卒業研究論文

再利用表分割による
自動メモ化プロセッサのヒット率向上

指導教員　　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 19 年度入学　19115156 番

山田 龍寛

平成 23 年 2 月 8 日

再利用表分割による自動メモ化プロセッサのヒット率向上

山田 龍寛

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，ある命令区間の入出力を表に登録しておき，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，過去に登録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．

自動メモ化プロセッサでは，関数とループを再利用可能な命令区間としており，この 2 種類の命令区間は検出方法や登録方法が異なっている．関数は call 命令の実行で検出され，その関数の return 命令後に入力と実行した結果を表に登録する．一方で，ループは後方分岐命令を実行することで検出され，次に同一の後方分岐命令を実行後に入力と実行した結果を表に登録する．しかし，ループの入力の 1 つであるイタレータ変数が単調に変化していくため，再利用することができない．そこで，自動メモ化プロセッサでは，入力を予測し投機実行コアによって事前実行を行い，表に登録しておくことでループ区間の再利用を試みている．また，これら 2 種類の命令区間は入出力情報の特性も異なっている．関数は，過去と同じ入力で呼び出される箇所が，プログラム中において局所的ではないため，関数の入出力情報はなるべく長い期間表に留めておく方が有効だと考えられる．一方，ループの入出力情報は，同じループ区間内で過去と同じ入力で実行される可能性は低く，入出力情報の有効な期間は局所的で，長い期間表に留めておくのは有効では無いと考えられる．

こういった性質の違いがあるにも関わらず，従来の自動メモ化プロセッサでは単一の表に関数とループのエントリが混在し，どちらの入出力情報かを区別することなく同一のアルゴリズムで表に対するあらゆる操作を行っている．そのため，入出力情報の性質の差異によって表を有効に利用できていない可能性がある．

本研究の目標は，これら入出力情報の性質の差異によって生じる不利益を軽減することで，再利用の成功率を高め，自動メモ化プロセッサの高速化を図ることである．そこで，関数の入出力情報とループの入出力情報をどちらのものかを区別して別々に扱う手法を提案する．

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手法のモデルを実装し，SPEC CPU95 ベンチマークでシミュレーションによる評価を行った．その結果，通常通り命令を実行するのと比較し，平均従来手法では平均 13.9%，最大 39.5%のサイクル数の削減だったのに対し，提案手法では平均 14.8%，最大 41.2%のサイクル数の削減に成功し有効性を確認できた．

再利用表分割による自動メモ化プロセッサのヒット率向上

目次

1 はじめに 1

2 研究背景 2

2.1 自動メモ化プロセッサ 2

2.1.1 再利用機構の構成 2

2.1.2 再利用機構の動作例 5

2.2 並列事前実行 8

2.3 パージアルゴリズム 9

2.3.1 TSID パージ 10

2.3.2 RFID パージ 11

3 提案 11

3.1 提案手法 11

3.2 MemoTbl の分割 13

3.2.1 境界型 13

3.2.2 ユニット分割型 14

4 実装 15

4.1 実装の概略 15

4.2 各専用再利用表のパージアルゴリズム 16

4.2.1 関数用のパージアルゴリズム 17

4.2.2 ループ用のパージアルゴリズム 18

4.3 多数分割を行うユニット分割型 18

5 評価 19

5.1 評価環境 19

5.2 評価結果 20

6 おわりに 24

謝辞 24

参考文献 25

1　はじめに

これまで，さまざまなプロセッサ高速化手法が提案されてきた．ゲート遅延が支配的であった時代には，微細化による高クロック化で高速化を実現できた．しかし配線遅延の相対的な増大に伴い，高いクロック周波数だけでは高速化を実現しにくくなったことで，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た．また，近年は電力効率と性能向上を両立させる観点から，複数コアを搭載したマルチコアプロセッサが主流となりつつあり，今後集積度の向上に伴ってコア数も増大していくと考えられている．これらはいずれもプログラム実行の並列化という方法で高速化を図るものであるが，本研究はそれとはまったく異なる計算再利用という手法に着目する．

計算再利用には，ハードウェアによるものとソフトウェアによるもの，またその両方によるものなど，様々なものが提案されている．しかしソフトウェアによる計算再利用は制約が大きく，オーバーヘッドも大きい．そこでハードウェアによるものが研究され，専用のハードウェアを用いることでバイナリの変更なしに既存のプログラムを実行できる自動メモ化プロセッサ [1] が提案されている．本論文では，この自動メモ化プロセッサのさらなる高速化手法を提案する．自動メモ化プロセッサは，関数及びループを計算再利用可能な命令区間とみなし，実行時にその入出力を MemoTbl と呼ばれる表に記憶しておく．そして再び同一命令区間を実行しようとした際に，その入力と過去に表に登録しておいた入力とを比較し，それが一致すれば過去の出力を読み出し，その命令区間の実行を省略することができる．既存の自動メモ化プロセッサでは，関数とループの入出力を同じ MemoTbl に区別なく記憶していたが，本研究では，入出力を登録する MemoTbl を分割して，関数とループの入出力を別々に記憶し，それら 2 種類の入出力を別々に扱う．これにより，従来モデルにおいて発生しうる，一方の登録によって他方の入出力を追い出してしまうなどといった，どちらかの登録によってもう一方の入出力情報が不利益を被るという状況を減らすことで，有限の表を有効に活用し再利用の成功率を高め高速化を図る．

以下，2 章では本研究が扱う自動メモ化プロセッサの動作モデルについて述べる．3 章では，MemoTbl を分割し，関数とループの入出力を別々に扱うことでことで高速化を図る手法を提案し，4 章でその実装方法について説明する．5 章で本提案手法の評価を行い，最後の 6 章で結論を述べる．

2　研究背景

本章では本研究で取り扱う自動メモ化プロセッサについて，その高速化方針と動作原理を解説する．

2.1　自動メモ化プロセッサ

メモ化 (Memoization)[2] とは，関数やループといったプログラム中の命令区間に対して，その入出力を計算再利用可能な状態で記憶しておくことである．このメモ化を，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行うプロセッサとして考案されたのが自動メモ化プロセッサである．

2.1.1　再利用機構の構成

自動メモ化プロセッサは単命令発行の SPARC V8 をベースアーキテクチャとしている．自動メモ化プロセッサのハードウェア構成を**図 1** に示す．自動メモ化プロセッサは通常のプロセッサと同様に，コア中には ALU，レジスタ (Reg)，1 次データキャッシュ (D$1)，コア外に共有の 2 次データキャッシュ (Shared D$2) を持つ．またそれら以外に，自動メモ化プロセッサ独自の機構として，命令区間およびその入出力を記憶しておく表である MemoTbl 及び再利用機構を管理するための機構 (Reuse System)，MemoTbl の作業用の小さなバッファである MemoBuf を持つ．MemoTbl はサイズが大きく，コアからのアクセスコストが大きい．そのため，入出力を検出する度に MemoTbl へアクセスするとオーバーヘッドが大きくなる．そこで，このオーバヘッドを軽減するため，作業用の小さなバッファである MemoBuf をコアの内部に設ける．検出された入出力を MemoTbl ではなく，MemoBuf に格納しつつ当該命令区間を通常実行し，実行終了時に MemoBuf の内容を一括で MemoTbl に格納する．この MemoBuf の働きにより，MemoTbl へのアクセス回数を減らしオーバーヘッドを削減している．なお，入力には関数の引数はもちろんのこと，当該命令区間で発生した主記憶参照も全て含まれる．出力には，関数の返り値及び当該命令区間で発生した主記憶書き込みが含まれる．

MemoTbl と MemoBuf の詳細な構成を**図 2** に示す．MemoBuf は複数のエントリを持ち，1 エントリには 1 つの命令区間の入出力セットを登録することができる．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を記憶する Read，出力セットを記憶する 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

