修士論文

再利用オーバヘッド削減スレッドによる自動メモ化プロセッサの高速化手法

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
修士課程創成シミュレーション工学専攻
平成 20 年度入学　20413520 番

神谷 優志

平成 22 年 2 月 4 日

再利用オーバヘッド削減スレッドによる
自動メモ化プロセッサの高速化手法

神谷 優志

内容梗概

ゲート遅延に対する配線遅延の相対的な増大に伴う消費電力や発熱量の増大から，マイクロプロセッサの動作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レベル並列性（ILP）に基づく高速化手法が注目されてきた．しかし，多くのプログラムは明示的な ILP を持たないことや，ILP を抽出できる場合でもメモリスループットなどの資源的制約があることから，これらの手法にも限界がある．そこで，既存のバイナリを変更することなく，計算再利用をハードウェアにより動的に行う自動メモ化プロセッサに関する研究が行われている．計算再利用とは，ある命令区間の入力に対応する出力を表に記憶しておき，次回に同じ命令区間を過去と同一の入力で実行する場合は，過去に記憶しておいた出力を読み出して，命令区間の実行を省略することにより，高速化を図る手法である．さらに，この自動メモ化プロセッサに並列事前実行という投機的手法を用いることで，複数のコアを有効活用する手法がこれまでに研究されてきた．

さて，この並列事前実行は，割り当てられるコア数が少ない時には有効であるが，割り当てられるコア数が多くなるにつれ，コア数を増やしたことによる効果を発揮しにくくなる．そこで本研究では，複数のコアを有効活用するための新たな手法を提案する．自動メモ化プロセッサが，計算再利用を適用する際に発生するコストを再利用オーバヘッドと呼ぶ．本研究の目標は，再利用オーバヘッドを複数コアを用いて隠蔽することにより，自動メモ化プロセッサのさらなる高速化を図ることである．そのために 2 つのスレッドを用いる．一つは命令区間の入力が表に記憶した値と部分的に一致した場合に，全ての入力が一致したものとして，投機的に後続区間の実行を行うスレッドである．もう一つは表に記憶した値と入力の比較を行わずに検索対象区間の実行を行うスレッドを用いて再利用オーバヘッドを削減するスレッドである．これらのスレッドを各コアに割り当てることで，従来有効活用されていなかったコアを有効利用できる．また，並列事前実行と提案手法を組み合わせて用い，命令区間毎に適用するスレッドを選択することで，更なる高速化を図る．

提案手法の有効性を検証するため，従来の自動メモ化プロセッサに提案モデルを実装し，シミュレーションによる評価を行った．その結果，Stanford ベンチマークでは

従来手法で最大 44.2%のサイクル数を削減，平均 11.2%サイクル数が増加していたが，提案手法では，最大 47.2%，平均 6.2%のサイクル数を削減できた．また SPEC CPU95 ベンチマークでは従来手法で最大 13.9%のサイクル数を削減，平均 0.1%サイクル数が増加していたが，提案手法では最大 41.5%，平均 11.1%のサイクル数を削減できた．

再利用オーバヘッド削減スレッドによる
自動メモ化プロセッサの高速化手法

目次

1 はじめに

ゲート遅延が支配的であった 2000 年代初頭までは，配線プロセスの微細化による高クロック化により，マイクロプロセッサの高速化を実現できた．しかし数年程前からは，ゲート遅延に対する配線遅延の相対的な増大に伴う消費電力や発熱量の増大から，マイクロプロセッサの動作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レベル並列性（ILP: Instruction Level Parallelism）に基づく高速化手法が注目されてきた．しかし，多くのプログラムは明示的な ILP を持たないことや，ILP を抽出できる場合でもメモリスループットなどの資源的制約があることから，これらの手法にも限界がある．

一方現在では，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向上を可能にするため，1 つの CPU に複数のコアを搭載したマルチコア CPU が広く普及している．一般の PC やワークステーションに用いられる汎用マルチコア CPU の例として，Intel 社 Core i7[1]，IBM 社 Cell/B.E.[2]，Sun Microsystems 社 UltraSPARC T2[3] 等が挙げられる．またネットワーク機器等で使用することを想定し，1 つのプロセッサに 64 個のコアを搭載した TILE64[4] などのメニーコア CPU も登場している．また CPU や GPU（Graphics Processing Unit）のみならず，DSP（Digital Signal Processor）等の組み込み向けプロセッサにまでマルチコア技術が幅広く普及してきた．今後は半導体のプロセスルールの縮小に伴い，単一プロセッサ当たりに搭載されるコア数が更に増加していくと考えられる．このようにプロセッサのマルチコア化が進んでいる現在，複数のコアを利用したプログラム当たりの高速化手法を検討する必要がある．

そこで，並列化されていないプログラムを複数のコアを用いて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）に着目してプログラムを複数のスレッドに割り当てられるよう分割し，それぞれのコアに割り当てる技術が研究されている．例えば並列化ライブラリを用いてプログラマが明示的に並列処理プログラムを記述したり，自動並列化コンパイラ [5, 6] を用いてコンパイラにより自動的に複数のコアにプログラムを割り当てる事ができる．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も存在し，ユーザー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．また，今後プロセッサ当たりのコア数の増加に伴い，既存の手法を利用しても，一部のコアに処理を割り当てられないという状況が発生する事も想定される．

一方で，プログラムの並列化による高速化手法とは別の概念として，これまでに計

算再利用と呼ばれる従来とは着眼点の異なる高速化手法を用いた自動メモ化プロセッサに関する研究が行われている [7, 8]．**メモ化（Memoization）**[9] とは，関数等の命令区間を計算再利用可能な形に変換する処理であり，命令の実行時に入出力セットの記憶および過去の入力セットとの比較を行うことで，同一入力による当該関数の再計算を省略し実行を高速化する．メモ化技術を用いた高速化に関する研究はソフトウェア分野では広く行われている [10, 11]．例えば，関数型言語 Haskell を拡張し，関数にメモ化を適用可能にした言語が提案されている [12]．しかし，ユーザーがメモ化する関数を明示的に指定する必要があり，プログラマの負担が大きい上，プログラムをコンパイルし直す必要があるため，ソースコードが提供されていないプログラムを高速化することはできない．さらに，特定のプログラミング言語による記述をユーザーに強制とするという問題もある．また，ソフトウェアによるメモ化はオーバヘッドが大きく，メモ化が適用可能なプログラムでも，高速化が可能なプログラムは科学技術計算など，特定のプログラムに限定される傾向がある．

これに対し本研究で扱う自動メモ化プロセッサは，既存のバイナリを変更することなく，ハードウェアを用いて動的に関数やループ等の命令区間を検出し入出力の記憶・比較を行うことで，命令区間に対してメモ化を自動的に適用する．さらに，マルチコアが一般化した事を踏まえ，複数のコアを有効に利用する方法として，並列事前実行という高速化手法が提案されている．

さて，命令区間によっては計算再利用を適用する際に非常に大きなオーバヘッドの発生する命令区間も存在することがこれまでの研究から明らかになっている．また，単一プロセッサ当たりに搭載されるコア数が増加していることから，複数のコアを有効に利用する必要もある．そこで，本論文では複数のコアを利用して，命令区間をメモ化する際に発生するオーバヘッドを削減する手法を提案する．これにより複数のコアを備えた自動メモ化プロセッサを設計する場合，従来活用されていなかったコアを有効利用して，高速化に寄与することが可能となる．さらに，本提案手法を従来の並列事前実行と組み合わせ，従来の自動メモ化プロセッサと比較して，更なる高速化を実現する手法を提案する．

以下 2 章では既存の自動メモ化プロセッサの構成と複数のスレッドを用いた高速化手法である並列事前実行について述べる．3 章では既存の自動メモ化プロセッサの問題点を挙げ，その解決案として複数のスレッドを用いて関数やループ等の命令区間をメモ化する際に発生するオーバヘッドを削減する手法について述べる．4 章では提案手法を実現するためのハードウェア構成について述べる．5 章で提案手法について評

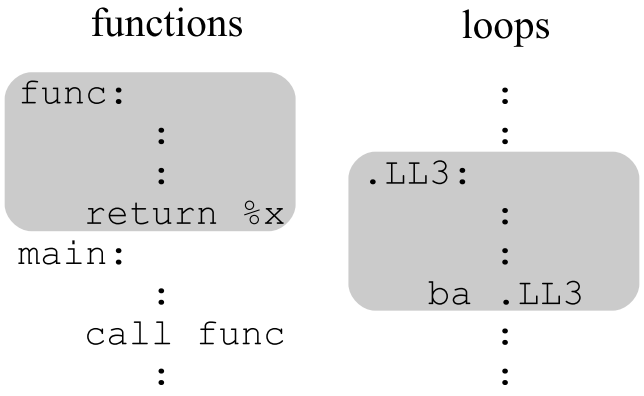

図 1: メモ化対象区間

価を行い，最後に 6 章で結論をまとめる．

2　自動メモ化プロセッサ

2.1　メモ化と自動メモ化プロセッサ

ソフトウェアの分野ではメモ化という高速化手法が知られている．メモ化とは，関数等の命令区間を**計算再利用**可能な形に変換する処理であり，命令の実行時にその入出力セットの記憶および過去の入力セットとの比較を行うことで，同一入力による当該命令区間の再計算を省略し，実行を高速化することができる．また，計算再利用とは，過去に実行した命令区間の入出力セットを記憶し，再度当該命令区間が過去と同じ入力で実行される時に，その実行を省略する手法である．具体的には，ある命令区間に対し，当該命令区間を実行中に出現した入出力セットを表に記憶する．再度同じ入力により当該区間を実行する場合には，予め表に記憶しておいた出力を利用し，命令区間の実行を省略する．

本論文で扱う**自動メモ化プロセッサ**は，過去に実行した関数やループ区間の入出力を表に記憶し，再度当該関数やループ区間を過去と同じ入力で実行する時に，その実行を省略することで高速化を図るプロセッサである．**図 1** には自動メモ化プロセッサがメモ化対象とする，関数およびループ区間の例をアセンブリプログラムで示す．関数は call 命令によるジャンプを行った直後の関数ラベルから，当該関数の return 命令までである．また図 1 のアセンブリプログラムでは，ラベル func から return 命令までが一つの関数である。一方，ループ区間は後方分岐命令と分岐先ラベルで挟まれた区間であり，図 1 のアセンブリプログラムでは分岐先ラベル.LL3 から分岐命令 ba までに対応する．メモ化対象区間が関数の場合，call 命令の検出から return 命令を検出す

```
int g, h;
int main(){
  g = 4, h = 2;
  int a = A(3);
}
int A(int x){
  return(B(x*2)+6);
}

int B(int y){
  int b = y;
  b += g;
  if(y > 3)
    while(g < 8){
      b += h;
      g++;
    }
  return b;
}
```

Loop C

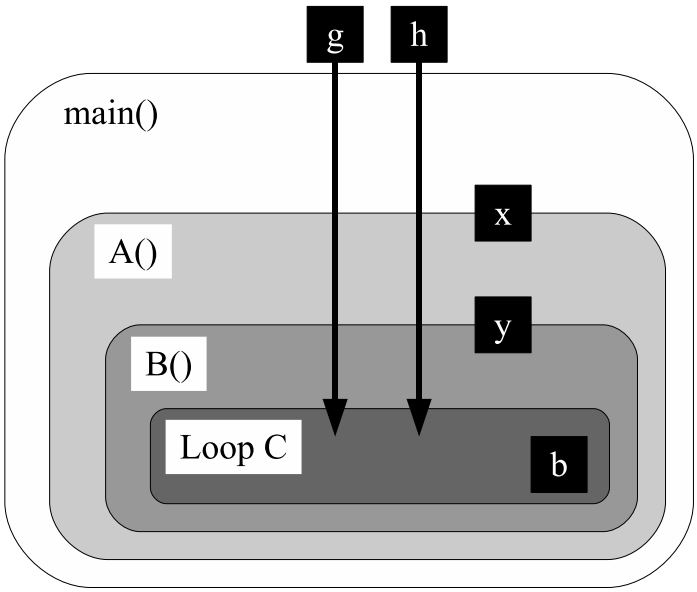

図 2: 各区間の入力セットと入力の影響範囲

るまでに出現した入出力セットを表に記憶する．一方，メモ化対象区間がループ区間の場合，後方分岐命令によりジャンプした直後の命令から再び後方分岐命令を検出するまでに出現した入出力セットを表に記憶する．ただし後方分岐命令は必ずしもループを構成するわけではないため，再び同一の後方分岐命令を検出した際にループ区間を構成すると判断する．そのため，1 回目のループイタレーションは，それをループ区間として認識することができない．

続いて，自動メモ化プロセッサがメモ化対象とする命令区間の入出力セットについて説明する．関数及びループ区間の入力は，関数の引数および関数内で参照される大域変数であり，出力は関数の戻り値および関数内で書き換えられる大域変数である．さらにループ区間の場合，1 回のループイタレーションの実行中に出現する大域変数に加え，そのループ区間を含む関数の局所変数もループの入出力として扱う．一方で関数の場合，関数内で扱う局所変数はその呼び出し元に影響を与えないため，関数の入出力に含めない．したがって関数に対しては，局所変数とそれ以外の変数を区別する必要がある．そこで自動メモ化プロセッサは，一般に OS がデータサイズおよびスタックサイズの上限を決定することを利用し，この上限および関数呼出が行われる直前のスタックポインタの値と変数アドレスとの関係から，局所変数とそれ以外の変数を判

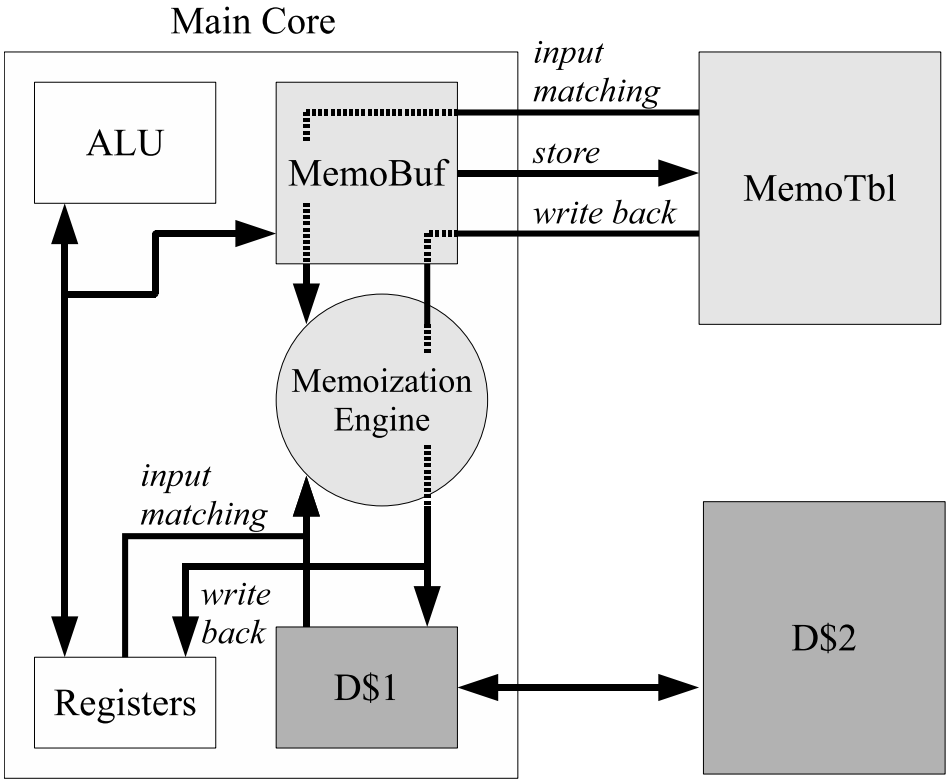

図 3: 自動メモ化プロセッサの構造

別する．**図 2** の左側に main 関数から関数 A を呼び出し，更に関数 A からループ C を含む関数 B を呼び出すプログラムの例を示す．また図 2 の右側に，当該プログラムに含まれる各関数及びループの入力を示す．図 2 の例でループ C の入力は，当該ループ内で参照される大域変数 g 及び h と，関数 B の局所変数 b である．また関数 B の入力はその引数 y と，ループ C 内で参照される大域変数 g 及び h である．さらに，大域変数 g 及び h は，関数 A の内部で呼び出される関数 B が参照しているため，関数 A の入力でもある．

2.2 自動メモ化プロセッサの構成と動作モデル

自動メモ化プロセッサの構成を**図 3** に示す．コアの内部には一般的なプロセッサコアが持つ ALU，レジスタ，1 次データキャッシュといった構造の他に，MemoTbl へ書き込む入出力セットを一時的に蓄えるバッファ（**MemoBuf**），メモ化機構を管理するための制御ユニットを持つ．また，コアの外部には一般的なプロセッサが持つ 2 次データキャッシュを持つ他に，命令区間の入出力セットを記憶するための表（**MemoTbl**）を持つ．MemoTbl はサイズが大きく CPU コアからのアクセスコストが大きい．その

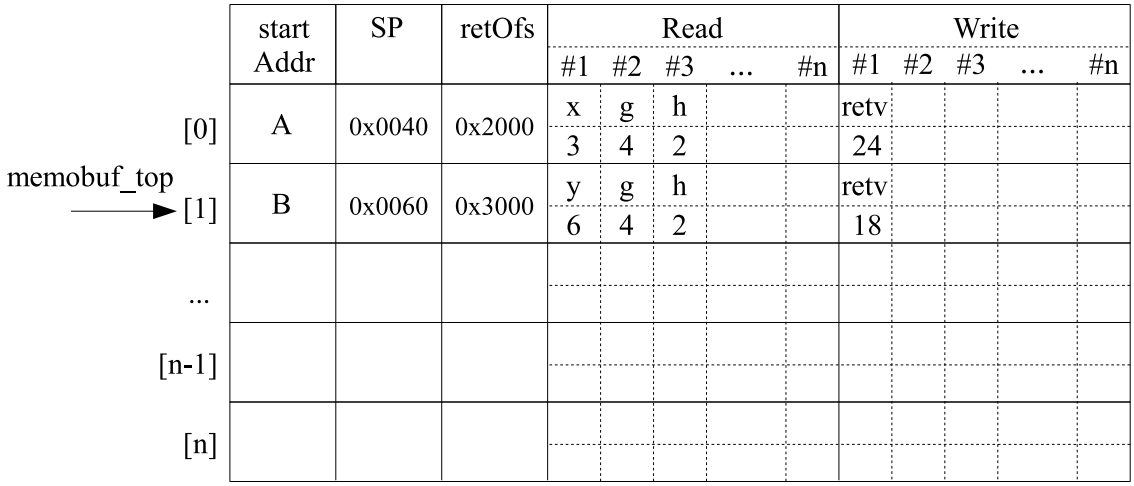

図 4: MemoBuf の構造

ため，メインコアが命令区間の入出力セットを登録する際，MemoTbl に対して頻繁に参照を行うと，そのオーバーヘッドが大きくなる．このオーバヘッドを軽減するため，作業用の書き込みバッファとして MemoTbl に比べてサイズの小さい MemoBuf が用いられ，各命令区間の実行終了時に MemoBuf の内容を一括して MemoTbl へと登録する．

