 サイクルを要している．

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図 8 中の 4 サイクル目で開始される．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，現在の関数内の命令をこの再利用テストとオーバラップして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，それによって，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が起

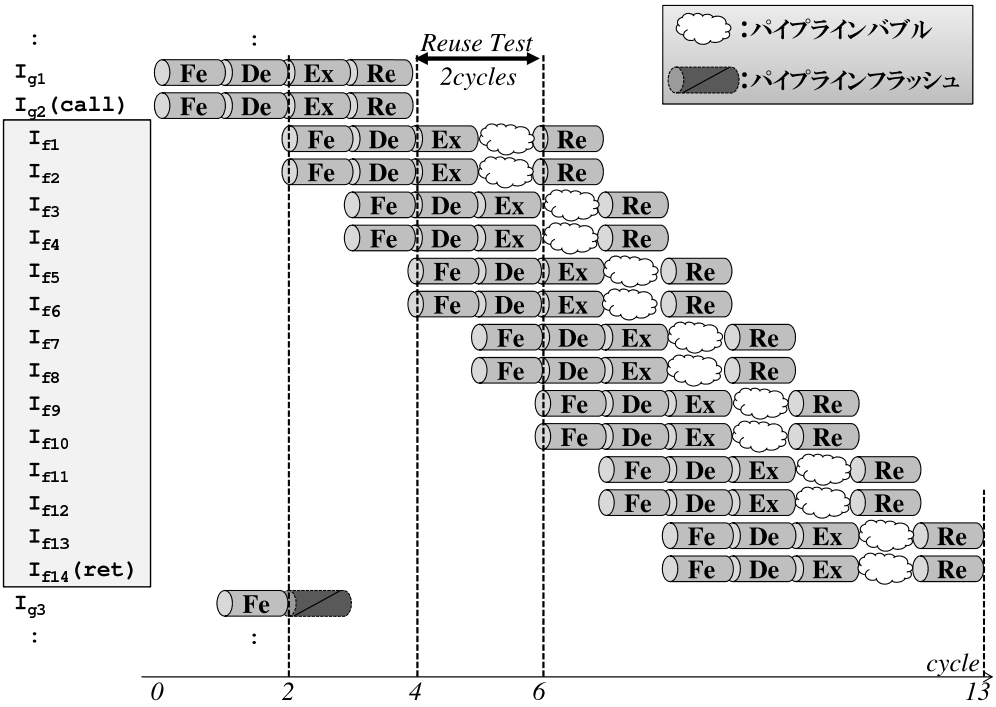

図 9: 再利用テスト失敗時のパイプライン実行の様子

こりうるためである．

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，過去の実行結果を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．そして，7 サイクル目にて，関数復帰先の命令（$\mathtt{I}_{g3}$）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 8 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，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．

2.4.2 再利用テスト失敗時

次に，再利用テスト失敗時のパイプライン実行の様子を**図 9** 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，再利用テストを開始する．また，再利用テストを行っている間，現在の関数内の命令はリタイアステージでの処理が禁止された状態で再利用テストとオーバラップして実行される．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したとすると，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8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　パイプラインのスループット向上手法

本章では，既存のスーパスカラ型自動メモ化プロセッサの問題点と，それを解決するための手法を提案する．

3.1　既存のスーパスカラ型自動メモ化プロセッサの問題点

2.4.1項で述べたように既存の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．この時，ベースプロセッサはパイプラインをストールさせずに，再利用テストとオーバラップして現在の関数内の後続命令を実行する．しかし，再利用テスト中，ベースプロセッサはキャッシュやメモリからレジスタへ値を転送する命令であるロード命令をオーバラップして実行することができない．これは，再利用テストではレジスタやメモリ上に存在する現在の関数の入力値と再利用表に登録されている過去の入力値との比較を行うため，再利用機構がキャッシュコントローラを独占的に使用するためである．そのため，このロード命令は再利用テストが終了するまでOP1ユニットで実行ステージでの処理を止められてしまう．これにより，ロード命令の実行ステージでの処理を再利用テストとオーバラップさせることができないことが原因でパイプラインバブルが発生し，パイプラインの使用効率が低下する．

ここで，ロード命令の実行ステージでの処理を再利用テストとオーバラップできない場合のパイプライン実行の様子を**図10**に示す．図10の例では，図8，図9で用いた命令列の，$\mathtt{I_{f1}}$，$\mathtt{I_{f2}}$，$\mathtt{I_{f4}}$がロード命令であった場合のパイプライン実行の様子を示している．なお，図10は再利用テストに失敗する場合の実行の様子を表している．図10中の3つのロード命令は，再利用テスト中に実行ステージでの処理がされないため，パイプラインストールが発生する（4サイクル目）．そして，再利用テストが終了するとベースプロセッサがキャッシュアクセス可能となるため，止められていたロード命令の実行ステージでの処理が再開される．しかし，この時点で実行ステージでの処理が止められている命令が3つ（$\mathtt{I_{f1}}$，$\mathtt{I_{f2}}$，$\mathtt{I_{f4}}$）存在している（6サイクル目）ため，$\mathtt{I_{f5}}$か

