卒業研究論文

再利用表の多段化による自動メモ化プロセッサの検索コスト削減

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 18 年度入学　18115014 番

板橋世里華

平成 22 年 2 月 8 日

再利用表の多段化による自動メモ化プロセッサの検索コスト削減

板橋世里華

内容梗概

これまでに，プログラムを高速に実行する手法として命令レベル並列性，スレッドレベル並列性，データ並列性などの様々な並列性に着目した研究が数多く行われてきた．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，高速化を図るというものである．それらとは全く異なる，値の局所性に着目した高速化手法に計算再利用というものがある．

ソフトウェアによる計算再利用に関する研究は従来から行われてきた．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というものが提案されている．メモ化とは，関数やループ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処理である．

自動メモ化プロセッサでは，再利用表の検索や，再利用成功時に出力を再利用表から書き戻すにはコストがかかり，それを再利用オーバヘッドと呼んでいる．再利用表を検索する際にかかるオーバヘッドを検索コストと呼び，再利用表から出力を書き戻す際にかかるオーバヘッドを書き戻しコストと呼ぶ．

本研究では，入力を登録する表を，今日の一般的なキャッシュのように多段化することで，検索コストを削減する手法を提案する．これにより，自動メモ化プロセッサの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手法を実装し，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べ，従来手法では最大で 3.9 %，平均で 0.9 %のサイクル数の削減だったのに対し，提案手法では最大で 6.3 %，平均で 1.1 %のサイクル数を削減できた．

再利用表の多段化による自動メモ化プロセッサの検索コスト削減

目次

1　はじめに

これまでに，プログラムを高速に実行する手法として命令レベル並列性，スレッドレベル並列性，データ並列性などの様々な並列性に着目した研究が数多く行われてきた．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，高速化を図るというものである．一方，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．

ソフトウェアによる計算再利用に関する研究は従来から行われてきた [1, 2]．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 [3, 4] というものが提案されている．メモ化 [5] とは，関数やループ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処理である．本論文ではこの自動メモ化プロセッサの更なる高速化手法として，メモ化によるオーバヘッドを削減する手法を提案する．

自動メモ化プロセッサは過去に実行した関数の入出力を表に登録しておく．その後再び同じ関数が呼び出されたときに，その入力と過去に表に登録しておいた入力とを比較し，それらが一致すれば過去の出力を利用し実行を省略する．本研究では，入力を登録する表を，今日の一般的なキャッシュのように多段化することで，検索する際のコストを削減し，自動メモ化プロセッサの更なる高速化を図る．

以下，2 章では，自動メモ化プロセッサとその問題点について説明する．3 章では，自動メモ化プロセッサにおける再利用オーバヘッド，主に検索コストを削減することにより，さらなる高速化を図る手法を提案する．4 章では，提案を実現するにあたって行った実装について説明する．5 章で提案手法について評価を行い，最後に 6 章で結論をまとめる．

2　自動メモ化プロセッサ

本章では本研究が対象とする自動メモ化プロセッサとその問題点について述べる．メモ化とは関数やループ等の命令区間の入出力を計算再利用可能な状態で記憶することである．このメモ化によって，ふたたび同じ入力で同一の命令区間を呼び出した時，計算結果の再利用が行われ，命令区間の実行を省略することができる．本章で述べる自動メモ化プロセッサは，バイナリの変更なしで既存のプログラムに対して自動的にメモ化を行い，実行することのできるプロセッサである．