次に MemoTbl 及び MemoBuf の詳細な構成について述べる．**図 4** に MemoBuf の構造を示す．MemoBuf は RAM で構成される表であり，複数のエントリから成る．MemoBuf の 1 エントリには 1 つの命令区間の入出力セットを登録する事ができる．各エントリは，メモ化対象となる命令区間の開始アドレスを記憶する startAddr，命令区間の実行開始時のスタックポインタ SP，命令区間の終了アドレスを記憶する retOfs，命令区間の入力セットと出力セットを記憶する Read 及び Write からなる．また，ポインタ memobuf_top により現在使用しているエントリを把握し，各 MemoBuf のエントリは番号の小さい方から順に使用される．自動メモ化プロセッサは，命令区間を検出すると，memobuf_top をインクリメントした後，命令区間を実行しながら，その実行中に出現した入出力を MemoBuf の Read 及び Write 領域に対して記憶していく．return 命令により関数の呼び出し元へ戻った場合や，ループ区間の分岐命令を検出した場合は，MemoBuf のエントリに記憶してきた当該命令区間の入出力を MemoTbl に書き込み，memobuf_top の値がデクリメントされる．このように命令区間の入出力セットを MemoBuf の各エントリに記憶することで，メインコアが現在実行している命令区間のネスト構造を保持し，入れ子構造になった命令区間もメモ化対象とすることができる．

図3からも分かるように，メインコアからみてMemoTblの手前にMemoBufが存在するため，メインコアは必ずMemoBufを経由してMemoTblへアクセスする．そのためMemoBufのエントリのうち，1つはMemoTblへのアクセス時にバッファとして使用される．よって，MemoBufのエントリ数がn個である場合，記憶可能な命令区間のネスト構造は$n-1$階層までである．これはMemoBufのエントリのうち，$n-1$個のエントリが使用されている時に，n個目のエントリをMemoTblにアクセスする際の一時的なバッファとして用いるためである．なお，命令区間の深さがMemoBufの保持できる$n-1$階層よりも深くなった場合，最も外側つまり上位の命令区間の入出力セットを記憶しているエントリの内容を削除し，内容を削除することで空いたエントリに新しい命令区間の入出力セットを登録する．

続いて，MemoBufにどのように入出力が登録されるかについて説明する．MemoBufの各エントリは現在の命令区間の実行状況に応じて適宜使用される．図4には，図2で示した関数A及び関数BがMemoBufに登録されている様子を示している．図2のプログラム例では，最内部で実行される関数Bの入力は，引数yと大域変数g及びhである．また関数Bの出力はint型の値18である．そのためMemoBufのエントリ1にはこれら入出力の値が格納されている．また，図2の右側にも示したように，関数Aの入力にはその内部で呼び出される関数Bの入力も含まれる．よってMemoBufのエントリ0には関数Aの直接の入力である引数xに加え，関数Bの入力の値も格納されている．

次にMemoTblの構造とその動作モデルについて説明する．MemoTblはその内部に4つの表を持つ．このうちRF，RA，W1はRAMで構成される表であり，RBは連想検索が可能なCAM(Content Addressable Memory)で構成される表である．以下これら構成要素について，その詳細を述べる．

RF: 登録済みの命令区間に関する情報を記憶する表である．RFに記憶する情報は多岐に渡る．まず，関数の開始アドレスやループの分岐命令のアドレス及びループのジャンプ先命令のアドレスを記憶する．それ以外にも，過去に計算再利用が成功又は失敗した回数や，最近登録した各命令区間の入出力値の履歴等を記憶する．

RB: 命令区間の入力値を記憶する表である．入力値を高速に検索可能とするため，CAMで構成される．

RA: 命令区間の入力アドレスを記憶する表である．

W1: 命令区間の出力値を記憶する表である．

続いて自動メモ化プロセッサの動作モデルについて**例1**のプログラムを例に説明す

例 1：自動メモ化プロセッサの動作モデル説明用プログラム

```
 1:  int weight = 1;
 2:  int array_mul(int num, int a[], int b[]) {
 3:      int i, v = 0;
 4:      for(i = 0; i < num; i++) /* 4行目から5行目までを */
 5:          v += a[i] * b[i];    /* ループAとする */
 6:      return (v / weight);
 7:  }
 8:  int main(void) {
 9:      int x, y, z, a[3] = {1, 2, 3}, b[3] = {4, 5, 6};
10:      x = array_mul(3, a, b);
11:      weight++, b[0] = 7;
12:      y = array_mul(3, a, b);
13:      weight--, b[0] = 4;
14:      z = array_mul(3, a, b);
15:      return (0);
16: }
```

る．例 1 のプログラム内の関数 array_mul は，2 つの配列について添字の一致する要素同士の積を求め，第一引数 num で指定された回数だけ配列の添字 0 から順に同様の処理を行い，それらの和を求め，その値を局所変数 v に代入する．その後局所変数 v を大域変数 weight で除算した結果を返す．また，関数 array_mul は main 文の中で複数回呼び出される．

さて，一般に関数やループ等の命令区間は，複数の異なる入力セットを用いて実行される．よってある命令区間の全入力パターンは木構造で表すことができ，関数やループの 1 入力セットはその木構造における 1 本のパスとして表現できる．**図 5** の上部に，例 1 の 10 行目及び 12 行目で関数 array_mul を呼び出した際に当該関数の入力が成す木構造を，それぞれ (a)，(b) として示す．引数 num の値及び配列 a の値は 2 つの入力セット (a) 及び (b) で一致する．しかし配列 b の値及び大域変数 weight の値が異なっているため，図 5 に示したように 3 つ目の入力から入力ツリーが分岐する．入力セットは RB 及び RA に木構造を保持したまま登録され，RB が節に，RA が枝に対応する．

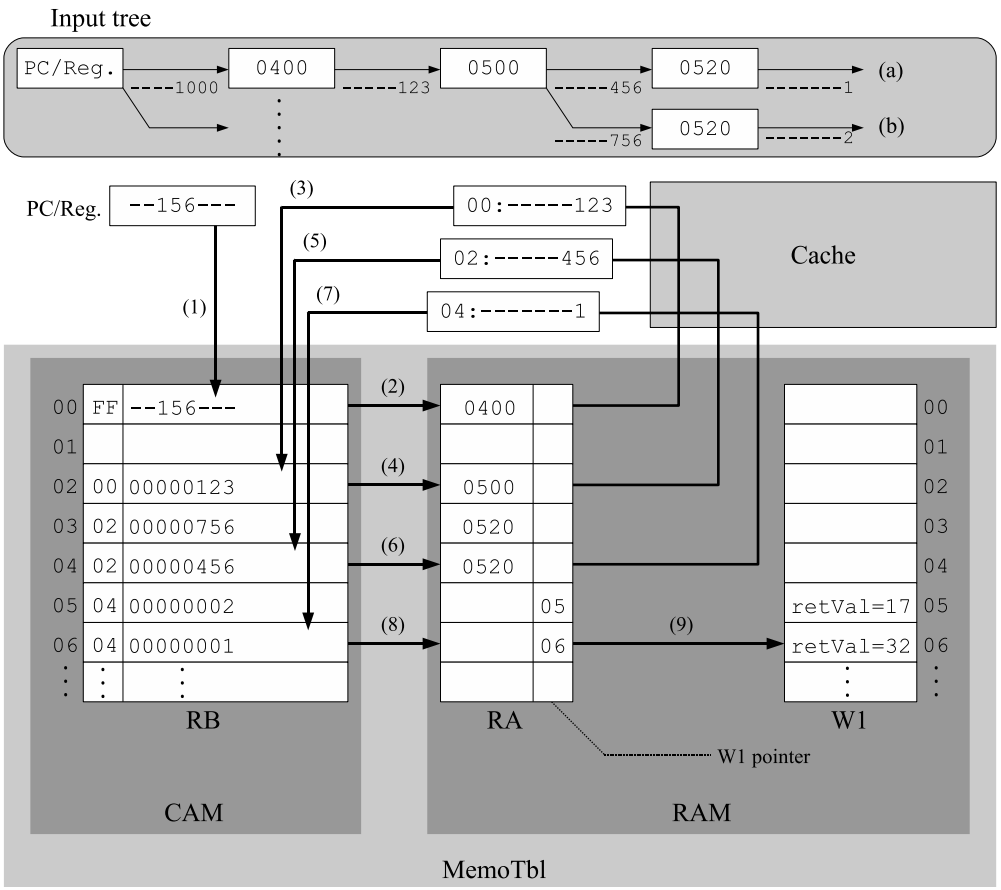

図 5: 入力のツリー構造と入力値検索の手順

命令区間の入出力セットを，その木構造を保ったまま RB 及び RA エントリに登録する事で，限られた容量の RB 及び RA エントリを有効に使用できる．

続いて，例 1 のプログラムを例に入出力セットの登録手順について述べる．例 1 のプログラム中の main 文が最初から実行され，10 行目で関数 array_mul が検出される．関数を検出すると，MemoTbl 内の RF，RB，RA を参照し，当該関数に対応する入力セットが MemoTbl 内に存在するか否かを確認する．これを**入力値検索**と呼ぶ．関数 array_mul は 10 行目の時点で初めて呼び出されるので，RF を検索しても対応する関数アドレスがまだ RF 上に存在しない．そのため関数 array_mul のアドレスを RF に登録し，2 行目から 7 行目までを通常通り実行する．自動メモ化プロセッサは，関数の実行中に出現した入出力を MemoBuf に登録しながら命令の実行を進め，MemoBuf には**図 6** の (a) に示したように入出力が登録される．関数 array_mul の入力は引数 num，配列 a，配列 b 及び大域変数 weight であり，出力は関数の戻り値である．MemoBuf 内に

(a)

start Addr	Read				Write			
	#1	#2	#3	...	#1	#2	#3	...
array_mul	a	b	weight		retv			
	00000123	00000456	00000001		00000032			

(b)

start Addr	Read				Write			
	#1	#2	#3	...	#1	#2	#3	...
array_mul	a	b	weight		retv			
	00000123	00000756	00000002		00000017			

図 6: MemoBuf の状態

ある Read 及び Write 領域は，1 エントリ当たり 32 バイトの値を記憶可能である．これはキャッシュ1 ライン分に相当する．そのため入力である配列 a 及び配列 b は 3 つの要素から成るが，それぞれ 1 つの Read エントリを使用して記憶できる．6 行目の return 文を検出し，関数の実行を終了する際，MemoBuf に記憶した入出力セットを MemoBuf から MemoTbl へと一括して登録する．同様に 12 行目の関数 array_mul の入出力セットも図 6 の (b) に示したように MemoBuf に登録され，当該関数の実行終了時に MemoTbl へと一括して登録される．

次に図 5 を用いて入力値検索の具体的な手順について述べる．プログラム例 1 では，10 行目及び 12 行目で関数 array_mul に対する入力値検索が失敗し，当該関数の入力セットが各 RB エントリに登録されているとする．引き続き 14 行目まで例 1 のプログラムを実行し，14 行目の関数 array_mul の呼び出し命令を検出すると，メインコアは命令実行を一時停止し，入力値検索を開始する．14 行目の関数呼び出し array_mul(3, a, b) に対する入力値検索の手順は図 5 中の (1) から (8) までに対応する．まず，関数のアドレス及び引数の値をキーとして RB を検索する (1)．本例では RB エントリ 00 の値と一致する事が分かる．その後，今一致した RB エントリと同一エントリ番号の RA を参照し (2)，次の入力値が格納されているレジスタ番号又は主記憶アドレスを得る．本例では配列 x の値を記憶している主記憶アドレス 0400 を RA エントリ 00 から得た後，検索キー 00 及び主記憶アドレス 0400 から読み出した値を用いて再度 RB を検索する (3)．検索キーは命令区間の開始アドレス又は入力を木構造として見た時の，親ノードの RA エントリ番号によって一意に決定される．検索の結果，主記憶アドレス 0400 から読み出した値は RB エントリ 02 と一致するため，先程と同様に RA エントリ 02 から

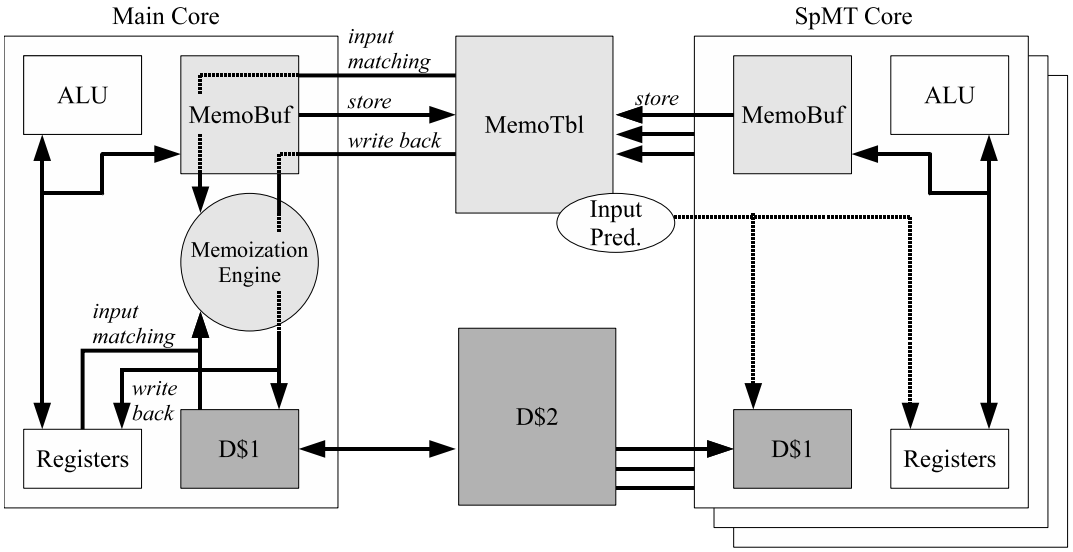

図 7: 並列事前実行機構を備えた自動メモ化プロセッサの構造

次の入力アドレス 0500 を得る (4)．同様の手順で検索を行い (5)(6)(7)，全ての入力が一致する事を確認すると (8)，入力値検索は成功となる．入力値検索が成功すると，終端エントリの W1 ポインタを用いて出力セットを W1 から読み出し (9)，メインコアのレジスタやキャッシュへ書き戻す．以上の手順により，14 行目の関数 array_mul(3, a, b) に対して計算再利用を適用することができる．

2.3 並列事前実行

マルチコアが一般的となっている背景から，単一のコアだけではなく，多くのコアを用いたモデルについて検討する必要がある．そこで自動メモ化プロセッサは，複数のコアを有効に利用する**並列事前実行**という仕組みを備えている．並列事前実行では，まず過去の命令区間の入力セットに基づき，ストライド予測を用いてその命令区間を将来実行する際に用いられる入力セットを予測する．そしてその予測された入力を用いて当該命令区間をメインコアとは別のコアで予め実行し，その命令区間の入出力セットを MemoTbl へ登録しておく．これにより，メインコアがまだ実行していない命令区間に対しても計算再利用を適用可能となり，高速化が可能となる．

並列事前実行機構を備えた自動メモ化プロセッサは，**図 7** に示すように従来のメインコアに加えて並列事前実行コアを持つ．また ALU，レジスタ，1 次データキャッシュ，MemoBuf は各コアが独立で持ち，MemoTbl，2 次データキャッシュおよび主記憶は全てのコアで共有される．MemoBuf をコアごとに持っているため，各命令区間の実行終

例 2：並列事前実行の動作モデル説明用プログラム

```
 1:  int i, u[10]= {0}, v[10] = {0};
 2:  ...
 3:  for(i = 0; i <= 7; i++)     /* 3行目から4行目までを */
 4:      u[i] = factrial(i);     /* ループAとする */
 5:  ...
 6:  for(i = 0; i <= 7; i++) {   /* 6行目から9行目までを */
 7:      v[i] = fibonacci(i);    /* ループBとする */
 8:      if(i == 5) i++;
 9:  }
10:  ...
```

了時に，それぞれのコアは入出力を MemoBuf から MemoTbl へ独立して登録することができる．メインコアは自身が MemoTbl に登録した入出力セットに加え，並列事前実行コアによって MemoTbl に登録された入出力セットを用いて計算再利用を適用することができる．並列事前実行コアは複数備えることが可能で，それぞれのコアは予測された入力を用いて，命令区間を並列に実行することができる．さらに並列事前実行は，キャッシュプリフェッチ機構としても有効に働く．並列事前実行コアが命令を事前に実行することで，2 次データキャッシュにまだ存在しない値が主記憶から読み出される可能性がある．その場合，将来メインコアが当該主記憶アドレスの値を読み出す際に主記憶にアクセスしなくても 2 次データキャッシュに当該アドレスの値が存在していることとなり，キャッシュミスを削減できる．また並列事前実行では，将来メインコアによって使用されない入力セットを用いて命令区間が事前に実行されたとしても，メインコアとは独立して命令を実行しているため，並列事前実行コアに再び新しい命令区間を割り当てる為のオーバヘッドは発生しない．ただし，エントリ数に制限のある MemoTbl に不必要なエントリが登録されることにより，有効なエントリが削除され，性能が低下してしまう可能性はある．

続いて並列事前実行機構を備えた自動メモ化プロセッサの動作モデルについて，**例 2** のプログラム例及び**図 8** を用いて説明する．並列事前実行を行うためには RB に登録された入力の履歴に基づき，将来の入力を予測して，並列事前実行コアへ渡す必要がある．このため，入力を予測して並列事前実行コアに渡すための小さなハードウェ

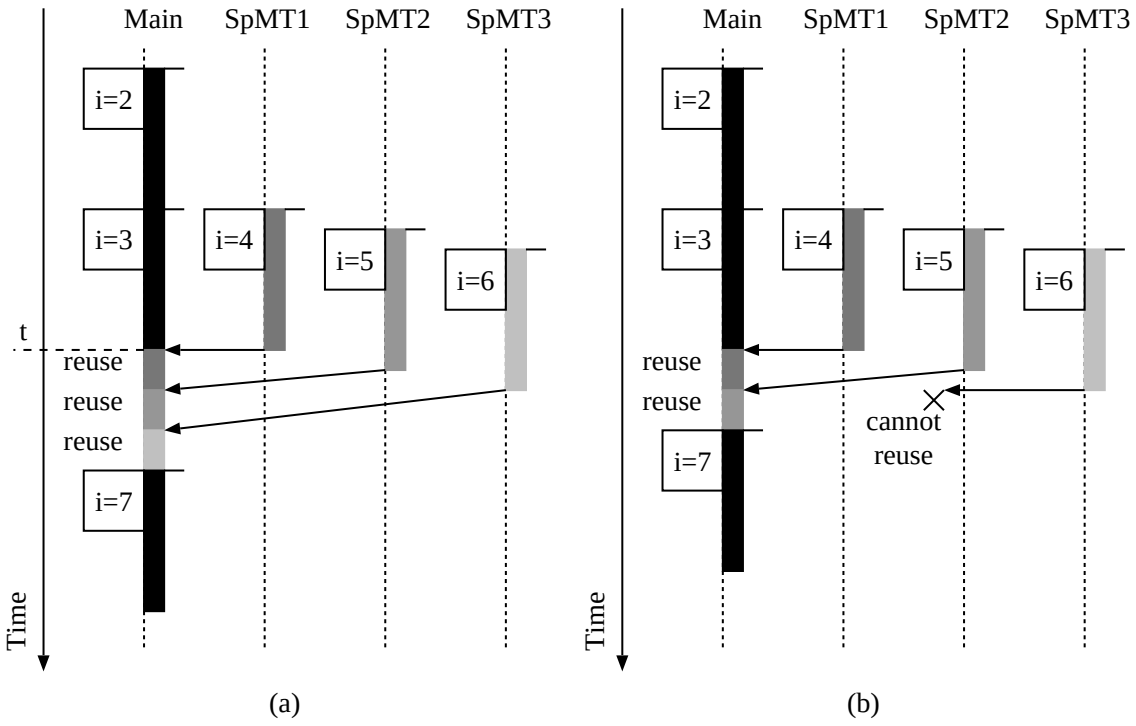

図 8: 並列事前実行の流れ

アを MemoTbl 内の RF に設ける．具体的には，同一命令区間を異なる入力で呼び出した際の入力の差分に基づいてストライド予測を行う．予測された入力セットに基づき，並列事前実行コアはメインコアと並行して当該命令区間の実行を開始する．図 8 はメインコアに加えて 3 つの並列事前実行コアを持つ場合の実行の流れを示しており，図 8 中の (a) は，プログラム例 2 のループ A に対して並列事前実行を行う様子を示す．自動メモ化プロセッサがループ区間に対応する後方分岐命令を検出するまでは，命令区間のブロックがループを構成することを検出できないため，例 2 の 3 行目でイタレーション変数 i の値が 0 の時は必ず通常通り実行される．続いて当該命令区間をイタレーション変数 $i = 1$ で実行し始めようとした時，当該命令区間はループを構成することが分かるため，メインコアはイタレーション変数 $i = 1$ に対応する区間の入力値検索を行う．ところが，自動メモ化プロセッサはイタレーション変数 $i = 1$ で当該ループをまだ実行していないため，対応する入力セットは MemoTbl に存在しない．そのため，メインコアは当該ループを通常通り実行する．同様にイタレーション変数 $i = 2$ に対しても，メインコアは当該ループを通常通り実行する．メインコアが当該ループをイタレーション変数 $i = 2$ まで実行した後，並列事前実行コアは当該命令区間の実行履歴に基づくストライド予測により，ループ A の入力である変数 i のストライドを 1 と予

