修士論文

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

指導教員　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学大学院工学研究科
修士課程情報工学専攻
平成 19 年度入学　19417595 番

新美　明仁

平成 21 年 2 月 5 日

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

新美　明仁

内容梗概

今日，半導体集積回路技術向上などから計算機の性能は飛躍的に向上してきた．しかし，科学技術計算や高度医療，工業製品の設計などの分野では依然として計算機の性能向上が求められており，性能向上に向け様々な研究が行われている．

これまでに計算機の高速化手法として命令レベル並列性（ILP：Instruction-Level Parallelism）に着目したSIMDやスーパスカラプロセッサの研究がされてきた．しかし，プログラム自体に存在するILPには限界があり，性能向上は頭打ちになりつつある．また，半導体集積技術の向上によりマルチコアプロセッサが普及した現在，スレッドレベル並列性 (TLP:Thread-Level Parallelism) に着目したプロセッサ技術の研究も行われてきた．しかし，たとえばコンパイラを用いてプログラムの自動並列化を試みても，TLPの抽出精度が低くなり性能が出にくい．また性能を出すためには，プログラマが明示的にスレッド化を行うプログラムを記述しなければならないが，大きな手間がかかる．そのためTLPを利用した高速化も限界をむかえつつある．

一方で，それらの手法とは異なった着眼点から，計算再利用を利用した自動メモ化プロセッサの研究が行われてきた．計算再利用とは，過去の計算結果を再利用することで高速化を図る手法であり，メモ化とは関数を対象に，計算し利用可能な形で記憶しておく手法である．この手法は，並列性を利用しないためILPやTLPに乏しいプログラムでも高速化が可能となる．さらにこの自動メモ化プロセッサでは専用のハードウェアを設けることで，既存のバイナリを変更することなく計算再利用を実現できるという特徴も持っており，プロセッサの性能向上を実現してきた．

しかし，これまでの研究では単命令発行のシンプルなプロセッサを想定していたことと，自動メモ化プロセッサの対象アーキテクチャであるSPARCにおいて，SPARCプロセッサ特有のレジスタウインドウに関する考慮が一般的ではないことから，より一般的な環境であるスーパスカラプロセッサへのメモ化の適用を試みた．

さらに本稿で提案する，メモ化を取り入れたスーパスカラプロセッサのモデルを考える過程で，メモ化に要するオーバヘッドの削減や更なる高速化の可能性についても考察できた．

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

目次

1 はじめに 1

2 自動メモ化プロセッサ 2

2.1 メモ化 2

2.2 自動メモ化プロセッサ 4

2.3 自動メモ化プロセッサのオーバヘッド 5

3 スーパスカラプロセッサへの自動メモ化機構の実装モデル 8

3.1 ARM プロセッサの仕様 9

3.1.1 ARM 命令セットの特徴 9

3.1.2 レジスタ 10

3.1.3 命令分解 13

3.1.4 パイプラインステージ 13

3.1.5 リオーダバッファ(ROB) 14

3.1.6 命令同士の依存解析 15

3.2 提案手法の概要と動作 15

3.2.1 再利用成功時の動作 16

3.2.2 再利用失敗時の動作 19

3.3 多重再利用時の検索 21

3.3.1 親関数優先検索 21

3.3.2 子関数優先検索 22

4 実装 23

4.1 SP 相対による MemoTbl への入力値登録 24

4.1.1 引数レジスタ数による再利用率の低下 24

4.1.2 ARM プロセッサへのメモ化機構適用時の問題点と改善策 . . . 25

4.2 その他の実装上の考慮点 28

4.2.1 MemoTbl 上のエントリ 29

4.2.2 キャッシュアクセスの競合 29

4.2.3 ソースレジスタ番号の伝搬 30

4.2.4 リオーダバッファの拡張 30

4.3 高速化 . . . 31
4.3.1 ライトバックと後続命令フェッチのオーバラップ . . . 31
4.3.2 キャッシュコントローラ内の先行要求削除によるライトバックの高速化 . . . 33

5 評価 **36**
5.1 評価環境 . . . 36
5.2 結果 . . . 38
5.3 考察 . . . 39

6 まとめ **41**

1　はじめに

今日，半導体集積回路技術向上などから計算機の性能は飛躍的に向上してきた．しかし，科学技術計算や高度医療，工業製品の設計などの分野では依然として計算機の性能向上が求められており，性能向上に向け様々な研究が行われている．

これまでに計算機の高速化手法として命令レベル並列性（**ILP：Instruction-Level Parallelism**）に着目したSIMDやスーパスカラプロセッサの研究がされてきた．しかし，プログラム自体に存在するILPには限界があり，性能向上は頭打ちになりつつある．また，半導体集積技術の向上によりマルチコアプロセッサが普及した現在，スレッドレベル並列性 (**TLP:Thread-Level Parallelism**) に着目したプロセッサ技術の研究も行われてきた．しかし，たとえばコンパイラを用いてプログラムの自動並列化を試みても，TLPの抽出精度が低くなり性能が出にくい．また性能を出すためには，プログラマが明示的にスレッド化を行うプログラムを記述しなければならないが，大きな手間がかかる．そのためTLPを利用した高速化も限界をむかえつつある．

一方で，それらの手法とは異なった着眼点から，計算再利用を利用した自動メモ化プロセッサの研究が行われてきた．計算再利用とは，過去の計算結果を再利用することで高速化を図る手法であり，メモ化とは関数を対象に，計算再利用可能な形で記憶しておく手法である．この手法は，並列性を利用しないためILPやTLPに乏しいプログラムでも高速化が可能となる．さらにこの自動メモ化プロセッサでは専用のハードウェアを設けることで，既存のバイナリを変更することなく計算再利用を実現できるという特徴も持っており，プロセッサの性能向上を実現してきた．

しかし，これまでの研究では単命令発行のシンプルなプロセッサを想定していたことと，自動メモ化プロセッサの対象アーキテクチャであるSPARCにおいて，SPARCプロセッサ特有のレジスタウインドウに関する考慮が一般的ではないことから，より一般的な環境であるスーパスカラプロセッサへのメモ化の適用を試みた．

さらに本稿で提案する，メモ化を取り入れたスーパスカラプロセッサのモデルを考える過程で，メモ化に要するオーバヘッドの削減や更なる高速化の可能性についても考察できた．

以降，2章では高速化手法であるメモ化とそのプロセッサモデルである自動メモ化プロセッサの動作を述べ，さらに問題点となっているメモ化にかかるオーバヘッドについて説明する．3章では提案手法であるスーパスカラプロセッサへの自動メモ化機構適用方法について述べる．さらにここでは，スーパスカラプロセッサにメモ化機構を

```
void main(){
  int i, x=1, y=2, tmp=0;
  for(i=0; i<10; i++){
    if(tmp % 2 == 0)
      tmp += (x + y); //tmpが偶数のとき実行
    else
      tmp += (x * y); //tmpが奇数のとき実行
  }
  printf("result=%d\n",tmp); //結果の出力
  return 0;
}
```

図 1: ILP を利用できないプログラム例

適用するために考慮した点や提案手法が従来の自動メモ化プロセッサよりオーバヘッドが削減できる理由についても説明する．4 章では実装における考慮点を述べ，更なる高速化についても説明する．そして，5 章で提案したプロセッサについて評価を行い，6 章で結論をまとめる．

2　自動メモ化プロセッサ

本章では，メモ化とメモ化を自動で行うことができる自動メモ化プロセッサについて説明する．

2.1　メモ化

メモ化（Memoization）とは，関数やループといったプログラム中の命令区間を計算再利用可能な形に変換し，記憶しておくことである．ここで，計算再利用とは，メモ化によって記録された過去の実行結果を用いることで，命令区間の再計算を省略し高速化を図る手法である．従来より研究が行われてきた値予測および投機的実行は，数多くの命令の投機や実行結果の破棄が必要であるのに対し，メモ化は実行する必要のある命令列そのものを削減できるという点で，従来とは発想の異なる高速化技術である．

メモ化の特徴は，入力値さえ一致すれば実行結果を検証する必要がない点，およびメモ化の対象とする区間が増えても必要となる機構の複雑さが増大しない点にある．

メモ化はソフトウェア分野では広く使われてきたプログラミング手法である．これは，プログラムの並列性とは無関係である．したがって，ILP の限界を越える可能性がある．その例を図 1 に示す．この例では，x と y を用いた計算結果を tmp へ加算していく処理を示している．ここで，このプログラムを並列化することを考える．まず，for ループで書かれた処理をループ展開しようした場合，ループの処理の 1 回目と 2 回目，2 回目と 3 回目と言うように，tmp 変数の加算が処理に含まれるため，それぞれの処理に依存関係が発生し，ループ展開ができないことがわかる．また，1 つ前の処理結果である tmp 変数が，偶数か奇数かによって if 文中のどちらが実行されるかが決定されるため，このことも並列化への障害となる．しかし，このプログラムにメモ化を適用することを考えると．命令区間（x + y）,（x * y）の 2 つをメモ化可能な区間として見ることができる．たとえば，この 2 つの計算結果をメモリに記憶しておき，つぎにこの演算を見つけたとに，メモリから結果を取りだし tmp に加算していけば 2 つの演算を省略でき高速化できる．今回のように，たとえ命令区間に依存関係がある場合でも，メモ化では同じ演算をしている命令区間を見つけることができれば高速化が可能である．つまりこれは，ILP がないプログラムでも高速化が可能であることを示している．また，ソフトウェアによるメモ化の利点は，メモ化可能な命令区間を柔軟に決めることができる点である．一方でメモ化には，ソフトウェアによるものの他にハードウェアを用いたもの，または両方を利用したものなど，さまざまなものが提案されている．ハードウェアによるメモ化には，Sodani らが提案している，単命令を対象とした汎用的なメモ化 [?] や Yang らが提案している，メモ化対象を load 命令に限定したものなど [?] があるが，どれも決められた対象区間をメモ化するものであり，ソフトウェアのように柔軟に対象区間を決定することはできない．

しかし，ソフトウェアによるメモ化はプログラムの書き換えを必要としたり，コンパイル時に再利用情報を埋め込まなければならない．そのため既存のロードモジュールやバイナリをそのままメモ化する事ができないという特徴も持つ．つまり，メモ化の効果はプログラムの書き方により影響を受けることとなる．また，書き換えたプログラムが必ずしもメモ化による性能向上の効果を受けられるとは限らない．かつソフトウェアによるメモ化ではメモ化のためのオーバヘッドも大きく，オーバヘッドを考慮したプログラミングが必要となる．一方で，我々が提案してきた自動メモ化プロセッサは，メモ化対象の命令区間を，多くの命令区間を含み，かつ始点と終点を容易に特定できる「関数」としている．これにより再コンパイルや静的解析に基づく付加情報の埋め込みを必要とせず，既存のバイナリに変更を加えることなく高速化が可能とな

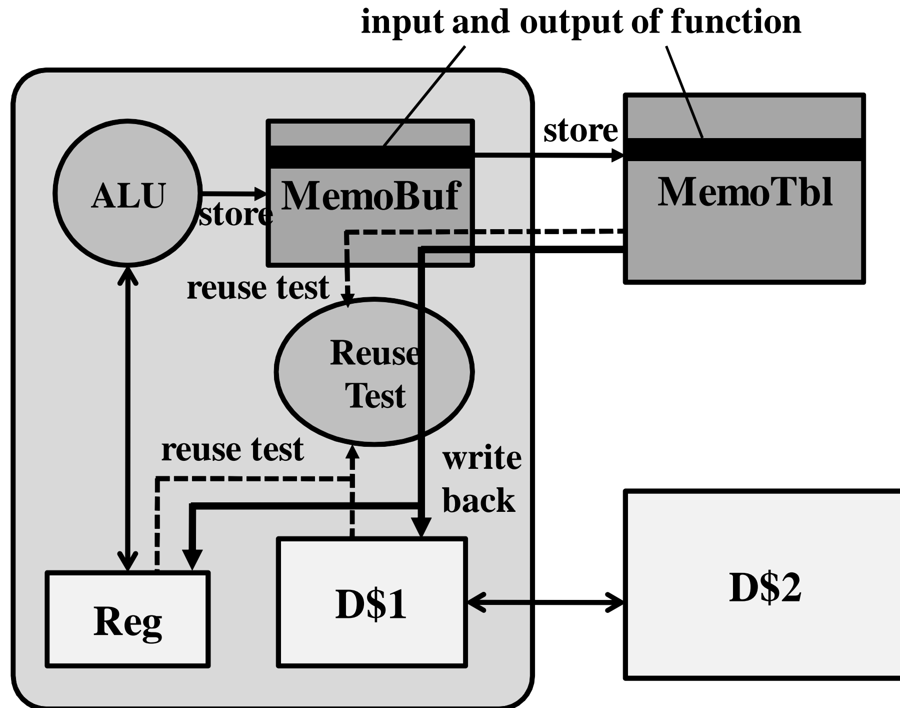

図 2: 自動メモ化プロセッサ

る．次節において自動メモ化プロセッサについて述べる．

2.2 自動メモ化プロセッサ

メモ化の従来研究では，単命令をメモ化対象区間とするものや，コンパイル時にメモ化のための情報を埋め込むものなどが提案されている．これに対し，我々はメモ化技術に基づいた汎用プロセッサとして，プログラム中のメモ化可能区間を自動的に抽出しメモ化を行う，自動メモ化プロセッサ [?] を提案してきた．これまでにも述べたようにメモ化の研究にはソフトウェアレベルで行うものもあるが，これはプログラムの書き換えによって実現する．つまり，関数自体に入出力を記録するためのコードと以前に記録した入力を検索するコードを付与することにより実現していた．しかし，我々の自動メモ化プロセッサは関数を動的に検出し，プログラムを書き換えることなく計算再利用を実現できるという特徴を持つ．

まずメモ化を実現するには，メモ化可能な命令区間を検出する必要がある．我々が提案する自動メモ化プロセッサのモデルを図 2 に示す．自動メモ化プロセッサでは，関数を命令区間としている．関数に含まれる命令の範囲は，call/jump 命令の分岐先から return 命令までであることから，call/jump 命令の分岐先から return 命令までをメ