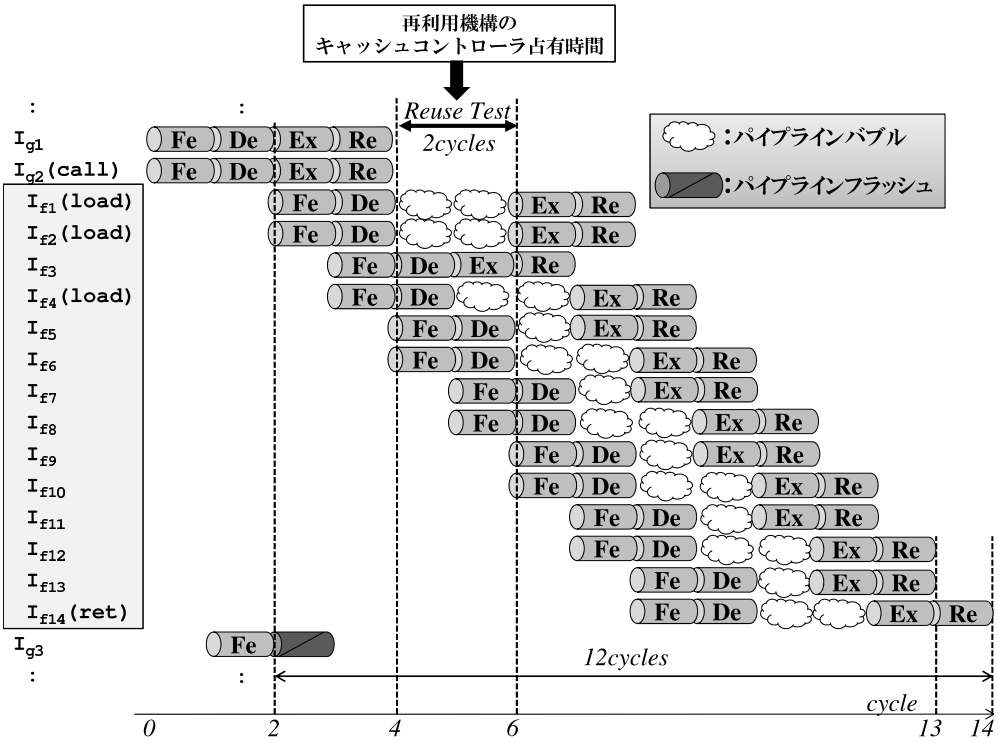

図 10: 再利用テスト中ロード命令実行不能の場合のパイプライン実行の様子

ら $\mathtt{I_{f14}}$ の命令も実行ステージに進むことができず，パイプラインストールが発生する．

3.2　パイプライン使用効率の向上手法

本節では，ロード命令の実行ステージでの処理を再利用テストとオーバラップさせる方法を述べる．そして，その際のパイプライン実行の様子を図に表し，提案手法によるパイプライン使用効率の向上を示す．

3.1 節で述べたように，再利用テスト中は，再利用機構がキャッシュコントローラを独占的に使用する．そのため，再利用テスト中，ベースプロセッサはキャッシュにアクセスすることができず，キャッシュやレジスタへのアクセスを必要とするロード命令は，実行ステージでの処理を OP1 ユニットで止められてしまう．しかし，ロード命令によりロードされる値を，キャッシュやメモリを介さずに取得することが可能であれば，再利用テスト中であってもロード命令を実行することができる．そこで，再利用テスト中に，ロード命令がキャッシュやメモリ以外から値を読み出すことを可能にするため，ロード命令のロード先アドレス，および，ロード命令の実行によりロードされた値を登録するためのバッファを用意する．このバッファに格納されている値を参照することで，キャッシュへのアクセスが禁止されている再利用テスト中にロード

命令が検出された際に，キャッシュを参照することなく，ロード命令を実行することが可能となる．

ここで，このバッファにアドレス，および，値を登録する手順を説明する．再利用テスト中は，再利用機構がキャッシュコントローラを独占的に使用するため，ベースプロセッサはキャッシュにアクセスすることができず，ロード命令の実行ステージでの処理を行うことができない．このとき，実行ステージでの処理を待機しているロード命令のロード先アドレスをバッファに登録しておく．その後，もし再利用テストに失敗した場合，関数の通常実行に戻るため，このロード命令は実行ステージでの処理が再開される．提案手法ではこのロード命令がリタイアされる際，ロードした値をアドレスと対応付けてバッファに登録する．一方で，プログラム内の命令を実行中にストア命令を検出した際，そのストア命令を実行してしまうと，バッファに登録されているエントリの値が最新の値ではなくなる可能性がある．これは，ストア命令がストア先アドレスと対応するキャッシュやメモリ上のエントリに値を転送する命令であるため，ストア先アドレスがバッファに登録されていた場合，そのストア命令が実行されることで，キャッシュやメモリ上の値と，バッファに登録されている値が異なってしまうからである．これを回避するため，ストア命令を検出し，そのストア命令のストア先アドレスでバッファ上を検索し，一致するエントリが存在する場合，ストアする値をバッファに登録されているアドレスと対応付けてバッファに登録する．以上に述べたような手順で，バッファにアドレスと値を登録していく．このバッファを使用することで，キャッシュへのアクセスが禁止されている再利用テスト中にロード命令が検出された場合であっても，ロード先アドレスでバッファ上を検索し，一致するエントリが存在する場合，バッファからそのエントリの値を読みだすことでロード命令の実行ステージでの処理が可能となる．

実際に，ロード命令の実行ステージでの処理を再利用テストとオーバラップして実行する場合のパイプライン実行の様子を，**図 11** に示す．図 11 では，図中に用いられている命令列において，$\mathtt{I_{f1}}$，$\mathtt{I_{f2}}$，$\mathtt{I_{f4}}$ の 3 つロード命令のロード先アドレスとロードする値がバッファに登録されているとする．図 11 の例では，バッファからロードする値を読み出し，ロード命令の実行ステージでの処理が再利用テストにオーバラップしている．そのため，Ex ステージでパイプラインバブルが発生していない．この例の場合，関数の実行に要したサイクル数は 11 サイクルである．一方，図 10 では，再利用機構がキャッシュコントローラを独占的に使用することにより，再利用テスト中にロード命令の実行ステージでの処理を行うことができなかった．そのため，Ex ステージで