測する．図 7 の例では 3 つの並列事前実行コアを持つ．そのため，メインコアが当該ループをイタレーション変数 i = 3 で実行している間に，並列事前実行コアはそれぞれイタレーション変数 i = 4 から i = 6 に対応する命令区間を実行し，その入出力セットを MemoTbl に書き込んでおく．メインコアは時刻 t の時点で，まだイタレーション変数 i = 4 から i = 6 でループ A を一度も実行したことがない．しかし当該区間に対応する入出力セットは並列事前実行コアによって事前に MemoTbl に登録されている．そのため，メインコアは当該命令区間に対して計算再利用を適用することができる．一方図 8 の (b) はループ B に対する並列事前実行の様子を示す．この場合はイタレーション変数 i の値が 5 の時に 8 行目の if 文の条件が成立するため，i の値がインクリメントされる．そのため，イタレーション変数 i = 6 では当該ループが実行されない．よって 3 つ目の並列事前実行コアの実行結果は利用されず，無駄になる．

さて，メインコアと並列事前実行コアは 2 次データキャッシュや主記憶を全てのコアで共有している．このため，これら共有領域に対して書き込みを行うと，メインコアや並列事前実行コアがプログラムを実行する際に値の不整合が生じてしまう．そこで，並列事前実行コアは主記憶に書き込みを行わず，MemoBuf の Read 及び Write 領域に書き込みを行う．以後，並列事前実行コアが当該アドレスの値を読み出す際は，主記憶ではなく MemoBuf 内の Read 及び Write 領域を参照する．これにより主記憶に値を書き込む事なく命令実行を継続できるため，主記憶値に矛盾は生じない．

2.4 再利用オーバヘッドとオーバヘッド評価機構

自動メモ化プロセッサは，MemoTbl の一部を CAM で構成することにより，その参照を高速に行うことができる．しかしある命令区間に対して計算再利用を適用するためには，回避不可能なオーバヘッドが生じる．MemoTbl 内の RB を構成する CAM 自体は 1 サイクルで検索可能であるが，現在の一般的な CPU コアのクロック周波数を 2GHz から 3GHz 程度と想定すると，CAM はその 1/10 程度のクロック周波数で動作する．そのため CAM の検索には，CPU 上での 10 サイクル程度に相当する時間を要することになる．なおこの比較コストは，入力値検索の成功・失敗に関わらず発生する．さらに入力値検索が成功した際には，出力を MemoTbl からレジスタやキャッシュへ書き戻すためのコストを要する．命令区間の入力の比較コストと出力の書き戻しコストを**再利用オーバヘッド**と呼ぶ．

さて，命令区間によっては再利用オーバヘッドが大きく，計算再利用を行わずに実際に命令を実行した方が早く実行を終えることができる場合も存在する．そうした場合，

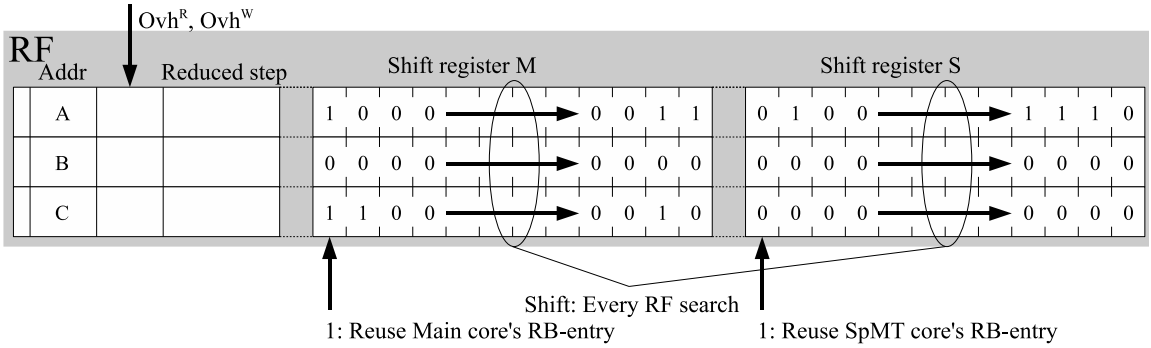

図 9: シフトレジスタの構造

計算再利用により性能が悪化するばかりか，必要としない入出力を MemoTbl に登録している事になり，MemoTbl が有効活用されない．そこで自動メモ化プロセッサは，MemoTbl への無駄なアクセスを抑制する再利用オーバヘッド評価機構を備えている．再利用オーバヘッド評価機構を使用して，再利用オーバヘッドと計算再利用により高速化できるサイクル数を見積もる．これにより，計算再利用による効果が再利用オーバヘッドを上回り，高速化可能と判断した区間に対してのみ入力値検索が行われる．

再利用オーバヘッド評価機構の動作モデルと実装ハードウェアについて述べる．プログラム中に出現する命令区間のうち，メインコアによる登録頻度が高く，さらに並列事前実行コアが登録した入出力セットの使用頻度も高いものが，最も並列事前実行による効果が得られる命令区間である．自動メモ化プロセッサは，この動的に変化する入出力の登録頻度や使用頻度を把握するために，一定期間における入出力セットの登録および計算再利用の状況を，RF に追加した 2 つのシフトレジスタを用いて記録している．このシフトレジスタの記録を基に再利用オーバヘッドの算出を行う．**図 9** に示すように，各 RF エントリはシフトレジスタ M 及び S を備える．メインコアが関数又はループ区間を検出した時，全ての RF エントリのシフトレジスタ M 及び S を右に 1 ビットシフトする．入力値検索に失敗し，メインコアが通常通り命令区間を実行した後，MemoTbl へ当該命令区間の入出力セットを登録する際には，その入出力セットがどのコアによって登録されたかを RF に記憶させる．シフトレジスタ M 及び S は，メインコア及び並列事前実行コアにより登録されたエントリが，過去一定期間の間に使用された回数を記憶する．ある命令区間に対して計算再利用を適用できた場合，メインコアが登録したエントリを用いた場合は対応する RF エントリのシフトレジスタ M の左端に 1 をセットする．また，メインコアが登録したエントリではなく，並列事前実行コアが登録したエントリを用いた場合は対応する RF エントリのシフトレジス

タ S の左端に 1 をセットする．これにより，効率よく計算再利用が適用可能な命令区間に対しては M や S に含まれる有効ビット数は多くなる．一方，計算再利用を適用することが困難な命令区間に対しては M や S に含まれる有効ビット数は少なくなる．

再利用オーバヘッド評価機構は，以下の計算式を用いて計算再利用の効果を見積もる．

$$(N^{Main} + N^{SpMT}) \times (S - Ovh^R - Ovh^W) \tag{1}$$

$$(T - N^{Main} - N^{SpMT}) \times Ovh^R \tag{2}$$

T をシフトレジスタのビット長とすると，ある命令区間に対して最近の一定期間 T の間に，メインコアが登録したエントリを用いた入力値検索成功回数 N^{Main} および並列事前実行コアが登録したエントリを用いた入力値検索成功回数 N^{SpMT} は，シフトレジスタ M 及び S の有効ビット数から得られる．これらの値と当該命令区間に対する過去の削減サイクル数 S から，実際に削減できたサイクル数を式 (1) により計算可能である．また，入力値検索に失敗し，検索対象の命令区間に対して計算再利用がが行われなかった場合の再利用オーバヘッドは式 (2) により計算可能である．発生した再利用オーバヘッド (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，計算再利用による効果が得られると考えられるため，このような命令区間に対しては入力値検索を行う．逆に発生した再利用オーバヘッドが削減できたサイクル数を上回る場合，入力値検索を行わないことで，無駄な再利用オーバヘッドの発生を抑制できる．なお，過去に計算再利用が行われたことがあり，その効果が得られる命令区間であっても，最近の一定期間 T の間に計算再利用が一度も行われなかった場合，$N^{Main} + N^{SpMT} = 0$ を満たしてしまう．そのため，計算再利用による効果が得られないと判断され，入力値検索は行われなくなる．よって，$N^{Main} + N^{SpMT} = 0$ を満たした際は，例外処理として N^{SpMT} の値を T の値で初期化し，入力値検索が行われなくなることを回避する．自動メモ化プロセッサは以上のような仕組みで計算再利用の効果を見積もり，効果の得られる命令区間のみに対して計算再利用の適用を試みる．

3　複数スレッドを用いた再利用オーバヘッドの削減

本章では，自動メモ化プロセッサが入力値検索を行う際に発生するオーバヘッドを削減することで，更なる高速化を実現する手法を提案する．まず予備評価を行い，既存の自動メモ化プロセッサの問題点を挙げる．その結果を踏まえ，複数のスレッドを用いて再利用オーバヘッドを削減する手法について述べる．

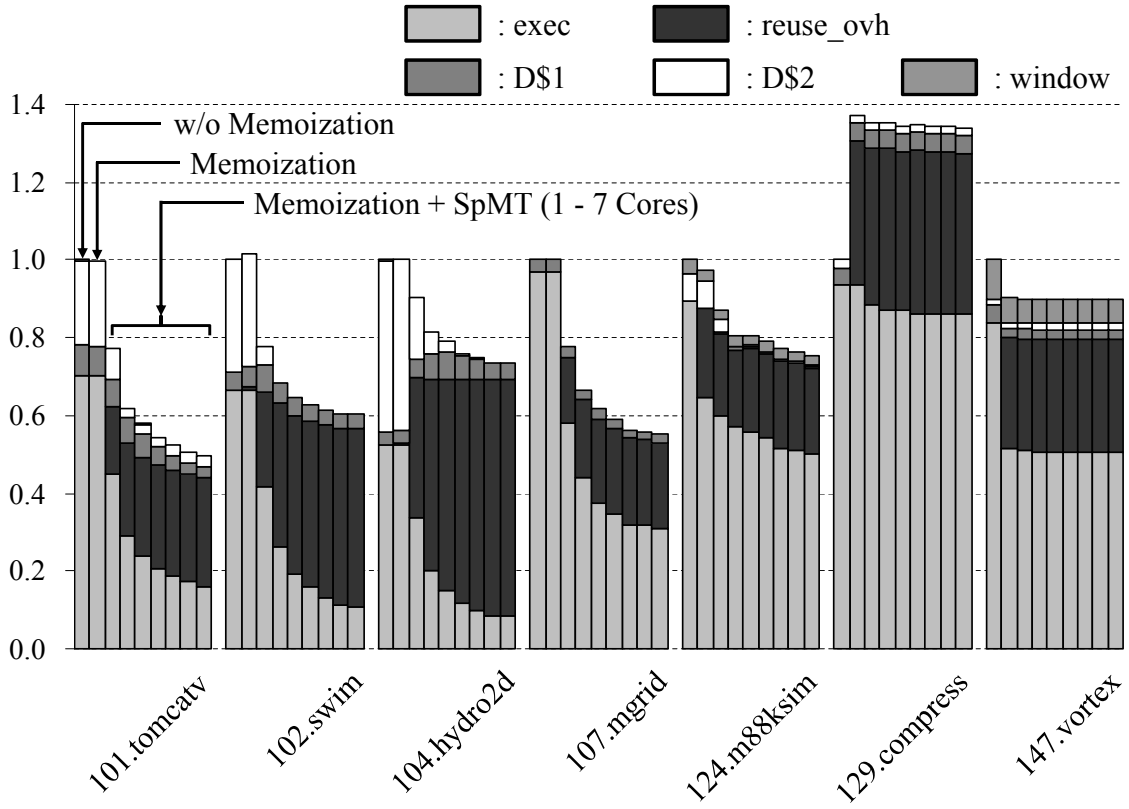

図 10: 予備評価

3.1 予備評価

既存の自動メモ化プロセッサにより，どの程度高速化が可能であるかを調査するために，汎用ベンチマークプログラムである SPEC CPU95 を用いて予備評価を行った．予備評価の結果を**図 10** に示す．図 10 のグラフは左から順にメモ化無し，メインコアのみを用いてメモ化を行った場合，左から 3 番目以降のグラフはメインコアに加えて並列事前実行コアを用い，並列事前実行コアの数を 1 コアから 7 コアまで変化させた場合の結果を示し，メモ化無しの場合の総サイクル数を 1 として正規化した．凡例は exec : 命令の実行サイクル数，reuse_ovh : 再利用オーバヘッド，D$1 : 1 次データキャッシュミス，D$2 : 2 次データキャッシュミス，window : レジスタウインドウミスを表わす．このレジスタウインドウミスは，自動メモ化プロセッサが想定する SPARC アーキテクチャ[13] 特有のものである．なお，予備評価では SPEC CPU95 ベンチマークのうち，並列事前実行により性能が向上するプログラム及び顕著な特徴が現れるプログラムを用いた．また並列事前実行の性能限界を正しく見積もるため，再利用オーバヘッド評価機構は無効として評価した．

まず，グラフ中の左から 4 つのプログラム（101.tomcatv，102.swim，104.hydro2d，107.mgrid）はメインコアのみによるメモ化では高速化できないものの，並列事前実行コアを組み合わせる事により高速化を実現している．これらのプログラムでは，並列事前実行コアの数が増加するにつれて性能は向上しているが，その性能向上の割合は徐々に小さくなっている．特に 102.swim では並列事前実行コアの数が 6 つまでは性能が向上するが，並列事前実行コアの数が 7 つになると逆に性能が悪化している．101.tomcatv，104.hydro2d，107.mgrid でもグラフ中に示した並列事前実行コア数が 7 つまではサイクル数が向上しているが，グラフ中にみられる傾向から，並列事前実行コアの数がある一定の数になると性能のピークに達し，それ以上並列事前実行コアの数を増やしても，性能が停滞もしくは悪化すると考えられる．並列事前実行コアの数が増加すると性能が悪化する原因は，並列事前実行コアによって MemoTbl に登録されたエントリが有効に利用されていないためと考えられる．一般に並列事前実行コアの数を増加させると，コア数が少ない場合よりも MemoTbl に書き込まれる入出力セットは増加する．しかし，MemoTbl のエントリ数は一定であるにも関わらず，並列事前実行コアによって多くの入出力セットが MemoTbl の容量以上に書き込まれる．その結果，MemoTbl に登録済みの有効なエントリが追い出され，計算再利用を適用できる可能性が低下し，性能も悪化すると考えられる．

また，129.compress 及び 147.vortex は並列事前実行コアを用いても，性能が向上しないプログラムである．特に 129.compress では大きな再利用オーバヘッドが発生するため，命令区間をメモ化することにより大幅に性能が悪化している．また，メインコアのみによるメモ化により高速化が可能な 124.m88ksim や 147.vortex でも，全体の実行サイクル数のうち，比較的大きな割合を再利用オーバヘッドが占めている．

性能悪化の大きな原因となっている再利用オーバヘッドがプログラム全体のサイクル数に対して占める割合を**表 1** に示す． 図 10 から分かるように，計算再利用が適用可能な命令区間を含むプログラムでは再利用オーバヘッドも発生する．計算再利用により高速化可能なプログラムでも，並列事前実行コア数などの条件によっては全体の実行サイクル数は削減できているものの，命令区間の実行サイクル数以上の再利用オーバヘッドが発生する場合もある．この傾向は 129.compress で特に顕著であり，命令の実行サイクル数は僅かしか削減できていないが，その削減サイクル数の数倍もの再利用オーバヘッドが発生しており，総サイクル数は大幅に増加している．

図 11 に発生した再利用オーバヘッドの内訳を示す．凡例は test(r) がレジスタと MemoTbl の値との比較コスト，test(m) が主記憶と MemoTbl の値との比較コスト，write

表 1: 再利用オーバヘッドがプログラム全体のサイクル数に対して占める割合

Benchmark	Memoization	Memoization + SpMT(2 cores)
101.tomcatv	0.38 %	23.87 %
102.swim	1.11 %	37.13 %
104.hydro2d	0.62 %	49.08 %
107.mgrid	0.01 %	20.05 %
124.m88ksim	23.01 %	19.76 %
129.compress	37.30 %	41.72 %
147.vortex	28.74 %	28.92 %

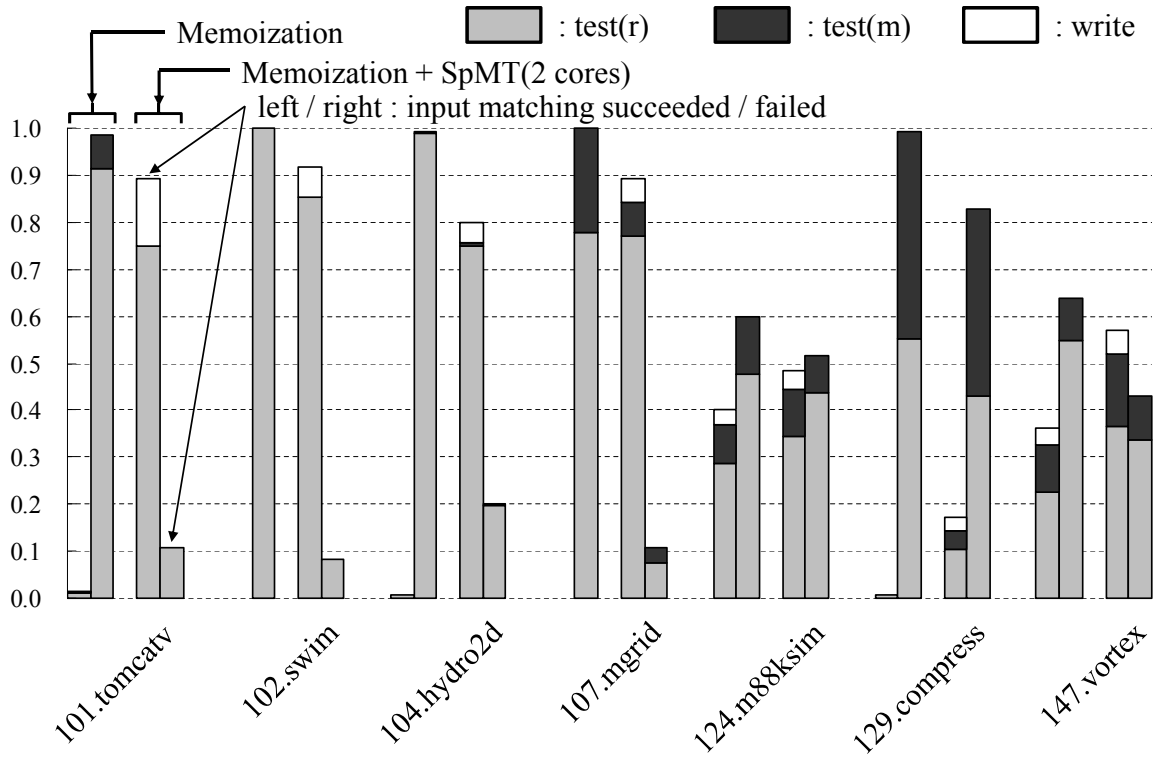

図 11: 再利用オーバヘッドの内訳

が出力を MemoTbl からレジスタや主記憶に書き戻すために要するコストである．グラフは発生する全ての再利用オーバヘッドを 1 として正規化した．また，1 つのプログラムについて，左 2 本のグラフはメインコアのみを用いた場合の入力値検索成功時及び入力値検索失敗時に発生したオーバヘッドである．また右 2 本のグラフはメインコアに加え，2 つの並列事前実行コアを用いた場合の入力値検索成功時及び入力値検索