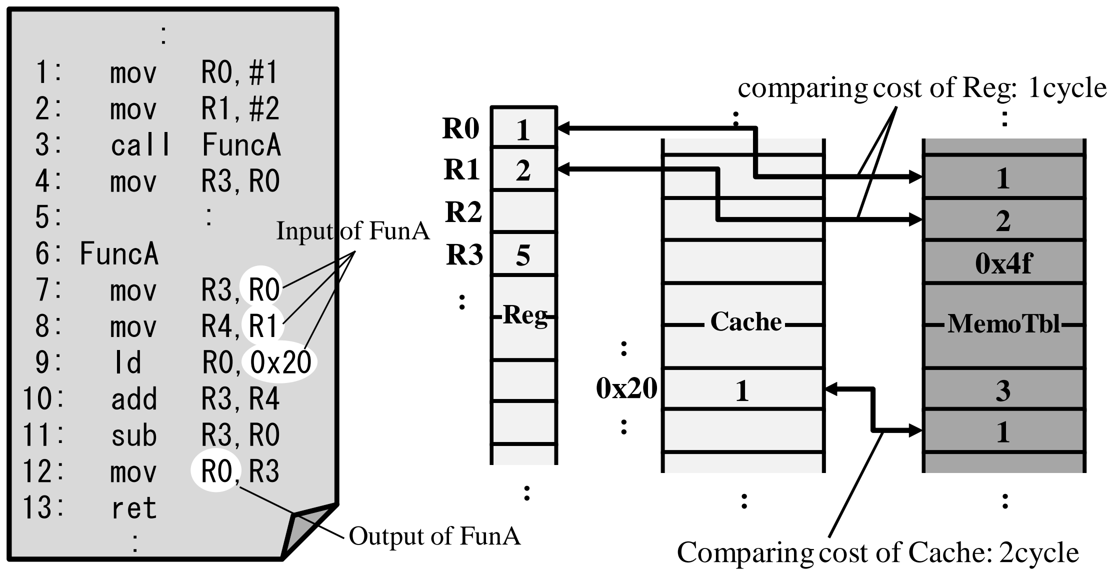

図 3: 自動メモ化プロセッサのオーバヘッド

モ化対象区間として定義し，この区間を動的に検出する．メモ化機構は，入出力を記憶する MemoTbl と呼ぶ記憶領域と MemoTbl への書き込みバッファである MemoBuf をもつ．自動メモ化プロセッサは，メモ化対象となる命令区間を検出すると MemoTbl から現在の入力と一致するエントリを検索し，完全に一致したエントリに対応する出力をキャッシュとレジスタに書き戻すことで命令区間の実行を省略する．また，一致するエントリが見つからなかった場合は通常通り命令区間を実行し，その際のレジスタおよびキャッシュの参照を入力，書き込みを出力として MemoBuf へ記憶していき，命令区間の最後まで実行したあとにまとめて対象命令区間の入出力セットを MemoTbl へ登録する．MemoTbl を CAM(Content Addressable Memory) を用いて構成することを仮定している．これは，入力一致比較を行うときの MemoTbl 検索オーバヘッドを小さく抑えるためである．MemoTbl 検索は命令区間を検出する度に行われることと，対象命令区間の入力が多ければ，その数に応じた比較回数を要するため，連想検索を高速で行える CAM が適している．

2.3 自動メモ化プロセッサのオーバヘッド

自動メモ化プロセッサの主なオーバヘッドは，現在の入力と MemoTbl 上の入力との一致比較にかかる時間である．MemoTbl は CAM で構成されており 1 度に全ラインを連想検索することができ，比較対象の入力をすばやく見つけることができる．そのため，一致比較コストにおいては，レジスタ参照やキャッシュ参照コストが支配的と

なる．故に一致比較にかかる総コストは，一致比較を入力数分だけ繰り返した総和として見積もることができるので，以下の式で表すことができる．

$$Ovh = Ovh_{Reg} \times N_{Reg} + Ovh_{mem} \times N_{mem} \tag{1}$$

式 1 の Ovh_{reg} はレジスタ参照のコスト，Ovh_{mem} はキャッシュ参照コスト，N_{reg}，N_{mem} はそれぞれレジスタとキャッシュ参照の数を示している．ここで具体的なプログラム例を用いて説明する．図 3 に，アセンブリプログラム例とそれに伴う一致比較の様子を示す．図 3 は，プログラム中の FuncA を再利用する場合の一致比較の様子を示している．FuncA の入力は，レジスタ R0，R1，とメモリのアドレス 0x20 上の値である．これは，プログラム中の 7 行目で R0 を，8 行目で R1 を，9 行目でメモリの 0x20 番地を参照していることから分かる．また，出力は R0 であることが，12 行目で関数内で計算した結果を R0 へ代入していることから分かる．つまり，関数 FuncA を再利用するには，FuncA を検出した段階で現在のレジスタ R0，R1，およびメモリの 0x20 内の値と MemoTbl 上の入力値を比較する．今回は，レジスタとの比較に 1cycle，キャッシュとの比較に 2cycle かかると仮定している．よって，FuncA の入力がレジスタ参照が 2 回，キャッシュ参照が 1 回であるため，式 1 を使ってコストを積もると，総サイクル数は，$1 \times 2 + 2 \times 1 = 4$ cycle となる．これが一致比較コストである．

自動メモ化プロセッサでは，関数を発見した段階で入力値一致比較を開始し，一致比較成功時に出力をレジスタとキャッシュへ書き戻すことで関数自体の実行を省略する．つまり一致比較成功時に削減されるサイクル数は，関数の実行にかかるサイクル数から一致比較にかかるサイクル数を除いた値である．そのため，小さな関数だが一致比較すべき入力の数が非常に多い場合には，関数そのものの実行時間より一致比較にかかるコストの方が多くなる可能性がある．その様なプログラム例を 4 に示す．この例では，Func_Sample の入力は，大域変数 a，b，c，d，e，f である．メモリ（キャッシュ）への参照を伴うため時間がかかり，オーバヘッドが大きい．このような場合は，Func_Sample を普通に実行した方が良く，メモ化を行わない方が良いといえる．

そのため我々の自動メモ化プロセッサではオーバヘッドフィルタと呼ばれるオーバヘッドの評価機構を備えている．オーバヘッドフィルタは，メモ化により削減されるサイクル数よりもメモ化操作に要するオーバヘッドの方が大きい関数に対し，メモ化の効果があるか否かを事前に判断し，効果が得られないと判断された関数をメモ化の対象から外す事により無駄なメモ化を削減する機構である．具体的には，命令区間のメモ化による削減サイクル数と，そのメモ化に要するオーバヘッドについて概算を行

```
int a=1, b=2, c=3, d=4, e=5, f=6; //大域変数
Func_Sample(){
  return (a+b+c+d+e+f); //大域変数の足し算
}

void main(){ //Func_Sampleを2回実行する関数
  printf("%d\n",Func_Sample());
  printf("%d\n",Func_Sample());
  return 0;
}
```

図 4: 入力数が多いプログラム例

うハードウェアを付加する．オーバヘッドフィルタによるメモ化時のコスト増加も考えられるが，ハードウェアで構成しているためそのコストは非常に小さい．

一方で，一致比較失敗時には失敗した時点で対象関数の実行を開始するため，一致比較に要したサイクル数が余分にかかってしまう．これは避けられないオーバヘッドである．これを図 5 に示す．図では, (a) 通常実行時, (b) 入力一致比較成功, (c) 入力一致比較失敗時のプログラム実行時間を示した．横軸を時間として右に向かって実行が進んで行く. (b) の例では，入力一致比較が成功し，関数 Func の実行が省略されるため高速化できる．しかし, (c) の場合では，入力一致比較の途中で入力値が一致しないことがわかり，その場で関数 Func の実行に移る．つまり, (c) の場合では，入力一致比較コストが上乗せされる分, (a) の通常実行より遅くなってしまう．この再利用失敗時のオーバヘッドは，どうしてもかかってしまう問題がある．

これまで自動メモ化プロセッサのオーバヘッドについて述べてきたが，現在の自動メモ化プロセッサでは一致比較失敗時のコスト発生が問題である．さらにオーバヘッドフィルタも現在のものは，動作速度を優先させるため，オーバヘッドコストの見積もりは簡単な概算を出しているだけに過ぎず，正確に見積もることができていない．つまり，このオーバヘッドの削減にはまだまだ課題があるといえる．そこで今回は，より一般的なプロセッサへメモ化機構の適用を考えて行く過程で，パイプライン機構によって，このオーバヘッドが隠蔽でき，自動メモ化プロセッサにおいて生じるオーバ

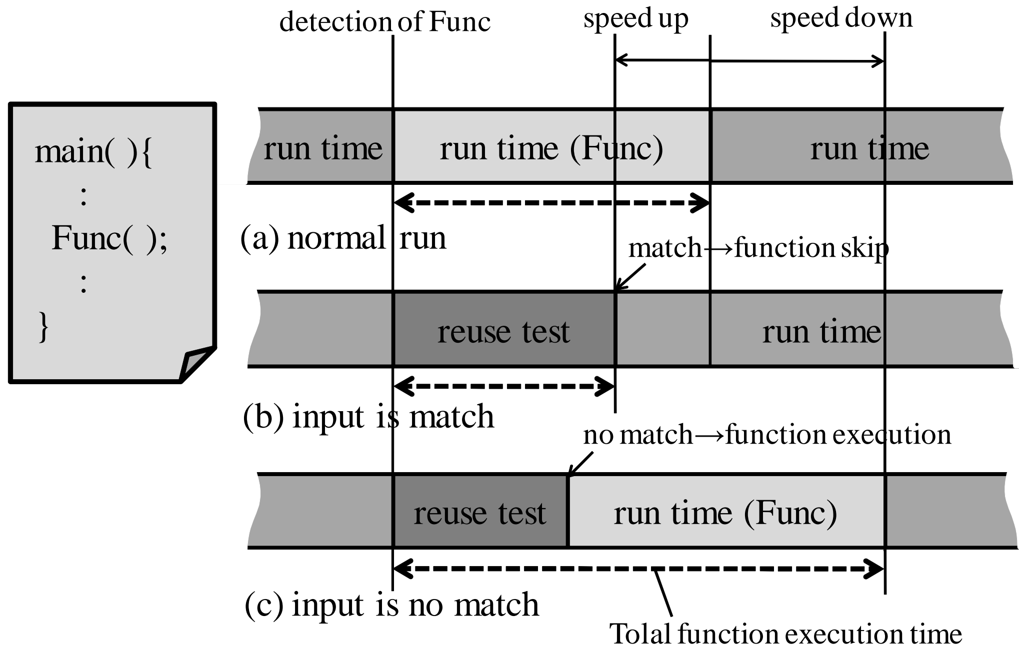

図 5: 再利用失敗時のオーバヘッド

ヘッドの改善方法を提案していく．次章では，提案手法であるスーパスカラプロセッサへのメモ化機構の適用に関して述べていく．

3 スーパスカラプロセッサへの自動メモ化機構の実装モデル

前章までは，これまで研究されてきた自動メモ化プロセッサについて述べてきた．しかし，メモ化の効果を確かめるための初期の検討として，プロセッサ構造が簡単な単命令発行のプロセッサを対象に研究してきた．そこで今回は，現在主流である，より複雑なアーキテクチャをもつプロセッサへの自動メモ化機構の実装を試みた．本章では，提案手法であるスーパスカラプロセッサへ自動メモ化機構を実装するための実装モデルについて述べる．今回は，現在一般的な環境となっているスーパススカラ機構をもつプロセッサとして，ARM プロセッサを対象とした．なお今回対象とするモデルでは，out-of-order 実行も可能としている．また，スーパスカラプロセッサは，命令レベルの並列性を利用したプロセッサの性能向上手法である．メモ化は命令レベルの並列性とは独立した，値の局所性を利用した手法であるため，お互いの手法を組み合わせて利用することができると考えられる．

3.1 ARM プロセッサの仕様

本節では，メモ化機構実装の対象とした ARM[?] プロセッサのアーキテクチャについて説明する．なお，今回対象とする ARM アーキテクチャは，命令分解型スーパスカラ機構を持つ ARM プロセッサとして，正確かつ詳細な動作シミュレーションを試みている OROCHI プロセッサ上の ARM プロセッサ部分である [?]．このプロセッサモデルは，ソフトウェアによるシミュレーションによる動作モデルの検証に始まり，FPGA による実現性の検証を経て，現在 ASIC(Application Specific Integrated Circuit) による遅延時間の検証が行われているなど，非常に現実的かつ実現性の高いプロセッサモデルである．しかし，本プロセッサモデルは，異種命令同時実行が可能なプロセッサを目指して設計されたプロセッサであり．通常の ARM プロセッサが持っていない特有の機構を取り入れている．具体的には，異種命令を同様に扱うために共通の内部命令に分解する，命令分解機構を取り入れている．さらに，分解した命令をスケジューリングするための VLIW キューを持つ．これらは，特異な機構であると思われるが，通常の out-of-order 実行可能なスーパスカラプロセッサが持つ，リザベーションステーションや命令ウインドウをもつ方式に比べて，フォワーディング機構や複雑な命令発行機構が不要であるため，動作周波数向上が期待できる．次からは，プロセッサモデルの概要を具体的に述べていく．

3.1.1 ARM 命令セットの特徴

まず ARM プロセッサの特徴について説明していく．ARM プロセッサは組み込み機器向けのプロセッサであり消費電力が低く高性能という特徴を持っている．現在では組み込み向けプロセッサとしては世界的なシェアを持つことで有名である．ARM プロセッサは RISC アーキテクチャを採用しているが，純粋な RISC の定義とは複数の点で異なる．以降にその特徴を述べる．

一部の命令実行サイクル数が変化する 一般的に RISC プロセッサは 1 命令 1 サイクルで構成される単純な演算命令で構成されるのが普通であるが，ARM の場合にはすべての命令が 1 サイクルで実行されるわけではない．複数ロード・ストア命令の実行サイクル数は転送されるレジスタ数によって変化する．

インラインバレルシフタ インラインバレルシフタは命令で使用される前に入力レジスタを前処理するハードウェアである．そのためレジスタに入っている値をシフトしてから足すという処理が 1 命令で行える．

Thumb16 ビット命令セット ARM はプロセッサコアを拡張し Thumb とよばれる第 2 の 16 ビット命令セットを追加している．このため ARM プロセッサは 16 ビット

命令も 32 ビット命令も実行でき，16 ビット命令を用いることでコード密度を高める事ができる．コード密度が高いとは，同じ処理をするために必要なコード量が少ないことを意味し，メモリ使用量を削減できる．メモリ使用量を抑えることは，組み込み用途のプロセッサにとっては必要不可欠な要素である．

条件実行　特定の条件を満たした場合にだけ命令が実行される．これにより，分岐命令が減り，性能とコード密度が高まる．

拡張命令　拡張ディジタル信号処理 (DSP) 命令が標準的な命令セットに追加されている．そのため高速に DSP 処理ができる．

以上の点が ARM 命令セットが組み込みアプリケーションに適している理由である．今回の実装モデルでは，基本的な ARM プロセッサアークテクチャである 32 ビットのアドレスサイズのものを参考とした．

3.1.2　レジスタ

次にレジスタについて説明する．まず，汎用レジスタはデータまたはアドレスを保持

表 1: ARM レジスタ構成

レジスタ	用途
R0〜R3	引数渡しレジスタ 1/作業レジスタ/結果格納レジスタ
R4〜R8	レジスタ変数
R9	レジスタ変数/静的ベースレジスタ
R10	レジスタ変数/スタック制限/スタックチャンク処理
R11	レジスタ変数/フレームポインタ
R12	ip(instruction pointer)/作業レジスタ
R13	sp(stack pointer) スタックフレームの最下部
R14	lr(link register)/作業レジスタ
R15	pc(program counter)
cpsr	条件コードレジスタ
spsr	条件コードレジスタの内容を待避するレジスタ

するレジスタである．ユーザモードで使用可能なレジスタを表 1 に示す．これは，ユーザモードすなわちアプリケーション実行時に通常使用される保護モードにおいて，アクティブなレジスタを示している．この場合，アクティブなレジスタは最大 18 個あり，

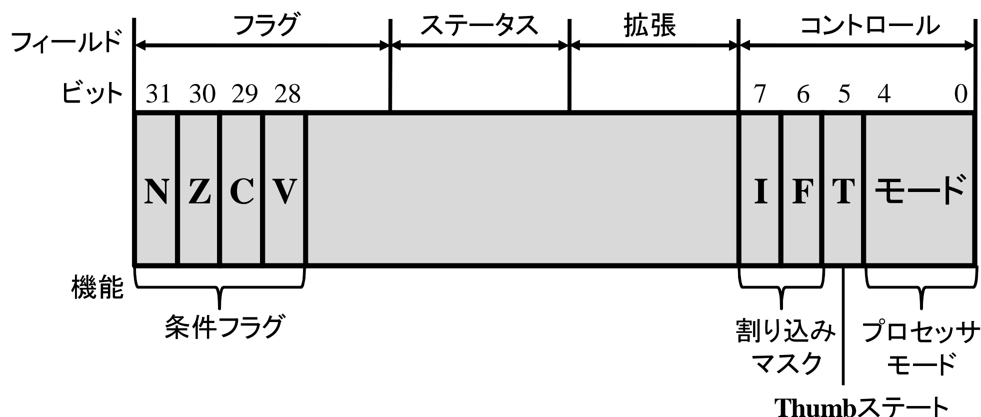

図 6: 一般的なプログラムステータスレジスタ (psr)