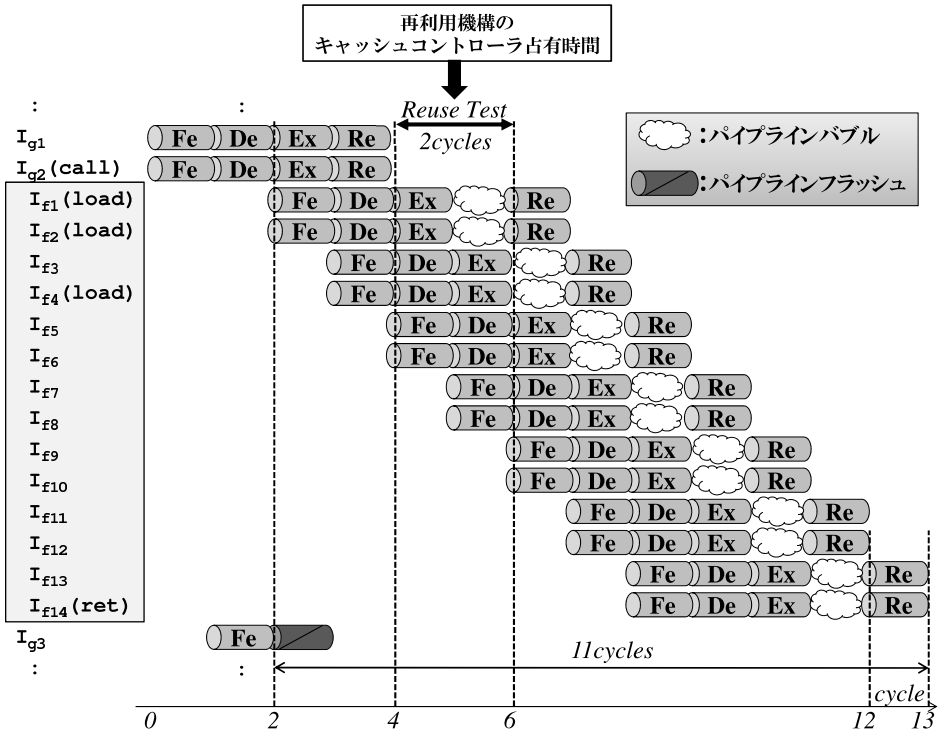

図 11: 再利用テスト中ロード命令実行可能の場合のパイプライン実行の様子

パイプラインバブルが発生し，関数の実行に 12 サイクルを要している． 章で述べたように提案手法では

4　独占回避バッファの実装

本章では，3 章で提案したパイプライン使用効率の向上を図る手法の，具体的な実装，動作について説明する．

4.1　独占回避バッファの構成

3 章で述べた提案手法を適用することで，再利用テストにオーバラップしてロード命令の実行を可能にする．これを実現するために，キャッシュやメモリにアクセスすることなく，ロード命令によりロードされる値を取得する必要がある．そこで，再利用テスト中に検出されたロード命令のロード先アドレス，およびそのアドレス上の値を記憶しておくバッファを追加する．本論文では，このバッファを**独占回避バッファ**と呼ぶ．

提案手法では，ロード命令によりロードされる値を独占回避バッファから読み出すことで，ロード命令を再利用テストにオーバラップして実行する．この独占回避バッ

独占回避バッファ

valid	addr.	value
1	addr1	1
0	addr2	2
1	addr3	3
⋮	⋮	⋮

図 12: 独占回避バッファの構成

ファの構成を**図 12** に示す.

独占回避バッファは，以下の 3 つのフィールドにより構成される.

- **valid** 当該エントリが有効であるか否かを示す.
- **addr.** 再利用テスト中に検出されたロード命令のロード先アドレスを登録する.
- **value** addr. フィールドに登録されているメモリアドレス上に記憶されている値を登録する.

4.2 独占回避バッファの動作

本節では，独占回避バッファを実装したスーパスカラ型自動メモ化プロセッサの動作について説明する．まず，4.2.1 項で独占回避バッファの動作の概要を示す．そして，その概要を元に，4.2.2 項から 4.2.6 項で各動作の詳細な説明を行う.

4.2.1 提案手法の動作概要

提案手法を実装した際の独占回避バッファの動作の概要を，**図 13**，**図 14** に示す．なお，3.2 章で述べたように，提案手法ではメモリアクセスが必要であるロード命令，およびストア命令の検出時に独占回避バッファを使用し，スーパスカラ型自動メモ化プロセッサの問題点の解決を図る．そこで，ロード命令検出時の提案手法の動作を示している図 13 と，ストア命令検出時の提案手法の動作を示している図 14 を用いて，それぞれの命令を検出した場合に分けて，提案手法の動作を順を追って説明する．また，図 13，図 14 中の四角の枠は，パイプラインステージでの命令の処理，丸の枠は独占回避バッファの動作を示している．また，図 13，図 14 中では，実装した独占回避バッファを LB と表記している.