失敗時に発生したオーバヘッドである．

まず，並列事前実行コアを用いない場合の結果をみると，メインコアのみでもメモ化を適用可能な命令区間の多い 124.m88ksim や 147.vortex では，入力値検索成功時，失敗時ともにオーバヘッドが発生しており，その割合に大きな偏りはない．一方，並列事前実行により，メモ化を適用可能な命令区間が増加する 101.tomcatv から 107.mgrid までのプログラムや，並列事前実行を行ってもメモ化による効果の得られる命令区間がほとんど存在しない 129.compress の検索コストは，そのほとんどが入力値検索失敗時に発生している．

次に並列事前実行コアと組み合わせた場合の評価として，図 10 の左から 3 番目及び 4 番目のグラフに該当する，並列事前実行コアを 2 コア用いた場合の結果をみる．124.m88ksim や 147.vortex では，メインコアのみの場合と比べ，入力値検索成功時と入力値検索失敗時に発生するオーバヘッドの割合は変化しているが，いずれも大きな偏りはない．また，129.compress では並列事前実行により若干ではあるが命令の実行サイクル数を削減できるため，それに伴う入力値検索成功時のオーバヘッドが発生している．メインコアのみを用いたメモ化とは大きな違いがみられるのが 101.tomcatv から 107.mgrid までのプログラムである．これらのプログラムでは並列事前実行が効果的であり，プログラム全体としての実行サイクル数を削減することに成功している．しかしこれら並列事前実行が有効なプログラムでも，並列事前実行コアの数を増やすと，再利用オーバヘッドが命令の実行サイクル数を上回る程度にまで増加し，大きなオーバヘッドが発生している．また，顕著な特徴として，これら CFP ベンチマークプログラムでは再利用オーバヘッドのほとんどがレジスタの値と MemoTbl の値との比較コストで占められている．一方，124.m88ksim や 147.vortex ではレジスタの値と MemoTbl の値との比較が多くの割合を占めるが，主記憶の値と MemoTbl の値との比較コストもある程度発生している．このように，多くのプログラムで再利用オーバヘッドがプログラムの高速化を妨げている事は明らかである．したがって自動メモ化プロセッサを更に高速化するためには，再利用オーバヘッドを削減する必要があると考えられる．また再利用オーバヘッドを削減するに当たり，図 11 より入力値検索成功時と失敗時の両方の場合でオーバヘッドが発生していることから，検索成功時，失敗時それぞれの場合のオーバヘッドを削減する必要があると考えられる．

3.2 再利用オーバヘッドの削減

前節の予備評価の結果を踏まえると，現在の自動メモ化プロセッサの問題点として，並列事前実行コアの数を増やしても，それらは全て有効に利用されるとは限らず，コア数が一定の数に達すると性能向上の限界に達する事が挙げられる．したがって，今後の技術革新により，プロセッサ当たりのコア数が更に増加していく事を考えると，並列事前実行とは異なった手法により，複数のコアを有効利用する必要がある．そこで，現在の自動メモ化プロセッサでは再利用オーバヘッドが大きい点に着目し，複数のスレッドを利用して再利用オーバヘッドを削減する手法を提案する．

提案手法では従来のメインコアと同等の動作をするメインスレッドと，従来の並列事前実行コアと同等の動作をする並列事前実行スレッドに加え，**投機スレッド**と**通常実行スレッド**を用いて再利用オーバヘッドを削減する．このうち投機スレッドは入力値検索成功時に発生する再利用オーバヘッドを，通常実行スレッドは入力値検索失敗時に発生する再利用オーバヘッドを削減する．提案モデルでは，従来の自動メモ化プロセッサと同様に，命令区間の検出や各スレッドの実行開始のタイミング決定などをハードウェアにより自動的に行うため，プロセッサは一切のソフトウェア支援を必要としない．**例 3** にこれらのスレッドの動作説明用のプログラムとこれらのスレッドが実行する命令区間を示す．以下 3.2.1 及び 3.2.2 では例 3 を用いながら投機スレッドと通常実行スレッドの動作モデルの詳細について述べる．

3.2.1 投機スレッド

メモ化の対象となる命令区間には，多くの場合複数の入力が存在する．複数の入力を持つ命令区間の入力セットは，複数の RB エントリを使用して記憶されているため，当該命令区間の入力となるレジスタや主記憶の値と，RB エントリとを複数回比較する必要がある．投機スレッドはこの比較コストを隠蔽する．これらは入力値検索成功時に発生する再利用オーバヘッドである．メインスレッドがある命令区間の開始を検出すると同時に，投機スレッドはメインスレッドのプログラムカウンタの値を自身のプログラムカウンタにコピーする．その後その命令区間の入力セットが，予め決められた個数の RB エントリと一致したとする．この時，投機スレッドは入力値検索に成功したものとして，メインスレッドのレジスタセットの内容と先ほどコピーしておいたメインスレッドのプログラムカウンタの値を用いて当該命令区間の後続区間を実行し始める．

例 3 には関数 array_calc の実行中に関数 array_mul を呼び出し，その実行終了後に元の関数 array_calc に戻る処理を行うプログラムを示す．また例 3 に示したプログラムを

例 3：投機スレッドと通常実行スレッドの動作説明用プログラム

```
 1: int a[3], b[3], weight = 2;
 2: int array_mul(int a[], int b[]) {
 3:   int v = 0;
 4:   v += a[i] * b[i];
 5:   return (v / weight);
 6: }                                        II
 7: int array_calc(void) {
 8:   int x, y;
 9:   x = array_mul(a, b);                   I
10:   weight--, b[0] = 4;
11:   ...
12:   y = array_mul(a, b);                   III
13:   ...
14:   return(0);                             IV
15: }
```

No-memoization thread

Preceding thread

Preceding thread

実行する際に，投機的再利用実行を行う動作モデルを**図 12** に示す．以下例 3 のプログラムの実行順に沿って投機スレッドの動作モデルについて説明する．まず図 12 の上側に示した従来モデル Original の動作について説明する．メインスレッドは命令区間 I を実行し，時刻 t_1 で関数 array_mul を検出したため，当該関数に対して入力値検索を行う．そしてメインスレッドは時刻 t_3 で入力が完全に一致した事を確認すると，出力を MemoTbl からレジスタ及びキャッシュへと書き戻し，時刻 t_4 で後続の命令区間 III の実行を開始する．その後，時刻 t_6 で関数 array_mul を検出し，先ほどと同様に当該関数に対して入力値検索を行う．しかし入力セットが MemoTbl に存在しなかったため，当該区間を時刻 t_7 から通常通り実行する．5 行目の return 命令により，関数 array_mul からその呼び出し元の関数 array_calc に戻った後は，命令区間 IV を通常通り実行する．

次に図 12 の下側 Proposal に示した提案モデルでの動作について説明する．ここでは説明の都合上，拡張後の自動メモ化プロセッサが持つコアに (A)，(B) と識別子を付し，プログラムの実行開始時点では各コアにはそれぞれ，メインスレッドと投機スレッドが割り当てられているものとする．メインスレッドを割り当てられているコア

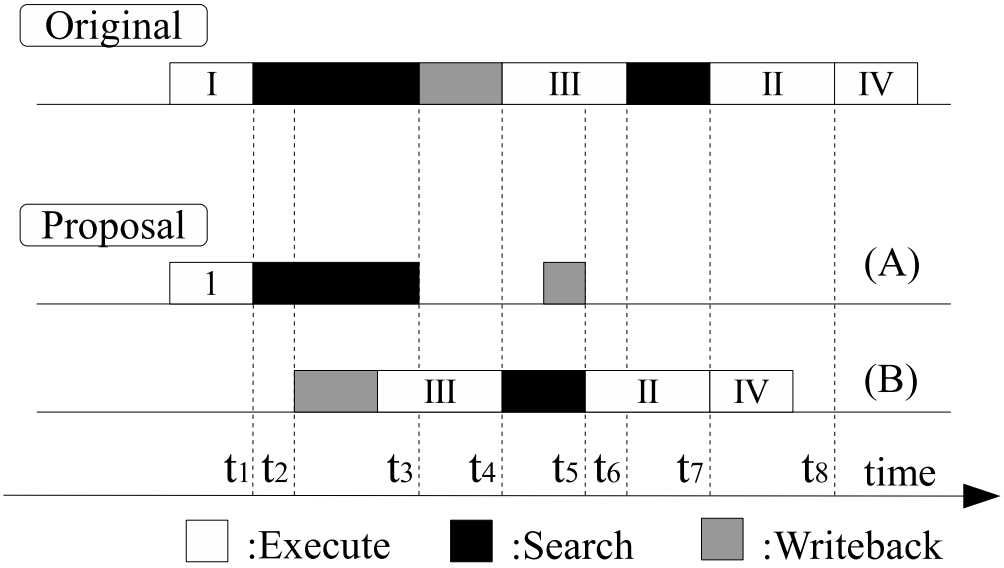

図 12: 投機スレッドの動作モデル

(A) は時刻 t_1 で 9 行目の関数 array_mul を検出し，入力値検索を開始する．それと同時に，コア (B) はコア (A) からプログラムカウンタの値を専用バスを利用してコピーし，コア (B) のプログラムカウンタにコピーした値を書き込む．コア (A) は従来のメインコアと同様に入力値検索を行い，時刻 t_2 で入力のうち配列 a の値が MemoTbl に登録されている値と一致したとする．このとき，コア (B) は過去に MemoTbl に登録した入出力セットの履歴をもとに 9 行目の関数 array_mul の呼び出しに対応する出力セットを予測する．そして，コア (B) は MemoTbl 内の W1 から出力セットを読み出す．その後，コア (B) は 9 行目の関数 array_mul の後続区間 III を実行し始める．時刻 t_3 でコア (A) が 9 行目の関数 array_mul の入力が完全に一致した事を確認すると，コア (B) が行っていた命令区間 III に対する投機的実行が成功したか否かを判定する．この判定は投機スレッドが出力を読み出す際に用いた W1 へのインデクスと，入力値検索の終端となった RA エントリが持つ W1 へのインデクスの値とを比較することで行われる．これらの値が等しい場合，投機的再利用は成功となり，メインスレッドを割り当てられているコア (A) と投機スレッドを割り当てられているコア (B) との間で割り当てるスレッドを切り替える．一方，投機スレッドが使用した W1 へのインデクスとメインスレッドが実際に読み出した値が異なる場合，投機スレッドは誤った出力セットを使用して後続区間を実行している事になるため，コア (A) とコア (B) との間でスレッドの切り替えは発生せず，投機スレッドの実行結果は squash される．

時刻 t_3 で投機的再利用に成功し，コア (B) は新しくメインスレッドを割り当てられたとすると，コア (B) は引き続き命令区間 III を実行する．その後時刻 t_4 でコア (B) は 12 行目の関数 array_mul を検出し，当該関数に対して入力値検索を行う．また投機スレッドを実行しているコア (A) は，入力のうち一つ目の入力である配列 a の値がある RB エントリに登録されている値と一致したため，先ほどの例と同様に後続区間 IV を実行し始める．引き続き検索を続け，時刻 t_5 でコア (B) は入力値検索に失敗することを確認する．この場合，12 行目の関数 array_mul に対して計算再利用自体が適用できなかったため，投機スレッドの実行結果も squash される．なお投機スレッドはメインスレッドとは独立して動作するため，メインスレッドの MemoTbl や主記憶へのアクセスを妨げることはなく，投機スレッドを用いたことによるオーバヘッドは発生しない．

なお本提案手法では実装の都合上，投機スレッドを割り当てられているコアが call 命令を実行して新たに関数を呼び出したり，return 命令を実行して現在実行している関数からその呼び出し元へ戻る事を不可能とした．さらに分岐命令の中でも，ループを構成する分岐命令もまた実行不可能とした．実装上の内容のため，詳細については次章で述べる．

さて，ループ区間ではループイタレーション単位で入出力セットが登録されているため，入力値検索を行い，1 回のループイタレーションに対する計算再利用を適用した直後に次のループイタレーションに対応する入力値検索を行う必要がある．しかしループを構成する分岐命令の実行を不可能としている制約上，投機スレッドは後続の命令区間がループに対応する後方分岐が不成立の場合のみしか命令を実行することができないため，投機スレッドはループ区間を入力値検索対象とする場合には適用できない．

3.2.2 通常実行スレッド

ある命令区間に対する入力値検索に成功した場合には，対象区間に対して計算再利用を適用することで，検索対象の命令区間の実行を省略でき，実行サイクル数を削減できる．しかし，入力値検索を行っても，命令区間に対応する入出力が MemoTbl に存在せず，入力値検索に失敗した場合は，対象命令区間を通常通り実行しなければならない．このとき，レジスタや主記憶の値と MemoTbl のエントリの比較に要したコストのうち，入力値検索の失敗が判明するまでに要したコストは無駄になってしまう．そこで，入力値検索を行わずに予め検索対象の命令区間を実行する通常実行スレッドを，メインスレッドとは別に用意する．メインスレッドがある命令区間の開始を検出して入力値検索を開始すると同時に，通常実行スレッドはメインスレッドのプログラ

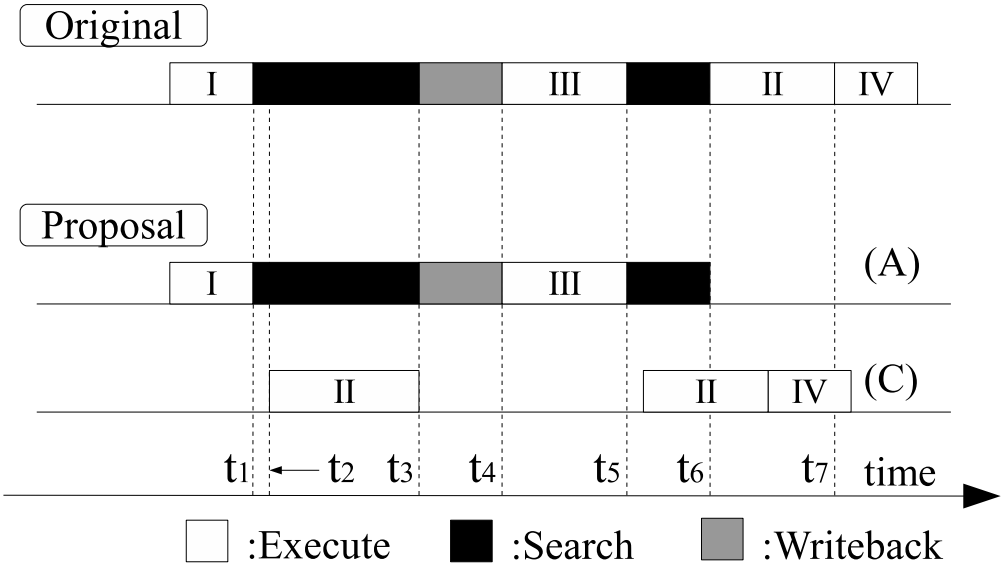

図 13: 通常実行スレッドの動作モデル

ムカウンタの値を自身のプログラムカウンタにコピーし，入力値検索対象となっている命令区間の実行をメインコアが行っている入力値検索と並行して行う．

前項でも用いた例 3 のプログラムを実行する際に，入力値検索に失敗したとして検索対象の命令区間を実行する通常実行スレッドの動作モデルを**図 13** に示す．以下前項と同様に例 3 のプログラムの実行順に沿って通常実行スレッドの動作モデルについて説明する．図 13 の上側に示した従来モデル Original の動作は図 12 と同様であるため，説明を省略し，図 13 の下側に示した提案モデルでの動作について説明する．ここでは説明の都合上，拡張後の自動メモ化プロセッサが持つコアに (A)，(C) と識別子を付し，プログラムの実行開始時点で各コアにはそれぞれ，メインスレッドと通常実行スレッドが割り当てられているものとする．メインスレッドを実行しているコア (A) は時刻 t_1 で 9 行目の関数 array_mul を検出し，入力値検索を開始する．それと同時に，コア (C) はコア (A) からプログラムカウンタの値を専用バスを利用してコピーし，自身のプログラムカウンタにコピーした値を書き込む．プログラムカウンタの値をコピーした時刻 t_2 の時点で通常実行スレッドを割り当てられているコア (C) は，入力値検索対象となっている命令区間 II を通常通り実行する．コア (A) はメインスレッドであるため入力値検索を行い，時刻 t_3 でコア (A) が入力の完全一致を確認する．よって 9 行目の関数 array_mul に対する入力値検索は成功となり，当該関数に対して計算再利用を適用することができる．この場合は入力値検索に成功したため，検索対象の区間を

通常通り実行していた通常実行スレッドの実行結果は不要となり，squash される．

引き続きメインスレッドを実行しているコア (A) は後続の命令区間 III を実行し，時刻 t_5 で 12 行目の関数 array_mul を検出すると，当該関数に対する入力値検索を行う．通常実行スレッドであるコア (C) も先ほどと同様に，当該関数を通常通り実行する．時刻 t_6 で入力の不一致により，入力値検索に失敗したことが確認されると，コア (A) とコア (C) との間で割り当てるスレッドを切り替える．以降コア (C) はメインスレッドを割り当てられ，コア (A) は通常実行スレッドを割り当てられる．通常実行スレッドは，コア (A) が入力値検索を行っている間に入力値検索対象の命令区間を実行しているため，コア (A) が入力値検索に失敗したことによる再利用オーバヘッドは隠蔽される．

3.3 並列事前実行との組み合わせ

前項で述べた投機スレッドと通常実行スレッドは既存の自動メモ化プロセッサを拡張し，再利用オーバヘッドを削減する．しかし，再利用オーバヘッドを削減するよりも，並列事前実行スレッドの数を増やした方が高速化可能なプログラムも存在する事が予備評価により確認されている．例えば，ループ区間には投機スレッドや通常実行スレッドを適用することができないため，ループ区間に計算再利用を適用することにより生じる再利用オーバヘッドを削減できない．そのため，ループ区間を多く含むプログラムに対しては，投機スレッドを用いるのではなく，並列事前実行スレッドを用いれば少しでも効果が得られると考えられる．一方，関数に対する計算再利用が有効なプログラムでは投機スレッドを適用できる可能性は高いが，並列事前実行による効果はあまり期待できない．この理由として，一般に関数に対して並列事前実行はあまり有効でない事が考えられる．関数の入力が単調に変化する場合，その関数はループ区間内に含まれている事が多く，外側のループに対して並列事前実行が有効となる．そのため，ループ内にある関数に対する並列事前実行はあまり有効でない事が多い．よって，関数に対する計算再利用が有効なプログラムには，投機スレッドを割り当てた方が効果が高いと考えられる．

本論文で提案するモデルでは，自動メモ化プロセッサが持つ各コアに対して割り当てるメインスレッド，並列事前実行スレッド，投機スレッド，通常実行スレッドをプログラムの実行中に動的に切り替える事で，効率的にコアを活用する．自動メモ化プロセッサのコア数を N とした時に，各コアに割り当てられるスレッドの種類及びスレッド数を**表 2** に示す．なお提案モデルでは，最低 3 つのコアを持つため，N は 3 以上である．まず，メインスレッドは命令実行のために必要不可欠であるため，必ずいずれ