データレジスタが 16 個，プログラムステータスレジスタが 2 個ある．プログラマはR0〜R15 のレジスタを使用することができる．その中でも R0〜R3 の 4 つのレジスタは関数の引数渡しに使用される．それとは別に ARM プロセッサでは特別な役割を担うレジスタが 3 つある．それは R13，R14，R15 のレジスタである．順に R13 はスタックポインタ (sp) の格納場所であり，R14 はリンクレジスタ (lr) と呼ばれ，サブルーチンを呼び出すときに現在のプログラムカウンタ (pc) を格納し，サブルーチンからの復帰時に使用する．R15 はプログラムカウンタ格納レジスタであり，プロセッサが次にフェッチするアドレスを保持する．

また，ARM にはプログラム･ステータス･レジスタというものがある．これは**cpsr(current program status register)** である．一般的なプログラム・ステータス・レジスタの基本レイアウトを図 6 に示す．cpsr はフラグ，ステータス，拡張，コントロールの 8 つのフィールドに分かれ，それぞれが 8 ビット幅である．拡張フィールドとステータス・フィールドは将来の使用に備えて予約されている．コントロール・フィールドには，プロセッサ・モード・ビット，Thumb ステート・ビット，割り込みマスク・ビットがあり．フラグ・フィールドには条件フラグがある．プロセッサ・モードはどのレジスタがアクティブで，cpsr レジスタ自体へのアクセス権を持つかを決定する．プロセッサ・モードは特権もしくは非特権モードのいずれかであり，特権モードでは cpsr すべてのフィールドの読み出し書き込みが可能である．非特権モードの場合，cpsr のコントロールフィールドは読み出しのみが可能である．ただし，条件フラグは読み出し書き込みが可能である．

プロセッサモードは合計 7 つあり，6 つの特権モード（アボート，高速割り込み要求，

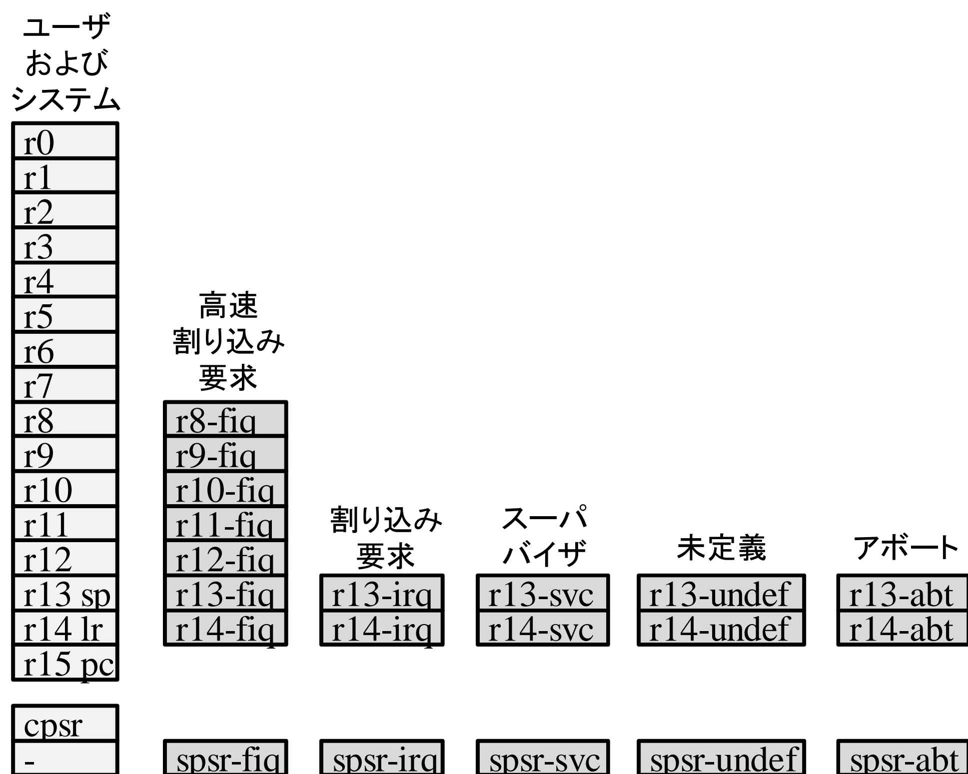

図 7: ARM レジスタ群（バンクレジスタ）

割り込み要求，スーパバイザ，システム，未定義）と 1 つの非特権モード（ユーザ）である．プロセッサはメモリアクセスに失敗するとアボートモードになる．また，高速割り込み要求モードと割り込み要求モードは ARM プロセッサの 2 つの割り込みレベルに対応する．スーパバイザモードは，リセット後のプロセッサモードであり，一般にシステムのカーネルが動作するモードでもある．システムモードはユーザモードの特殊版で，cpsr への読み出し，書き込みを許容する．未定義モードは，未定義命令の実行時のモードであり，ユーザモードはプログラムとアプリケーションの両方で使用されるユーザアプリケーションが一般に実行されるモードである．また，図 7 にレジスタファイルにある 37 個のレジスタを示した．このうち 20 個のレジスタは場合によってプログラムから隠される．このようなレジスタを**バンクレジスタ**と呼ぶ．図 7 ではユーザおよびシステムモード時のレジスタ以外が対応する．これらのレジスタは，現在のプロセッサモードにより見えるレジスタが異なる．ユーザモードを除いたすべてのプロセッサモードは，cpsr のモードビットに直接書き込むことでプロセッサモード変更することができる．またバンクレジスタはユーザモードのレジスタと 1 対 1 で対応しており，プロセッサモードを変更すると，既存モードのバンクレジスタと新しい

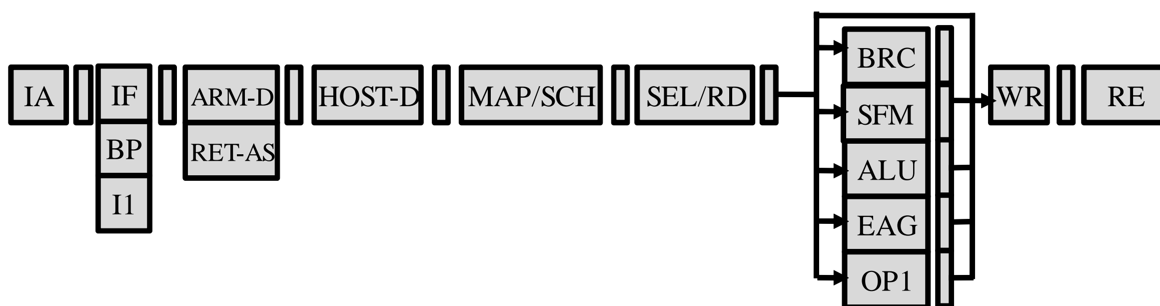

図 8: ARM プロセッサパイプラインステージ

モードのバンクレジスタが入れ替わる．図 7 には，割り込みモードで現れる新しいレジスタも示している．前のモードに戻るときに使用するレジスタである **spsr(saved program statusregister)** は，特権モード時しか変更できず，ユーザモードには存在しない．

ここまでは，ARM プロセッサの汎用レジスタについて説明してきたが，対象プロセッサモデルには，これらのレジスタに加えて，6 個の拡張レジスタが存在する．この拡張レジスタは，命令分解後の内部命令が使用するためのレジスタであり，この内部命令用のレジスタを **IREG(Implicit Register)** と呼び．対してユーザモードで使用する R0〜R15 までの汎用レジスタを **EREG(Explicit Register)** と今後呼ぶこととする．さらに，条件コードレジスタ cpsr も内部命令用のレジスタを cpsr とは別に持つ．次に，命令分解について説明する．

3.1.3 命令分解

本プロセッサモデルでは ARM 命令を RISC 型内部命令へ分解するモデルをとっている．これは，本プロセッサモデルが，異種命令混在実行を目指して設計されたプロセッサであるからである．また，命令分解を採用しているのは，ARM 命令セットが RISC 設計でありながら，1 命令で複雑な処理も可能な命令形態をとっていることも理由の一つである．パイプラインで命令を効率よく実行するには，命令が常に同じサイクルで実行されるほうが良いとされている．そのため，本モデルでは ARM 命令を内部命令に変換する手段をとっている．一方で既存の命令分解機構に，Intel 社の Netburst 機構があるが，本モデルの分解機構は演算機をより単純化している．

3.1.4 パイプラインステージ

次に，ARM プロセッサのパイプラインステージについて述べる．図 8 にパイプラインステージのモデルを示す．ただし，図 8 では VLIW キューは省いている．以下，それぞれのステージの役割を説明する．

IA(Instruction Address) ステージ　命令フェッチアドレスをプログラムカウンタから計算し，セットする．

IF(Fetch) ステージ　g-share 分岐予測機構を用いて命令キャッシュから連続 2 命令を読み出す．

ARM-D(Decode) ステージ　ARM 命令を RISC 型内部命令に変換する．また，1 サイクルに可能な分解は，最大 2 個の ARM 命令から，最大 4 個の内部命令までである．4 命令を越える分解は複数サイクルにより行う．例として，乗算命令は最大 22 命令に分解する．

HOST-D(Decode) ステージ　実行条件付命令を条件に関わらず依存関係が変化しない 2 つの命令列に分解する．

MAP/SCH(register mapping/schedule) ステージ　汎用レジスタ EREG と拡張レジスタ IREG を合わせた論理レジスタから，命令ウインドウである **ROB(Reorder Buffer)** へのマッピングを行うとともに，VLIW キューに命令をセットする．

SEL/RD(select/read) ステージ　命令発行を行うステージであり，演算機からのバイパスが利用可能かどうか調べるとともに，依存関係の待ち合わせを行う．

BRC/SFM/ALU/EAG/OP1(Instruction Execution) ステージ　命令実行ステージである．BRC は分岐命令を処理するユニットであり，SFM はシフト演算と積和演算用の補助演算を，ALU は演算命令を，EAG はアドレス計算と積和補助演算を，OP1 はロード，ストア命令をそれぞれ処理するユニットを示している．

WR(WriteBack) ステージ　ROB への書き込みを行うステージである．

RE(Retire) ステージ　先行命令がすべて完了した命令を ROB から論理レジスタへ書き戻し，すべての処理を終了する．なお，Retire ステージで処理される命令の順番は，元の分解前命令列と同じであることが保証されている．

なお本プロセッサモデルでは，各ステージのユニットを単純化することで，命令を各ステージ，すべて 1cycle で動作するよう設計されている．むろんキャッシュミスや分岐予測ミスなどのパイプラインハザードが起った時，もしくはソフトウェア割り込みや周辺機器からの外部割り込みが起った時などは例外である．

3.1.5　リオーダバッファ(ROB)

次に ROB について説明する．ROB は，命令ウインドウのことであり，命令のオペコードやレジスタ番号など，実行中の命令に関する様々な情報を保持している．

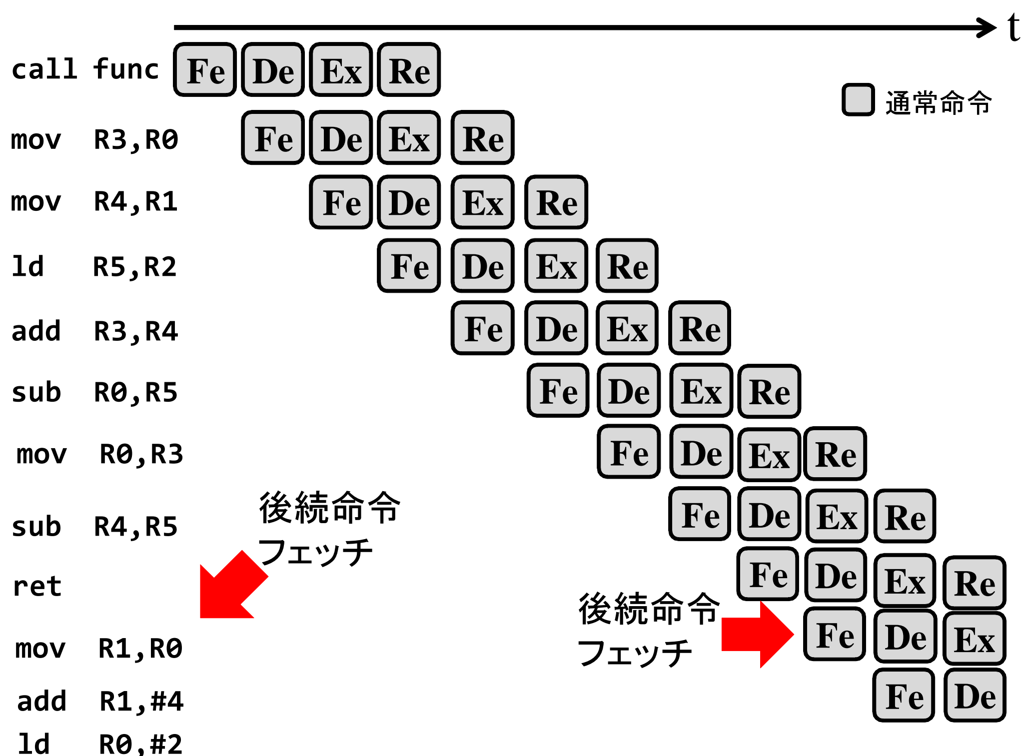

図 9: パイプライン動作例（通常実行時）

3.1.6 命令同士の依存解析

命令には依存関係があるものが 存在する．本プロセッサは out-of-order 実行を想定している．out-of-order 実行は命令の順序を入れ替えることにより，命令同士の依存関係を原因とするパイプラインのストールを削減する手法である．そのため，依存依存解析が必要となる．本プロセッサでは，命令の依存関係を先行命令のデスティネーションレジスタ番号と後続命令のソースレジスタ番号を比較することで行っている．依存関係を解析したあとその情報をもとに VLIW キューへ命令を投入することでスケジューリングを行う．