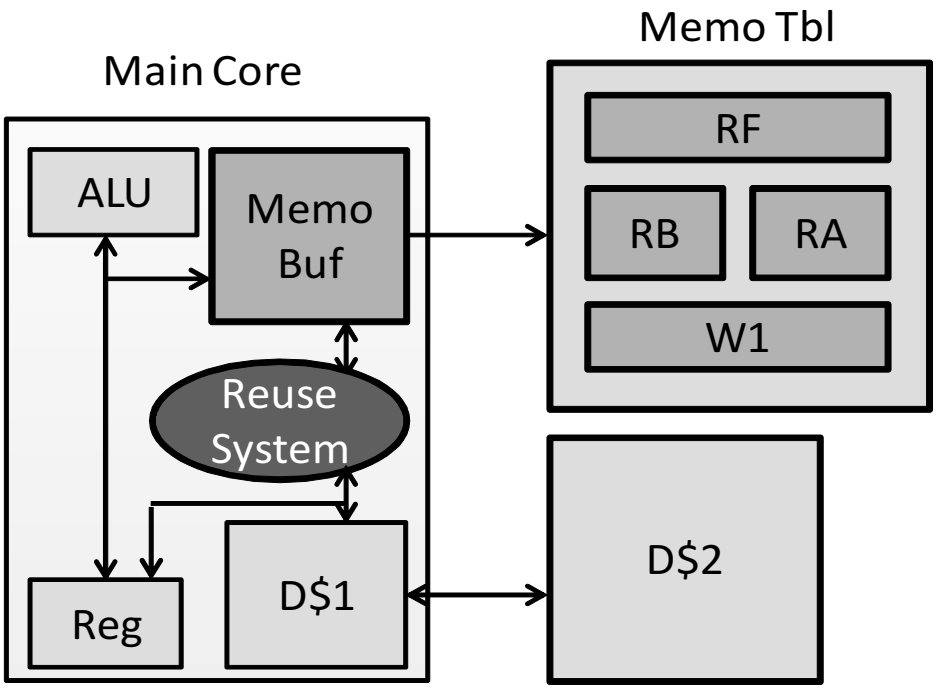

図 1: 自動メモ化プロセッサのハードウェア構成

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，ループ区間の後方分岐命令を再度実行した場合は，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する．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．

MemoTbl は，命令区間を記憶する表 RF，入力値を記憶する表 RB，入力アドレスを記憶する表 RA，及び出力値を記憶する表 W1 から構成される．RF は，各再利用対象命令区間に対応する行を持っており，その行番号を RFindex と定めている．また，各行にはメモ化のためのフィールドがあり，それぞれ関数及びループの区別（Type(F or L)），命令区間開始アドレス（fadr），命令区間の終端アドレス（eadr）を記憶する．ただし関数の場合は終端アドレスの代わりに戻りアドレスを記憶する．また後述する並列事前実行の入力ストライド予測に用いるための直近の入力値セット（Pred dist）を記憶するフィールドも持つ．

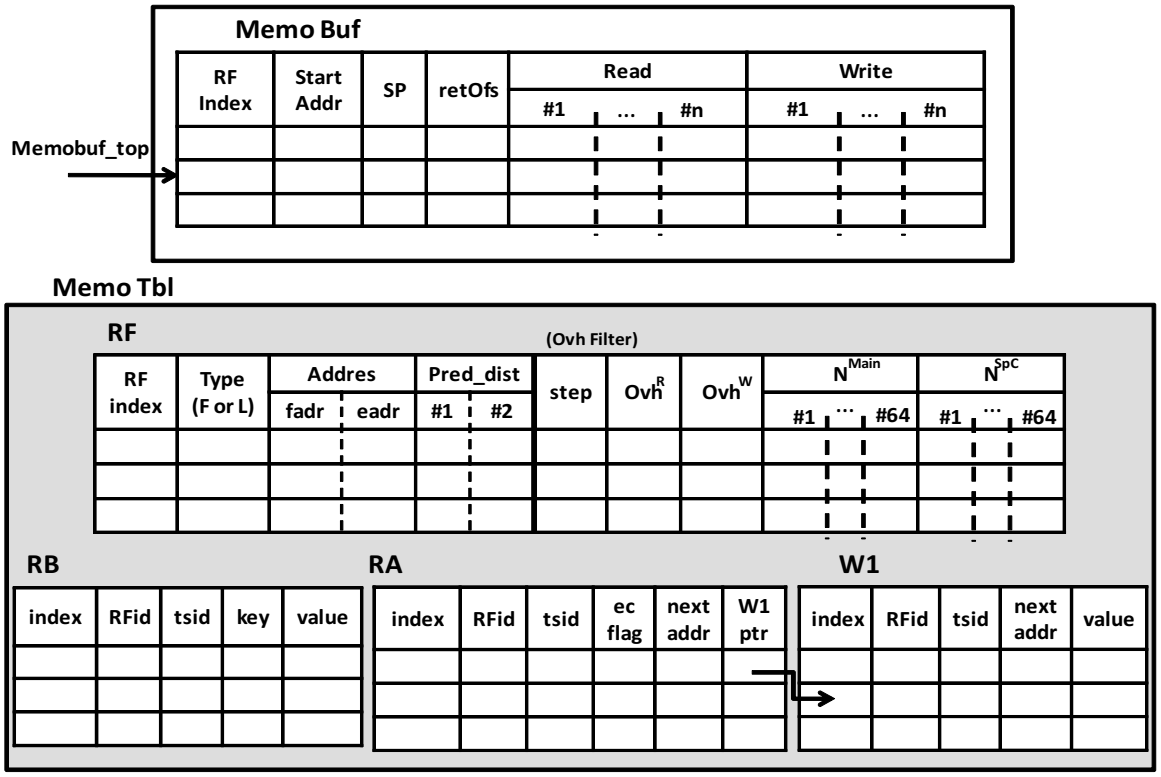

図 2: MemoBuf と MemoTbl の構成図

RB は，命令区間の入力値を記憶する表である．各行は RF の行番号 RFindex に対応する RFid を持ち，どの命令区間の入力値を記憶しているのかをこの値により判別する．一般に命令区間内では，複数の入力が順に参照され使用されるが，同じ命令区間であっても入力の値が異なると，そのエントリの次の入力値のアドレス (次入力アドレス) が異なる場合がある．これは，主記憶アドレス値自体が入力値として用いられる場合や，条件分岐の存在が原因である．つまりある命令区間の入力アドレスの列はその入力値によって分岐していくので，全入力パターンはツリー構造で表現できる．そのため，RB はこのツリー構造を折り込んで格納する必要がある．そこで，各エントリは入力値を記憶する value フィールドに加えて，当該エントリの親エントリを指す key フィールドを持つ．

RA は，入力アドレスのセットを記憶する表であり，先述した次入力アドレスを保持している．RA は RB と同数のエントリを持ち，同じ Index を持つ RB エントリの次入力アドレスを，next addr フィールドで記憶している．つまり，RB と RA のエントリは 1 対 1 対応している．この他に，入力セットの終端エントリか否かを保持する ec flag フィールド，及び RB と同様に RFid を保持している．また，入力が完全に一致した場合は，その入力に対する出力セットを参照できるように，入力列の末尾を構成する RA エントリは，

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

図 3: サンプルプログラム

対応する W1 エントリのエントリ番号を W1 ptr フィールドに保持している．

W1 は命令区間の出力を記憶する表であり，命令区間の出力値を value フィールドで保持している．また，RB と同様に RFid を保持している．MemoTbl と入力であるレジスタやキャッシュの状態を一致比較し，入力が完全に一致した場合は RA が持つ W1 へのポインタにより当該入力に対応する出力を読み出し，レジスタやキャッシュ及びメモリへ書き込むことで，命令区間の実行を省略することができる．

2.1.2 再利用機構の動作例

計算再利用を行うための再利用機構の動作に，MemoTbl へのエントリ登録と，当該命令区間が再利用可能か入力一致比較を行う再利用テストがある．

エントリの登録

メモ化対象となる命令区間の実行が終了したとき，命令区間の実行開始から MemoBuf に蓄えてきた命令区間の開始アドレスや入出力セット等の情報が，MemoTbl へ登録される．このとき命令区間の開始及び終端アドレスは RF へ，入力の値は RB へ，入力のアドレスは RA へ，出力は W1 へ登録される．

一般に，関数やループの命令区間内では，ある入力の値によって次に参照するべき入力アドレスが変化する．例えば，変数に主記憶アドレスが格納されている場合や条件分岐により次の入力が変化する場合である．そのため，ある命令区間の全入力パターンを保持するために，入力が登録されている入力エントリは木構造を成している．よって，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 3 に示すサンプルプログラムの実行を通じて，自動メモ化プロセッサ上で関数 func の入力がどのように登録されていくかについて説明する．関数の入力には引数と，その関数が参照する大域変数や上位関数の局所変数が含まれる．関数 func の場合，引数 x，大域変数 a,b,c が入力である．この関数の入力が RB 中のエントリにおいて木構造を成す様子を**図 4** に示す．図 4 中の input1，input2，input3 は何番目の入力であるかを示している．