表 2: プログラム実行中に割り当てられるスレッド

スレッド名	スレッド数	役割
メイン	1	従来のメインコアと同様
投機	0 or 1	入力値検索対象の後続命令区間を実行
通常実行	1	入力値検索対象の命令区間を通常通り実行
並列事前実行	$N-3$ or $N-2$	命令区間を事前実行し MemoTbl へ入出力を登録

かのコアに割り当てられる．また予備評価の結果より，多くのプログラムでは再利用オーバヘッドのうち，入力値検索失敗時に発生するオーバヘッドが多くの割合を占めている．さらに通常実行スレッドは，投機スレッドと異なり，命令区間の出力セットを予測して読み出す必要がなく，確実に再利用オーバヘッドを削減することができる．よって，通常実行スレッドは多くのプログラムに対して有効であると考えられるため，常にいずれかのコアに割り当てられる．一方，投機スレッド及び並列事前実行スレッドはその効果が見込めるプログラムが限定的であるため，これらのスレッドを割り当てるコア数はプログラム実行中に動的に変化させる．

4　ハードウェア実装

本章では 3 章で述べた動作モデルを実現するための具体的なハードウェア実装について述べる．

4.1　アーキテクチャ概要

本節では提案手法のアーキテクチャ概要について述べる．提案モデルのアーキテクチャを**図 14** に示す．提案モデルでは最低で 3 つのコアを持ち，各コアにはそれぞれメインスレッド，投機スレッド，通常実行スレッドが割り当てられる．4 つ目以降のコアには並列事前実行スレッドが割り当てられる．図 14 は 5 つのコアを持つ場合の例である．各コアは従来モデルと同様に，Local MemoBuf を持ち，2 次データキャッシュは全てのコアで共有される．提案モデルでは Local MemoBuf に加え，各コアで共有される共有 MemoBuf を持つ．また従来モデルと大きく異なり，各コアは 2 つのレジスタセットを持つ．2 つのレジスタセットのうち，一方はメインスレッドと並列事前実行スレッドが使用し，もう一方は投機スレッドと通常実行スレッドが使用する．このうち，投機スレッドと通常実行スレッドが使用するレジスタセットを **SpRF(Speculative Register File)** と呼ぶ．さらに，各コア間で値を通信するための専用バスを持つため，各コアの

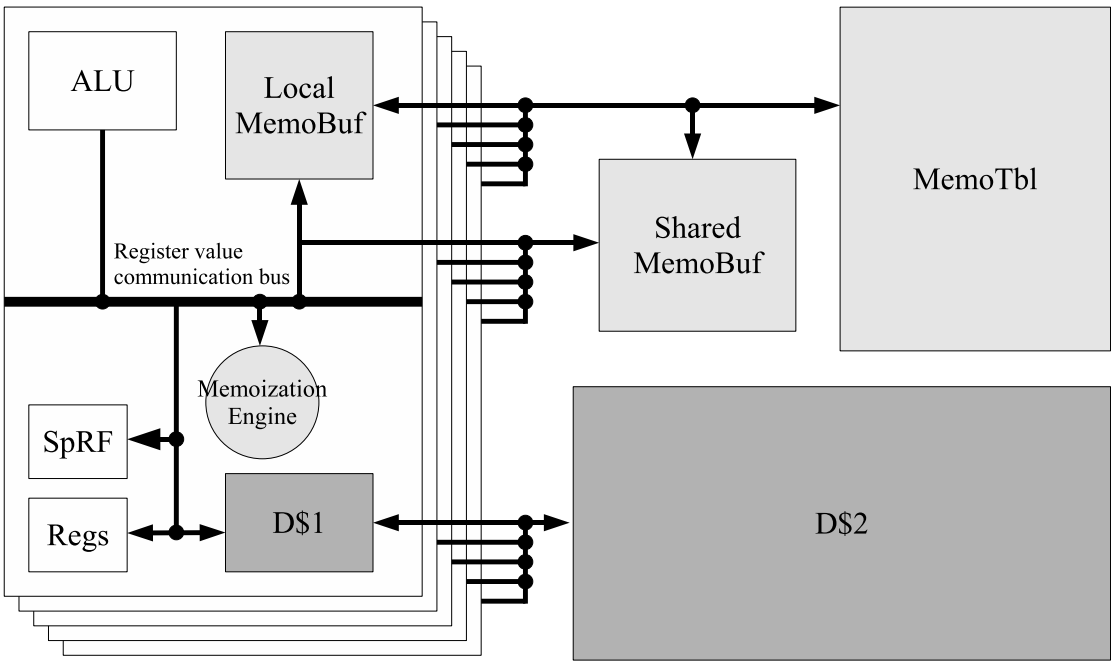

図 14: 提案手法を実装した自動メモ化プロセッサの構造

ALU 出力は全てのコアのレジスタセットに書き込むことができる．コア間でレジスタの値を通信するバスは文献 [14] を参考に実装した．

各コア毎に持つ Local MemoBuf に加え，共有 MemoBuf を実装した理由は，投機的再利用の成功や入力値検索の失敗により，メインスレッドを割り当てられるコアが切り替わった際，新しくメインスレッドを割り当てられたコアが迅速に命令実行を開始できるようにするためである．共有 MemoBuf は全てのコアから同時にアクセス可能であるが，同一のエントリには 1 つのコアしかアクセスできない．投機スレッド及び通常実行スレッドは命令区間を実行中に出現した入出力セットを，共有 MemoBuf に登録する．共有 MemoBuf は全てのコアからアクセス可能であるため，投機スレッドや通常実行スレッドが割り当てられていたコアに新しくメインスレッドが割り当てられても，コストを要することなく共有 MemoBuf に登録されている入出力セットにアクセス可能である．なお，従来モデルと同様に Local MemoBuf は，並列事前実行スレッドが命令区間を実行中に出現した入出力を一時的に記憶するために用いたり，主記憶に値を書き込む代わりに用いられる．

共有 MemoBuf の具体的な使用方法について，**図 15** を用いて説明する．共有 MemoBuf の各エントリの構成は Local MemoBuf の構成とほぼ同等であるが，Local MemoBuf のエントリ数よりも 2 エントリ多い合計 $n+2$ 個のエントリを持つ．$n+2$ 個のエント

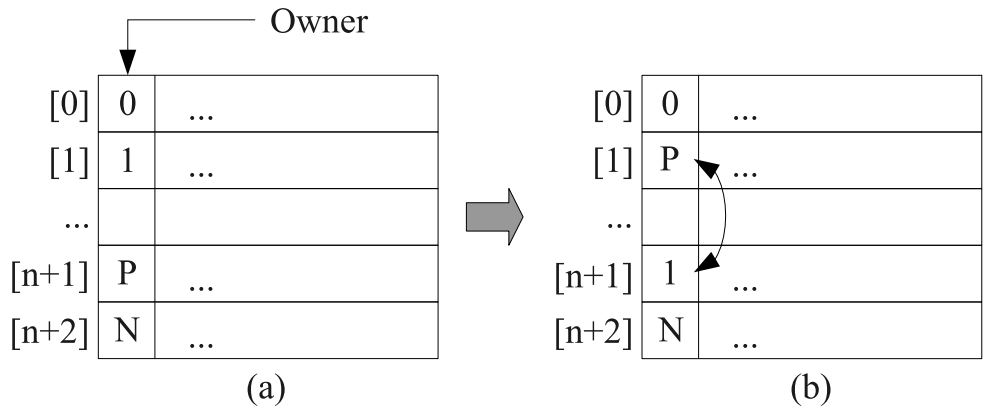

図 15: 共有 MemoBuf の動作例

リのうち，2 つのエントリは投機スレッド及び通常実行スレッドによって使用され，各エントリがどのスレッドによって使用されているかを Owner フィールドに記憶している．Owner フィールドの値のうち，P は投機スレッドによって使用されるエントリであり，N は通常実行スレッドによって使用されるエントリを示す．なお，その他のエントリはメインスレッドによって使用される．図 15 の例ではメインスレッドによってエントリ 0 とエントリ 1 が使用されているため，これらエントリの Owner フィールドにはメインスレッドが記憶しているネスト構造の階層を示す値 0 及び 1 が記憶されている．メインスレッドを割り当てられるコアが切り替わった時，MemoTbl の各エントリを使用するスレッドも変更される．図 15 の (a) では，エントリ $n+1$ を投機スレッドが使用している．その後，メインスレッドと投機スレッド又は通常実行スレッドを割り当てられるコアが切り替えられると，図 15 の (b) に示すようにこれまでメインスレッドが使用していたエントリ 1 を投機スレッドが使用し，新しいメインスレッドはエントリ $n+1$ を使用する．スレッドが切り替わった際に各エントリの内容を他のエントリにコピーしなくてもよいため，新しいメインスレッドはスレッドの切り替え後，すぐに実行を開始することができる．

以下，本項で述べた提案手法のアーキテクチャが，どのように使用されるかについて説明する．4.2 では，命令を実行中に各コアに割り当てるスレッドを動的に切り替える際に用いるハードウェアの詳細及びその使用方法について述べる．その後 4.3 及び 4.4 では，具体的なプログラム例を用いて命令を実行中にハードウェアがどのように用いられるかを説明する．

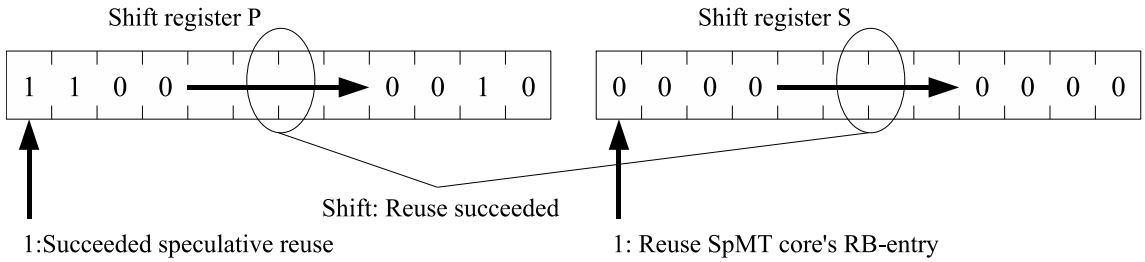

図 16: スレッド性能見積もり用ハードウェア

4.2 割り当てるスレッドの決定

提案モデルでは，プログラムの特性に応じて効果が期待できるスレッドを割り当てるために，割り当てるスレッドを動的に切り替える．そのためには，前章で述べたように投機スレッドや並列事前実行スレッドの有効性を動的に判断し，各コアにスレッドを割り当てる必要がある．そこで，スレッドの有効性を判断するための簡単なハードウェアを自動メモ化プロセッサに追加した．実装に当たっては，既存の自動メモ化プロセッサが備えている再利用オーバヘッド評価機構に使用されているハードウェアを参考にし，できるだけハードウェアコストが小さく済むように考慮した．実装したハードウェアの構成を**図 16** に示す．スレッドの性能見積もりに使用するハードウェアは，図 9 で示した再利用オーバヘッドの見積もりに用いるシフトレジスタと同等である．

まず，図 16 の左側に示した投機スレッドの効果を見積もるために用いるシフトレジスタについて説明する．入力値検索を行い，全ての入力が一致する度にこのシフトレジスタを 1 ビット右シフトする．さらに投機スレッドによる投機的再利用が成功した場合，最上位ビットに 1 をセットする．これにより，T をシフトレジスタのビット幅とすると，最近 T 回の入力値検索成功回数のうち，投機スレッドによる投機的再利用に成功した頻度を記録できる．

次に図 16 の右側に示した並列事前実行スレッドの効果を見積もるために用いるシフトレジスタについて説明する．入力値検索を行い，全ての入力が一致する度にこのシフトレジスタを 1 ビット右シフトする点は投機スレッドの効果を見積もるシフトレジスタと同様である．もし，MemoTbl から読み出した出力セットが，過去に並列事前実行スレッドによって登録されたものであった場合，シフトレジスタの最上位ビットに 1 をセットする．これにより，最近 T 回の入力値検索成功回数のうち，並列事前実行スレッドが登録したエントリを用いて計算再利用を適用できた頻度を記録できる．

シフトレジスタ P 及び S は，図 9 に示した再利用オーバヘッドの見積もりに用いるシフトレジスタのように，各 RF エントリに設けるのではなく，MemoTbl 内の独立し

例 4 : スレッドを動的に切り替えるモデルの説明用プログラム

```
 1:  int weight = 1;
 2:  int array_mul(int num, int a[], int b[]) {
 3:      int i, v = 0;
 4:      for(i = 0; i < num; i++)
 5:          v += a[i] * b[i];
 6:      return (v / weight);
 7:  }
 8:  int main(void) {
 9:      int x, y, z, i, a[3] = {1, 2, 3}, b[3] = {4, 5, 6};
10:      x = array_mul(3, a, b);
11:      x = array_mul(1, a, b);
12:      y = array_mul(3, a, b);
13:      ...
14:      z = array_mul(3, a, b);
15:      ...
16:      for(i = 0; i <= 1023; i++)
17:          z += (int)sqrt(i);
18:      return (0);
19: }
```

た領域に設ける．そのため，スレッド性能の見積もりに使用するシフトレジスタは，命令区間によって区別されない．これは各命令区間毎の投機スレッドや並列事前実行スレッドによる高速化効果を把握したいわけではなく，プログラム全体としてのスレッドの有効性を把握したいためである．

続いて**例 4** のプログラムを例に，プログラム実行中の図 16 に示したシフトレジスタの具体的な動作を，**図 17** のシフトレジスタの状態図を用いて述べる．なお，図 17 中の記号 (a) から (f) は時系列順に割り振られている．例 4 のプログラムを実行し，11 行目までの命令実行を終えた時点で，MemoTbl には関数 array_mul(3, a, b) 及び array_mul(1, a, b) に対応する入出力セットが登録されている．その後，12 行目で関数 array_mul(3, a, b) に対する入力値検索に成功したとする．入力値検索に成功した時点で，シフトレ

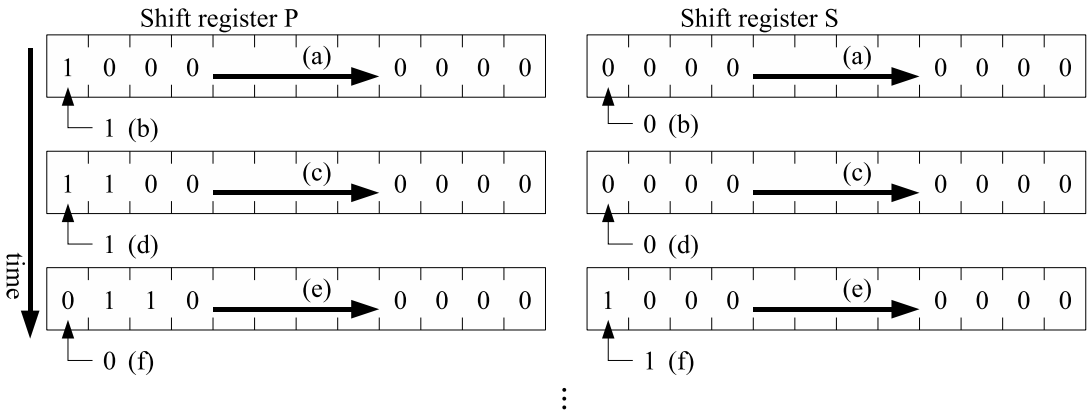

図 17: シフトレジスタの動作説明図

ジスタ P 及び S を 1 ビット右シフトする (a)．その後，当該関数に対して投機的再利用にも成功したとすると，シフトレジスタ P の最上位ビットに 1 をセットする．また，12 行目の関数の入出力セットは並列事前実行スレッドによって登録されたエントリではなかったとすると，シフトレジスタ S の最上位ビットに 0 をセットする (b)．14 行目の関数 array_mul(3, a, b) に対しても投機的再利用が成功したとすると，12 行目の場合と同様の操作を行う (c)(d)．以上より 14 行目の関数 array_mul に対する入力値検索が終了した時点では，P の上位 2 ビットに 1 が格納されており，S の全ビットは 0 である．引き続き命令を実行し，16 行目から 17 行目までで構成されるループ区間に到達したとする．このとき，並列事前実行スレッドが登録したエントリを用いて当該ループに対する入力値検索が成功したとすると，シフトレジスタ P 及び S の各ビットを 1 ビット右シフト (e) した後，1 回の入力値検索成功につきシフトレジスタ S の最上位ビットに，1 が登録される (f)．

提案モデルではシフトレジスタ P 及び S の有効ビットの数を比較する事により，各スレッドの有効性を判断し，スレッドの切り替えを行う．各スレッドの有効性の判断には以下の計算式を用いる．

$$N^P > a \tag{3}$$

$$N^S - N^P > b \tag{4}$$

なお，N^P 及び N^S はそれぞれシフトレジスタ P 及び S 中の有効ビットの数であり，a 及び b は定数である．式 (3) の条件を満たし，なおかつどのコアにも投機スレッドが割り当てられていない場合，投機スレッドが有効であると判断し，並列事前実行スレッドの 1 つを投機スレッドに切り替える．一方，式 (4) の条件のみを満たし，いずれかの

コアに投機スレッドが割り当てられている場合，投機スレッドより並列事前実行スレッドの方が有効性が高いと判断し，投機スレッドを並列事前実行スレッドに切り替える．投機スレッドの切り替えはシフトレジスタ S 中の有効ビット数に依存せず，シフトレジスタ P 中の有効ビットの数のみを切り替えの基準に用いる．式 (3) 及び (4) の両方の条件を満たさない場合，各コアに割り当てるスレッドの切り替えは行われない．現在の実装では，並列事前実行スレッドは複数のコアに割り当てることができるが，投機スレッドは 1 スレッドしか割り当てることができない．そのため，投機スレッドが有効であると判断した場合は，並列事前実行スレッドよりも投機スレッドを優先して割り当てる実装とした．

4.3 レジスタ及び主記憶の一貫性

これまでに説明した各スレッドが正しく命令を実行するためには，各コアがレジスタや主記憶の値を管理し，メインスレッドの命令実行に影響を与えないようにする必要がある．このうち，主記憶への書き込みについては，メインスレッド以外はその実行中に主記憶書き込み命令を検出すると，自身の Local MemoBuf に対して書き込みを行う．メインスレッドが主記憶を参照する際には，まず Local MemoBuf を参照し，読み出しアドレスに対応する値が Local MemoBuf に存在しない場合はキャッシュ及び主記憶から値を読み出す．なお各スレッドの実行結果が不要になり，スレッドの計算結果を squash する必要が生じた時には，各コアが Local MemoBuf に書き込んだ内容を削除する．このようにメインスレッド以外のスレッドはキャッシュや主記憶に対して書き込みを行わないため，主記憶値に矛盾が生じることはない．

また，レジスタへ値を書き込む際，メインスレッド及び並列事前実行スレッドは通常のレジスタセット（以後 **Regs** と表記）を使用し，投機スレッドと通常実行スレッドは SpRF に対してレジスタ書き込みを行う．SpRF は投機的に書き込んだ値を保持しているため，スレッドの切り替えが発生せず，投機スレッドや通常実行スレッドの計算結果が squash される場合は，SpRF に書き込んだ値も削除される．また，こうしたレジスタ書き込みに関する情報を管理するために，各コアは **RegMask(Register Mask)** と呼ぶビットマスクを持つ．RegMask の構成を**図 18** に示す．RegMask は，各レジスタから値を読み出す際，Regs と SpRF のどちらを参照すべきかを表すビット列で構成される．このうち，レジスタ番号に対応するビットが有効（ビットに 1 がセット）であれば，SpRF に投機的な値が書き込まれているため，SpRF から値を読み出す．一方，レジスタ番号に対応するビットが無効（ビットに 0 がセット）であれば，SpRF