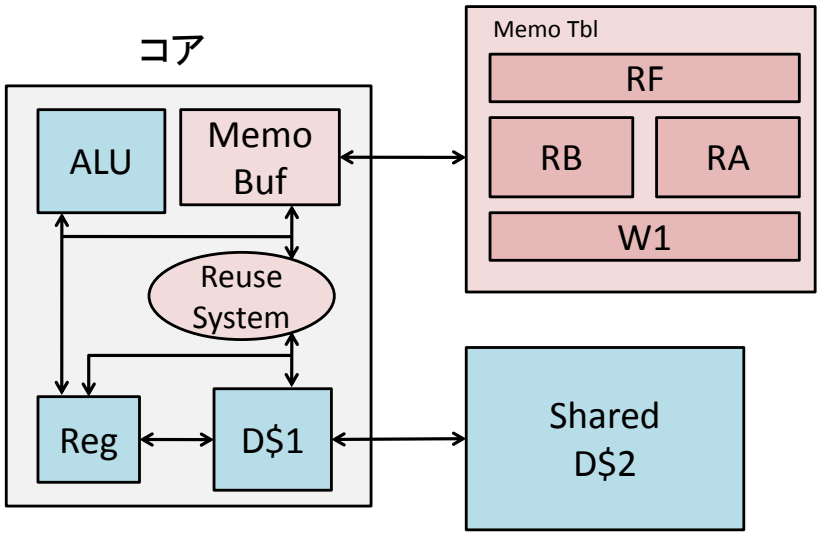

図 1: 自動メモ化プロセッサのアーキテクチャ図

2.1 ハードウェア構成

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，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，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 は命令区間およびその入出力を記憶するための表であり，計算再利用を行うために必要となる．MemoTbl と MemoBuf の詳細な構成を**図 2** に示す．

MemoTbl は，命令区間を記憶する表 RF，命令区間の入力値セットを記憶する表 RB，入力アドレスのセットを記憶する表 RA，及び出力値を記憶する表 W1 から構成される．なお，MemoTbl において命令区間の入力値セットを保持する RB は，高速な連想検索が可能な CAM(Content Addressable Memory) で実装されると仮定している．また，コアが命令区間の入出力を再利用表に登録する際，入出力を検出する毎にサイズの大きい再利用表に対して参照を行うと，オーバヘッドが大きくなる．そこで，このオーバヘッドを軽減するため，作業用の小さなバッファである MemoBuf をコアの内部に設け，各命令区間の実行終了時に一括して MemoTbl に登録する．

まず MemoBuf について説明する．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

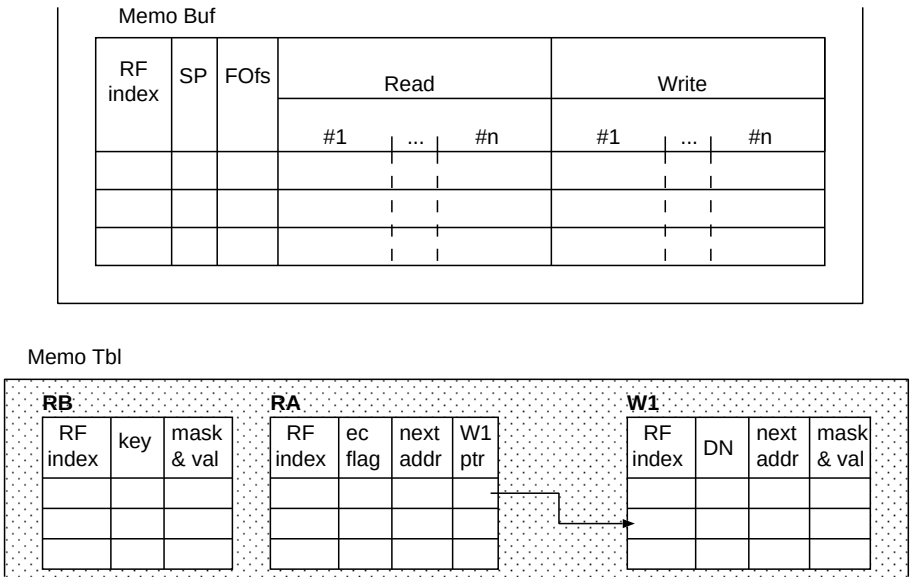

図 2: MemoBuf と MemoTbl の構成図

象命令区間の開始アドレスを記憶する RFindex，命令区間の実行開始時のスタックポインタ SP，関数の終了アドレスを示すオフセット FOfs，命令区間の入力セット Read 及び出力セット Write の各フィールドからなる．自動メモ化プロセッサは命令区間の実行を進めながら，命令区間実行中に発生した入出力を Read 及び Write フィールドに記憶していく．そして命令区間の実行終了時に当該命令区間に対応する MemoBuf エントリを MemoTbl 本体に格納する．

MemoBuf に複数のエントリが存在するのは，コアが実行中に呼び出した関数のネスト構造を保持するためである．現在 MemoBuf のどのエントリを使用しているかを把握するために，ポインタ memobuf_top を用いる．エントリはエントリ番号の小さい方から順に使用され，関数呼び出し (call) によってネストが深くなると，それに対応して memobuf_top がインクリメントされる．逆に関数から復帰した場合 (return) は memobuf_top がデクリメントされる．このようにして MemoBuf は，コアが現在実行している関数のネスト構造を保持する．