入力は多分木で構成され，RB エントリが節に，RA エントリが枝に対応する．まず，関数 func が 1 を引数として呼び出される．その時の関数 func の入力値の候補は，引数と大域変数 a,b,c であるが，いま引数の値が 1 で a の値が 4 であるため，関数 func 中の 2,3,4 行目が実行される．したがって入力エントリは図 4 の (A) のように登録される．この時変数 c は関数 func 内で参照されていないので入力とならず，(A) にもそのエントリは登録されない．次に変数 b の値のみが変化し，2 回目の関数 func の呼び出しが行われると，図 4 の (B) のように入力が登録される．2 回目の実行は 1 回目と同じく 2,3,4 行目が実行される．引数と変数 a の値は変化していないので，input2 までは入力エントリの部分木を 1 回目と共有し，変数 b=6 を input3 として登録している．このように，同じ命令区間で入力値の共通する部分が存在する場合は，途中までの入力エントリを共有することで，有限である RB テーブルを効率的に利用できる．そして 3 回目の関数 func が呼び出されたとき入力候補の 1 つである変数 a が 13 行目の代入文で変化しているので，関数 func 内の条件分岐が不成立となる．そのため，関数 func 内では 2,3,6 行目を実行することとなるので，入力エントリは図 4 の (C) のように登録される．そして，変数 a の次に参照される値が変数 b ではなく変数 c となり，次に参照するべき入力アドレスが変化する．そのため，次の入力値の枝分かれの部分では，異なる RA エントリ (枝) が用いられる．

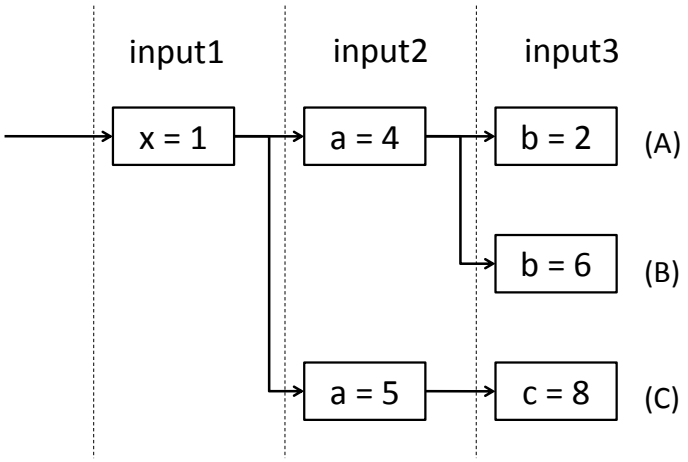

図 4: 入力エントリが成す木構造

再利用テスト

計算再利用を行うためには，関数やループの入力値と既に MemoTbl に登録されているエントリとを比較する必要がある．**図 5** に MemoTbl の検索手順の様子を示す．ここでは，図 3 のプログラム例の 14 行目までを実行した後の MemoTbl の状態を示している．

図 5 中の RF,RB,RA,W1 は 2.1.1 項で述べたフィールドの値をそれぞれ記憶している．ただしこの図では，RF には命令区間の名前とそれに対応する RF index のみを記述し，RB,RA,W1 の RFid は全て同じ値であるため省略している．図 4 を例にすると input3 の親エントリは input2，input2 の親エントリは input1 であり，input1 は親エントリを持たない．親エントリを持たないエントリをルートエントリと呼ぶ．ルートエントリは key 値として，一番目のエントリであることを示す FF を持つ．

関数やループの命令区間を検出した際，その開始アドレスを用いて RF を検索する．ここでは，サンプルプログラムの 16 行目の関数呼び出しが行われたとして，どのように MemoTbl を検索するのか説明する．まず，RF から得られた RFindex を基に RB エントリの検索を開始する．このとき，入力セットのうち最初のエントリであることを示す FF を key 値として持ち，1 つめの入力値 1 を value として持つエントリを検索する (a)．条件に一致した RB エントリに対応する RA エントリから，次に参照するアドレスを得てレジスタや主記憶を参照し，次の入力値を得る (b)．そして，主記憶を参照した結果，次の入力値は 4 であることがわかる．次に，先ほど一致した RB のインデクスである 00 を key

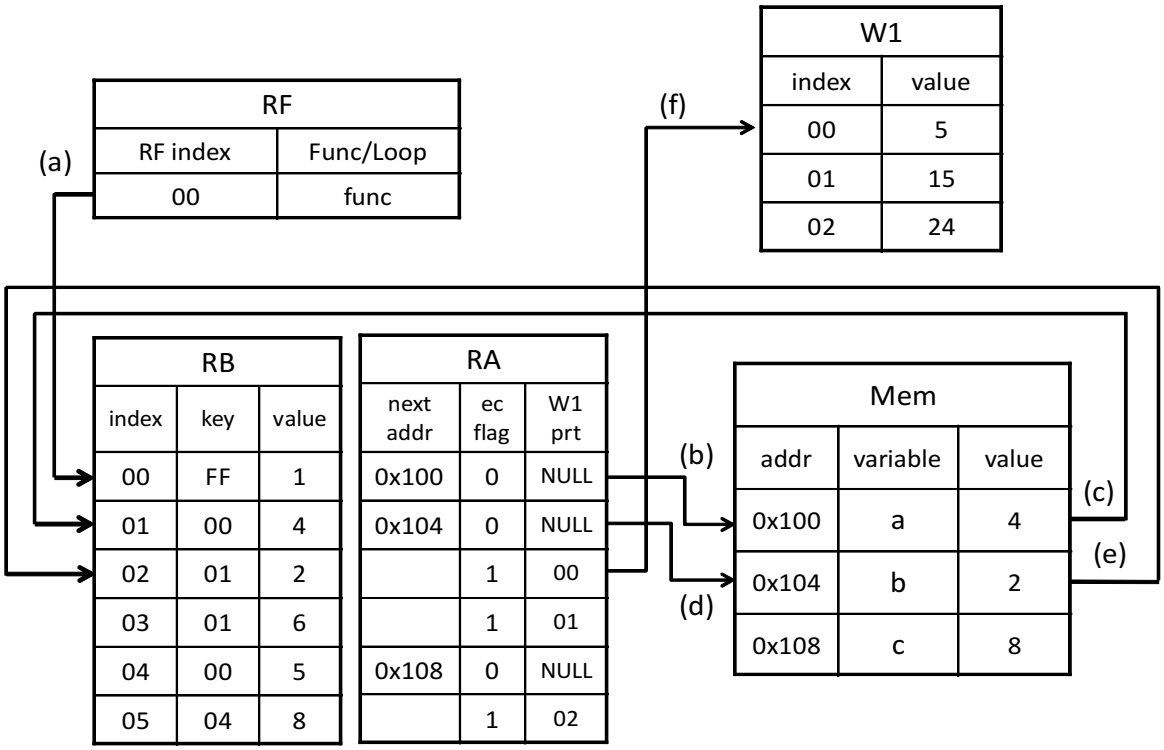

図 5: MemoTbl の検索手順

値として持ち，なおかつ，取得した 2 つ目の入力値 4 を value として持つエントリを再度 RB から検索する (c)．同じように RB エントリに対応する RA エントリから次に参照するアドレスを得てレジスタや主記憶を参照し入力値を得て (d)，その値を value として持ち，key 値に RB インデクス 01 を持つエントリを RB から検索する (e)．すると，検索したエントリに対応する RA エントリの ec flag が 1 ，つまり入力セットの終端エントリに達したため，全ての入力値の一致比較に成功し再利用が可能となる．再利用テストに成功したとき，検索の終点となった RA エントリの W1ptr に格納されているポインタによって W1 を参照し，当該命令区間の出力値を読み出す (f)．そして，読み出した出力をレジスタやメモリに書き戻すことで，当該区間に対して計算再利用が適用できる．

2.2 並列事前実行

自動メモ化プロセッサは計算再利用に基づく手法であるため，当然ながらある命令区間を過去に完全に同一の入力セットで実行した場合にのみ効果が得られる．よって入力の 1 つであるイタレータ変数が単調に変化していくループでは，全く効果が得られない．

そこで，計算再利用を行いながら実行を進めるメインコアとは別に，メインコアが実行

した命令区間の過去の入力に基づいて，同一命令区間をメインコアに先駆けて実行する投機実行コアを複数備えるシステムを考える．これを並列事前実行とよび，以下この投機実行コアを SpC(Speculative Core) と呼ぶことにする．**図 6** に並列事前実行機構を備えた自動メモ化プロセッサのハードウェア構成を示す．各 SpC は，それぞれ MemoBuf と 1 次キャッシュを持ち，MemoTbl と 2 次キャッシュは全コアで共有する．