3.2 提案手法の概要と動作

本節では，提案手法の概要と再利用成功時の動作および失敗時の動作について述べる．

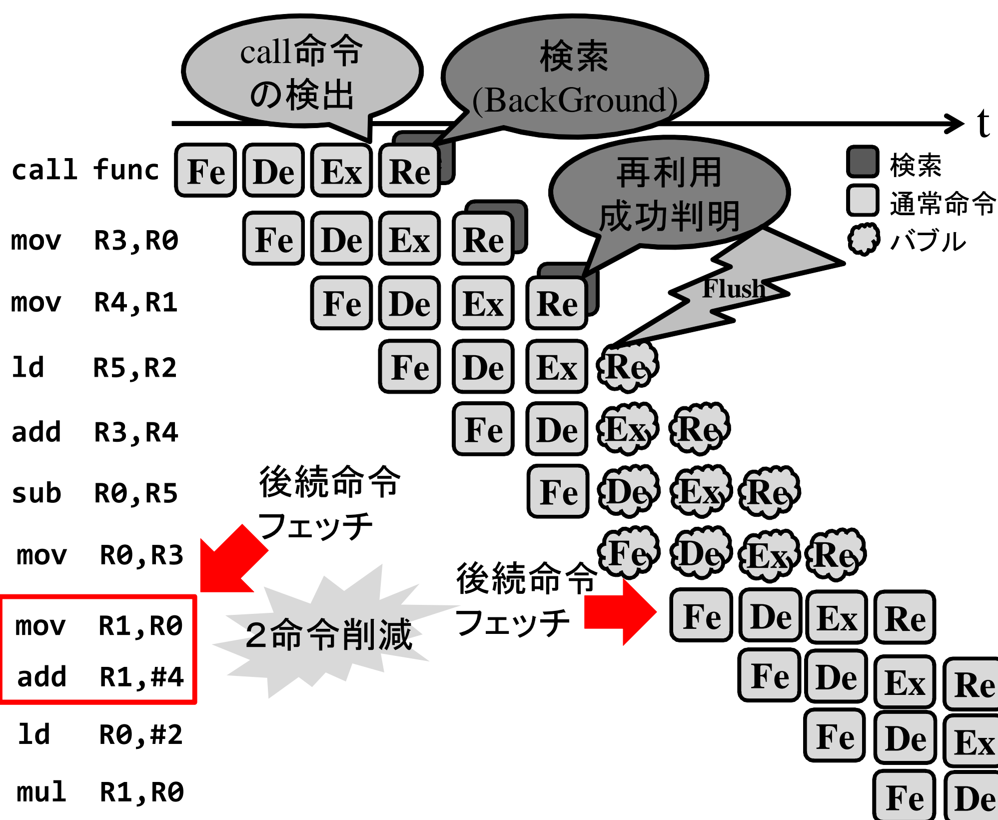

図 10: パイプライン動作例（再利用成功時）

3.2.1 再利用成功時の動作

提案手法における再利用成功時の動作について説明する．なお，提案手法のモデルでは，MemoTbl 検索とパイプラインでの命令実行は平行して行うこととする．また，検索はリタイアステージで行う．本来なら命令をデコードした段階で，検索開始のために必要な call 命令の検出が可能であるため，デコードステージでも検索が可能であるように思われる．しかし，実際は検索における入力値の一致比較は，過去の入力と現在の入力（レジスタの内容やキャッシュの内容）とを比較するものであり，デコードステージの段階では，関数呼び出し call 命令以前の命令が実行されていない可能性がある．そのため関数以前の命令が全てリタイアされてから，つまり，関数の入力となる値が全てレジスタやキャッシュに存在することが保証されていなければならない．そこで，call 命令がリタイアされたことが確認できれば，関数以前の命令がすべて終了したことが保証されることを利用し，リタイアステージのタイミングで call 命令の検出と検索処理を行うこととした．

ここで，図 9 に示すような命令列を本提案プロセッサで実行する場合の動作を考え

る．図 9 は，関数の開始から終了までのパイプライン処理の様子を示したものである．まず 1 行目で call 命令により，関数 func が呼び出され，9 行目の ret 命令で関数 func を終了したのち，関数が終わった後の後続命令をフェッチし命令実行が進んでいった時のパイプラインの様子を模式図的に表した図である．また，横軸は時間 t を表し，右方向へパイプライン中を命令が進んでいく．パイプラインステージは，それぞれ以下のように定義する．

- Fe:フェッチステージ
- De:デコードステージ
- Ex:実行ステージ
- Re:リタイアステージ

なお，今回の例では説明を簡略化するためにパイプラインステージを以上のような 4 つのステージとし，1 ステージ 1 サイクルで処理される．また，load，store 命令などメモリへのアクセスに関しては Ex ステージに含め，演算結果の Write 処理は Re ステージに含めている．まず，図 9 は再利用を行わない場合に，命令がパイプラインを流れていく様子を表している．この例では，すべての命令が 1 サイクルで処理され，パイプラインハザードは起ってはいない．よって関数 func を実行するのにかかるサイクル数は，call func 命令がフェッチされてから ret 命令がリタイアされるまでの総サイクル数なので，実行に 12 サイクルかかっていることが分かる．

次に再利用時の動作を図 10 を使って説明する．この例での命令列は図 9 で示したものと同じである．まず，call func 命令がパイプライン中の Re ステージに来て，call 命令を検出すると MemoTbl を検索し始める．なお検索は，命令実行と並行して行うことができるため，検索は命令実行に対してバックグラウンド (BackGround) で動作している様に捉えることができる．この例では，3 サイクルで検索がヒットし，再利用が可能であることが判明する．再利用可能であることが分かると，次の 1 サイクルでパイプラインフラッシュを行う．パイプラインフラッシュは，パイプライン中の命令を削除する処理であり，一般的には分岐予測が失敗したときや，割り込みが検出されたときなどに行われる．具体的には，パイプラインレジスタの内容であったり，ROB 内の命令，VLIW キュー内の命令が消去される．その後，関数の出力を MemoTbl から取りだし，レジスタとキャッシュに書き戻す．なお図 10 では，書き戻す（ライトバック）処理は説明の都合上省略している．その後，後続命令（関数が終了したあとに最初に実行する命令）をフェッチし，プログラムの実行が続いていく．これら一連の処理を行うことで計算再利用を実現しており，今回の例では，2 サイクルの性能向上を

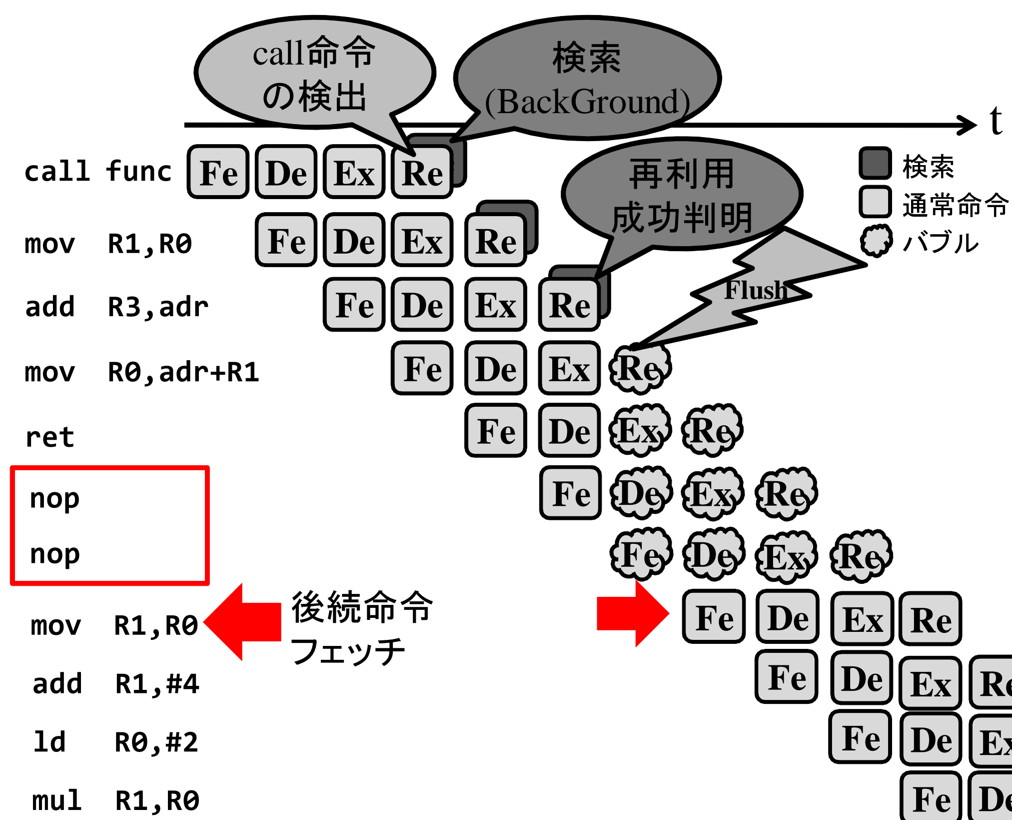

図 11: パイプライン動作例（パイプラインフラッシュのデメリット発生時）

実現している．

ここで，再利用が成功すると判明した時点でパイプラインフラッシュを行う理由について説明する．ここでのパイプラインフラッシュは，関数内のまだ実行していない命令を削除する処理である．もし，パイプラインフラッシュを行わないと，関数内の命令がパイプラインに残っているため，それらの命令がライトバックによってレジスタやキャッシュに書き込まれた値を上書きしてしまう恐れがある．つまり，その後のプログラム動作が変わってきてしまい，正しく動作しない．今回のモデルでは，検索がヒットした段階で，パイプラインフラッシュを行い，関数の出力をライトバックした後，後続命令をフェッチしている．しかし，この方法にも欠点がある．図 11 の場合である．図 11 は，とても小さな関数を想定してメモ化を行った場合である．この例では，検索に 3 サイクル要し，再利用成功が判明したあと，前例と同様にパイプラインフラッシュし，出力をライトバックした後，後続命令をフェッチしている．しかし，この例では再利用を行わなければ ret 命令の後に，すぐ後続命令 mov をフェッチできていたはずである．今回は結果的に再利用したことにより，後続命令実行が遅れてしま

う例を示した．この原因としては，検索が成功した段階で再利用しようとするためであり，この関数を本当に再利用した方が良いかの判断基準に，関数の大きさ（関数内の命令数）などが全く考慮されていないためである．この様な例に対応するのに，理想的な方法は，パイプライン上の関数内命令のみをフラッシュして，関数の結果をライトバックした後に，既に再利用成功が判明する前にフェッチが終わっている後続命令の実行に移る方法である．しかし，本プロセッサはout-of-order 実行であり，分解された命令が順序を入れ替えられてパイプラインレジスタや VLIW キューに入っているため．この命令が関数内命令であるかを判断できず，関数内命令だけを消去することができないため今のモデルのままでは不可能である．さらに，もう一つ方法が考えられる．関数終了後の後続命令がフェッチされたことを検出し，再利用を中止すればその方が高速化できるのではないかと考えられる。しかし，この方法では，パイプライン中に残る再利用対象の関数の命令を実行したときのサイクル数と再利用にかかるサイクル数を比較する必要がある．このうち再利用対象の命令の実行サイクルを見積もるのは非常に難しい．たとえば，実行する命令の中にキャッシュへのアクセスを伴う命令が含まれていた場合，キャッシュミスが発生する場合があるため対象サイクル数予想は困難であるといえる．そこで，今回はこの様なプログラムでも決まった再利用のための動作を行う事とした．

3.2.2　再利用失敗時の動作

次に再利用が失敗したときの動作について説明する．我々がこれまで研究してきた自動メモ化プロセッサでは，再利用失敗時には，必ずオーバヘッドが発生していたが，今回提案するプロセッサでは再利用失敗時にそのオーバヘッドが生じない．これは，MemoTbl 検索動作を通常の命令実行と同時に行っているため，再利用が失敗したときに，一から命令をフェッチし実行し始める必要がないからである．つまり，再利用のための動作がパイプライン上の動作と独立しているため，再利用失敗時のミスペナルティがまったく存在しない．

この手法は，再利用失敗時のオーバヘッドが無いという以外にもメリットを持っている．再利用失敗時の動作例を図 12 に示す．この例は，関数の大きさ（命令数）は小さいが，検索に時間がかかるプログラム例を示している．検索に時間がかかる関数としては，関数の入力数が多い場合や，今回の例のようにメモリへのアクセスが多いときが考えられる．この例では，本来再利用成功が判明するのは，検索から 6 サイクル目である．しかし，その前に ret 命令がリタイアステージで検出されたため再利用検索を終了できる．この処理は，対象関数を再利用するのに費やされる総サイクル数が，

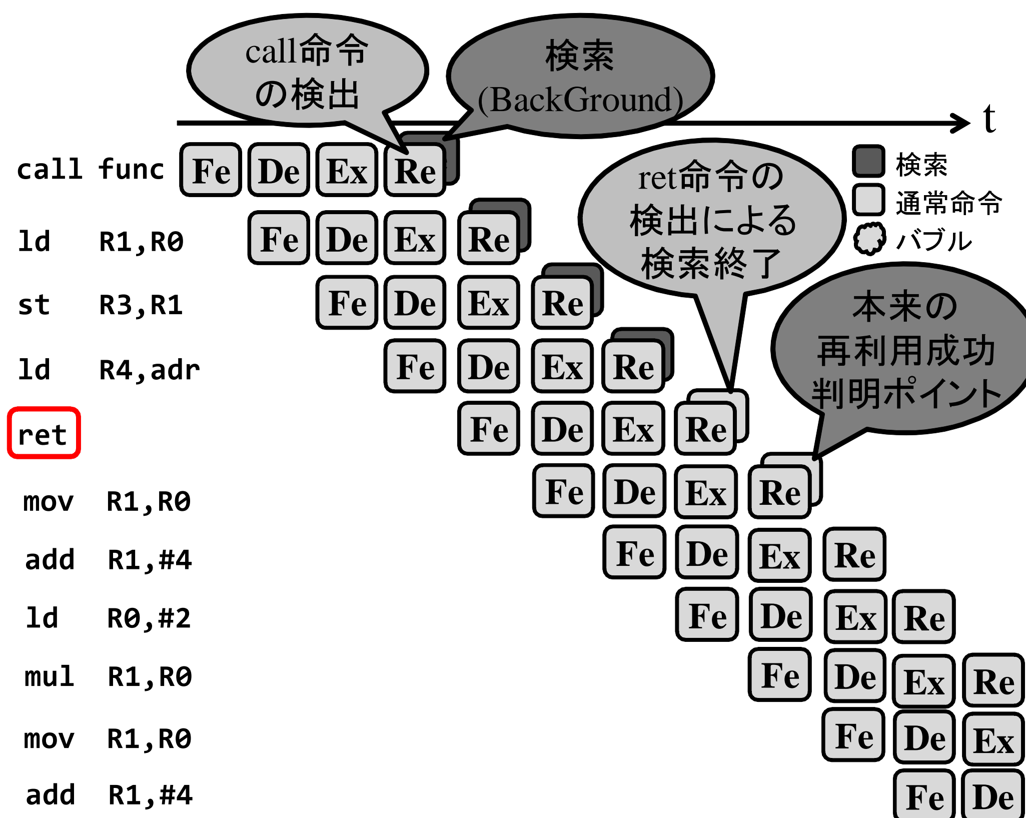

図 12: パイプライン動作例（再利用失敗時）

対象関数の実行にかかる総サイクルより多い場合に，再利用を行わないことを意味し，再利用した方がかえって遅くなってしまう様な関数を再利用対象から外すことに相当する．つまり，再利用が効果的でない関数に対するフィルタリングをしているようなものである．これは，従来の自動メモ化プロセッサでも行っていた．従来の方法は，再利用で削減できるサイクル数と再利用オーバヘッドを専用のハードウェアを用いて見積もり，再利用しないほうが良い関数を再利用対象から外していた．しかし，この機構は高速に動作させるために，複雑な見積もり計算は行っておらず，見積もりの精度はあまり良くない．また専用のハードウェアを追加する必要もあるため，実装コストがかかってしまう．その点今回の提案手法では，計算によりオーバヘッドを見積もる必要がないことや専用のハードウェアを追加する必要もないため有効な手法であると考えられる．