ロード命令

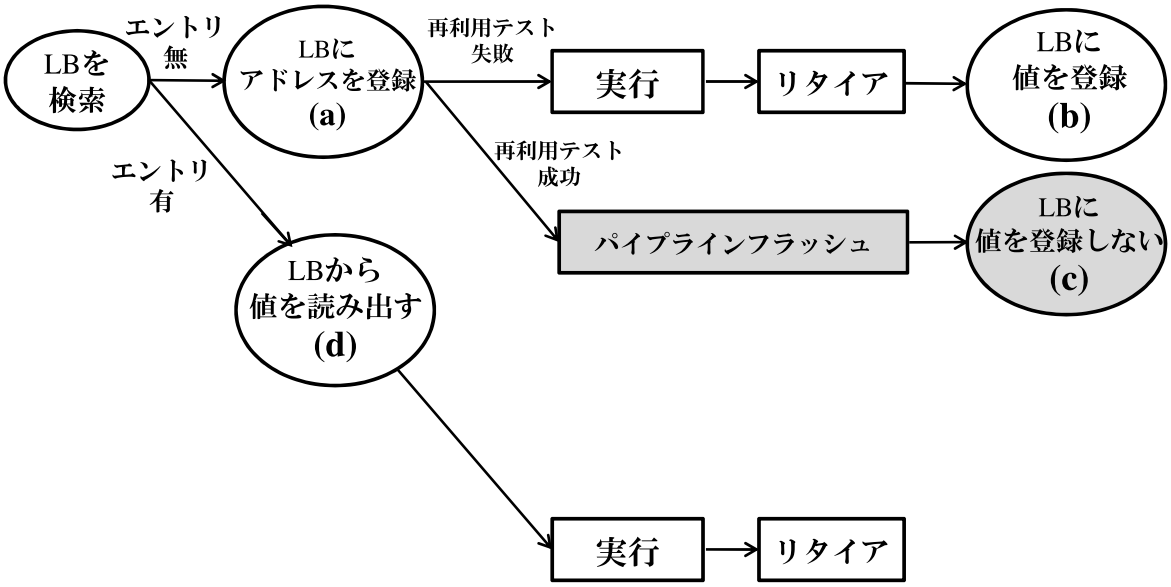

図 13: 提案手法の動作概要（ロード命令検出時）

ストア命令

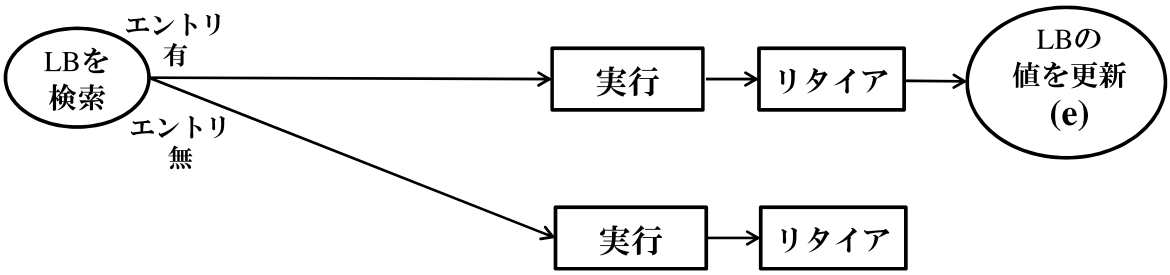

図 14: 提案手法の動作概要（ストア命令検出時）

まず，提案手法では再利用テスト中にロード命令が検出されると，フェッチ/デコードステージでの処理が終了した後，ロード命令のロード先アドレスを独占回避バッファ上で検索する．

独占回避バッファを検索して一致するエントリが存在しない場合は，そのロード命令のロード先アドレスを独占回避バッファに登録する（図 13(a)）．アドレスを独占回避バッファに登録した後，再利用テストに失敗する場合は，関数の通常実行に戻るため，ロード命令の実行ステージでの処理が再開される．実行ステージで処理された後，

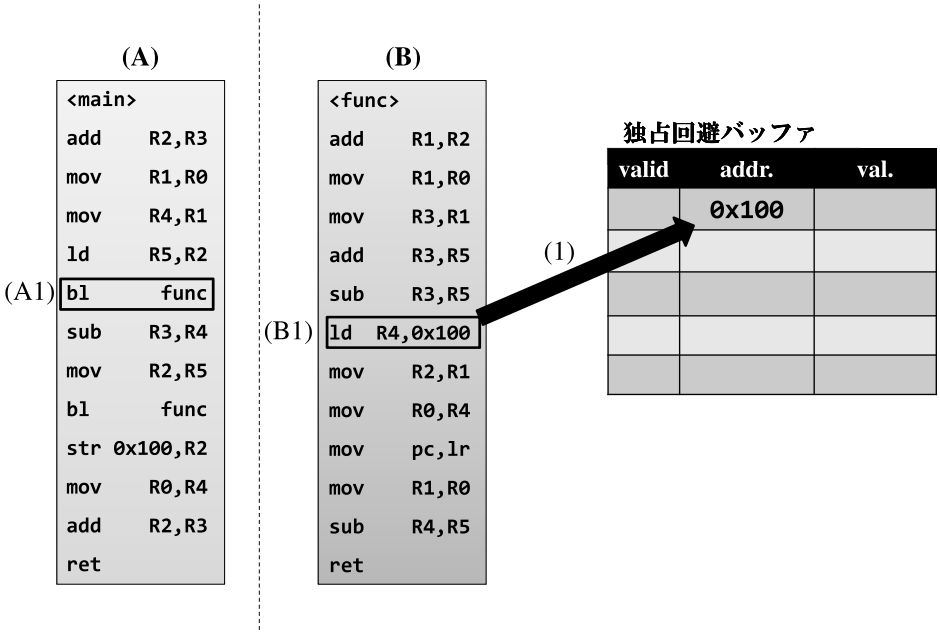

図 15: 再利用テスト中の動作例

ロード命令はリタイアされるため，ロードされた値を独占回避バッファに登録する（図 13(b)）．一方で，アドレスを独占回避バッファに登録した後，再利用テストに成功する場合は，関数内命令の実行が省略されるため，パイプラインがフラッシュされる．そのため，このロード命令によりロードされる値を独占回避バッファに登録することができない（図 13(c)）．

これに対し，独占回避バッファを検索して一致するエントリが存在する場合は，そのエントリから値を読み出すことで，ロード命令の実行ステージでの処理を行う（図 13(d)）．

なお，ストア命令が検出され，そのストア命令が実行されると，ストア先アドレスに対応するキャッシュやメモリ上の値が書き換えられてしまうことで，独占回避バッファ上に登録されているエントリの値が最新の値ではなくなる可能性がある．そのため，ストア命令が実行される際，独占回避バッファ上の value フィールドを，ストアする値で更新する（図 14(e)）．