ループの命令区間の入出力セットの登録方法は，関数の入出力セットの時とは異なっている点がある．メインコアは後方分岐命令を実行して初めて再利用を適用するループを検出し，その後方分岐命令の分岐先から同一の後方分岐命令までを再利用可能な命令区間として RF に登録する．そして，次のループイタレーションの実行結果を RB,RA,W1 へと登録していく．各ループイタレーションに対応する同一命令区間を数回実行した後に，RF に記録されたこの命令区間の直近の入力値セット (Pred dist) を用いて，メインコアは将来の入力値セットを予測し，この予測した値を用いて SpC にメインコアと並行して同一区間を投機実行させる．そして，実行に使用した入力セット及び実行の結果得られた出力値セットを，コア間で共有された MemoTbl に登録する．値予測が正しかった場合，メインコアが次に実行しようとする命令区間のイタレーションは既に SpC により実行済であり MemoTbl に結果が格納されているため，実行を省略できる．また値予測が誤っていた場合も，メインコアは当該区間を通常実行するだけであるので，MemoTbl 検索のコストは発生するものの，投機実行ミスに起因するオーバヘッドは発生しない．ただし，大きさが有限である MemoTbl に再利用が適用されないエントリが登録されることにより，有効なエントリが追い出され，再利用の成功率が低下してしまう可能性がある．なお，この入力値セットの値予測アルゴリズムにはさまざまな複雑な手法を用いることも可能であるが，必要となる追加ハードウェア量等の観点から，現在は直近の過去 2 回の実行に用いられた入力値セットの差分に基づく，単純なストライド予測を用いることを想定している．

また，メインコアと SpC では 2 次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，SpC がこれらの共有領域に対して書き込みを行うと，他の SpC やメインコアがプログラムの実行を行う際にデータの不整合が生じてしまう．よって，SpC では MemoBuf を主記憶書き込みの際のバッファとして使用することで，メモリデータの不整合の発生を回避している．

2.3 パージアルゴリズム

過去の入出力を記憶しておく MemoTbl は有限である．そのため，命令区間や入力，出力を記憶し続けていくと RF,RB,RA,W1 はそれぞれ，ついにはエントリで溢れてしまい，

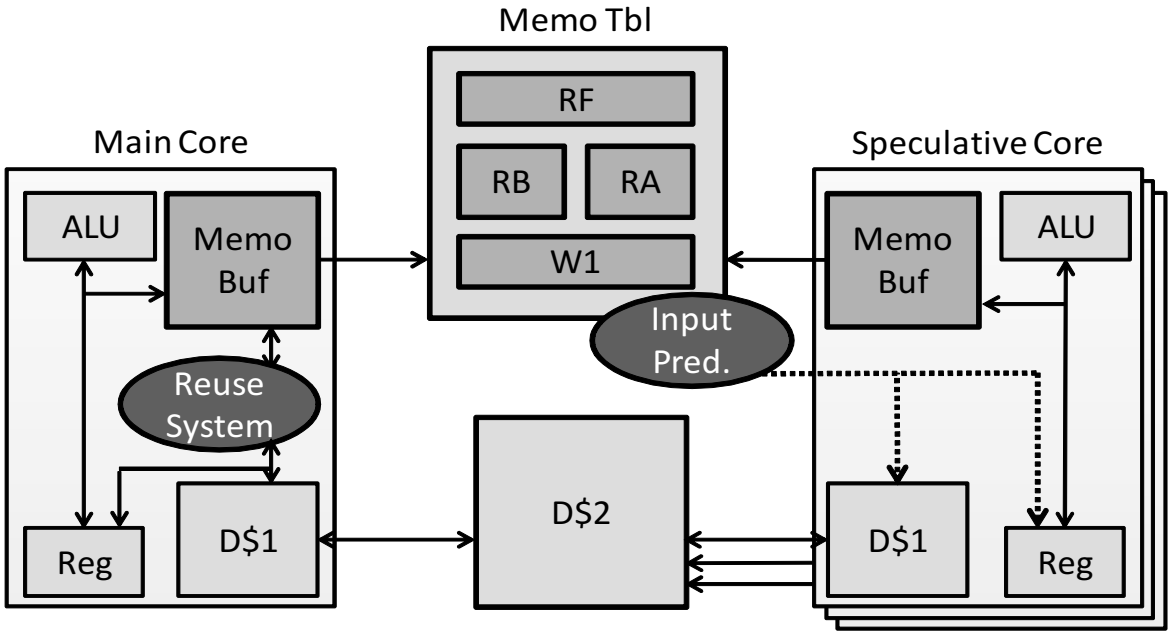

図 6: 並列事前実行機構を備えた自動メモ化プロセッサのハードウェア構成

それ以上エントリを登録することができなくなってしまう．そこで，MemoTbl の各表からエントリをパージする必要がある．現在，自動メモ化プロセッサはそのパージアルゴリズムとして，LRU に基づく TSID パージと，命令区間を指定してそのエントリ群を削除する RFID パージの 2 つを用意している．

2.3.1 TSID パージ

LRU に基づいたエントリの追い出しを行うために，リングカウンタを用いて時刻管理を行っている．また，RB,RA,W1 にエントリを登録する際に，エントリを登録した時のリングカウンタの値を tsid というフィールドに記憶していく．また，エントリが検索ヒットした場合，そのエントリ群の tsid は検索時のリングカウンタに更新される．これにより，tsid はタイムスタンプの役割を果たしている．リングカウンタは，入出力エントリの登録が一定回数行われる毎にインクリメントされ，リングカウンタが更新されたとき更新後の値と同一の tsid をもつエントリをパージする．

しかし，新しく入出力セットを登録しようとした際に，途中までのエントリを共有できる場合，新たに追加するエントリのみの tsid をリングカウンタと同じ値にしてしまうと，共有部分が追加部分よりも古い tsid を持つことになる．この状態で tsid を用いてパージしてしまうとルートに近いエントリがパージされてしまうことになり，入力エントリの木構造が崩れてしまう．そこで，エントリの登録の際には，共有されるエントリの tsid も現

在のリングカウンタの値に更新することで，これを防いでいる．TSID パージは入力セットを登録する直前に行われ，定期的に RB,RA,W1 内の不要なエントリを追い出すことで，RB,RA,W1 が溢れるのを未然に防いでいる．なお，SpC が MemoTbl にエントリを登録する場合，メインコアによる登録エントリを保護するために，メインコアによるリングカウンタの更新や TSID パージを行わない．そのため，SpC による登録が大量に発生すると，MemoTbl が溢れてしまう．このような場合に備えて，もうひとつのパージアルゴリズムである RFID パージが用意されている．

2.3.2 RFID パージ

RB,RA,W1 エントリはどの命令区間の入力 (出力) エントリであるかを表すために，命令区間の登録されている RF エントリの index を RFid フィールドに持つ．この RFid の値を利用して命令区間毎のエントリ群をパージする．このパージ方法を RFID パージと呼ぶ．このパージが行われる場面は 2 種類ある．

ひとつは，RF に命令区間を登録しようとする際に RF が溢れた場合が挙げられる．この場合，新しい命令区間を登録するために，既に登録されている命令区間のうち，最近検索や登録が行われておらず再利用成功回数も低い命令区間を追い出す．また，その際に，パージ対象となった命令区間の RFid を持つエントリを RB,RA,W1 から全てパージする．

もうひとつは，RB,RA,W1 に入出力セットを登録しているときにどれかの表が溢れた場合が挙げられる．この場合，新しい命令区間を登録するために，当該命令区間の RFid を持つエントリを RB,RA,W1 から全てパージする．また，登録途中の入出力セットも同時にパージされるため，パージ後に登録を再開しても親エントリが存在しないエントリが生成されてしまう．そのため，パージが終了した際には，現在の登録を中止している．

3 提案

自動メモ化プロセッサでは，関数とループを計算再利用の対象区間としている．そのため入出力エントリには関数の入出力エントリとループの入出力エントリが存在している．現在はどちらの入出力エントリかを区別せず扱っているが，後述する関数とループのエントリの性質の違いに着目し，それぞれを区別して登録，検索，パージを行う手法を提案する．

3.1 提案手法

従来モデルでは，関数及びループの入出力エントリを 1 つの MemoTbl に登録してきた．そのため，MemoTbl には関数とループの入出力エントリが混在している．2.2 節で

述べたように，関数の入出力エントリと異なり，ループの入出力エントリの多くは予測した値を使用して SpC が投機実行を行った結果が MemoTbl に登録されている．このように，関数とループではエントリの登録の仕方が異なっている．また，それらのエントリの性質にも異なる点がいくつかある．ここでは，エントリの登録回数，登録量，及び有効期間などに着目してその違いを述べる．以降，関数の入出力エントリを関数エントリ，ループの入出力エントリをループエントリと呼ぶ．

関数エントリ

関数はプログラム中のどの場所からも呼び出される可能性があるため，同一入力による呼び出しに関して必ずしも局所性があるとは限らない．そのため，関数エントリの入出力セットはなるべく長い期間 MemoTbl に保持しておきたいと考えられる．また，関数呼び出し 1 回あたりに 1 つの入出力セットしか MemoTbl に登録されないため，エントリの登録量は比較的少ない．