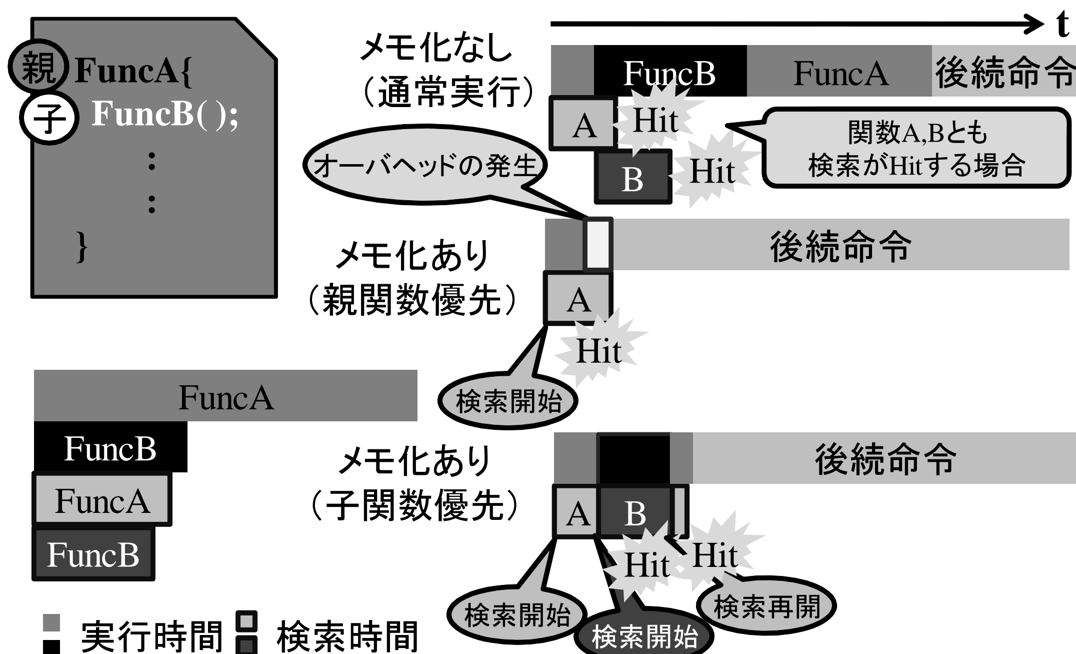

図 13: 親関数優先検索例

3.3 多重再利用時の検索

次に従来の自動メモ化プロセッサでも対応していた，関数の多重再利用について，本提案手法でのモデルを述べる．提案するプロセッサでは，多重再利用をする場合には気をつけなければならないことがある．入れ子構造である関数を多重再利用する場合に，親関数を検出すると同時に MemoTbl 検索を始め，親関数の検索がまだ途中であるにも関わらず，新たに子関数を検出した場合，はたして親関数と子関数のどちらを優先して検索したらいいかを考えなければならない．そのメリット，デメリットについて以降述べていく．

3.3.1 親関数優先検索

まず，親関数を優先して検索する場合について図 13 を用いて説明する．図 13 は，親関数優先検索時の動作と子関数優先検索時の動作をタイムライン形式で示している．図の左上の様な関数があった場合，関数それぞれの実行時間と検索時間が左下の様なタイムラインであったとする．さて，このような入れ子関数を実行する場合の動作を右上に示した．これは，メモ化なし，つまり通常実行時の様子を示している．まず，関数 FuncA の実行を開始し，すぐに関数 FuncB を見つけるため，FuncB の実行に移り，FuncB 終了後に FuncA の残りの命令区間を実行するという様子が分かる．ここで，検索に注目する．まず，FuncA を検出した段階で FuncA の検索を始めるが，すぐに FuncB

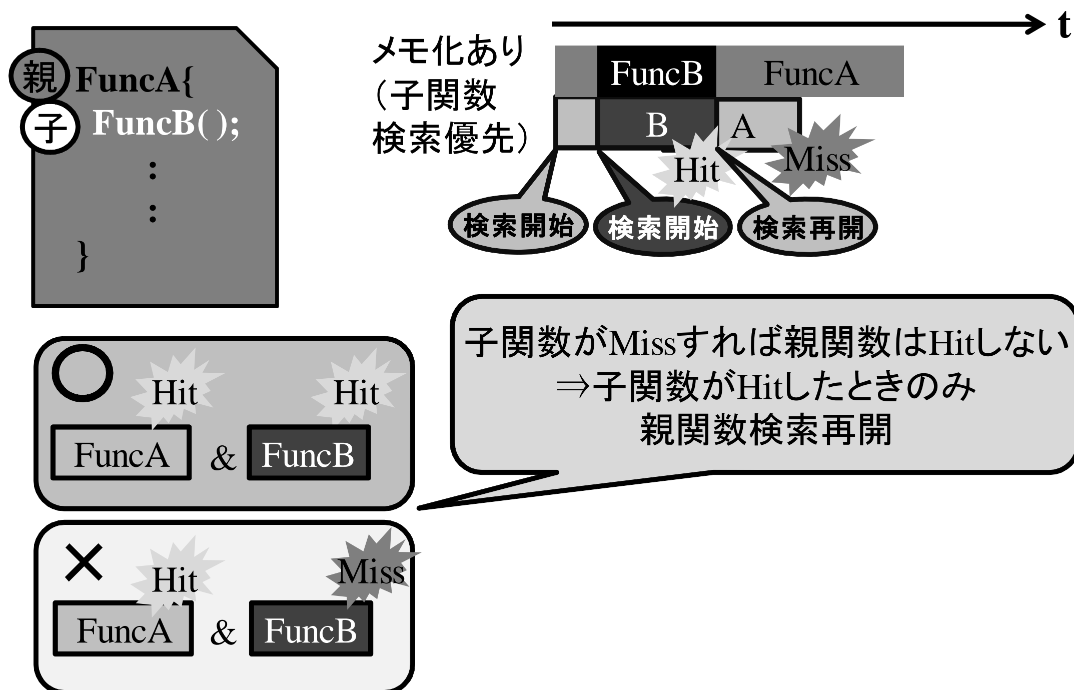

図 14: 子関数優先検索例

を検出してしまうため．そこからの検索をどうするかが問題となる．それは，検索が一度に一つの関数にしか行えないためである．なお，今回の例では，FuncA，FuncB ともに検索がヒットする例を用いた．

ここで，メモ化あり（親関数優先）の例について見ていく．FuncA 実行と同時に検索も開始すし，検索途中に FuncB を検出する．ここで親関数優先検索の場合，親関数である FuncA の検索が終わるのを待つため図のようにオーバヘッドがかかってしまう．しかし，今回の例では FuncA の検索がヒットするため FuncB の実行を省略できる分，後続命令をより早く実行できるメリットがある．それに比べ右下の例，メモ化あり（子関数優先）では，FuncA 検索途中でも FuncB を検出した場合には，子関数検索に切り替えるため後続命令の実行が遅れてしまうというデメリットがある．つまり，この様な親関数，子関数ともにヒットする場合には子関数の検索が省けることによるメリットが得られる．

3.3.2　子関数優先検索

次に，子関数優先検索時の場合を図 14 を例に説明する．この例では，子関数に対するる検索はヒットするが，親関数の検索はミスする場合を示している．まず，FuncA の実行と同時に FuncA の検索も開始する．子関数優先検索なので FuncB を見つけた段階

で FuncB の実行と検索に移る．そして，FuncB の検索がヒットするため FuncB を再利用し FuncB の実行が省略される．その後，FuncA の残りの実行とともに FuncA の検索途中から検索を再開するという動作である．図からも分かるように，子関数優先検索の場合には，新たな関数を検出した場合，すぐにその関数の実行と検索に移るため，親関数優先時の様にオーバヘッドが生じないというメリットがある．さらに，図 14 の左下の例のように子関数がヒットする場合，親関数はヒットする可能性があるが，子関数の検索がミスした場合，親関数の検索もミスするという事が言える．故に，子関数がミスしたら親関数の検索をする必要がなく，再利用機構の余分な動作を減らすことができるため，電力の削減効果が見込める．

しかし，子関数優先検索手法にはデメリットもある．それは，子関数検索が成功した場合である．子関数の検索が成功した場合には，親関数の検索を再開する必要があるため，子関数を呼び出す前までに検索していた，親関数検索の途中のポイントを記憶しておかなければならないと考えられる．そのためには，検索途中のポイントを記憶しておくためのバッファ（スタック）を追加しなければならない．スタックを用いるのは，関数の入れ子構造の階層が深いときにも対応するためである．このバッファを設けるのはハードウェアコストがかかるのでデメリットであると言える．

ここまでに，2 つの方法を述べてきたが，今回は比較的簡単に実装ができると考えられる親関数優先検索手法を採用した．具体的な理由としては，MemoBuf への入出力登録に関して，今回は MemoTbl 検索とパイプライン上の命令実行を並行して行う仕様にしているため，従来の自動メモ化プロセッサでは再利用が失敗したときに，MemoBuf への入力の登録作業を行っていたものを，call 命令検出時に関数内の命令実行と並行して，MemoTbl 検索と登録の両方を行わなければならないからである．そのため，子関数を優先する場合には，子関数を見つけるとすぐに子関数の実行に移らないといけないため，親関数の MemoTbl 上の入力を MemoBuf へ登録している場所を上書きせずに残しておき，子関数の登録場所をその上の MemoBuf 上のラインとする必要があるので，MemoBuf の管理が複雑になることや MemoBuf の数（深さ）をさらに増やさなければいけないという理由から親関数優先検索手法を適用した．

4　実装

本章では，提案するメモ化機構を追加した ARM プロセッサの実装について述べる．また，実装上考慮した点などについても説明する．

4.1 SP 相対による MemoTbl への入力値登録

本節では，メモ化において関数の入出力を MemoBuf，MemoTbl へ登録する方法について説明する．今回，ARM プロセッサにメモ化機構を実装する上で，メモ化機構のモデルを変更したため，その理由や実装方法についても述べる．

4.1.1 引数レジスタ数による再利用率の低下

これまで研究されてきた従来の自動メモ化プロセッサでは，引数レジスタとして 6 つのレジスタを用いる事ができた．これは，引数の数が 6 つまでの関数についてメモ化が可能であることを示している．逆を言えば 6 つより多い引数を持つ関数をメモ化することはできていなかった．一方で，ARM プロセッサでは引数レジスタは 4 つと定義されているため，従来と同じメモ化方法では，メモ化可能な関数が引数 4 つのものに限定されてしまう．これは，従来の自動メモ化プロセッサにおいて，引数 5 個，6 個のものがメモ化対象からはずれることに相当し，再利用率の低下や，実行速度の低下を招く可能性がある．実際，事前評価として従来の自動メモ化プロセッサを用いて，関数の引数が 4 つ以下のものだけをメモ化対象とする制限を加えたのち，SPEC CPU95 ベンチマークを用いて評価を行ったが，ほとんどのベンチマークにおいて同等か性能悪化が確認された．この結果だけを見ると，メモ化において引数が多いものが，再利用率などの性能に関わっていることが推測できる．

なぜこのような結果が得られたかを考えると，まず考えられるのは関数の引数と関数を再利用したときに削減されるサイクル数の関係である．一般的に，関数の引数が多い方がその関数を再利用するための MemoTbl 検索コストは大きい傾向がある．しかし，引数が多い関数ほど関数の大きさ（関数内の命令数）が大きいことが多く，再利用が成功すれば削減されるサイクル数も大きいという特徴がある．そのため SPEC CPU95 においては，メモ化による効果が，引数の大きい関数によるものに大きく影響しているということがこの結果から分かる．また，引数が多ければ，一致比較すべき入力数も一般的には増加し，異なった引数パターンで関数が呼び出されることも多いのではないかと，これまでは考えられてきた．たくさんの入力パターンが存在すれば，それだけ再利用が当りにくくなるはずである．しかし，今回の結果から SPEC CPU95 では，異なった引数パターンで関数が呼ばれることは少なかったと考えられる．

よって，以上のことから ARM 版の提案プロセッサに従来と同じメモ化機構を実装し，SPEC CPU95 において評価しても再利用による効果はわずかである可能性がある．もちろん，命令セットが異なるため，これまでに述べた特徴が ARM 命令セットにおいても同様に見られるとは限らない．しかし，命令セットが違っても，引数の大

きい関数ほど命令数も多くなる傾向があるのは確かであり，一般的なプログラミングを行った場合は当然そうなる．この傾向は，命令セットが変わったからといって変わるものではない．そのため，関数の引数が 4 つ以下のものだけをメモ化対象とする仕様を改善しなければならない．

4.1.2 ARM プロセッサへのメモ化機構適用時の問題点と改善策

ここまで，ARM プロセッサでは引数レジスタが 4 つしか使用できないため性能低下につながる可能性があることを述べてきた．これは，従来通りのメモ化機構を ARM に適用した場合に起こることであり，メモ化機構のモデルを変更する必要がある．

そもそもなぜ従来のメモ化方法では，引数レジスタの数によってメモ化可能な関数が制限されてしまうのか，その理由について説明する．まず，自動メモ化プロセッサでメモ化を行うには，関数の入出力を検出し，MemoBuf へ登録する必要がある．そこで，どの値が入出力となるかを判断しなければならない．例として関数 f を再利用するときについて説明する．関数 f の入力となるのは，f が参照する大域変数と明示的引数，そして f の親関数中の局所変数である．

そこで，f の入出力をメモリマップ上で識別するために，**ABI(Application Binary Interface)** の定義を使用する．従来の自動メモ化プロセッサは SPARC[?] を対象に考えていたため，SPARC ABI を用いて判断していた．図 15 に SPARC のスタックフレーム概要を示す．図 15 から分かるように SPARC では，スタックフレームの使い方が ABI により明確に規定されている．図 15 の%fp はフレームポインタ，%sp はスタックポインタを示す．また，%sp 以上の領域のうち，%sp+0〜63 はレジスタ待避領域，%sp+68〜91 は引数待避領域であり，いずれも関数の入出力ではない．その他に，構造体を返す場合の暗黙的引数は%sp+64〜67 に格納され，明示的引数は，アウトレジスタ%o0〜5(%sp+68〜91)，および，%sp+92 以上の領域に置かれる．さらに，大域変数は，LIMIT 未満の領域に置かれると定められている．

ここで問題となるケースを図 15 を用いて説明する．FuncA 実行時には，LIMIT 未満の領域に命令および大域変数，また%sp 以上に有効な値が格納されている．ここで注目すべきなのは FuncB に対する明示的引数の先頭 6 ワードはレジスタ待避領域である%o0〜5(%sp+68〜88) に格納され，第 7 ワード以降は%sp+92 に格納されることである．ベースレジスタを%sp とするオペランド%sp+92 が出現した場合には，この領域は引数の第 7 ワードすなわち FuncB の明示的引数である．一方で，オペランド%sp+92 が出現しない場合，この領域は FuncA の局所変数である．このように，FuncA 実行時には FuncA の局所変数と FuncB の明示的引数を区別することができる．一方で，FuncB