以降では，本項で述べた図 13，図 14 中の (a) から (e) のそれぞれの動作について，図を用いて詳細に説明する．

4.2.2 再利用テスト中ロード命令検出時の動作

再利用テスト中にロード命令を検出した際の動作例を**図 15** に示す．図の例では，プロセッサは図中左の命令列（A）内の命令を順番に実行していくものとする．また（B）

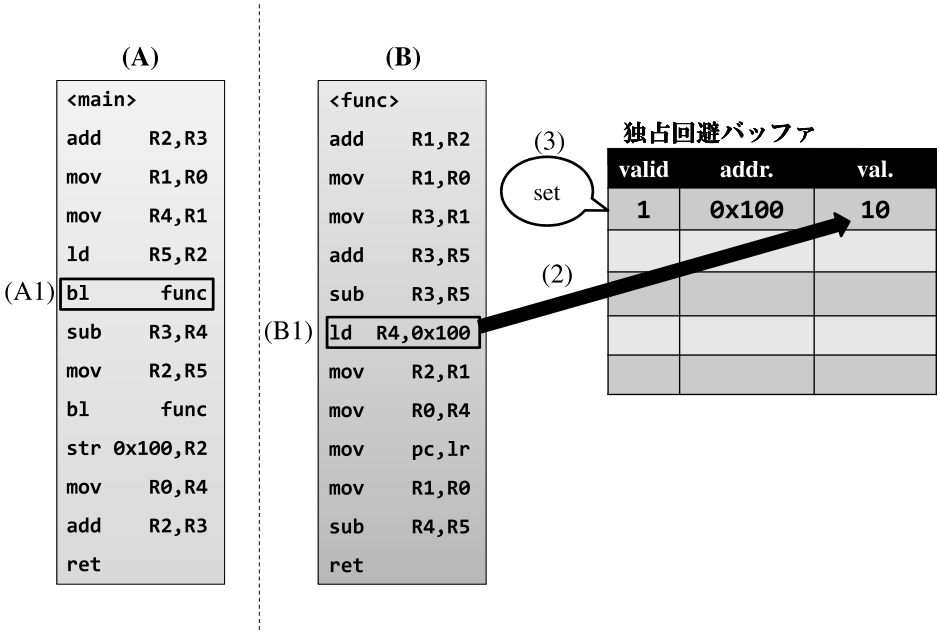

図 16: 再利用テスト失敗時の動作例

は，関数 func というルーチンを構成する命令列であり，（A）内で関数 func が呼び出された際には，（B）内の命令の実行に移行する．そして，関数呼び出し命令が検出された場合，プロセッサは図中右の命令列（B）内の命令を実行していくものとする．まず，ベースプロセッサは命令列 (A) 内の命令を順番に実行していき，関数呼び出し命令（A1）をリタイアすると，メモ化制御機構は再利用テストを開始する．このとき，ベースプロセッサは再利用テストと命令列（B）内の命令の実行をオーバラップさせる．命令列（B）内の命令を実行している間，ロード命令（B1）が検出された場合，ロード命令の実行をストールさせる．これは，再利用テスト中は再利用機構がキャッシュコントローラを使用するため，ベースプロセッサはキャッシュコントローラが使用可能となるまで，メモリアクセスを必要とするロード命令の実行を待機させる必要があるからである。このロード命令（B1）の実行が待機されているとき，ロード先のアドレス 0x100 を，独占回避バッファの addr. フィールドに登録する (1)．これは，以降のプログラム内の命令の処理で再び再利用テスト中にロード命令が検出された際，ロード命令のロード先アドレスを独占回避バッファ上で検索し，アドレスの一致したエントリから値を取り出すためである．

4.2.3 再利用テスト失敗時の動作

メモ化制御機構が再利用テストに失敗した場合の動作例を，**図 16** に示す．再利用テストに失敗すると，関数 func に計算再利用を適用できないことが判明し，命令列（B）

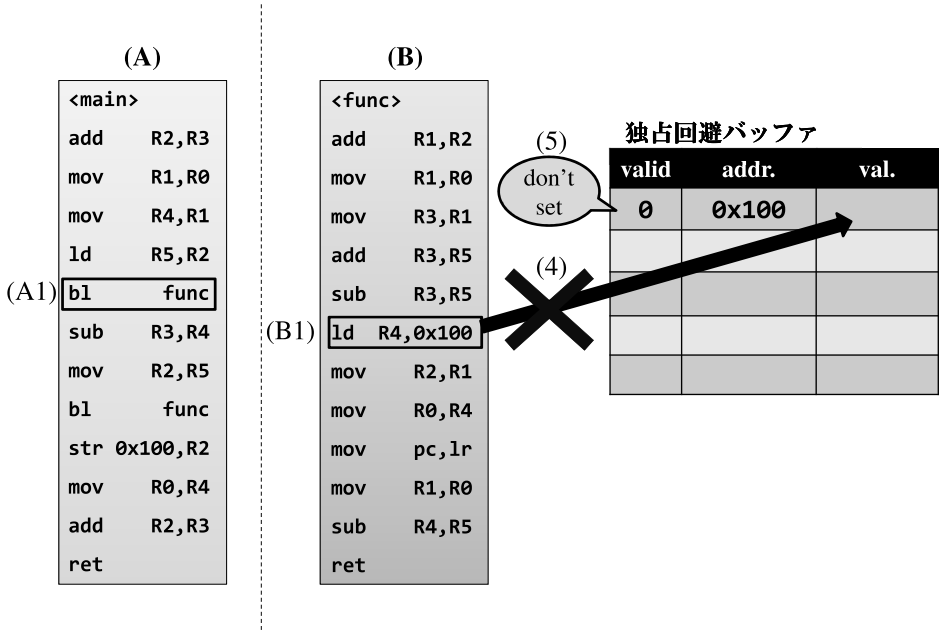

図 17: 再利用テスト成功時の動作例

内の命令の通常実行を行う．このとき，再利用機構はキャッシュコントローラの使用を終えるため，ベースプロセッサはキャッシュコントローラを使用することができる．そのため，再利用テストに失敗すると，OP1 ユニットで止められていたロード命令（B1）の処理が再開され，値が判明する．仮にそのロードしてきた値を 10 とすると，addr. フィールドにロード先アドレス 0x100 が登録されているエントリの value フィールドに，ロードした値である 10 を登録し (2)，valid をセットする (3).