ループエントリ

ループにおいては，現在実行中の命令区間に対し，予測によって得た将来の入力を用いて投機実行し，その結果が登録される．そのため，その入力セットは，そのループ区間を実行している間に限り有効であると考えられる．また，一般にイタレーション毎に入力が変化していくため，一度再利用が成功した入出力セットがそのループ中に再利用される可能性はかなり低い．つまり，ループエントリの入出力セットは 1 度再利用された場合，MemoTbl にそれ以上保持しておくメリットは少ないと考えられる．また，予測した値を用いて実行した結果を登録しているため，その値予測自体が外れた場合には，投機実行により登録されたエントリがほぼ全て無駄となってしまうだけでなく，RB,RA,W1 に登録されている有効なエントリを追い出してしまう可能性がある．さらに，1 つのループに対して，予測された入力セットとそれを用いて実行した結果が次々に MemoTbl に登録されるため，1 呼び出しあたり 1 セットしか登録されなかった関数に比べ多数のエントリを消費する．

このように，関数エントリはループエントリに比べ比較的登録量が少ないが入出力セットが有効と考えられる期間は長く，一方でループエントリは登録量が多いが入出力セットの有効期間は短いという性質の違いがあると考えられる．しかし，既存モデルでは，これら性質の異なる関数エントリとループエントリを単一の MemoTbl に混在させて登録し，同じパージアルゴリズムを適用してしまっている．このため，関数かループどちらかのエントリの性質によって，他方のエントリが不利益を被っている可能性がある．例えば，関数エントリはできるだけ長い期間 MemoTbl に保持していたいにも関わらず，ループエン

トリの大量の登録によってリングカウンタのカウントが加速し，短期間でTSIDパージが実行され関数のエントリ群が追い出されてしまうといった場合が考えられる．

このような，エントリの性質の差異によって生じる不利益を解消させるために，従来ではエントリの種類に関係無く同じ方法で登録や検索，パージを行っていたのに対して，エントリの種類に応じて異なった登録や検索，パージを行う手法を提案する．これにより，従来では有効であるにも関わらずパージされてしまっていた入力セットに対して，再利用テストを行えるようになるうえ，関数・ループそれぞれに適したパージアルゴリズムを個別に設定することが容易になり，再利用ヒット率の向上が期待できる．

3.2 MemoTblの分割

従来の自動メモ化プロセッサのMemoTblを構成するRF,RB,RA,W1のうち，入出力を記憶するRB,RA,W1を分割することで，エントリの種類に応じて異なった操作を行えるようにする．これ以降RB,RA,W1をまとめて入出力表と呼ぶ．その入出力表の分割案を2種類提案する．1つは，従来の入出力表に対して境界値を定め分割を実現する方法，もう1つは，物理的に入出力表を複数ユニットに分割する方法である．これら2種類の方法を以下それぞれ境界型，ユニット分割型と呼ぶ．

入出力表を分割することによって，2種類のエントリを別々に扱えるというメリットがあるが，デメリットも存在する．従来の入出力表では，関数およびループそれぞれが使用できるエントリ数の上限は規定されておらず，入出力表全体を一方で占めることも可能であった．これに対し入出力表を分割する手法では，それぞれの上限は表全体の大きさよりも小さくなってしまう．例えば単純に入出力表を等分割する場合，それぞれの使用可能エントリ数は従来の半分となり，表の使用効率が低下する可能性がある．一般に，関数とループの存在する割合や，1つの入出力セットに必要なエントリ数の割合はプログラムによって様々に異なる．これらに偏りのあるプログラムの場合，入出力表の分割によって，かえって性能が低下してしまう可能性がある．境界型，ユニット分割型はそれぞれの方法でこの問題に対処している．

3.2.1 境界型

境界型は，従来の入出力表に対して境界となる値を定め，その境界値に対して入出力表の上位アドレスと下位アドレスをそれぞれの専用表とする手法である．境界型のMemoTblのモデルを**図7**に示す．入出力表のうちRA,W1は省略しているが，RA,W1もRBと同様に構成するものと想定し，境界値を記憶して置くためにレジスタ(border)を追加している．命令区間を登録するRFには手を加えず，入力エントリをRBに登録するときに，

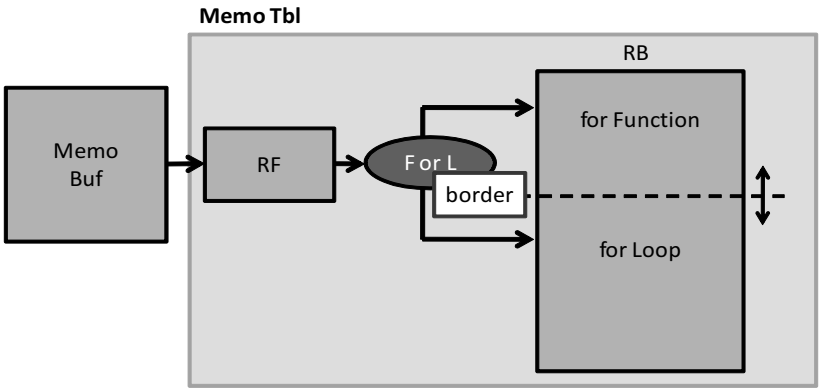

図 7: 境界型の MemoTbl モデル

RF エントリの Type フィールドからこの命令区間が関数エントリかループエントリかを判別する (F or L)．そして，図の例では，境界値に対して上位アドレスに関数エントリを，下位アドレスにループエントリを登録する．また，プログラム実行中に境界値を動的に変化させることで，入出力表の利用効率の低下を軽減する．例えばループが非常に多く関数が極端に少ないプログラムはループ入出力表の登録可能領域を増やすことで，より多くのループ入出力セットを保持しておくことが可能となる．

さて，入出力表のうち RA,W1 は RAM で構成されているが RB は CAM で構成されている．CAM はあるデータワードを指定した場合，全内容からそのデータワードを検索し，そのデータワードが見つかれば，そのワードのアドレスを返す．このように CAM は表全体を操作対象とするため，CAM の仕様上一部分を操作対象とすることはできない．しかし，境界型の登録や検索，パージなどの操作は，定めた境界値より上位，下位アドレスを操作対象とすることになり，部分操作をしなければならない．そのため，CAM の部分操作を行えるように，CAM に対して追加のハードウェアを実装する必要がある．

3.2.2 ユニット分割型

ユニット分割型は，関数及びループ専用の入出力表を複数用意し，それぞれについて関数専用かループ専用かの役目を割り当てる手法である．ユニット分割型の MemoTbl のモデルを**図 8** に示す．この図では，RB を 2 つに分割している．また，図 7 と同様に RA,W1 は省略している．境界型と同様に，RF には手を加えず，どちらのエントリかを判定する (F or L)．そして，その判定結果に基づき，関数専用の RB，ループ専用の RB に振り分けて登録する．

境界型と異なり，物理的に入出力表を分割しているので分割比をプログラム実行中に変更することができないが，ユニットを多数に分割しそれぞれのユニットの役割を切り替え

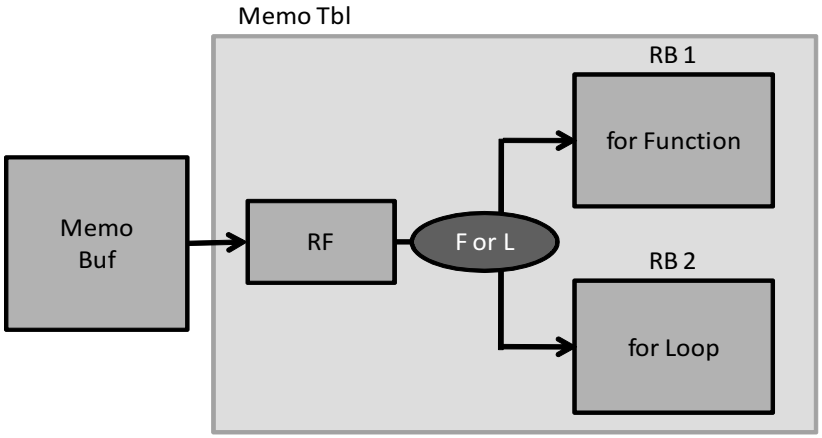

図 8: ユニット分割型の MemoTbl モデル

ることで，近似的に分割比の変更をすることが出来る．ユニット分割型は，どちらかのエントリ専用の RB が単体ユニットとして用意されるので，境界型のように CAM での部分操作をする必要がない．そのため，CAM に追加のハードウェア実装を行う必要無しに，各 RB ごとに独自の操作を行わせることが可能である．これによって，関数，ループそれぞれのパージアルゴリズムや，関数とループどちらかのみに行いたい操作を容易に実現することができる．