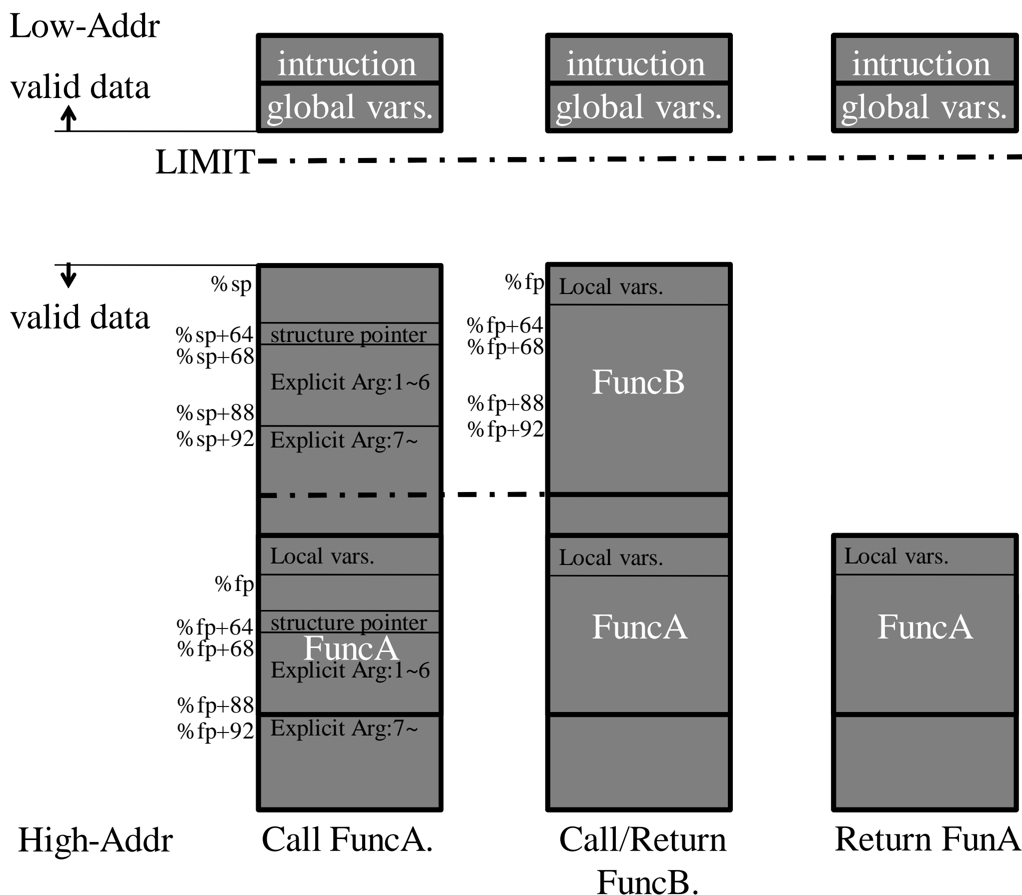

図 15: スタックフレームの概要 (SPARC)

実行時の図を見ると明示的引数が入力，返り値が出力，大域変数および FuncA の局所変数からの読み出し，書き出しが入出力となりうる．ただし，FuncB は可変長引数を受け入れる場合があるため、一般には%fp+92 以上の領域が FuncA の局所変数か FuncB の明示的引数かの区別ができない．そこで FuncA 実行時に引数の第 7 ワード以降を検出した関数は再利用対象から除外し，第 7 ワード以降を検出しない関数に関しては，直前に%sp+92 のアドレスを記録しておく．そして，FuncB 実行時において主記憶参照アドレスが，記録しておいた%sp+92 の値以上のときは FuncA の局所変数，小さい場合には FuncB の明示的引数として区別していた．つまり，関数の呼び出される側 (callee) では，明示的引数と親関数の局所変数の区別がつかないことが問題である．これは，ARM プロセッサでも言えることである．

一方で，ARM プロセッサでは，従来のメモ化方式を適用し，関数の第 5 引数以上を検出したとき，その関数を再利用対象から除外することができないと言う問題がある．

それは，ARM プロセッサでは，SPARC の様に ABI でスタックフレームの使い方が詳細に決っておらず．コンパイラがコンパイル時にそのプログラムごとにスタックの積み方を決定している．よって，関数の第 5 引数の検出そのものが困難であり，従来手法をそのまま適用することはできない．

そこで今回は，スタックを介して受け渡される引数と，入力となる親関数の局所変数を全て SP 相対形式で Memobuf，および MemoTbl へ登録するモデルを提案する．第 5 引数以上は SP 相対でアクセスされるため，その登録も SP 相対形式で行われるのが正しい．しかし，スタックフレーム上に置かれた引数と親関数の局所変数の区別がつかないため，親関数の局所変数も SP 相対で記憶する事とした．逆に上記の登録を絶対アドレスで行った場合，第 5 ワード以上の引数の一致比較は，関数が呼び出されたときのそれぞれの SP が異なるとき，つまりスタックフレームの状態が異なった時に一致する保証がないため，絶対アドレスでの登録は行うことができない．

ここで，登録すべき入力の登録方法を整理し，以下に示した．

- 大域変数 → 絶対アドレス形式で登録
- 自身の局所変数 → 不登録
- 親関数の局所変数 → SP 相対形式で登録
- 第 5 ワード以降の引数 → SP 相対形式で登録

また，MemoBuf と MemoTbl への登録は，どちらも SP 相対形式で登録し，MemoTbl の検索も SP 相対形式で行うこととする．

一方で，SP 相対形式での登録方法を採用する上で，再利用が成功したときの MemoBuf 上の動作を変更したので，そのことについて説明する．図 16 に再利用が成功したときの MemoBuf の動作を示す．図では，現在プログラム中の FuncC を実行しているものとする．通常は，関数実行と同時進行で MemoBuf への入出力登録と MemoTbl 検索を行っている．そこで実行時の入出力登録と MemoTbl 上の入出力を MemoBuf へ登録する処理を同時に行わなければならない．これは，子関数の入出力は親関数の入出力に含まれることがあり，再利用が成功した場合にその入出力を必要があれば親関数の入出力として登録しなければならないからである．そこで，MemoTbl 上の入出力を MemoBuf へ持ってくることで，再利用が成功したときに，親関数に対して結果を反映する．再利用が成功した場合，実行自体を省略してしまうため，実行時に登録していた入出力は途中までしか登録されておらず，親関数に反映できない．そのため MemoTbl 検索時に登録しておいた入出力を図の様に用いる．また，MemoTbl，MemoBuf では SP 相対形式で入出力が登録されている．しかし，その入出力を親関数に反映する場合，

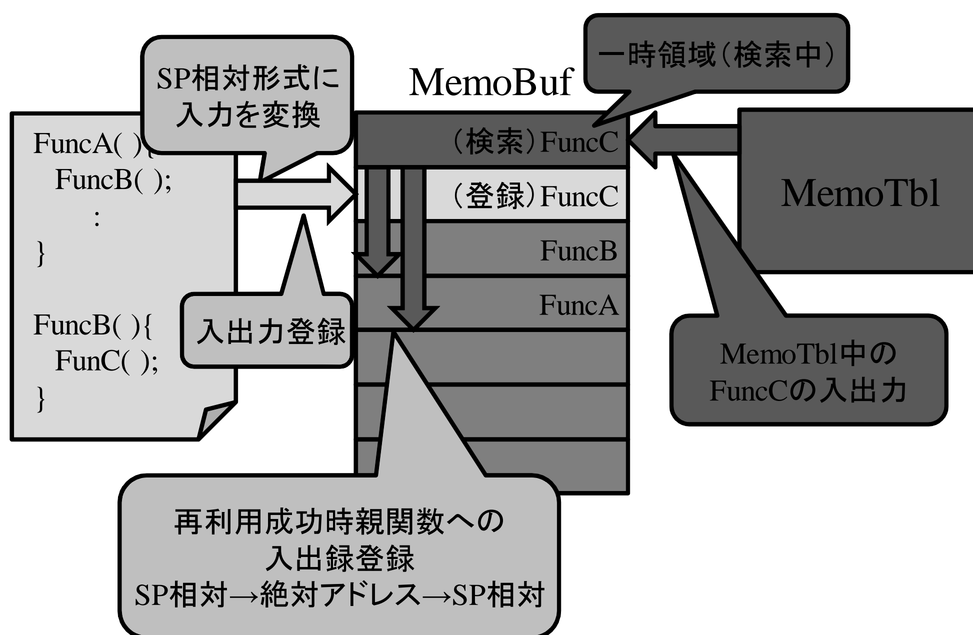

図 16: 再利用成功時の MemoBuf 上の親関数への登録値の反映

親関数とでは当然 SP が違うため，一度絶対アドレスに変換した後，親関数のスタックポインタを見て，その関数に適した SP 相対形式に計算しなおしてから親関数に反映する必要がある．

4.2 その他の実装上の考慮点

本節では，前項までに述べてきた以外の，提案手法を実装するうえで考慮した点について説明する．ここで本提案プロセッサの実装モデル図を図 17 に示す．これは，図 8 で示したパイプライン機構の図に，キャッシュやメモリ，ストアバッファ，MemoTbl，MemoBuf，キャッシュコントローラを追加したもので，提案手法プロセッサの全体図を示している．図の I$1 は，1 次命令キャッシュを表し，D$1 は 1 次データキャッシュ，$2 は共有 2 次キャッシュ，MEM はメモリを表している．また，L2_ctl は 2 次キャッシュコントローラ，L2_ctl_buf は 2 次キャッシュコントローラがあふれたときのためのバッファ，MEM_ctl はメモリコントローラを表しており，それぞれ命令フェッチからのフェッチ要求，ロード・ストア命令からの要求，MemoTbl からの入力一致比較時のキャッシュ参照要求などを管理している．また，STBuf はストアバッファを表しており，ストア命令を一時的に保存しておくためのキューである．なお，ロード命令は，キャッシュから値がロードされるまで，パイプラインはストールするが，ストア命令は，ストアバッファに空きがあれば，ストアバッファに命令を投入することで OP1 ステージ

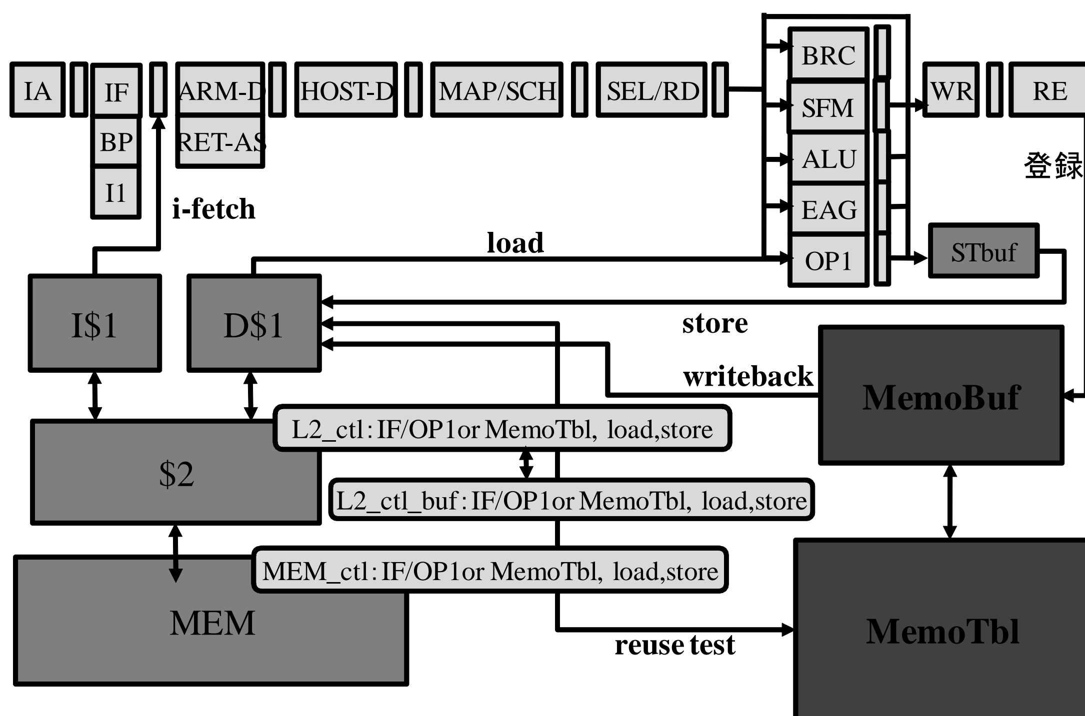

図 17: 実装モデル

での処理を終えたと判断し，パイプラインがストールすることはない．

4.2.1 MemoTbl 上のエントリ

まず，MemoTbl の 1 ラインサイズを提案するプロセッサのキャッシュラインサイズに合わせて 64Byte とした．これにより，MemoTbl 上にキャッシュ1 ラインが乗り，キャッシュと MemoTbl の入力一致比較において 1 ライン同士を比較することができる．また，MemoTbl の 1 ラインには，4Byte サイズのエントリなら 16 個格納できる．そこで，引数レジスタを登録する場合では，1 エントリ目に PC を登録し，2 エントリ目にはメモ化対象関数の情報が登録されるが，今回は特に使用しない．3〜6 エントリ目に引数渡しのためのレジスタ R0〜R3 を登録する．引数レジスタ登録ラインの 7 エントリ以降は，空のままであるため CAM の使用効率の低下を招く可能性があるので，今後使用効率を高める登録方法に改善していく必要がある．

4.2.2 キャッシュアクセスの競合

本提案手法では，パイプラインの命令実行と並行して MemoTbl 検索を動作させるため，同サイクル中にパイプライン中の load 命令によるキャッシュアクセスと MemoTbl 検索中のキャッシュアクセスの競合が起る可能性がある．キャッシュへのアクセスは，キャッシュコントローラが管理している．たとえば，load 命令がキャッシュを占有して

いる場合には，MemoTbl からのキャッシュアクセスを待たせるよう管理している．しかし，同サイクル中に，load 命令，もしくは store 命令からの要求と MemoTbl からの要求が同時にキャッシュへアクセスした場合，キャッシュメモリのポート数が足らないことから同時アクセスはできない．そのため，どちらかを優先してアクセスさせる他ない．今回は，load，store 命令からの要求を優先させることを考えた．これは，検索途中の関数が再利用できるかはわからないのに，関数を検出するごとに，毎回 MemoTbl からの要求を優先して行うのは，本来のパイプライン上の命令実行を著しく妨げる可能性があるためである．しかし，これはキャッシュアクセスのタイミングの問題なので，プログラムごとに全く異なる結果が得られることが予想される．よって，実際に評価してみないと分からないことであるため，今後の課題として評価して行くこととする．

4.2.3　ソースレジスタ番号の伝搬

本プロセッサでメモ化を行う場合には，パイプラインステージの後段までソースレジスタ番号を伝搬させる必要がある．これは，リタイアステージにおいて MemoBuf へ入力値登録を行うとき，ソースレジスタ番号が必要だからである．なぜ，ソースレジスタ番号が途中で失われてしまうかという理由は，本プロセッサが命令分解機構を取り入れており，途中で元のソースレジスタ番号が必要なくなるからである．命令分解すると，汎用レジスタと拡張レジスタを使う命令列に変換される．その命令が，マップステージにおいて，命令ウインドウも兼ねる物理レジスタのリオーダバッファにマッピングされるため，そこからのパイプラインレジスタには，もとのソースレジスタ番号が存在しない．そこで，パイプラインレジスタとリオーダバッファを拡張し，元のソースレジスタ番号をリタイアステージまで伝搬させた．さらに，命令実行ステージ以降では，値自体も伝播させる．なお，このソースレジスタ番号およびその値は，メモ化のための入出力登録時にのみ使用される．

4.2.4　リオーダバッファの拡張

まず，ソースレジスタ番号を保持するために，リオーダバッファへの拡張も行った．さらに，分解後の命令の最後にあたる命令に，終端を示すフラグをつけるため，リオーダバッファを拡張した．この拡張が必要な理由としては，本プロセッサでは ARM における関数を呼び出す時に使う命令，BL(Branch and Link) 命令も分解されることがある．分解が行われる場合には，BL 命令が条件付き実行命令であるときがある．この場合，分岐命令と条件コードを処理する命令として分解される．本来，メモ化対象関数の入出力を登録するのは，関数に飛んだ先の命令からである．そのため，BL 命令を検