4.2.4 再利用テスト成功時の動作

一方，メモ化制御機構が再利用テストに成功した場合の動作例を，**図 17** に示す．再利用テストに成功すると，命令列（B）内の命令の実行を省略しパイプラインをフラッシュする．つまり，OP1 ユニットで止められていたロード命令（B1）は処理が再開されないため，ロード先アドレス上の値を知ることができない．そのため，このロード命令（B1）のロード先アドレスを登録したエントリの value フィールドに，値を登録することができない (4)．値を登録しない独占回避バッファのエントリは，無効にする必要があるため，独占回避バッファの対応するエントリの valid をセットしないようにする (5),

4.2.5 独占回避バッファを用いたロード命令実行時の動作

再利用テスト中にロード命令が検出され，独占回避バッファに登録されている値を用いてそのロード命令を実行する際の動作を説明する．4.2.2 項で述べたように，再利

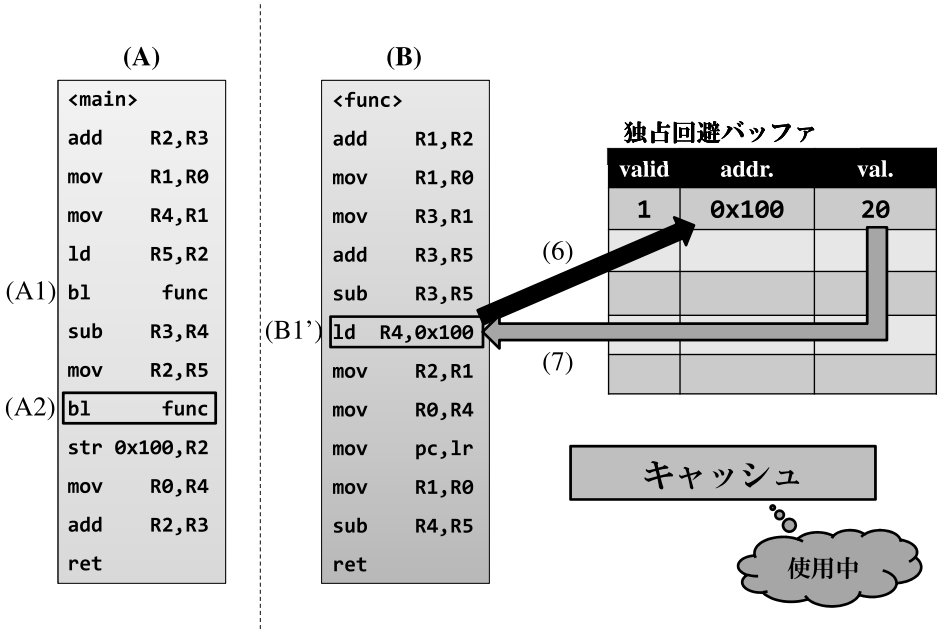

図 18: 独占回避バッファを用いたロード命令実行の動作例

用テスト中は再利用機構が入力値の一致比較を行い，キャッシュコントローラを独占的に使用するため，ロード命令の実行が止められてしまう．これを回避するため，まず，ロード命令のロード先アドレスを独占回避バッファ上で検索する．そして，一致するエントリが存在する場合，そのエントリの value フィールド上の値を読み出す．この独占回避バッファから読み出した値を使用することで，キャッシュにアクセスすることなくロード命令の実行ステージでの処理が可能となる．なお，独占回避バッファ上で検索し，一致するエントリが存在しなかった場合，ロード先のアドレスを，独占回避バッファの addr. フィールドに登録する．

ここで，再利用テスト中にロード命令が検出され，独占回避バッファから値を読み出すことでロード命令を実行する際の具体的な動作例を**図 18** に示す．命令列（A）内の命令を実行中に，関数呼び出し命令（A2）がリタイアされると再利用テストが開始される．そして，この再利用テストと関数内の後続命令の実行がオーバラップされているときにロード命令（B1'）が検出される場合，まず，ロード先アドレス（0x100）を独占回避バッファ上で検索する (6)．この例では，一致するエントリが存在するため，そのエントリの value フィールドに登録されている値である 20 を読み出すことで (7)，キャッシュコントローラを使用することなくロード命令を実行することができる．これにより，ロード命令の実行ステージでの処理を再利用テストとオーバラップさせることができるようになるため，パイプラインの使用効率が向上し，スーパスカラ型自