4 実装

本章では，3 章で述べた分割方法のうち，ハードウェアコストや実現可能性を重視して，ユニット分割型による分割を採用し，入出力表の分割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 実装の概略

ユニット分割のもっとも単純なモデルである，2 ユニット分割を実装した．これは，従来の自動メモ化プロセッサの MemoTbl を構成する 4 つのテーブルのうち，入出力を管理する表 RB,RA,W1 をハードウェアレベルで 2 つに等分割したモデルである．2 ユニット分割を行った MemoTbl と SpC を含めたハードウェア構成を**図 9** に示す．命令区間を記憶する RF には手を加えず，関数専用 (RB_F,RA_F,W1_F)，ループ専用 (RB_L,RA_L,W1_L) の入出力表をそれぞれ用意している．ある入出力セットの登録 (検索) を行う際には，RF の Type フィールドを参照し，それを基に振り分け機構 (F or L) によってどちらの入出力表に登録 (検索) を行うかを決定する．そしてそれぞれの専用入出力表に対して登録 (検

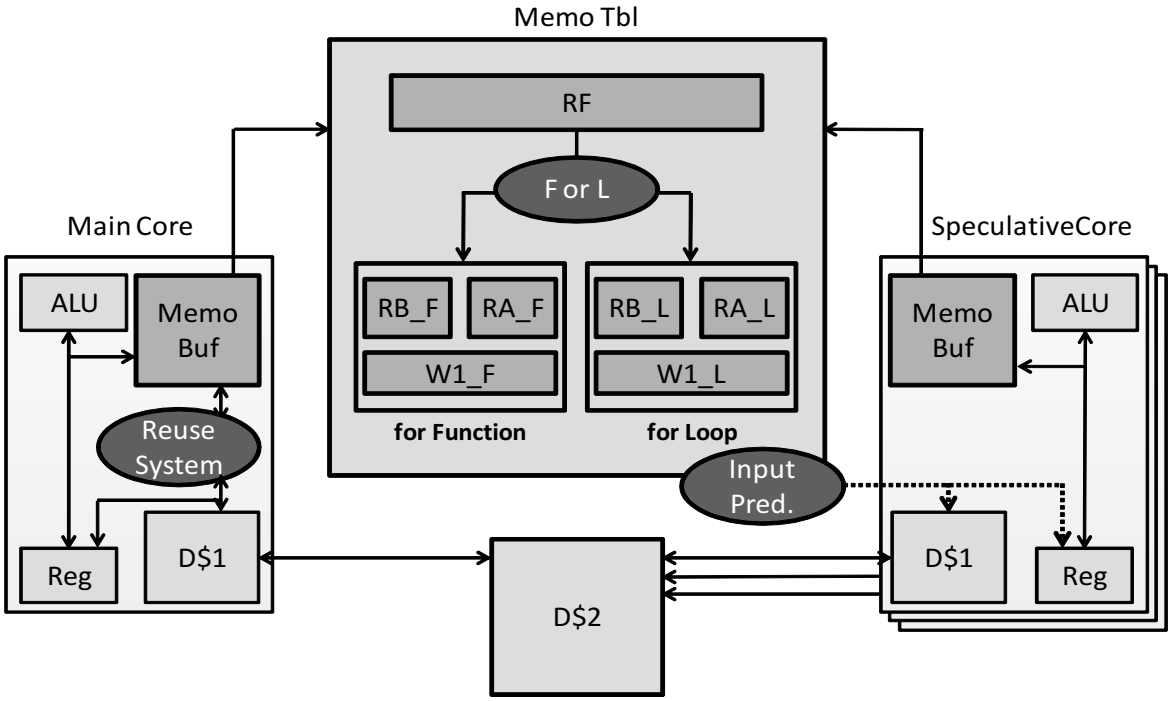

図 9: 2 ユニット分割型のハードウェア構成図

索) 操作を行う．入出力表への登録や検索方法は，どちらの入出力表にアクセスするのかを選択する必要があるという点以外に従来モデルとの違いはない．

4.2 各専用再利用表のパージアルゴリズム

2 つのユニットに物理的に分割したことで，関数専用の入出力表に対するパージアルゴリズムと，ループ専用の入出力表に対するパージアルゴリズムをそれぞれ個別に設定することが容易になった．加えて，入出力表を分割することによって，関数・ループそれぞれに適したパージアルゴリズムを用いることの重要性は増す．なぜなら，3.2 節でも述べたように，それぞれが使用できる最大エントリ数が減少することで，特にそれぞれの使用率に偏りのあるプログラムでは性能低下が引き起こされる可能性があるためである．この性能低下の原因を，3.2.2 項で少し触れた方法で分割比を変更して改善することが出来たとしても，両方のエントリの登録量が多い場合には，それぞれの表にエントリを登録しきれなくなり溢れてしまう．そのため，この最大エントリ数の減少を，分割比の変更だけでなく，エントリ利用効率の向上によってカバーする必要がある．そこで，それぞれのエントリの性質に適したパージアルゴリズムを実装し，エントリをより効率よくパージする必要がある．

4.2.1 関数用のパージアルゴリズム

3.1 節で述べたように，関数呼び出しはプログラム中のどの場所にも存在する可能性があるため，同一入力による呼び出しに関しても必ずしも局所性があるとは限らない．そのため，なるべく関数の入出力エントリは長い期間 MemoTbl に保持しておきたいと考えられる．しかし，同一入力で同じ関数がいつ呼び出されるかを正確に予測するすることは困難である．そこで，関数用のパージアルゴリズムには最長不使用エントリをパージする LRU を採用する．

従来の入出力表は，ある一定数のエントリ登録が行われるとエントリセット登録前に，TSID パージによってエントリがパージされる．この TSID パージを行わないことで，従来より長い期間関数エントリを保持することが出来る．しかし，TSID パージを行わないと入出力表の空きエントリは減少を続け，ついにはなくなってしまう．この時，従来では RFID パージが行われるが，RFID パージは登録しようとしている入出力エントリの登録を中止してしまい，最新の実行情報の登録を行わないだけではなく，登録しようとしていた入出力エントリと同じ命令区間の入出力エントリも全てパージしてしまう．この RFID パージの動作は MRU に近いといえる．そこで，入出力表が溢れた場合にパージするエントリを選択するのにも LRU を用いるようにする．入出力表が溢れたタイミングで，LRU を実現している TSID パージを行うことも可能だが，従来の TSID パージのままでは，リングカウンタによる時刻管理に問題が生じてしまう．TSID パージを従来のタイミングで行わないため，登録されてからリングカウンタが 1 周した後もエントリは残り続ける．そのため，更新されたリングカウンタと同じ tsid を持つエントリがパージされず，新しく登録されたエントリと同じ tsid を持つことになる．つまり，リングカウンタ 1 周分異なるタイミングで登録されたエントリ同士が同じ tsid を持つことになるため，より古いエントリが存在しているにも関わらずパージされてしまうエントリが存在してしまう．

そこで，リングカウンタによる時刻管理とは別に，LRU を実現するために入出力セット毎の登録順序 (order) をエントリに付加する．この order は入出力セットが登録される度にインクリメントされていき，古いエントリほど order は小さい値となり，新しいエントリほど order は大きい値を持つ．そのため，エントリは入出力セット毎に異なる order を持ち，入出力表が溢れた場合に，最も参照されていない入出力セットが 1 セットパージされる．また，tsid と同じように，木構造を壊さないために共有エントリの order 値は更新される．また，検索されたエントリをより長く入出力表に記憶しておくために，再利用テスト時に検索を行ったエントリの order 値も最新のものに更新される．

4.2.2 ループ用のパージアルゴリズム

3.1 節で述べたように，ループエントリはイタレーション毎に入力が変化していくため，一度再利用が成功した入出力セットがそのループ中に再度実行され再利用される可能性はかなり低いと考えられ，MemoTbl に長い期間保持しておいてもメリットが少ない．そこで，従来の TSID パージや RFID パージに加えて，新たに再利用テストが成功した入出力セットをパージする．これにより，値予測が成功し再利用されるループ区間に対しては入出力表が溢れる回数が減少すると考えられ，再利用テストを行う前の入出力セットまでパージしてしまう RFID パージの回数を減らすことができる．これにより，これまで再利用テストを行うことが出来なかった有効な入出力セットに対して再利用を適用することが出来るようになると考えられる．