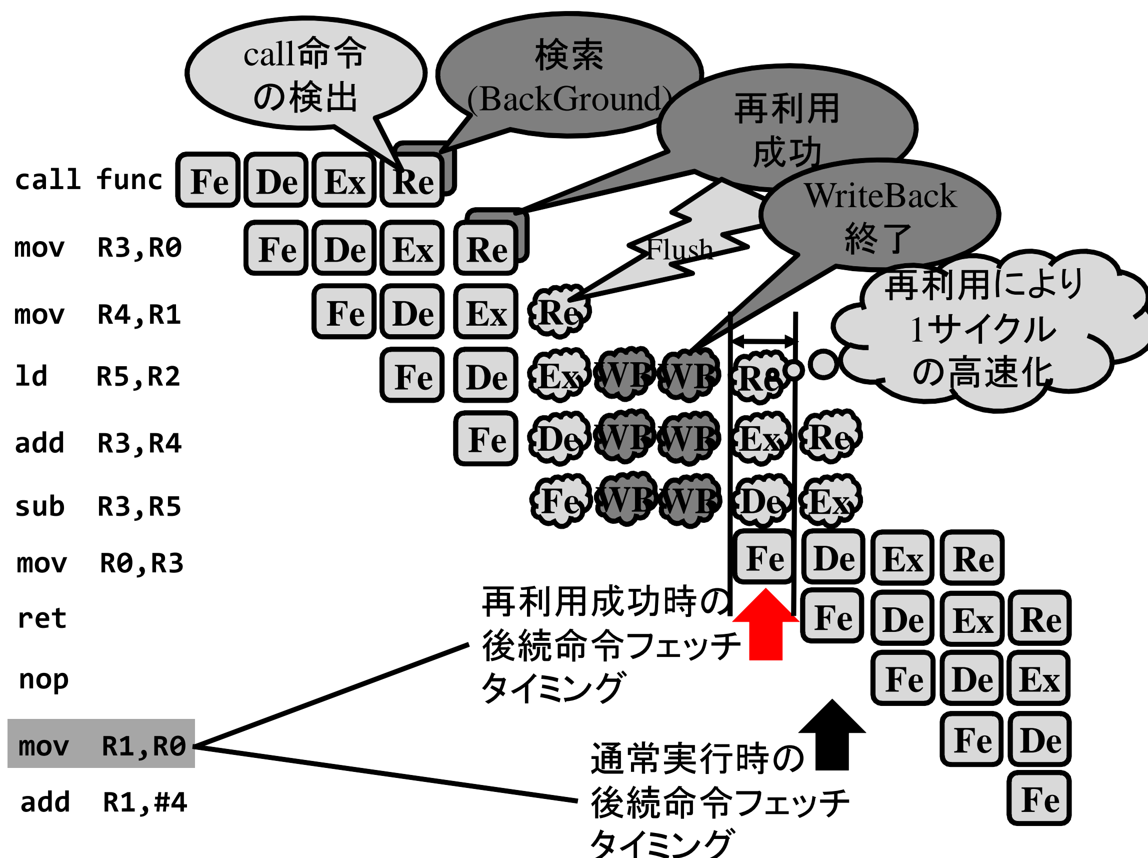

図 18: ライトバックを含むパイプライン動作

出した段階で，MemoBuf への入出力登録を開始してはいけない．BL が分解されていた場合，まだ処理するべき命令が残っているからである．そのため，関数に飛ぶまでに処理すべき命令を全て処理し終わったことを保証する必要がある．そのため，分解命令の終端を管理するフラグをリオーダバッファに設け管理する．なお，out-of-order 実行によりパイプライン途中では命令の順序が入れ替わるが，命令分解時では終端命令が分かっていることや，リタイアステージにおいては命令の処理順序がもとの命令列と同じであることが保証されているため，正しく動作することが保証できる．

4.3 高速化

本節では，図 17 のモデルを元に，高速化の余地を探り．高速化案を提案する．

4.3.1 ライトバックと後続命令フェッチのオーバラップ

今回注目したのは，再利用が成功したときのライトバック動作である．図 18 は，ライトバックにかかるサイクル数を追加したパイプライン動作図である．図 18 では，ま

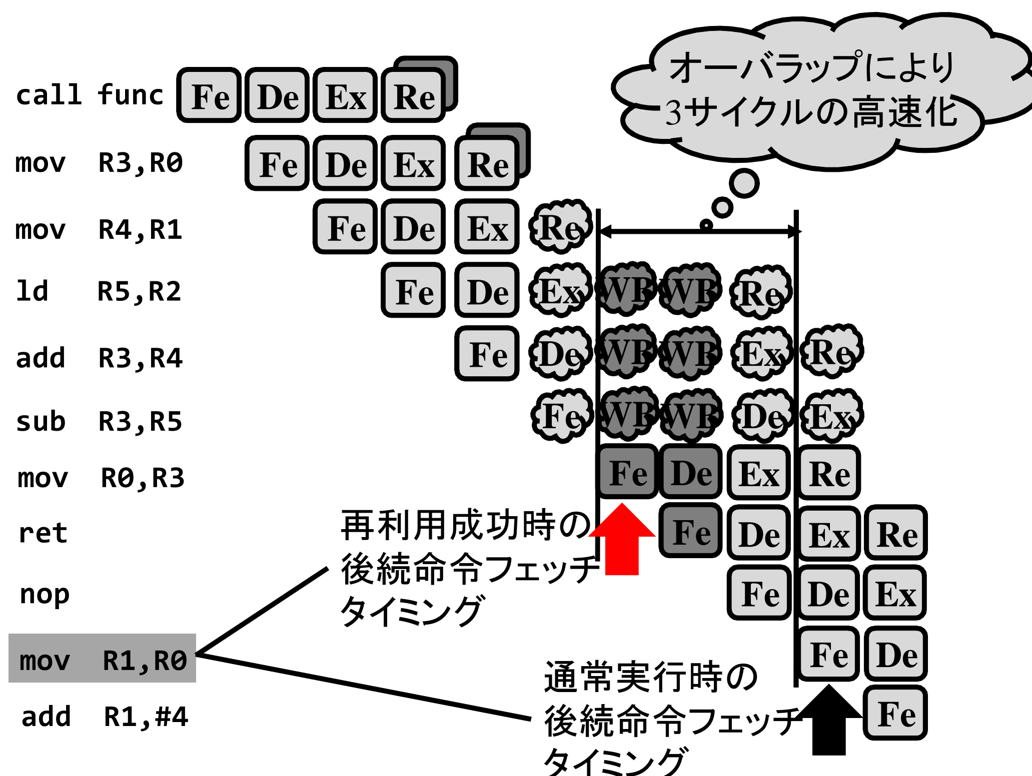

図 19: ライトバックと後続命令フェッチのオーバラップ時のパイプライン動作

ず関数呼び出し call func をリタイアステージで検出するとすぐに検索を開始する．2 サイクル目で再利用ができることが判明し，次の 1 サイクルでパイプラインフラッシュを行った後，関数の結果をキャッシュとレジスタにライトバックする．そしてライトバック完了を確認してから後続命令をフェッチしている．これが普通に考えられる実装である．しかし，この例のようにせっかく再利用が成功してもパイプラインフラッシュやライトバックをしなければならず，再利用が成功した場合の後続命令フェッチ位置と再利用をしない場合である通常実行時の後続命令フェッチ位置を比べると，1 サイクルにとどまってしまっている．これは，1 サイクルの高速化を意味しているが，これではせっかく関数の再利用ができても得られる効果がわずかになってしまっている．これは，非常に残念なことであり，改善が必要である．ここで，ライトバックの処理に注目する．ライトバック処理は，再利用できた関数の結果をレジスタとキャッシュに書き戻す処理のみを行っている．つまり，ライトバックが占有しているのは，キャッシュとレジスタのみであるため，それ以外のユニットは遊休状態であると言える．そこでこの遊休状態のユニットを有効利用したい．しかし，現在，パイプラインをフラッシュしているため，パイプライン上には命令は存在せず，パイプラインは空のままで

ある．そこで，ライトバック開始時，正確にはパイプラインフラッシュ終了時に，同時に後続命令をフェッチするように実装モデルの変更を行った．図19に，先ほどの図18と同じプログラムで，ライトバックと後続命令フェッチをオーバラップしたときのパイプラインの様子を示す．今回の例では，オーバラップを行ったことで，オーバラップを行わなかった場合に比べて2サイクル，合計で3サイクルの向上を図れていることがわかる．この例では2サイクルだが，さらにライトバックに多くの時間がかかる場合には，さらなる高速化が望めると考えられる．

では，理想的にはどこまで高速化が見込めるのだろうか．ライトバックは，論理レジスタとキャッシュへ出力を書き込む処理である．まず，考えられるのは，後続load，store命令がライトバックのキャッシュアクセスを追い越してしまうことである．この場合，キャッシュアクセス順序がくずれ結果が異なってしまう．そこでload，store命令はOP1ステージでとめる必要がある．次に，ライトバックが論理レジスタに書き込む処理を後続命令が追い越してしまう場合である．ほとんどの命令が論理レジスタアクセスを伴うものである．それらの命令が論理レジスタにアクセスするのは，sel/rdステージなので，load，store命令以外がsel/rdステージまできた場合には，そのステージで処理を止める必要がある．以上のことを考慮すると，得られる理想の性能は，関数の最初にload命令がきたときであり，7ステージ（IA，IF，ARM-D，HOST-D，MAP/SCH，SEL/RD，OP1）での命令実行のオーバラップが可能であり，7サイクルの成功向上が得られると予想される．これは，理想性能だが，これらのオーバヘッドが毎回削減できれば，積み重なるとかなりの効果が得られるのではないかと考えられる．

4.3.2　キャッシュコントローラ内の先行要求削除によるライトバックの高速化

再利用成功が判明した後の手順として，まずパイプラインのフラッシュを行い，その後ライトバックが完了し，後続命令をフェッチすることで再利用のための処理を完了する．しかし，このライトバック処理には時間がかかる場合があるため後続命令フェッチが遅れる場合があり，例を以下に示す．

- ライトバック時にキャッシュへの書き込みミスが起る場合
- ストアバッファに先行ストア命令の要求がまだ残っている場合
- ライトバック時において，既にキャッシュコントローラ内に先行ロード・ストア命令の要求が入っている場合

以上の中には，高速化の余地が含まれている．まず，1つ目のキャッシュへの書き込みミスが起こる場合であるが，これは，キャッシュの問題なので改善することはできない．次に，2つ目のライトバッファに先行ストア要求が入っている場合であるが，これ

は，先行ストア要求より先にライトバックによるキャッシュ書き込みをしてしまうと，メモリ書き込みの順序が崩れるため，同一ラインへの書き込みが発生する場合に動作が異なってしまうため，高速化は望めない．最後に，3 つ目の場合であるが，ライトバック時，キャッシュコントローラ内に既に先行要求が格納されていた場合，本来なら，全ての先行要求を処理した後に，ライトバックのキャッシュ書き込み要求を処理するのが普通である．しかし，このキャッシュコントローラ内要求の中には，削除しても問題ない要求が含まれている．この要求を削除することができれば，ライトバックまでの時間を少しでも短縮することができるはずである．

まず，ライトバック時に，キャッシュコントローラ内に入っている可能性のある要求としては，大きく Read 要求と，Write 要求に分けられる．Read 要求には発行元が記録されており，IF ステージからの命令フェッチによる Read 要求 (**IF-Read-Request**)，OP1 ステージからの load 命令を処理するための Read 要求 (**OP1-Read-Request**)，MemoTbl からの入力一致比較処理のための Read 要求 (**Memo-Read-Request**) であるかが判断できる．IF-Read-Request は，キャッシュから命令をフェッチする処理であり，フェッチが完了されるまで，パイプラインはストールする．さらに、IF-Read-Request は in-order で処理される。そのため，再利用成功時においてキャッシュコントローラに存在する要求は，再利用対象の関数内命令であるか関数実行後の後続命令であることが保証される．つまり，削除したとしても再利用後の処理には一切影響がない．OP1-Read-Request も同様の理由で削除可能である．最後に，Memo-Read-Request であるが，これはそもそも，再利用が成功した時点（一致比較が終了した時点）でキャッシュコントローラ内に要求が存在していることはない．つまり，キャッシュコントローラ内の Read 要求に関しては，全て削除可能である．

次に，Write 要求について述べる．Write 要求も同様に発行元による判断が可能であり，その種類にはストアバッファ(STbuf) からの Write 要求 (**STb-Write-Request**)，MemoTbl からのライトバックのための要求 (**Memo-Write-Request**) が存在する．また，この他にもキャッシュ書き込みミス時に，対象キャッシュラインが dirty だった場合に発生する，キャッシュライン追い出し要求もあるが，この要求に関しては，キャッシュミス時に対象ラインが dirty かどうかを判断し要求を出すため，要求を削除してもキャッシュライン上の情報が残ってさえいれば問題ない．まず，Memo-Write-Request は，ライトバック自身の要求なのでこれは消してはならない．次に，STb-Write-Request であるが，この要求は削除できない．この要求はもともと，OP1 ステージで処理する store 命令によるものである．OP1 ステージでは，store 命令を処理する場合，まず STbuf が

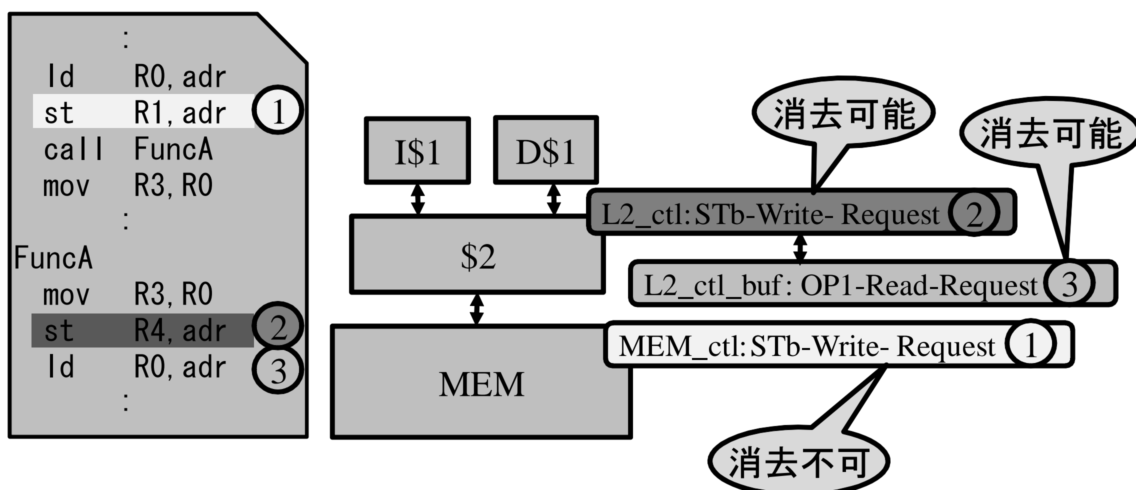

図 20: キャッシュコントローラの削除例

いっぱいでないかを確認し，STbuf に空きがあれば，store 命令を STbuf へ投入することで処理を終了する．つまり，STbuf がいっぱいであればパイプラインはストールするが，そうでない場合は，実際にはキャッシュへの値の書き込みが行われていないにもかかわらず，パイプラインは流れ，store 命令はリタイアされる．つまり，再利用が成功した時点でも，まだ関数が呼び出される以前のストア命令が処理されないで残っている可能性が考えられる．図 20 は，その様子を表している．図の例では，関数を呼ぶ call FuncA 実行以前に ld 命令と st 命令を実行している．さらに，FuncA 内でも st 命令，ld 命令を順番に実行しているときの再利用成功時のキャッシュコントローラ内の様子を示している．再利用が成功した場合，省略される FuncA の関数内の命令は実行する必要がないため削除可能である．具体的には図中の 2，3 の命令である．しかし，1 の命令は，関数外の命令で実行しなければプログラムの動作に影響を及ぼしてしまうため，削除してはならない．よって，単純に Write 要求を消去することはできない．厳密には，関数内の store 命令による STb-Write-Request は消去することができるが，そのためには，要求を出した命令がいったいどの関数内の命令なのかを記憶しておく必要があり，キャッシュコントローラおよび STbuf の拡張が必要となる．