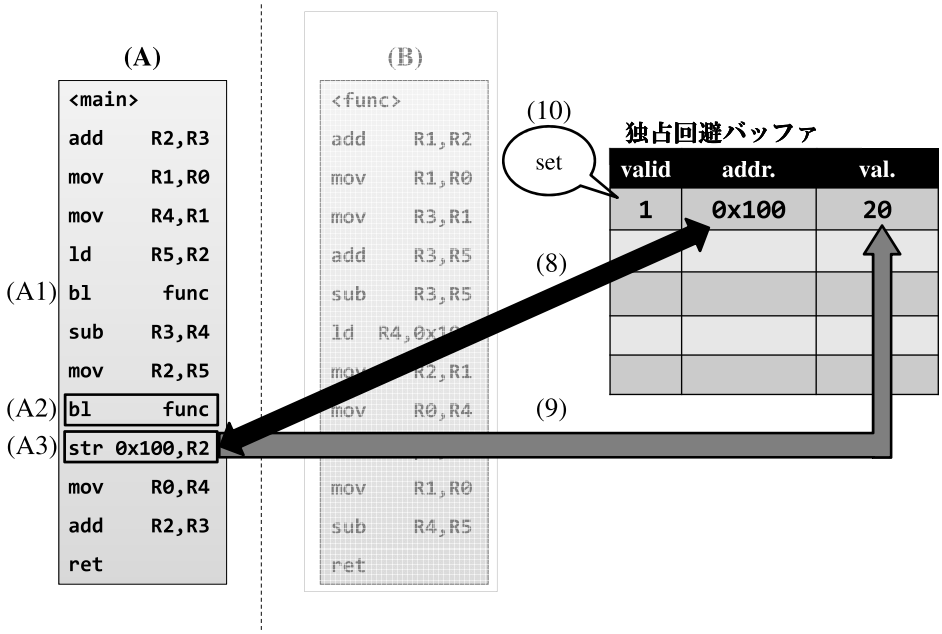

図 19: ストア命令検出時の動作例

動メモ化プロセッサの高速化が見込める．なお，独占回避バッファ内に一致するエントリが存在しなかった場合，4.2.2 項の動作のようにロード命令のロード先のアドレスを独占回避バッファに登録する．

4.2.6　ストア命令検出時の動作

命令列（A）内の命令を実行しているときに，ストア命令が検出された際の動作を説明する．ストア命令が実行されると，ストア先アドレスに対応するキャッシュやメモリ上の値が書き換えられる．そのため，もし addr. フィールドにそのアドレスを登録しているエントリが独占回避バッファ上に存在する場合，そのエントリの value フィールド上の値が最新の値ではなくなる可能性がある．そこで，まずストア命令が検出された場合，ストア命令のストア先アドレスを独占回避バッファ上で検索し，一致するエントリが存在するかを調べる．そして，一致するエントリが存在した場合，ストアする値でそのエントリの value フィールドを更新する．それによって，独占回避バッファに登録されている値を，最新の値とすることが可能となる．

ここで，ストア命令が検出された際の具体的な動作例を**図 19** に示す．この例では，関数呼び出し命令（A1）により呼び出された，命令列（B）内の命令をを実行した後の状態を示しており, 独占回避バッファ内には命令列（B）に存在するロード命令のロード先アドレスが登録されている．ストア命令（A3）を検出すると，ストア命令（A3）のデスティネーションオペランドの値 `0x100` を独占回避バッファ上で検索する (8)．こ

の図の例では，一致するエントリが独占回避バッファ上に存在するため，仮に，ストア命令によりストアされる値を 20 とすると，当該エントリの value フィールドに登録されている値を，ストアする値である 20 で更新する (9)．このとき，この更新したエントリは有効なエントリとなるため，valid フィールドがクリアされている場合はその valid フィールドをセットする (10).

なお，ストア命令のデスティネーションオペランドの値 `0x100` を独占回避バッファ上で検索した際，一致するエントリが存在しなかった場合は, 独占回避バッファの値は更新しない.

5　評価

以上で述べたスループット向上手法を，ARM 型スーパスカラプロセッサのシミュレータに実装し，ベンチマークプログラムを用いてその性能を評価した.

5.1　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** に示す．MemoTbl 内の InTbl に用いる CAM の構成は MOSAID 社の DC18288 [8] を参考にした．なお，プロセッサのクロック周波数は 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，再利用テスト中に検出される全てのロード命令のロード先アドレスを登録できるようにするため，独占回避バッファのエントリ数を 8192 として実装を行った．なお，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，関数を対象とした計算再利用のみで評価した.

5.2　評価結果

3.2 節で述べた提案手法の有効性を検証するために，以下の 2 つのプロセッサの実行サイクル数の評価を行った.

(A)　既存のスーパスカラ型自動メモ化プロセッサ

(P)　再利用テストとロード命令を同時に実行できるようにした場合のスーパスカラ型自動メモ化プロセッサ

評価の結果を**図 20** に示す．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て評価を行った．グラフは総実行サイクル数を示しており，メモ化を行わないモデルの実行サイクル数を 1 として正規化している．また，凡例はサイク