4.3 多数分割を行うユニット分割型

入出力表を 2 つに分割するだけでは，関数専用とループ専用の入出力表の分割比をプログラム実行中に変更することができず，プログラムによっては性能が低下してしまう可能性がある．しかも，今回実装する 2 ユニット分割は入出力表を 1 対 1 の割合で分割しているため，入出力セット数がどちらかに偏っているようなプログラムの場合，分割によるデメリットの影響を大きく受けてしまう可能性が高い．そこで，入出力表を 2 つに分割するのではなく，3.2.2 項で少し触れたさらに多くのユニットに分割する手法を提案する．複数に分割した各ユニットを，関数専用かループ専用どちらの入出力表として扱うかをプログラム実行中に切り替えることで，3.2.1 項で述べた境界型には劣るものの比較的柔軟に入出力表の分割比を変化させることができる．この分割方法を複数分割型と呼ぶ．

図 10 に複数分割型の MemoTbl のモデルを示す．この図の例では入出力表を 5 等分し 5 ユニット体制としており，関数専用のユニットを 2 つ，ループ専用のユニットを 3 つの，4 対 6 の割合で入出力表を使用している様子を表している．ここで例えば，プログラム実行中に関数呼出し回数または関数エントリの登録量が少ないと判断し，ループに割り当てるエントリ数を増やしたいと考えた場合，図 10 で関数専用となっているユニット 2 を関数専用からループ専用に切り替えることで登録できるループエントリの数を増加させることが出来る．このとき，入出力表の使用比は 2 対 8 に変化する．逆に関数呼び出しまたは関数エントリの登録数が多いと判断したとき，ユニット 3 を関数用に切り替えることで関数の登録できるエントリ数を増やすことも出来る．この切り替え対象を決定するアルゴリズムを考案することは今後の課題である．なお，複数のユニットに分割したことによってそれぞれの CAM サイズが小さくなり，CAM の検索に要するオーバーヘッドが，現在想

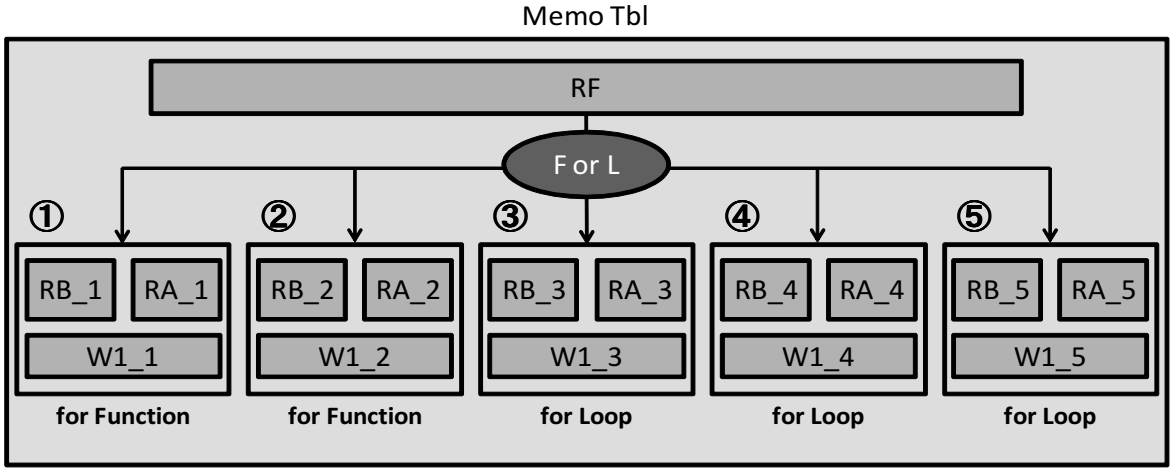

図 10: 複数分割型の MemoTbl のモデル

定しているものよりも抑えられる可能性がある．

しかしながら，1 ユニットあたりの登録できるエントリ数がさらに少なくなるため，エントリの取捨選択がより重要となってくる．本稿では，2 ユニット型を実装し関数とループを別々に扱うことの必要性を評価し，複数分割型を実装する上での足掛かりとする．そしてそこで得られた知見に基づき，将来的に複数分割方式の実現を目指す．

5 評価

本提案手法を実現するために既存の自動メモ化プロセッサに対して追加実装を行った．そして, 提案手法の有効性を示すためベンチマークプログラムを用いて評価と考察を行った．

5.1 評価環境

実行環境には，計算再利用のための機構を実装した単命令発行の SPARC-V8 シミュレータを用いた．シミュレーション時のパラメータを**表 1** に示す．なお，キャッシュパラメータや命令レイテンシは SPARC64-III[3] を，MemoTbl の RB を構成する大容量 3 値 CAM は MOSAID 社の DC182888[4] の構成を参考にし，プロセッサのクロック周波数が CAM のクロック周波数の 10 倍と仮定して検索オーバヘッドを見積もっている．また，提案手法では入出力表を等分割するため，1 つあたりの RB のライン数は 512 となる．

表 1: 評価環境

D1 Cache 容量	32KBytes
ラインサイズ	32Bytes
ウェイ数	4ways
レイテンシ	2cycles
Cache ミスペナルティ	10cycles
共有 D2 Cache 容量	2MBytes
ラインサイズ	32Bytes
ウェイ数	4ways
レイテンシ	10cycles
Cache ミスペナルティ	100cycles
Register Window 数	4sets
Window ミスペナルティ	20cycles/set
RB サイズ (既存)	32bytes × 1K 行 (32KBytes)
RB サイズ (等分割後)	32bytes × 512 行 (16KBytes)
レジスタ ⇔ CAM	9cycles/32byte
メモリ ⇔ CAM	10cycles/32byte
CAM ⇒ レジスタ or メモリ	1cycle/32byte

5.2 評価結果

SPEC CPU95 の 11 のプログラムを gcc-3.0.2 (-msupersparc -O2) 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て評価を行った．また，入力データセットとしては train を用いた．結果を**図 11** に示す．

図 11 中の凡例はサイクル数の内訳を示しており, exec は命令サイクル数, test(r), test(m) はそれぞれレジスタ/キャッシュと RB(CAM) との一致比較オーバヘッド，write は再利用成功時に発生する結果の書き戻しオーバヘッド，D$1, D$2, window はそれぞれ一次/二次キャシュミスペナルティとレジスタウィンドウミスペナルティである．

評価は，再利用を行わないモデル，従来モデル，提案モデルについて行った．提案モデルについては，入出力表を分割し従来のパージアルゴリズムを適用したものと，4.2 節で述べたパージアルゴリズムを適用したものについて評価を行っている．そして，2 つの提案モデルの結果から，後述する手法を適用したものについても評価している．なお，全て

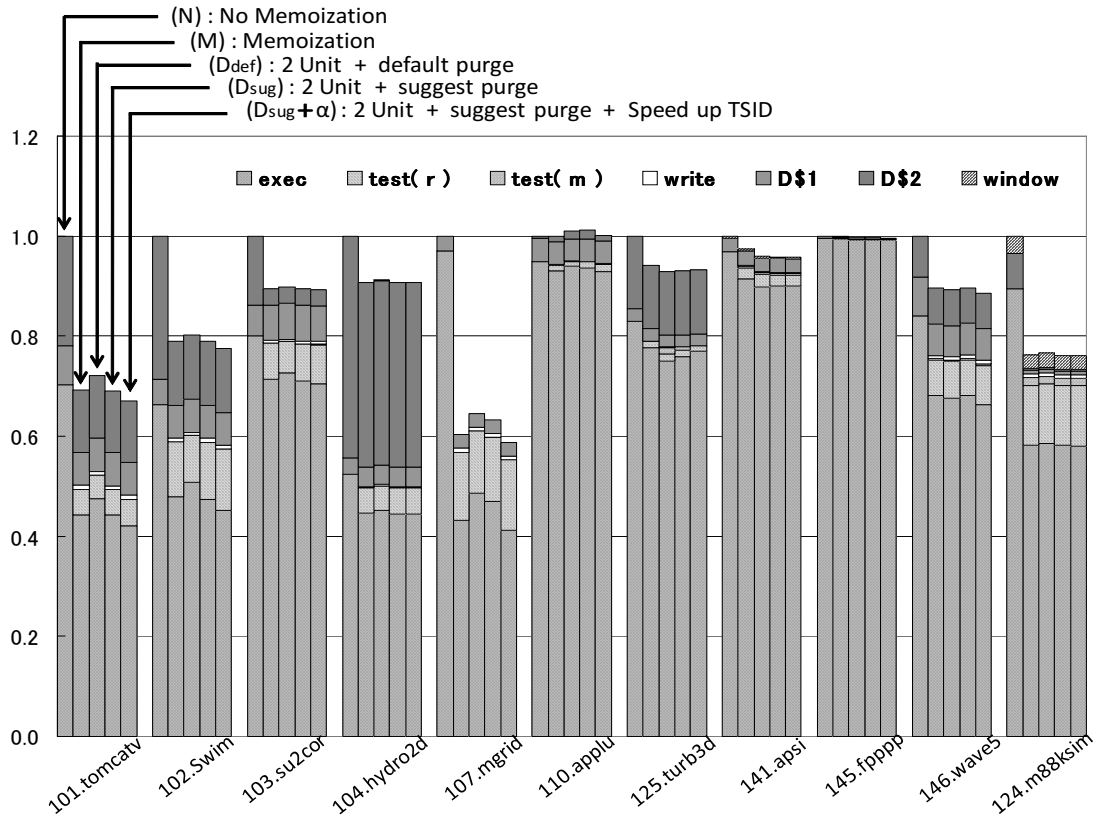

図 11: 評価結果

のモデルはメインコア 1 つに加え，SpC 3 つの合計 4 コア構成とした．図 11 中で各ベンチマークプログラムの結果を 5 本のグラフで示しているが，それぞれ左から順に