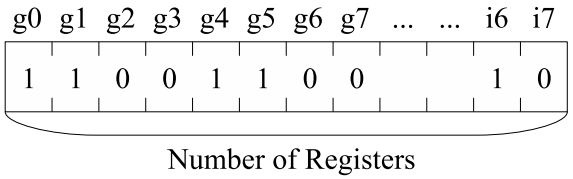

図 18: RegMask の構成

例 5 : SpRF と RegMask の説明用プログラム

```
1:  int a[3] = {1, 2, 3}, b[3] = {4, 5, 6}, c, d;
2:  int main(void) {
3:      ...
4:      c = sum(a, b);
5:      ...
6:      d = min(c, 6);
7:      ...
8:      return (0);
9: }
```

に投機的な値が書き込まれていないため，Regs から値を読み出す．また，SpRF に投機的な値を書き込んだ時には，RegMask の対応するレジスタ番号のビットを有効化する．次項からはプログラム**例 5** を用いて，各スレッドの実行中に SpRF や RegMask がどのように使用されるかについて詳しく説明する．

4.3.1 メインスレッド

メインスレッドは，自身及び投機スレッド，通常実行スレッドが割り当てられているコアの Regs 及び SpRF の双方に対してレジスタ書き込みを行い，それらが同一の内容になるようにする．これは，投機スレッドや通常実行スレッドが迅速に実行を開始できるようにするためである．例 5 のプログラムを実行する時のメインスレッドの動作について**図 19** を用いて説明する．図 19 は 4 行目までの命令実行を終え，5 行目以降の実行を開始した直後の各コアの状態を示す．4 行目の関数 sum に対しては計算再利用が適用でき，命令実行を省略できたとする．各コアの Regs 及び SpRF はコア 2 のレジスタ g3 を除き，同一の値を保持している．メインスレッドはプログラムの実行中，入力値検索を行っている時以外は並列事前実行スレッド以外を割り当てられているコアの Regs 及び SpRF に対して同一の値を書き込む．例 5 のプログラムでは，4 行目の

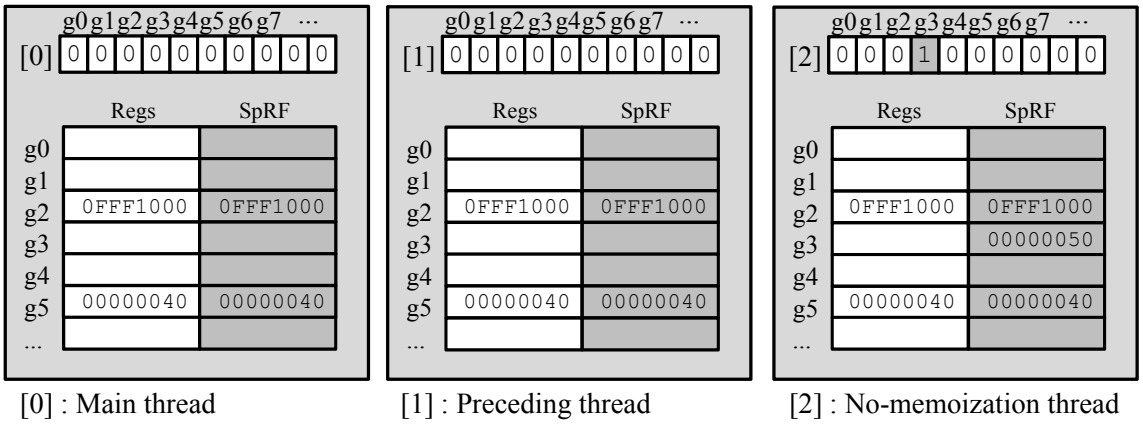

図 19: メインスレッドの動作説明

関数 sum を実行中に通常実行スレッドがレジスタ g3 に対して書き込みを行ったため，コア 2 の SpRF が保持している g3 の値は 00000050 となっており，自身の Regs や他のコアの Regs 及び SpRF の値とは異なる．そのため，コア 2 の RegMask のうち，g3 に対応するビットが有効化されている．引き続き命令実行を行い，6 行目で関数 min の呼び出し命令を検出する．ところが，まだコア 2 の SpRF に書き込まれている g3 の値に変化はなく，g3 に対応する RegMask のビットが有効である．このまま通常実行スレッドが命令区間 min を実行し，レジスタ g3 の値を読み出すと，誤った実行結果となってしまう．これを回避するため，入力値検索対象の命令区間を検出したとき，全てのコアの RegMask の各ビットを調べ，有効なビットが存在しているか否かを確認する．もしいずれかのコアの RegMask に有効なビットが存在していた場合，そのコアにメインスレッドが割り当てられていればその値を，それ以外のスレッドが割り当てられていればメインスレッドを割り当てられているコアの対応する Regs の値を全てのコアに書き込む．その後，RegMask の当該レジスタ番号に対応する有効ビットを無効化する．図 19 の例ではコア 2 の SpRF の g3 にコア 0 の Regs の g3 が記憶している値 0FFF1000 を書き込み，コア 2 のレジスタ g3 に対応する RegMask のビットを無効化する．

RegMask の有効ビットを調べた後，コア間でレジスタの値をコピーする必要が生じた際にはコピーオーバヘッドが発生する．このコピーオーバヘッドをできる限り隠蔽するため，メインスレッドはその実行中に発生したレジスタ書き込みの内容を，並列事前実行スレッドが割り当てられているコア以外に対して反映させる．そのため，命令を実行中に RegMask の対応ビットが有効なレジスタに対して書き込みが行われた場合，レジスタ値は上書きされ，メインスレッドの Regs の内容と同一になる．よって，対応する RegMask のビットを無効化することができ，一部のオーバヘッドを隠蔽で

きる．

4.3.2 投機スレッドと通常実行スレッド

投機スレッドと通常実行スレッドは，自身の SpRF に対してのみレジスタ書き込みを行う．これらのスレッドが動作するのは，メインスレッドが入力値検索を行っている間に限られる．入力値検索の際にレジスタ書き込みは発生しないため，投機スレッドや通常実行スレッドが動作している時にはこれらのスレッドが割り当てられているコアに対してメインスレッドからレジスタ書き込みが行われることはない．投機スレッドや通常実行スレッドの SpRF には，メインスレッドが入力値検索対象としている命令区間を検出する直前のレジスタ内容が格納されている．そのため，これらのスレッドはメインスレッドからそのプログラムカウンタの値をコピーするだけで，入力値検索対象の命令区間及びその後続区間の実行を開始できる．

次に，例 5 のプログラムと**図 20** を用いて，投機スレッドと通常実行スレッドの動作について説明する．プログラムを実行し，4 行目で関数 sum を検出すると，メインスレッドは入力値検索を行う．この時，投機スレッドは当該関数の出力を予測して MemoTbl から読み出し，後続区間の実行を開始する．投機スレッドが後続区間を実行中に，レジスタ g4 に対して書き込みを行う命令があったとする．すると，投機スレッドを割り当てられているコア 1 は自身の SpRF に対して書き込みを行うと同時にレジスタ番号 g4 に対応する RegMask のビットを有効化する．これと同様に通常実行スレッドも 4 行目の関数 sum を実行する過程でレジスタ g3 に対して書き込みを行う命令があったとすると，自身の SpRF に対して書き込みを行い，レジスタ番号 g3 に対応する RegMask のビットを有効化する．ここまでは図 20 の (A) に対応する．この後，メインコアが入力値検索に成功し，さらに投機的再利用にも成功したとする．すると，メインスレッドはコア 1 に割り当てられるため，コア 1 はこれまで投機スレッドが投機的な値を書き込むために使用していた SpRF を以後 Regs として扱う．この様子は図 20 の (B) に対応する．また通常実行スレッドを割り当てられているコア 2 の実行結果は不要となるが，既に SpRF に書き込んだ値は無効化されない．次の入力値検索対象区間を検出するまでに RegMask が有効，すなわち SpRF に書き込まれている値が他のコアの Regs や SpRF のものと異なる場合，次回の入力値検索を開始するまでにその SpRF に書き込まれている値を，メインスレッドが割り当てられているコアの Regs の値と同一の値となるように試みる．具体的には，まず命令実行の過程で，SpRF に対して上書きが発生すれば，メインスレッドの Regs の値と同一になる．また，キャッシュミス発生時及び実行に数サイクルを要する乗除算命令や浮動小数点演算命令の実行時など，命令

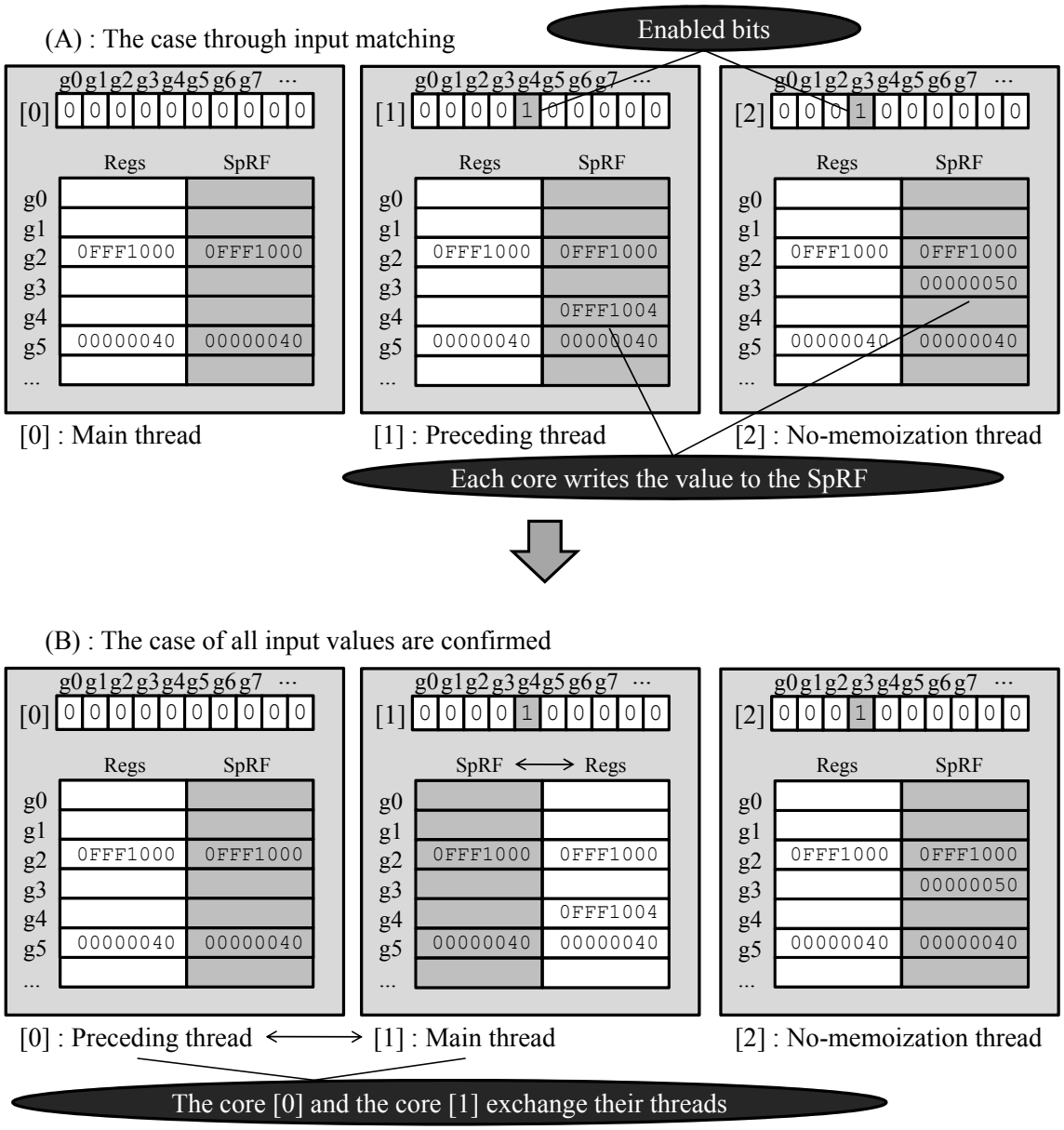

図 20: 投機スレッドと通常実行スレッドの動作説明

を実行しておらず，レジスタ書き込みポートが空いている際にも，RegMask のビットが有効となっているコアの SpRF に対して書き込みを行う．

このようにできる限り通常の命令実行の過程で全てのコアの Regs と SpRF がメインスレッドを実行しているコアの Regs の値と同一になるように試みる．しかし，全ての値をコピーできない場合も存在する．その場合は命令区間を検出し，入力値検索を開始する前にレジスタ値のコピーを行うため，オーバヘッドが発生する．以上の手順により，投機スレッドと通常実行スレッドは，レジスタの値に不整合が生じるのを回避

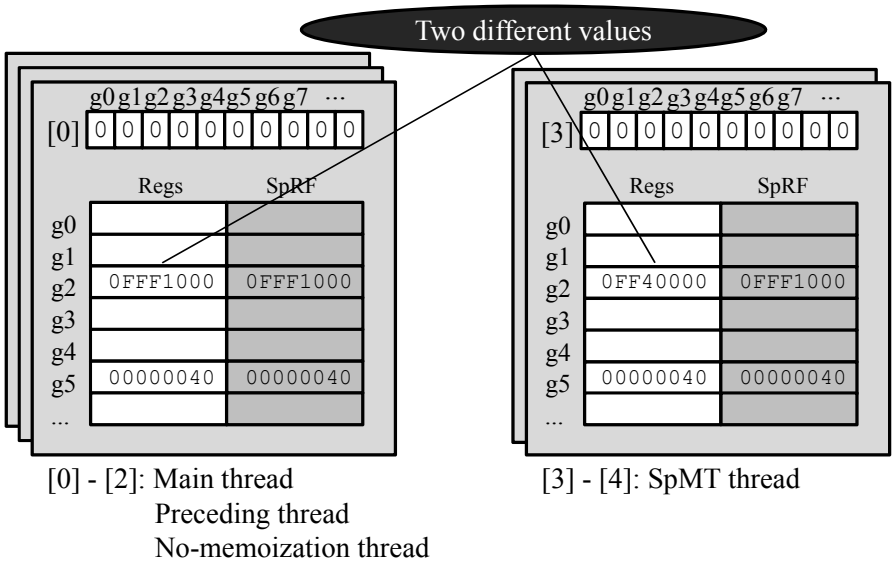

図 21: 並列事前実行スレッドの動作説明

する．

また，投機スレッドから並列事前実行スレッドへの切り替えは，入力値検索が終了した際に行う．もし，投機スレッドを割り当てられているコアが動作している時にスレッドを切り替えると，投機的実行は途中で終了してしまう．そのため，投機スレッドを割り当てられているコアが動作していない時に，割り当てるスレッドの切り替えを行う．並列事前実行スレッドは，命令区間の入力セットを予測し，予測した入力を Regs に書き込んでから命令実行を開始する．そのため，入力値検索終了後の投機スレッドが動作していない状況では RegMask や Regs，SpRF の状態に関わらず，並列事前実行スレッドを開始することができる．

4.3.3 並列事前実行スレッド

並列事前実行スレッドが割り当てられているコアは，自身の Regs を使用し他のコアのレジスタセットへの書き込みは行わない．また，メインスレッドが割り当てられているコアからレジスタの値を書き込まれる事もない．よって並列事前実行スレッドは，他のスレッドのレジスタの状態に関係なく，他のスレッドとは独立して命令を実行する．

次に**図 21** を用いて並列事前実行スレッドの動作について説明する．並列事前実行スレッドは他のスレッドとは非同期にレジスタ書き込みを行う．そのため，プログラム実行中に並列事前実行スレッドと投機スレッドを切り替える際には，これまで並列事前実行スレッドを割り当てられていたコアの Regs の内容を，メインスレッドの Regs

と同一の内容にする必要がある．したがってメインスレッドの Regs の値を，新たに投機スレッドを割り当てるコアにコピーしてからスレッドの切り替えを行う．ところが，スレッド切り替えにより，新しく並列事前実行スレッドを割り当てられるコアにメインスレッドから Regs の値をコピーする際，1 つのレジスタセットは合計 104 本のレジスタを備えるため，レジスタの値を全てコピーするには大きなオーバヘッドが発生する．そこでこのオーバヘッドを隠蔽するため，メインスレッドの Regs 値のコピーは入力値検索を行っている間に行う．1 回の入力値検索の間に全ての値をコピーできなかった場合は，次の入力値検索の間にコピーを行う．この過程を繰り返し，全ての値をメインスレッドを割り当てられているコアからコピーできた時，これまで並列事前実行スレッドを割り当てられていたコアに対して投機スレッドを割り当てる．

4.4 予測ポインタ

入力値検索対象となっている命令区間の後続区間を投機スレッドが実行し始めるためには，メインスレッドが入力値検索を行っている間に，検索対象の命令区間に対応する出力セットを MemoTbl から得る必要がある．このため，MemoTbl の各 RA エントリに**予測ポインタ**を追加する．予測ポインタは従来から RA が備えている W1 ポインタに類似した働きをする．予測ポインタはメインスレッドが入力値検索を行い，入力が部分的に一致したことを確認したとき，投機スレッドが W1 から対応する出力セットを読み出すために参照される．命令区間の入力は木構造を成して MemoTbl に登録されているため，入力が部分的に一致した時点では検索の終端となる RA エントリを一意に特定できない．この問題を解決するため，予測ポインタは入力値検索を行っている関数の入力に対応する W1 へのインデクスの候補を保持する．この候補は MemoTbl へ関数の入出力セットを登録した際に登録され，計算再利用に成功した際に更新される．

図 22 は命令区間の入力が，図上部の入力ツリー (b) で表される入力である場合に MemoTbl を検索する様子を示している．MemoTbl の RB や RA を参照する際，どのエントリを参照したかを RA に記憶していく．そして入力値検索に成功し，出力を MemoTbl の W1 から読み出す際に，終端の RA エントリが保持している W1 へのインデクスを，これまで記憶してきた全ての RA エントリが持つ予測ポインタの記憶領域に対して書き込む．これにより投機スレッドは，メインスレッドが入力値検索を行っている途中であっても，予測ポインタを参照する事により，入力セットに対応する出力セットの候補を読み出す事ができる．