以上のことをまとめると，厳密に削除してはならない要求は，STb-Write-Request 中のメモ化対象関数の呼び出し以前に処理されるべき要求のみということが分かった．しかし，これを実現するためにはキャッシュコントローラの拡張などが必要であることから，今回は，既存の機構を用いて実装できる Read のみを削除する機構を実装することとした．

```
int b=2; //大域変数
int recursive_func(int a){
  if(a > 0)
    return(b + a * recursive_func(a-1));          … (4)
  else
    return(1);
}

int main(void){
  int x,y,z;
  x = recursive_func(5); //2+(5*4*3*2*1)          … (1)
  y = recursive_func(6); //2+(6*5*4*3*2*1)        … (2)
  z = recursive_func(5); //2+(5*4*3*2*1)          … (3)
  printf("%d,%d,%d\n",x,y,z);
  return(0);
}
```

図 21: 再帰関数

5　評価

本章では，今まで述べてきた提案手法のプロセッサモデル（高速化手法を除く）をシミュレータに実装し，テストプログラムおよびベンチマークプログラムを用いて評価をおこない，その結果について考察する．

5.1　評価環境

以下に評価に用いた特徴的な関数を持つプログラム例を示す．図 21 は，再帰関数を示している．図の例では関数 recursive_func が引数 1 つを取る．たとえば，図 21 において関数 recursive_func が引数 5 を取る場合，関数 recursive_func 側では，1〜5 までの積をもとめ，さらに大域変数との和を求めたものが，結果となってかえってくる様子を表している．このプログラム例をもとに多重再利用について次の項で考察する．

次に，図 22 に引数の多い関数を持つプログラム例を示す．このプログラムは，13 個

```
int arg_much(int a, int b, int c, int d, int e, int f,
         int g, int h, int i, int j, int k, int l, int m){
  retrun(a * b + c + d + e + f + g + h + i + j + k + l + m);
}

int main(void){
  /*1〜13までの和*/
  printf("%d\n",arg_much(1,2,3,4,5,6,7,8,9,10,11,12,13));
  /*2〜14までの和*/
  printf("%d\n",arg_much(2,3,4,5,6,7,8,9,10,11,12,13,14));
  /*1〜13までの和*/
  printf("%d\n",arg_much(1,2,3,4,5,6,7,8,9,10,11,12,13));
  return(0);
}
```

図 22: 引数の多い関数

の引数を取り，各引数の総和を計算するプログラムである．従来の自動メモ化プロセッサでは，明示的引数の数を制限していたため 6 個以上の引数を持つ関数を再利用することはできない．しかし，今回の実装モデルでは，引数レジスタを用いる明示的引数と大域変数である入力を除き，Memobuf および MemoTbl への入力の登録方法を SP 相対形式に変更したことにより，理論上引数の数に制限を受けずに再利用が可能であるはずである．また，評価には N-Queen 問題を解くベンチマークプログラムも用いた．

一方で，シミュレータのパラメータについて表 2 に示す．なお，入力一致比較時の MemoTbl とキャッシュまたは，レジスタとの比較コストは，それぞれ 1cycle，2cycle として見積もっている．

また，今回評価に用いたプログラムは，コンパイルオプション（arm-elf-gcc-4.1.1 -msoft-float -mach=armv4 -O2）をもちいた．このオプションは，整数除算や浮動小数点演算命令を生成せずに，代わりに演算ライブラリを呼び出すオプションである．これを用いた理由は，現在の ARM プロセッサの多くが浮動小数点演算などを演算ライブラリでおこなっているためである．

表 2: シミュレーション時のパラメータ

I1 Cache 容量	16KBytes
I1 Cache ミスペナルティ	8cycles
I1 ラインサイズ	64Bytes
I1 ウェイ数	4
D1 Cache 容量	32KBytes
D1 Cache ミスペナルティ	8cycles
D1 ラインサイズ	64Bytes
D1 ウェイ数	4
共有 2 次 Cache 容量	2MBytes(direct-map)
共有 2 次 Cache ミスペナルティ	40cycles
共有 2 次 Cache ラインサイズ	64Bytes
共有 2 次 Cache ウェイ数	4
分岐予測方式	gshare
RET-AS	8entry
ストアバッファ	8entry
ROB	32entry
CAM Size	4kline
CAM Line Size	64B

5.2 結果

前項で示した，再帰関数を含んだプログラムと引数の多い関数を含んだプログラムの評価結果を図 23 に示し，N-Queen 問題を解くベンチマークプログラムの評価結果を図 24 に示す.

まず，図 23 であるが，縦軸をサイクル数，横軸をプログラム例の名前を示している．図 24 も同様である．また，1 番左のグラフから提案するプロセッサにおいて，メモ化を行わない場合のサイクル数を示しており，左から 2 番目のグラフが，メモ化を行うと共にに検索中はパイプラインの動作を止めるモデル，3 番目のグラフがメモ化を行うと共に検索動作とパイプラインの動作を平行して行う提案プロセッサモデルを示している．ここで，左から 2 番目に示したグラフである，検索中にパイプライン動作を止めるモデルは，従来の単命令発行の自動メモ化プロセッサで用いていたモデルであ

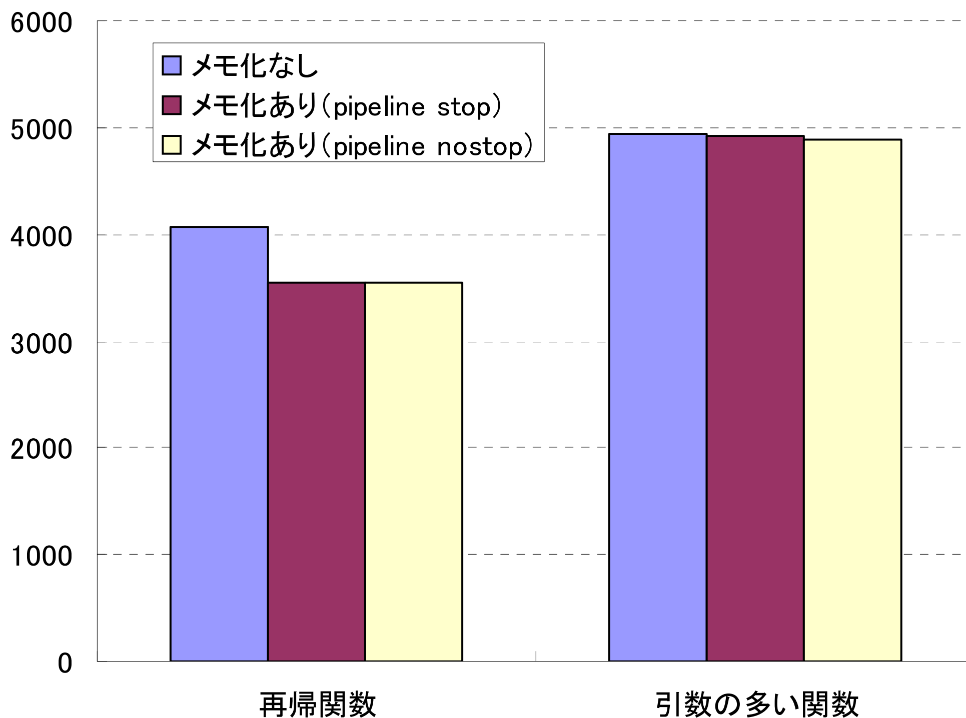

図 23: 結果 1

る．この 2 番目の手法では，再利用失敗時のオーバヘッドが必ず発生してしまう欠点がある．さらに，再利用が成功したとしても，削減されるサイクル数が入力一致比較コストおよび出力のライトバックコストの合計よりも少ない場合，速度低下につながることがあると考えられる．

5.3 考察

図 23 は，再帰関数を含んだプログラム例の実行結果を示している．図 23 では，再帰関数である，recursive_func が再利用可能なメモ化対象関数である．関数 recursive_func の入力となるのは，関数に渡される引数である．図中の（1）では 5,（2）では 6 である．そのほかに関数内から参照される大域変数も入力である．また，再帰関数の場合，多重再利用が適用される．多重再利用を実現するために，MemoBuf は階層構造をとっている．今回の提案プロセッサでは，6 階層の MemoBuf を用意している．MemoBuf がいっぱいであるときには，登録されている関数のうち，いちばん上位にあたる親関数を削除してから新しい関数の登録を始める．つまり，関数が 6 つ MemoBuf に入ると思われるかも知れないが，実際は，再利用が成功したときの為に，MemoTbl 上の入力

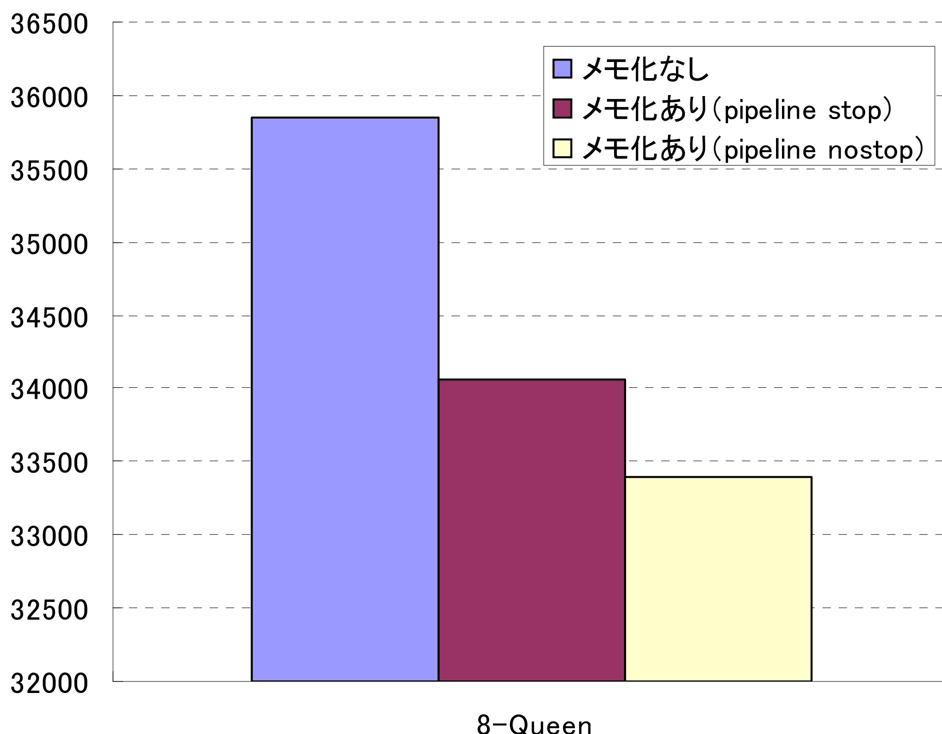

図 24: 結果 2

を格納しておくスペースが 1 つ必要であるため，5 つの関数が登録可能である．そのため，このプログラム例では，再帰の深い方から数えて 5 つが再利用可能となる．今回の例で，メモ化あり（pipeline_stop）とメモ化あり（pipeline_nostop）とで，差がでなかったのは図 23 の（3）で再利用が成功するとき，再帰一回目で検索が成功するため，検索コストが少ないため差がでなかったと考えられる．

次に図 23 中の引数の多い関数であるが，メモ化あり（pipeline_nostop）の場合において高速化が図れていることがわかる．この例の特徴は，引数が 13 個と多いことである．前項でも述べたが，これは SP 相対形式で入力を登録しているためできることである．そのため，引数の多い関数に対応できることがこの結果より分かる．

最後に 24 であるが，これは 8-Queen 問題を解くベンチマークプログラムである．プログラム的には，Queen をおける場所を再帰的に探がしていくプログラムである．この関数において再利用の効果がある関数は，引数の数が 6 つであるので，SP 相対形式で入力登録することで対応できているため，高速化が図れているといえる．

6　まとめ

本稿では，計算再利用に基づいて研究されてきた自動メモ化プロセッサを，より一般的な環境であるスーパススカラプロセッサでも実現することを目的に，対象プロセッサ（ARM）に適した自動メモ化機構のモデル化，および実装と評価を述べてきた．そして，本評価において，提案モデルの実現性，有効性を示すことができた．よって，out-of-order 実行のスーパススカラプロセッサにおいても自動メモ化手法を適用することできることがわかり，自動メモ化プロセッサの実現性がより高まったといえる．しかし，提案手法では，パイプランでの命令実行と入力一致比較を同時に行うため，キャッシュの競合によるわずかばかりの性能低下も見られた．そのため，今後キャッシュアクセスに関する詳しい評価も行っていきたい．

一方で，更なる高速化案についても述べ，検討を行った．今回は 2 つの改良の余地を述べたにとどまったが，今後は更なる詳しい評価をおこなうと共に更なるオーバヘッド削減の余地を考えていきたい．具体的な課題としては，MemoTbl を検索し始めるタイミングをより早いステージで行うことである．現在のモデルでは，検索時に比較するための現在の入力（レジスタやキャッシュ上の値）が関数の call 命令を検出した時点で確定していることが保証されなければならないため，本来の命令列の順番が保証されるリタイアステージで，call 命令の検出および MemoTbl の検索を行っていた．しかし，call 命令はデコードした段階で検出可能であるため，デコードステージで検索を投機的に始めることができないか検討中である．

謝辞

本研究の為に多大な御尽力をいただき，日頃から熱心な御指導を賜った名古屋工業大学の津邑公暁准教授，奈良先端科学技術大学院大学の中島康彦教授に深く感謝いたします．

また，本研究のために多大な御協力をいただいた名古屋工業大学の高木伴彰さんに深く感謝致します．

さらに，本研究に対しての熱心な御協力をいただきました，名古屋工業大学の松尾啓志教授に深く感謝致します．

加えて，たびたびご検討・助言をしていただいた同大学の斎藤彰一准教授や松井俊浩助教授にも深く感謝致します．最後に，本研究の際に多くの助言・協力をしていただいた松尾・津邑研究室ならびに斎藤研究室のみなさまに対しても深く感謝致します．

参考文献

[1] Sodani, A. and Sohi, G. S.: Dynamic Instruction Reuse, *Proc. 24th International Symposium on Computer Architecture*, pp. 194–205 (1997).

[2] Yang, J. and Gupta, R.: Load Redundancy Removal through Instruction Reuse, *International Conference on Parallel Processing*, pp. 61–68 (2000).

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

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

[5] Nakada, T., Nakashima, Y., Shimada, H., Kise, K. and Kitamura, T.: *OROCHI: A Multiple Instruction Set SMT Processor* (2008).

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