(N) メモ化を行わないモデル

(M) 既存の MemoTbl を用いてメモ化を行うモデル

($\mathbf{D}_{def}$) 入出力表を等分割したモデル (提案モデル 1)

($\mathbf{D}_{sug}$) 入出力表を分解し専用パージアルゴリズムを適用したモデル (提案モデル 2)

($\mathbf{D}_{sug}+\alpha$) (D_{sug}) においてループ入出力表のリングカウンタを SpC による登録でも更新するモデル (提案モデル 3)

が要したサイクル数を表している．なお，各サイクル数はメモ化なし (N) の結果を 1 とする正規化を行っている．

まず，(D_{def}) の結果より，等分割することで従来モデルに対して，125.turb3d，141.apsi，145.fpppp，146.wave5 の 4 つは総実行サイクル数を削減できている．しかしながら，これら 4 つのプログラム以外では総実行サイクル数が増加してしまっている．特に 107.mgrid では 6.7%と大きく増加してしまっている．つまり，入出力表を分割するだけ

では，分割によるデメリットの方が大きくなり，あまり効果が得られないのがわかる．それに対して，(D_{sug}) の場合では，分割によってサイクル数が増加してしまっているプログラムに対して，総実行サイクル数を削減できており，分割によるデメリットを軽減できているのがわかる．特に 101.tomcatv では，分割することでサイクル数が従来モデルに対して 4.2%増加してしまっていたが，(D_{sug}) では逆に 0.2%削減することができている．また，110.applu は命令サイクル数である exec が (D_{def}) より減少しているため，再利用率が上昇しているといえるが，再利用によって削減できるサイクル数より検索によるオーバーヘッド (test(r)+test(m)) の方が大きく増加してしまっているため，(D_{def}) より (D_{sug}) の方が総サイクル数が増加している．従来モデル (M) と提案モデル (D_{sug}) の結果をまとめると，SPEC CPU95 ベンチマークでメモ化なし (N) に比べ，従来手法では平均 13.9%，最大 39.5%のサイクル数の削減だったのに対し，(D_{sug}) では平均 13.8%，最大 36.7%のサイクル数の削減に留まった．

従来モデル (M) と提案モデル (D_{sug}) を比べると，サイクル数を削減できているプログラムでも大きな削減が見られず，期待した結果とならなかった．この原因を探るために，命令区間毎の再利用率や RFID パージの回数を，102.swim を用いて調べた．102.swim は，入出力表を分割することで従来モデルに対して総サイクル数が 1.6%増加してしまうが，専用のパージアルゴリズムを適用することで従来モデルより 0.1%総サイクル数を削減できるプログラムである．

(M)，(D_{def})，(D_{sug}) の 3 つのモデルを比較して，特に再利用回数が異なっているループ区間の再利用成功回数と RFID パージの回数を**表 2** に示す．表 2 から (M) は極端に RFID パージ回数が少なく，再利用成功回数が最も多いことがわかる．(M) に比べ提案モデル 2 つは RFID パージの回数が多くなっている．これは，分割することでループエントリの最大エントリ数が減少しているためであると考えられる．また，提案モデルの (D_{def}) に対して (D_{sug}) は RFID パージの回数が少なくなり，より再利用が成功するようになっている．これにより，ループエントリ用のパージアルゴリズムが有効に働いているといえる．しかしながら，(D_{sug}) も既存モデルほど再利用が成功しておらず，エントリの追い出しが十分でないため，表が溢れて RFID パージが行われ (M) に比べ再利用回数が悪化していると考えられる．ループ入出力表に対して RFID パージが行われると，予測した将来の入出力セットを再利用テストを行う前にパージしてしまい，性能が低下してしまう．

ここで，2.3.1 項で述べたように，メインコアの登録したエントリを保護するために SpC が登録する際には，どちらの入出力表も従来通り TSID パージのリングカウンタを更新していない事に着目する．入出力表を分割する提案モデルでは，ループの入出力表において

表 2: ループ区間 (1aee4 ～ 1af98) の再利用成功回数と RFID パージの回数

	再利用成功回数	RFID パージ回数
(M) 従来モデル	101711	4
(D_{def}) 提案モデル 1	58975	10075
(D_{sug}) 提案モデル 2	84277	6234
($D_{sug}+\alpha$) 提案モデル 3	132893	0

は SpC による登録が大半を占める．このため，ループ入出力表のリングカウンタの更新速度は，エントリの登録速度に対し非常に遅くなる．これにより TSID パージが起こらないままエントリが増加していき，結果として RFID パージが頻発していると考えられる．

そこで，ループ入出力表のリングカウンタを SpC による登録の際にも更新するように (D_{sug}) を変更したモデルが ($D_{sug}+\alpha$) である．図 11 からわかるように, ($D_{sug}+\alpha$) は従来モデルに比べて全体的に総サイクル数を削減できているのがわかる．特に 101.tomcatv では従来モデルに比べ, 総サイクル数の 3%を削減することが出来ており, 102.swim,107.mgrid, 141.apsi,146.wave5 の 4 つも 1%を越えるサイクル数を削減できている．また，110.applu では，(D_{sug}) を適用することで性能が 1%以上悪化してしまっていたが，($D_{sug}+\alpha$) では 0.3%の悪化に留まっている．一方で，124.m88ksim は (D_{sug}) よりも命令サイクル数を削減できているが，検索オーバーヘッドが多くなってしまったため総サイクル数が増加してしまっている．また，表 2 で示した命令区間に関しては，($D_{sug}+\alpha$) では RFID パージが行われなくなり，再利用成功回数が大きく増加しているのがわかる．しかしながら，($D_{sug}+\alpha$) でも命令区間によっては表が溢れ RFID パージが行われている．そのため，ループ入出力表のパージアルゴリズムをさらに改良する必要があるといえる．例えば，イタレータ変数が単調に変化していく場合には，再利用が成功した入出力セットより古い入出力セットをパージするといった方法も考えられる．また，再利用ヒット率を**表 3** に示す．表より従来手法に比べヒット率が向上しており，ループのパージアルゴリズムだけではなく関数のパージアルゴリズムも有効に働いているのがわかる．

結果をまとめると，メモ化なし (N) に比べ，従来手法では平均で 13.9%，最大 39.5%のサイクル数の削減だったのに対し，($D_{sug}+\alpha$) では平均 14.8%，最大 41.2%のサイクル数の削減に成功した．

表 3: 再利用ヒット率

	全体	関数	ループ
(M) 従来モデル	9.7%	7.9%	14.0%
($D_{sug}+\alpha$) 提案モデル 3	10.4%	8.5%	16.3%

6 おわりに

本研究では，従来までの自動メモ化プロセッサに対して更なる高速化手法として，関数エントリとループエントリを別々に扱う手法を提案し，入出力表を分割することで実装した．SPEC CPU95 ベンチマークを用いて評価した結果，提案手法の有効性を示した．従来モデルでは平均 13.9%，最大 39.5%であったサイクル数削減率が，提案するモデルでは平均 14.8%，最大 41.2%まで向上させることができた．

本研究の今後の課題として，短期的な課題と長期的な課題が挙げられる．短期的な課題は，多数分割型を実装することである．この多数分割型を実装する上で，いくつか考えなければならないことがある．1 つは，どのユニットをどちらの専用表に切り替えるのかを選択する方法である．また，関数及びループのいずれのエントリが多いという判断をどのように行うのか，複数の専用表のうちどのユニットにエントリを登録を行うのかなども考えなければならない．長期的な課題としては，ソフトウエ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ことが考えられる．バイナリのみでなく，ソースコードが公開されているプログラムについては，プログラム内の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こと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エントリが登録できるようにするなど，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，性能向上すると考えられる．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室および齋藤研究室の方々に深く感謝致します．

参考文献

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

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

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

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