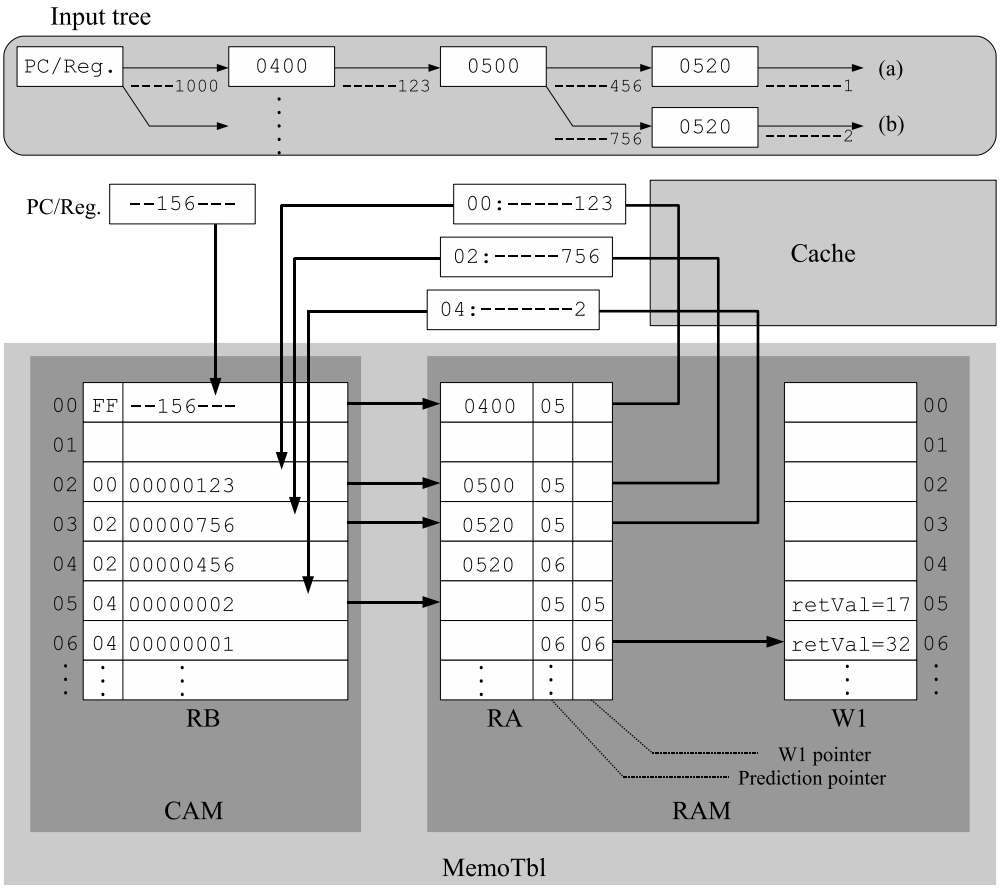

図 22: 予測ポインタ

5 評価

5.1 評価環境

前章までで述べた拡張を自動メモ化プロセッサシミュレータに実装し，シミュレーション評価を行った．シミュレータは単命令発行の SPARC V8 をベースとしている．シミュレーション時のパラメータを**表 3** に示す．なお，キャッシュや命令レイテンシは SPARC64-III[13] を参考にした．表 3 に示した切り替えに用いる閾値 a 及び b の値は，それぞれの値を変化させて評価を行い，最適と判断した値である．閾値 a 及び b の値を変化させても，実行サイクル数やスレッドの割り当て状況に大きな変化は見られなかったため，閾値 a 及び b は表 3 に示した値とした．また，ベンチマークプログラムには，Stanford ベンチマークと SPEC CPU95 ベンチマークを gcc-3.0.2(-O2 -mcpu=supersparc) 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．MemoTbl 内の RB に用いる CAM の構成は，ルネサステクノロジ

社の R8A20410BG[15] を参考にしている．R8A20410BG は 360MHz のクロック速度で動作し，1 秒間に 360M 回の検索が可能である．現在の一般的なプロセッサの動作クロックは 2GHz から 3GHz 程度である．そこで，CAM の 1 ラインを 32 バイトとし，レジスタと CAM の 1 ラインを比較するのに 9 サイクル，1 次データキャッシュと CAM の 1 ラインを比較するのに 10 サイクルを要するものとした．なお，RB エントリと主記憶の値とを比較する際，比較する値が 1 次データキャッシュに存在せず，2 次データキャッシュ及び主記憶へのアクセスが発生した場合には，そのアクセスレイテンシも生じる．

5.2 評価結果

評価結果のグラフは，左から順に

(N) ：メモ化を行わない場合
(M) ：メモ化のみを行う場合
(S) ：メモ化に並列事前実行を組み合わせた場合
(P) ：メモ化に提案手法である投機的再利用を組み合わせた場合
(HS) : (S) と (P) を組み合わせ，並列事前実行スレッドを静的に割り当てた場合
(HD) : (S) と (P) を組み合わせ，各スレッドを動的に割り当てた場合

の場合の総実行サイクル数であり，それぞれメモ化を行わない場合 (N) を 1 として正規化した．(S) 及び (P) は合計 3 つのコアを持つ．また (HS) は合計 5 つのコアを持ち，並列事前実行スレッドを静的に 2 スレッド割り当てた．なお，(HS) は，投機スレッドと並列事前実行スレッドを動的に切り替える事により得られる効果を確認するために掲載した．また (HD) は合計 5 つのコアを持ち，初期状態で並列事前実行スレッドを 2 スレッド割り当てた．なお，再利用オーバヘッドを削減する提案モデルの有効性を検証するため，提案手法を適用できないループ区間のみに対してのみオーバヘッド評価機構を使用し，関数に対してはオーバヘッド評価機構を使用しないこととした．Stanford ベンチマーク及び SPEC CPU95 ベンチマークを用いた評価結果をそれぞれ**図 23**，**図 24** に示す．図 23 及び図 24 の凡例は左上から順に，exec は命令の実行に要したサイクル数，reuse_ovh は再利用オーバヘッド，reg_copy はコア間でレジスタの値をコピーするのに要するコスト，D$1 は 1 次データキャッシュミスにより要したサイクル数，D$2 は 2 次データキャッシュミスにより要したサイクル数，window は SPARC アーキテクチャ特有のレジスタウインドウミスによって要したサイクル数である．

まず，図 23 の Stanford ベンチマークの評価結果について述べる．Stanford ベンチ

表 3: シミュレーションパラメータ

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
共有 MemoBuf	64 kBytes
Local MemoBuf	48 kBytes (×5)
MemoTbl CAM	128 kBytes
比較コスト（レジスタ，CAM 間）	9 cycles/32bytes
比較コスト（キャッシュ，CAM 間）	10 cycles/32bytes
ライトバック（MemoTbl からレジスタ，キャッシュ）	1 cycle/32bytes
SpRF	416 Bytes
レジスタコピー	1 cycle/32bits
入力の部分一致	1 entry
投機スレッドの有効性判定の閾値（a）	3
並列事前実行スレッドの有効性判定の閾値（b）	10

マークの全体的な傾向として，並列事前実行を行わないと，多くのプログラムで高速化不可能である点が挙げられる．その中で唯一 Puzzle のみが，メモ化 (M) により (N) と比較して 46.5%と大幅な高速化を実現している．また Perm や Towers など，再利用オーバヘッドにより (M) ではメモ化無し (N) に比べ，大幅に総サイクル数が増加しているプログラムも存在し，これらは並列事前実行を用いた (S) でもほとんどサイクル数を削減できていない．一方，(M) では (N) に比べ総サイクル数が増加していた Intmm や Mm，FFT といったプログラムでも，(S) の結果より並列事前実行による高速化を実

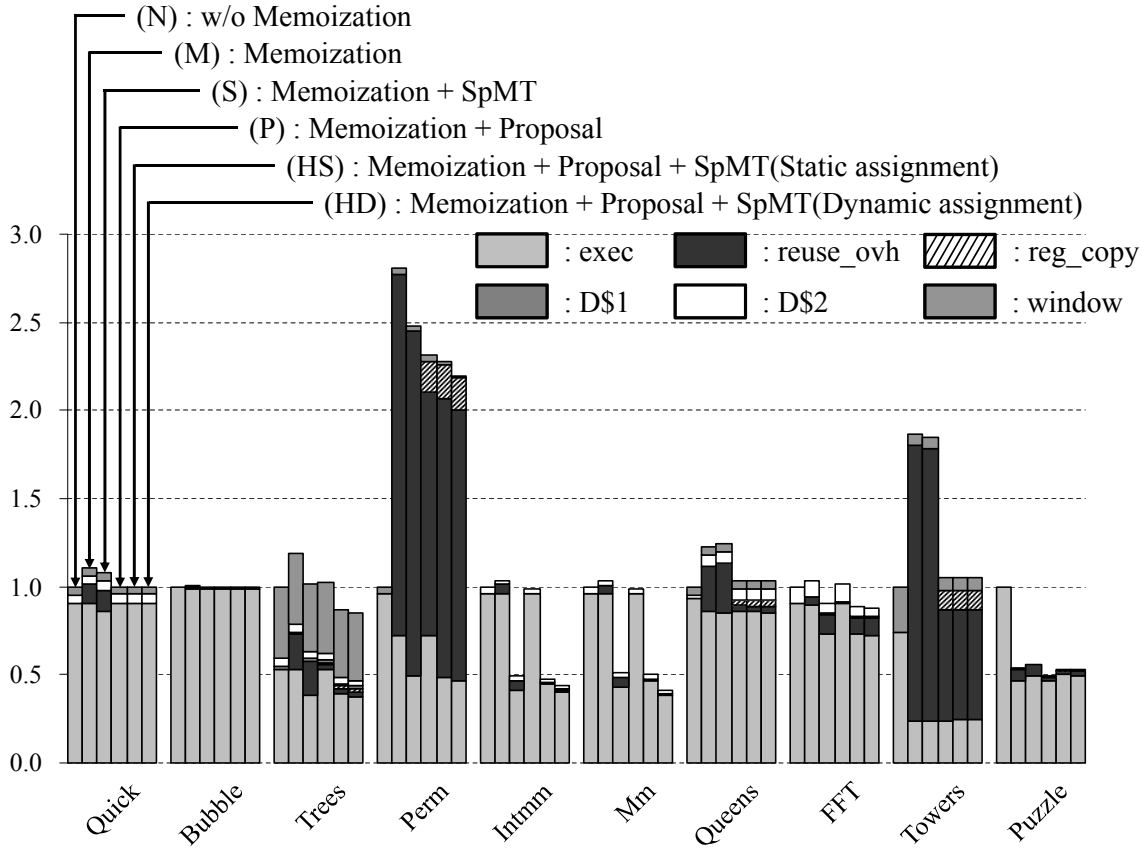

図 23: Stanford での評価結果

現している事が分かる．続いて再利用オーバヘッドを削減するモデル (P) について述べる．従来のメモ化 (M) により (N) に比べ総サイクル数が増加していた Trees，Queens，Towers では，再利用オーバヘッド削減モデル (P) により，総サイクル数を (N) と同程度にまで抑制することができた．しかし，Perm に対しては (P) でも (M) に比べ若干のサイクル削減に留まっており，総サイクル数は (N) に比べ大幅に増加している．

次に提案モデルの評価結果について述べる．まず，並列事前実行スレッドを静的に割り当てたモデル (HS) では，並列事前実行により高速化可能なプログラムについては，概ね (S) と同程度の総サイクル数となっている．また，(M) や (S) でサイクル数が増加したプログラムでは，再利用オーバヘッドを抑制でき，(P) と同程度の総サイクル数となっている．ところで，Trees では特徴的な結果がみられる．(S) では再利用オーバヘッドにより，命令の実行サイクル数が削減できているにも関わらず，総サイクル数は (N) とほとんど同等である．しかし，(HS) では，再利用オーバヘッドを削減することで，(N) よりも総サイクル数を削減できた．また，(S) と (P) を組み合わせ，投機スレッドと並列事前実行スレッドを動的に割り当てたモデル (HD) では，(HS) に比べ

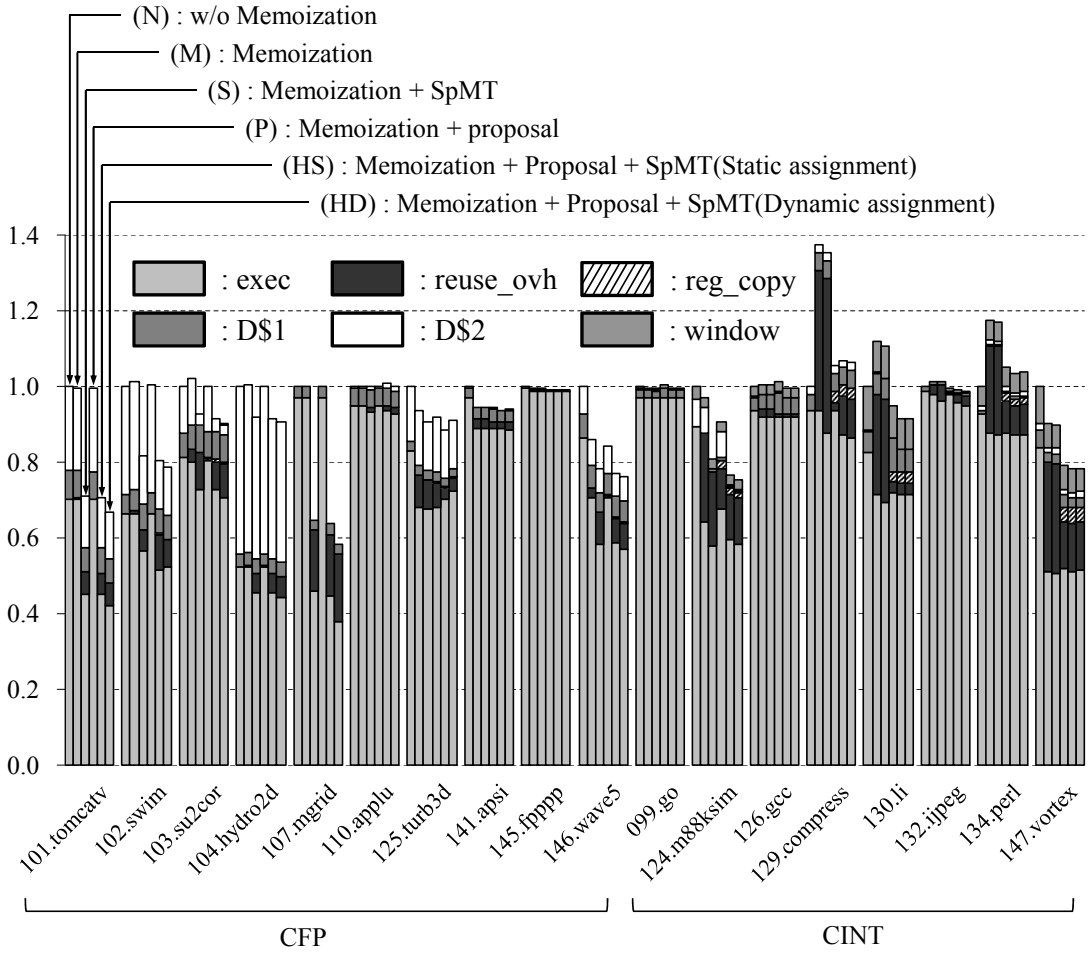

図 24: SPEC CPU95 での評価結果

て更なる高速化を実現しており，Quick，Perm，Queens，Towers 以外のプログラムで (N) よりも高速化を実現できた．

続いて，図 24 の SPEC CPU95 ベンチマークの評価結果について説明する．SPEC CPU95 では，Stanford ベンチマークのように従来のメモ化 (M) により，(N) よりも大幅に総サイクル数が増加しているプログラムは少ない．それでも，129.compress や 130.li，134.perl など，(N) よりも総サイクル数が増加しているプログラムも存在する．これらのプログラムに対しては，若干ではあるが計算再利用により命令実行に要するサイクル数を削減可能である．しかし，計算再利用により削減できたサイクル数を再利用オーバヘッドが大幅に上回ったため，(N) よりも総サイクル数が大きく増加した．また，(M) のみで高速化が可能なプログラムは 125.turb3d，146.wave5，124.m88ksim，147.vortex のみであり，最も高速化可能な 146.wave5 でも (N) と比べ 13.9%の高速化に留まっている．

一方，(M) のみではSPEC CPU95の多くのプログラムを高速化するのは不可能であるが，(S) では多くのプログラムに対して高速化を実現している．CFP ベンチマークではその傾向が顕著であり，110.applu と 145.fpppp を除き，並列事前実行による高速化を実現している．特に (S) では (N) と比較して，101.tomcatv では 28.9%，107.mgrid では 35.1%のサイクル数を削減できた．また (S) によりキャッシュミスが減少しているプログラムもみられることから，並列事前実行は，キャッシュプリフェッチ機構としても有効に働いたと考えられる．ただし，並列事前実行は 124.m88ksim を除く CINT ベンチマークに対しては有効でなく，147.vortex では (S) は (M) と比較して，若干サイクル数が増加した．147.vortex ではメモ化可能な関数が多く，(M) でも MemoTbl のエントリ数が十分ではない．よって，並列事前実行によりエントリの登録頻度が増し，有効なエントリも追い出されてしまうことにより，総サイクル数が増加したと考えられる．

また，提案モデルを並列事前実行と組み合わせずに用いたモデル (P) では，129.compress や 130.li，134.perl など，(M) や (S) でサイクル数が増加したプログラムに対して総サイクル数の増加をある程度抑制できている．さらに 124.m88ksim や 147.vortex では (M) よりも総サイクル数を削減できた．(S) と (P) の結果から分かるように，両方の手法により効果が得られるプログラムは 124.m88ksim 以外に存在せず，一般に (S) か (P) のうち，いずれか一方のモデルが有効であるプログラムが多い．こうした傾向を踏まえて (HS) 及び (HD) をみると，全てのプログラムで (S) 及び (P) のうち，サイクル数を削減できたモデルの結果と同程度か，それ以上に総サイクル数を削減できた．特に (S) により高速化が可能な CFP ベンチマークの多くや，(P) により高速化が可能な 124.m88ksim や 147.vortex では (M) に比べサイクル数を削減できた．

参考として，関数に対してもオーバヘッド評価機構を用いた場合の評価結果を**図 25** に示す．(HD) と同等のコア数で評価を行うため，5 コアのうち，1 コアをメインコア，4 コアを並列事前実行コアとしたモデル (S4) を評価に用いた．また (HD) は関数にもオーバヘッド評価機構を用いた上で，投機スレッドや通常実行スレッドを動作させた．図 25 の結果より，提案モデル (HD) は 4 つのコアを並列事前実行コアとして用いた (S4) よりも平均的にはサイクル数を削減できている．特に 130.li や 147.vortex 等，並列事前実行が有効でないプログラムに対しては (N)，(M)，(S4) に比べ，高速化を実現できた．オーバヘッド評価機構を関数にも適用することで，129.compress や 134.perl では (N) に比べ大幅にサイクル数が増加する事を抑制できた．こうしたプログラムに対しても提案手法 (HD) は有効で，オーバヘッド評価機構では削減できなかった再利用オーバヘッドも削減できた．なお，再利用オーバヘッド削減手法が有効でないプログラム

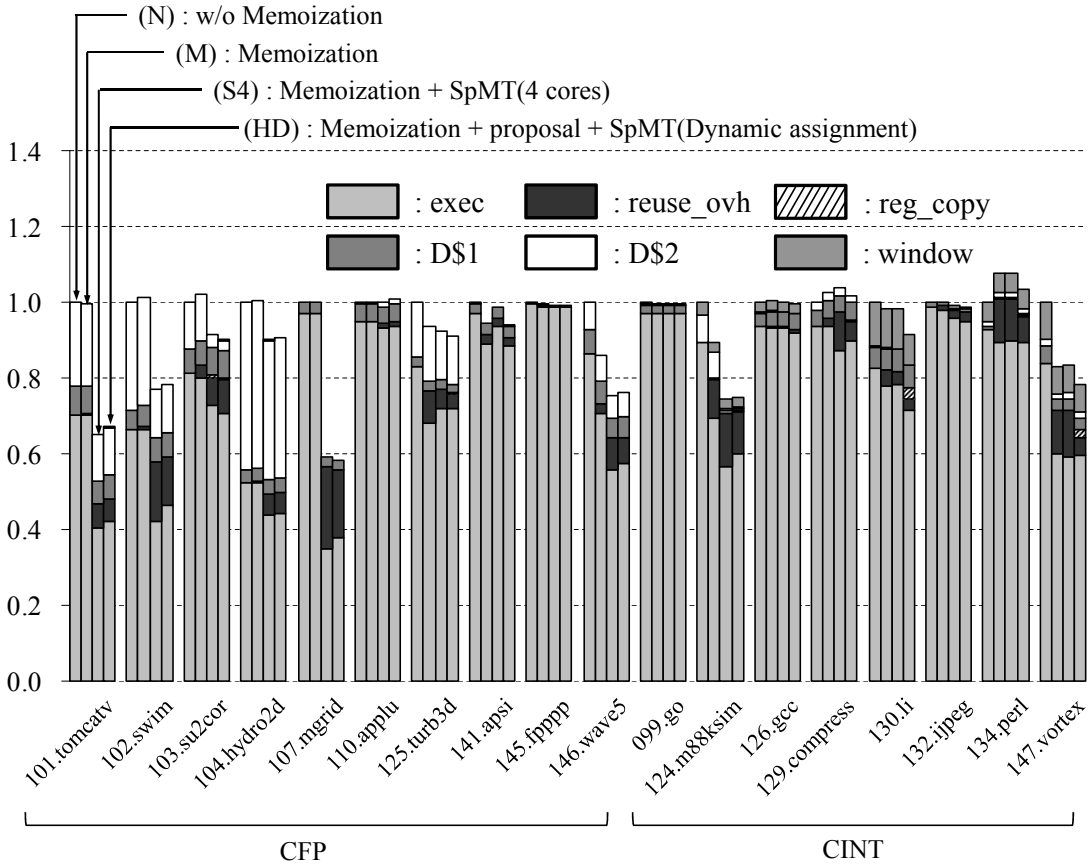

図 25: SPEC CPU95 での評価結果（関数に対するオーバヘッド評価機構有効）

でも，103.su2cor や 107.mgrid のように，(HD) の方が (S4) よりもサイクル数を削減できているものも存在する．これは再利用オーバヘッド評価機構により，本来効果の得られる命令区間に対する計算再利用が抑制されてしまったためと考えられる．