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

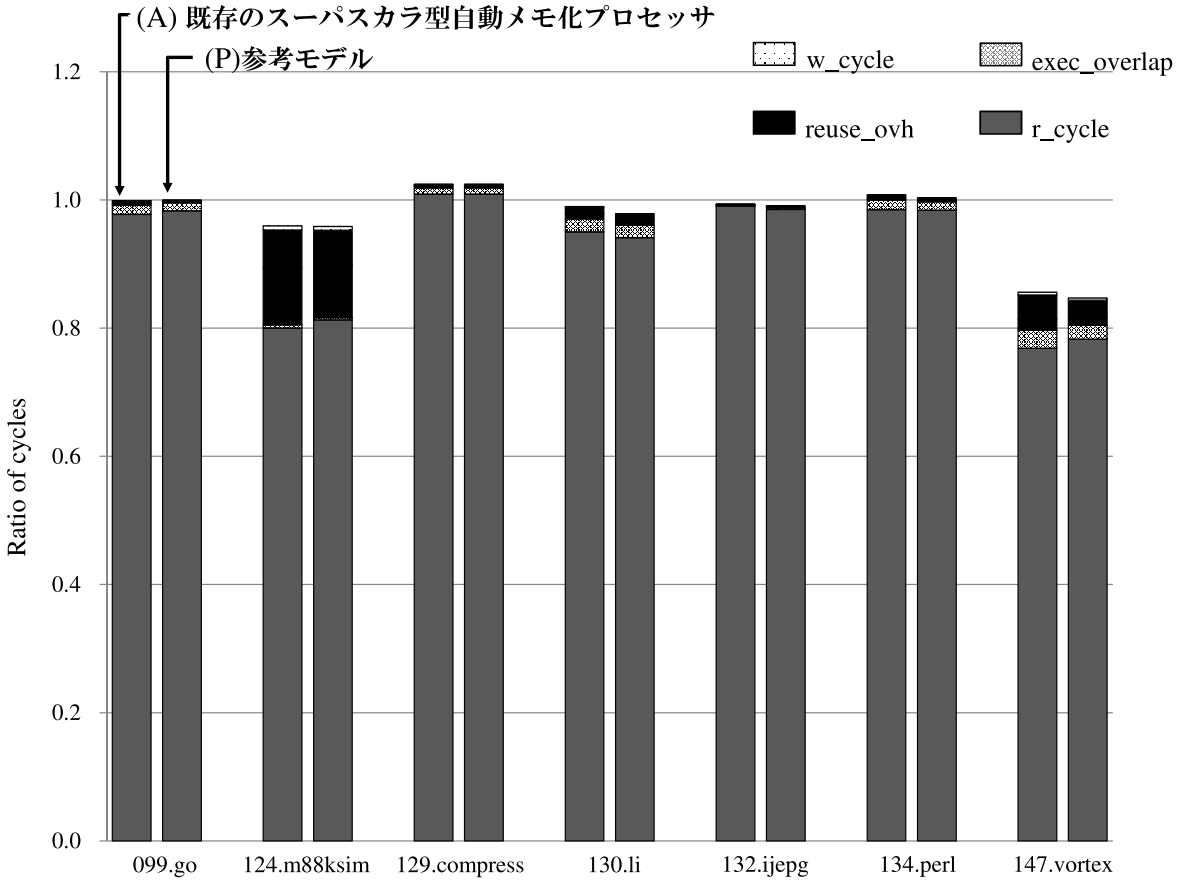

図 20: 評価結果

ル数の内訳を示しており，図 20 の r_cycle 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，exec_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，reuse_ovh は，再利用テスト成功時に無駄になった実行サイクル数，w_cycle は，MemoTbl の出力をレジスタやメモリに書き戻す際に要したサイクル数を表す．評価の結果，既存のスーパスカラ型自動メモ化プロセッサと比べて総実行サイクル数は平均 0.17%，最大 1.20%の削減率となり, 高速化手法の有効性を確認した．

5.3 考察

まず，130.li に着目すると，提案手法により r_cycle が削減されていることがわかる．これは，ロード命令の実行ステージでの処理を再利用テストとオーバラップさせることができるため，再利用テスト失敗時に実行できる命令数が増加していたからである．このことから，提案手法では既存のスーパスカラ型自動メモ化プロセッサよりも，再

利用テスト失敗時に実行が進んでいると考えられる．次に，147.vortex に着目してみると，r_cycle が増加し，reuse_ovh が削減されて性能が向上しているが，これはオーバヘッドフィルタ機構の影響であると考えられる．オーバヘッドフィルタとは，2.2 節で述べたように，再利用を適用した際に削減できる r_cycle よりも，再利用テストの際に生じる検索オーバヘッドや書き戻しオーバヘッドの合計が大きい場合に，再利用を適用しないようにすることで，性能低下を抑制するための機構である．147.vortex の実行結果を調査したところ，ロード命令が再利用テスト中に実行できるようになったことで，その関数に再利用を適用した際に削減できるサイクル数が減少し，その結果オーバヘッドフィルタによって再利用の適用を中止される命令区間が増加していることが分かった．このため，再利用オーバヘッドが減少し，命令実行サイクル数が増加したと考えられる．

6　おわりに

自動メモ化プロセッサは，メモ化をハードウェアにより自動的に行い，プログラム実行の高速化を図る．このうち，既存のスーパスカラ型自動メモ化プロセッサには，ロード命令の実行ステージでの処理を再利用テストとオーバラップさせることができず，これによりパイプラインの使用効率が低下してしまうという問題が存在した．これは，再利用テスト中，再利用機構がキャッシュコントローラを独占的に使用し，ベースプロセッサがキャッシュにアクセスすることができないためである．そこで本研究では，再利用テスト中にロード命令を検出した際，バッファにロード命令のロード先アドレス，およびロード命令の実行によりロードされた値を記憶し，そこから値を読み出すことで問題点の解決を図った．提案手法の有効性を確認するために，独占回避バッファと呼ぶバッファをスーパスカラ型自動メモ化プロセッサに実装し，SPEC CPU95 INT ベンチマークを用いて評価した．その結果，既存のスーパスカラ型自動メモ化プロセッサと比べて平均 0.17%，最大 1.20%の高速化を実現した．

本研究の今後の課題として, 独占回避バッファにとって適切なエントリ数を考察し，独占回避バッファのエントリ数をそのエントリ数に変更することが挙げられる．さらに，ロード先アドレス登録時に独占回避バッファのエントリが溢れることを回避するために，独占回避バッファ内の無効なエントリや使用率の低いエントリを追い出す方法を実装することも必要である．また，再利用テスト中にロード命令が検出され，再利用テストが成功した場合に，提案手法ではそのロード命令によりロードされる値を独占回避バッファに登録することができない．これを解決するため，再利用テスト成

功時に再利用表を検索し，再利用表から必要な値を取り出し独占回避バッファに登録することで有効なエントリを増加させ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, 協力をして頂いた松尾・津邑研究室, 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柴田裕貴氏, 津村高範氏に深く感謝致します．

参考文献

[1] 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).

[2] Shibata, Y., Tsumura, T., Tsumura, T. and Nakashima, Y.: An Implementation of Auto-Memoization Mechanism on ARM-based Superscalar Processor, *Proc. Int'l Symp. on System-on-Chip 2014 (SoC2014)*, pp. 1–8 (2014).

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

[4] 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).

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

[6] 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 (20061128).

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

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