結果をまとめると，(HD) は (N) に比べ Stanford ベンチマークでは最大で 47.2%，平均で 6.2%のサイクル数を削減でき，SPEC CPU95 ベンチマークでは最大で 41.5%，平均で 11.1%のサイクル数を削減できた．また既存のオーバヘッド評価機構を併用し，提案モデルと同数のコアに並列事前実行スレッドを最大限割り当てた場合 (S4) との比較でも，SPEC CPU95 ベンチマークで平均 1.8%のサイクル数を削減でき，良好な結果が得られた．

5.3 評価結果の比較・考察

まず，各ベンチマークプログラム毎の入力値検索成功率と削減サイクル数との関係について考察する．**表 4** に SPEC CPU95 ベンチマーク及び Stanford ベンチマークに

表 4: 入力値検索成功率

SPEC CPU95		(M)		(HD)
101.tomcatv		2.2%		15.3%
102.swim	*	0.0%		10.8%
103.su2cor	*	0.0%		10.5%
104.hydro2d	*	0.7%		3.2%
107.mgrid	*	0.0%		43.2%
110.applu	*	3.3%	*	0.8%
125.turb3d		0.5%		1.5%
141.apsi		26.9%		61.7%
145.fpppp		2.2%		2.1%
146.wave5		4.4%		6.0%
099.go		0.0%		0.0%
124.m88ksim		47.8%		18.1%
126.gcc	*	0.4%	*	0.6%
129.compress	*	0.4%	*	0.6%
130.li	*	10.2%		8.6%
132.ijpeg	*	2.0%		61.7%
134.perl	*	1.7%	*	1.8%
147.vortex		18.2%		23.2%

Stanford		(M)		(HD)
Quick	*	0.0 %	*	3.1%
Bubble	*	0.0%		0.2%
Trees	*	0.0%		3.6%
Perm	*	47.2%	*	70.8%
Intmm	*	0.0%		7.4%
Mm	*	0.0%		13.2%
Queens	*	1.3%	*	1.4%
FFT	*	11.1%		13.9%
Towers	*	46.5%	*	46.1%
Puzzle		2.8%		2.9%

ついて，メモ化 (M) のみを行うモデル及びスレッドを動的に割り当てるモデル (HD) の入力値検索成功率を示す．入力値検索成功率とは，検索が行われた回数と検索成功回数の比であり，オーバヘッド評価機構によって入力値検索を行うべきでないと判断され，検索自体が行われなかった場合は検索が行われた回数に含めない．なお，表 4 中の*は (M) 及び (HD) の総サイクル数が，メモ化無し (N) よりも増加したプログラムである．図 23，図 24，表 4 から分かるように，入力値検索成功率と総サイクル数の間には必ずしも相関関係が無い．これは削減可能なサイクル数が，メモ化可能な命令区間の長さに依存するためと考えられる．例えば Perm や Towers では (M)，(HD) ともに入力値検索成功率は非常に高いが，総サイクル数は (N) よりも大幅に増加している．

特に Perm ではプログラム全体の命令実行サイクル数 exec が 100 万サイクル程度であるにも関わらず，入力値検索対象となるループ区間は 10 万回程度実行される．Perm では計算再利用を適用することにより，ループ区間に対してオーバヘッド評価機構を使用しているにもかかわらず，有効に働かない．そのため (M) や (HD) では，(N) と比べて大幅にサイクル数が増加したと考えられる．Perm ほど極端ではないものの，同様の傾向は Towers でもみられる．計算再利用による効果の得られない命令区間に対しては，当該命令区間に対して計算再利用を適用することを回避するオーバヘッド評価機構によって，入力値検索を行わないようにするのが理想である．しかし，Perm や Towers のようにプログラム全体の実行サイクル数に比べ，ループの呼び出し回数が極端に多く，十分な時間の経過を待たずに MemoTbl に登録したエントリが次々と削除されていく．そのため，各 RF エントリが備えているオーバヘッド評価用のシフトレジスタの値も頻繁に初期化され，オーバヘッド評価機構が十分な効果を発揮しなかったと考えられる．

なお，入力値検索成功率と高速化可能なサイクル数との間には必ずしも相関関係は無いものの，124.m88ksim や 147.vortex 等，計算再利用により高速化を実現しているものについては，10%以上の入力値検索成功率となっている．また (M) ではほとんど高速化できていないが (HD) による高速化が可能なプログラムでは，(M) では 1%以下であった入力値検索成功率が，(HD) では数 10%以上に向上している．特に 107.mgrid の入力値検索成功率は 43.2%となっており，大幅に入力値検索成功率が向上している．よってある程度入力値検索成功率が高くないと，計算再利用の効果も得られていない事も分かる．

続いて，投機スレッドと並列事前実行スレッドを各コアに動的に割り当てる仕組みが有効に動作したか否かを検証するために，全体の命令実行サイクル数のうち，投機スレッドが割り当てられていたサイクル数と投機スレッドの代わりに，並列事前実行スレッドが割り当てられていたサイクル数の分布を調査した．その結果を**図 26** に示す．なお，調査は特徴的な傾向を持つプログラムに限定して行った．まず並列事前実行が有効な 102.swim では，命令実行中にほとんど投機スレッドが割り当てられず，並列事前実行スレッドが割り当てられている．また，並列事前実行と投機的再利用の両方が有効な 124.m88ksim では，プログラム実行中に数回のスレッド切り替えが発生し，適切なスレッドを選択しながら命令を実行している．一方 126.gcc では，頻繁に割り当てるスレッドの切り替えが発生しているものの，元々投機スレッドと並列事前実行スレッド共に有効でない．そのため計算再利用による高速化が見込めず，全体の性能

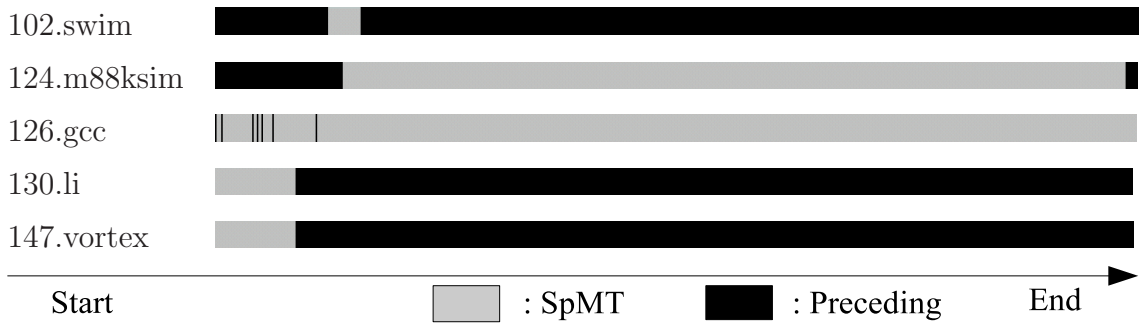

図 26: スレッドの割り当て状況

表 5: 投機的再利用に関する情報

	予測成功率	平均入力比較数	削減サイクル数
124.m88ksim	20.4%	1.67	0.2%
130.li	11.8%	1.92	0.1%
147.vortex	35.2%	3.37	3.1%

にはほとんど影響を与えない．また，投機スレッドが有効な 130.li や 147.vortex では，いずれのプログラムでも実行開始直後から暫くの間，並列事前実行スレッドが割り当てられる．その後は数回のスレッド切り替えは発生するものの，概ね投機スレッドが割り当てられている．以上の結果から，プログラムの特徴に応じて，その実行中に適切なスレッドを選択できている事が確認できた．

最後に投機スレッド及び通常実行スレッドによる効果について検証する．まず，投機スレッドが予測ポインタを使用して正しい出力を読み出した割合である予測成功率，各命令区間に計算再利用を適用する際に比較すべき入力数の平均，投機スレッドによって削減できたサイクル数の割合を**表 5** に示す．なお，表 5 には投機スレッドによる効果がある程度得られるプログラムに限定して掲載した．まず，比較的投機スレッドによる効果が得られる 147.vortex での予測成功率は 35.2%である．一方，124.m88ksim や 130.li では 10%から 20%程度の予測成功率であるものの，投機スレッドを用いた事による削減サイクル数はごく僅かに留まっている．この原因は，投機スレッドが命令区間を実行開始直後に関数の call 命令を検出し，命令を実行できずに停止する場合が非常に多いためである．例 6 は 130.li のソースコードの一部である．このうち，3 行目の関数 xlsave は投機的再利用の成功回数が最も多い関数である．例 6 の 3 行目の関数 xlsave を検出後，投機スレッドがその後続区間を実行する．しかし，4 行目に関数

例 6：投機的再利用の効果が無いプログラムの例（130.li）

```
1:  LOCAL NODE *evalhook(NODE *expr) {
2:  ...
3:  oldstk = xlsave(&ehook,&ahook,&args,(NODE **)NULL);
4:  args = consa(expr);
5:  rplacd(args,consa(xlenv));
6:  ...
```

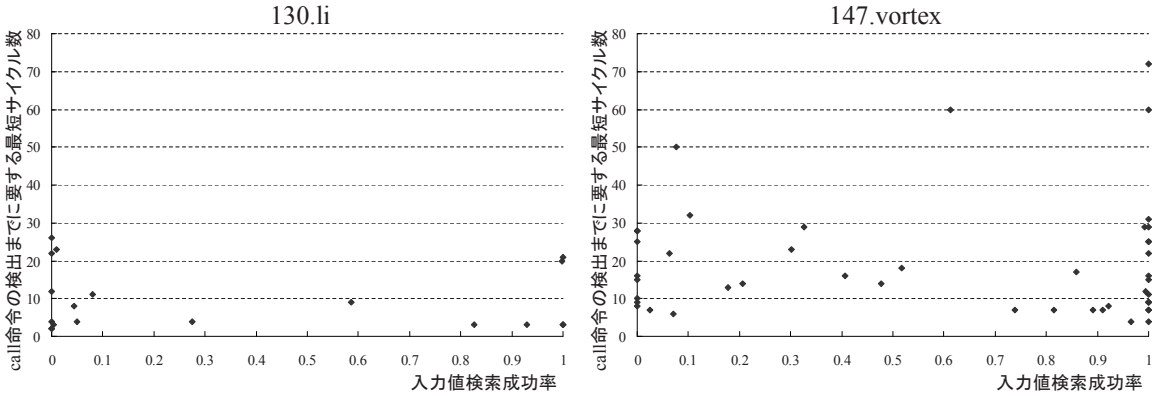

図 27: 入力値検索成功率と call 命令の検出までに要する最短サイクル数

consa の呼び出し命令が存在するため，投機スレッドは実行開始直後に停止してしまう．このような場合，投機的再利用を適用しても全く高速化は見込めず，予測成功率が高いと削減サイクル数も大きいとは限らない事が分かる．4 行目の関数 consa に対する 5 行目の関数 rplacd も，同様である．一方，147.vortex ではプログラムの実行開始直後からしばらくの間に実行される命令区間に対して投機的再利用が有効であり，比較的削減サイクル数も大きい．147.vortex は，データベースのプログラムという特性上，大きなサイズの構造体を入力とする関数が多数存在し，検索オーバヘッドも大きい．また 124.m88ksim や 130.li に比べ，実行開始直後に投機スレッドが停止してしまう場合も少ない．そのため，投機的再利用が適用できた際の効果は大きく，比較的投機スレッドによる高速化を実現できたと考えられる．

また投機スレッドとは異なり，通常実行スレッドは SPEC CPU95 CINT の多くのプログラムに対して有効である．通常実行スレッドは命令区間の入力セットに対応する出力セットを予測する必要がないため，入力値検索失敗時に発生する再利用オーバヘッドを確実に隠蔽できる．**図 27** に 130.li 及び 147.vortex に含まれる関数のうち，再利用オーバヘッド評価機構により，計算再利用による効果が得られると判断された関

数の入力値検索成功率と，関数の実行開始後に別の関数の call 命令を検出するまでに要する最短サイクル数の分布を示す．図 27 に示した 2 つのプログラムでは，グラフの左下にサンプルが集中している．そのため，これらのプログラム内には，計算再利用による効果が得られる関数でも，入力値検索成功率が低い関数は多数存在する事が分かる．また平均的には，入力値検索が終了する前に call 命令を検出し，通常実行スレッドは命令実行を停止してしまう可能性も高い．表 5 及び図 27 から分かるように，実際に 130.li や 147.vortex では，平均入力比較数の検索を終える前に call 命令が検出される関数は多数存在する．しかし通常実行スレッドは投機スレッドとは異なり，入力の一部が一致する事を確認しなくても命令実行を開始できるため，もし入力値検索が終了する前に call 命令検出により命令実行が停止しても，一部のオーバヘッドは隠蔽できる．そのため検索開始後，入力の一部が一致するまでのオーバヘッドを全く隠蔽できない投機スレッドとの間で削減可能なサイクル数に大きな差が開き，通常実行スレッドによる効果が投機スレッドによる効果を上回ったと考えられる．

6　おわりに

本論文では，既存の自動メモ化プロセッサを拡張し，複数のコアを有効に利用して自動メモ化プロセッサの高速化を図る手法を提案した．提案手法の有効性を確認するため，Stanford ベンチマーク及び SPEC CPU95 ベンチマークを用いて評価を行った．その結果，Stanford ベンチマークでは最大で 47.2%，平均で 6.2%，SPEC CPU95 ベンチマークでは最大で 41.5%，平均で 11.1%のサイクル数を削減でき，提案手法の有効性が確認できた．

本研究の今後の課題として，まず短期的な目標には，次の 2 つの課題が挙げられる．1 つ目の課題として，現段階の実装では非常に単純な実装であり，予測成功率があまり高くない予測ポインタの改良が考えられる．4 章の図 22 から分かるように，現在の実装では各 RA エントリの予測ポインタに，最後に登録した入力セットに対応する W1 ポインタの値を登録している．この実装だと入出力セットによっては，実際に入力値検索が成功した時に用いる W1 ポインタの値と投機スレッドが参照する予測ポインタの値が一致せず，予測が外れる場合も多い．そこで，各 RA エントリに複数の W1 ポインタの値を記憶可能にしておき，各 W1 ポインタの利用頻度やそれらを用いた場合の予測成功率等に基づいて，投機スレッドが W1 から実際に読み出す出力セットを決定するといった仕様変更が考えられる．

2 つ目の課題として，前章の終わりでも述べたように，投機スレッドの仕様を改良

する事が考えられる．その一案として，現在の実装では１つのコアにしか割り当てる事のできない投機スレッドを，他のコアにも割り当てられるように改良する事が考えられる．現段階では投機スレッドにより，最大でも数%程度しかサイクル数を削減できない．サイクル数を削減できない理由の一つに，投機スレッドが命令区間を実行中に関数を検出した時，その時点で実行を停止してしまう事が挙げられる．そこで複数のコアに投機スレッドを割り当てる事により，投機スレッドが関数を検出した際にそのスレッドは当該関数に対する入力値検索を行い，それとは別の投機スレッドが当該関数の後続区間を実行できるように現在のモデルを拡張する．これにより投機スレッドは実行中に関数を検出しても停止せず，次々とその後続区間を実行可能となり，更なる性能向上が可能であると考えられる．

最後に，本研究の長期的な目標としては，ソフトウェアにより自動メモ化プロセッサを支援し，更なる高速化を目指す事が考えられる．バイナリのみではなく，ソースコードが公開されているプログラムについては，プログラム内にメモ化のためのヒント情報を埋め込む等のソフトウェアによる支援を行う事で，更なる高速化が可能であると考えられる．このような研究は，既にある程度行われており，有用であることが示されている．しかし，それをマルチスレッドへ応用した場合については未だ検証されていない．そこで，並列事前実行の際に用いるストライド値をプログラム内に埋め込んで，並列事前実行により確実に有効な入出力セットを MemoTbl へ登録できるようにする等，ソフトウェアによるメモ化とマルチスレッド技術を組み合わせることで，新たな知見が得られると考えられる．

謝辞

本研究のために指導教員として多大な御尽力をいただき，日頃から熱心なご指導を賜った名古屋工業大学の松尾啓志教授，津邑公暁准教授に深く感謝致します．また，たびたびご検討・助言をして頂いた同大学の齋藤彰一准教授，松井俊浩助教にも深く感謝致します．最後に，本研究の際に多くのご協力をして頂いた松尾・津邑・齋藤研究室の学生の皆様，中でも本研究に関して色々な相談や議論を通じ，多大なる支援を頂いた高木伴彰氏，池谷友基氏，板橋世里華氏をはじめ，本論文の作成にあたり，校正をして頂いた方々に深く感謝の意を表します．

参考文献

[1] Intel Corp.: First the Tick, Now the Tock: Next Generation Intel Microarchitec-

ture (Nehalem), Intel Whitepaper (2008).

[2] Sony Computer Entertainment: *Cell Broadband Engine Architecture*, 1.01 edition (2006).

[3] Sun Microsystems, Inc.: *UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007* (2007).

[4] Tilera Corporation: *Product Brief: TILE64 Processor* (2007).

[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] Kamiya, Y., Tsumura, T., Matsuo, H. and Nakashima, Y.: A Speculative Technique for Auto-Memoization Processor with Multithreading, *Proc. 10th Int'l. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'09)*, pp. 160–166 (2009).

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

[10] Swadi, K., Taha, W., Kiselyov, O. and Pasalic, E.: A Monadic Approach for Avoiding Code Duplication when Staging Memoized Functions, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, ACM Press (2006).

[11] Beame, P., Impagliazzo, R., Pitassi, T. and Segerlind, N.: Memoization and DPLL: Formula Caching Proof Systems, *Computational Complexity, Annual IEEE Conference on*, Vol. 0, p. 248 (2003).

[12] 森本武資, 岩崎英哉, 竹内郁雄: 枝刈り機構とメモ化機構をもつ言語, コンピュータソフトウェア, Vol. 21, No. 4, pp. 55–60 (2004).

[13] HAL Computer Systems/Fujitsu: *SPARC64-III User's Guide* (1998).

[14] Barli, N., Tashiro, D., Iwama, C., Sakai, S. and Tanaka, H.: A Register Communication Mechanism for Speculative Multithreading Chip Multiprocessors, *In Proc. of the SACSIS 2003*, Vol. 2003, No. 8, pp. 275–282 (2003).

[15] ルネサステクノロジ: ニュースリリース: R8A20410BG (2009).