次に MemoTbl の各構成要素について説明する．RB は入力値を記憶するための表であり，CAM で構成されているため高速な検索が可能である．RA は次に参照すべき入力の主記憶アドレス (nextaddr) を記憶するための表である．RB と RA の各エントリは 1 対 1 に対応する．RA はそのエントリが入力値の組の最後のエントリであることを示すフラグ (ecflag) を持ち，最後のエントリは出力を記憶している W1 のエントリ

例 1

```
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);
         …
```

を指すポインタ (W1ptr) を持つ．MemoTbl と入力一致比較を行い，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．

また，MemoTbl の大きさは有限であるため，MemoTbl の登録エントリが溢れる前に不要なエントリを削除する必要がある．このエントリの削除アルゴリズムには LRU(Least Recently Used) 方式を用いている．

2.2 動作モデル

例 1 に示すサンプルプログラムを実行したときの自動メモ化プロセッサの動作について説明する．関数の実行が終了したとき，関数の実行開始から MemoBuf に蓄えてきた関数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

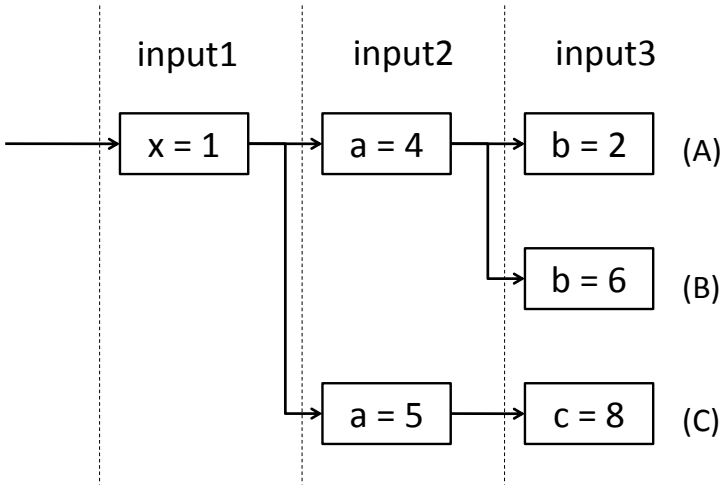

図 3: 入力エントリがなす木構造

き入力の値は RB へ，入力のアドレスは RA へ，出力は W1 へ登録される．

関数の入力は木構造をなすため，RB 中のエントリも木構造をなすように登録される．サンプルプログラムの関数 func は入力として引数 x，大域変数 a，b，c を用いる．この関数の入力が RB 中のエントリにおいて木構造ををなす様子を**図 3** に示す．

図 3 中の input1，input2，input3 は何番目の入力かを示している．サンプルプログラムの 10 行目の関数呼び出しの入力は (A) に対応している．10 行目の関数呼び出しでは，関数 func 中の 2，3，4 行目が実行されるので，1 番目の入力は引数 x で値は 1 となり，2 番目の入力は大域変数 a で値は 4 となり，3 番目の入力は大域変数 b で値は 2 となる．大域変数 c は参照されないため，入力とはならない．次に 12 行目の関数呼び出しの入力は (B) に対応している．12 行目の関数呼び出しでも，関数 func 中の 2，3，4 行目が実行されるので，1 番目の入力は引数 x で値は 1 となり，2 番目の入力は大域変数 a で値は 4 となり，3 番目の入力は大域変数 b で値は 6 となる．ここでも大域変数 c は参照されないため，入力とはならない．ここで，先ほどの 10 行目を実行したときの入力の組 (A) の 1 番目と 2 番目の入力が (B) の 1 番目と 2 番目の入力と同じであるため，(A) と (B) は図 3 のように 1 番目と 2 番目の入力を共有している．次に 14 行目の関数呼び出しの入力は (C) に対応しており，14 行目の関数呼び出しでは，関数 func 中の 2，3，5，6 行目が実行されるので，1 番目の入力は引数 x で値は 1 となり，2 番目の入力は大域変数 a で値は 5 となり，3 番目の入力は大域変数 c で値は 8 となる．14 行目を実行したときの入力の組 (C) は，1 番目の入力が (A)，(B) と同じであるため，1 番目の入力を (A)，(B) と共有する．

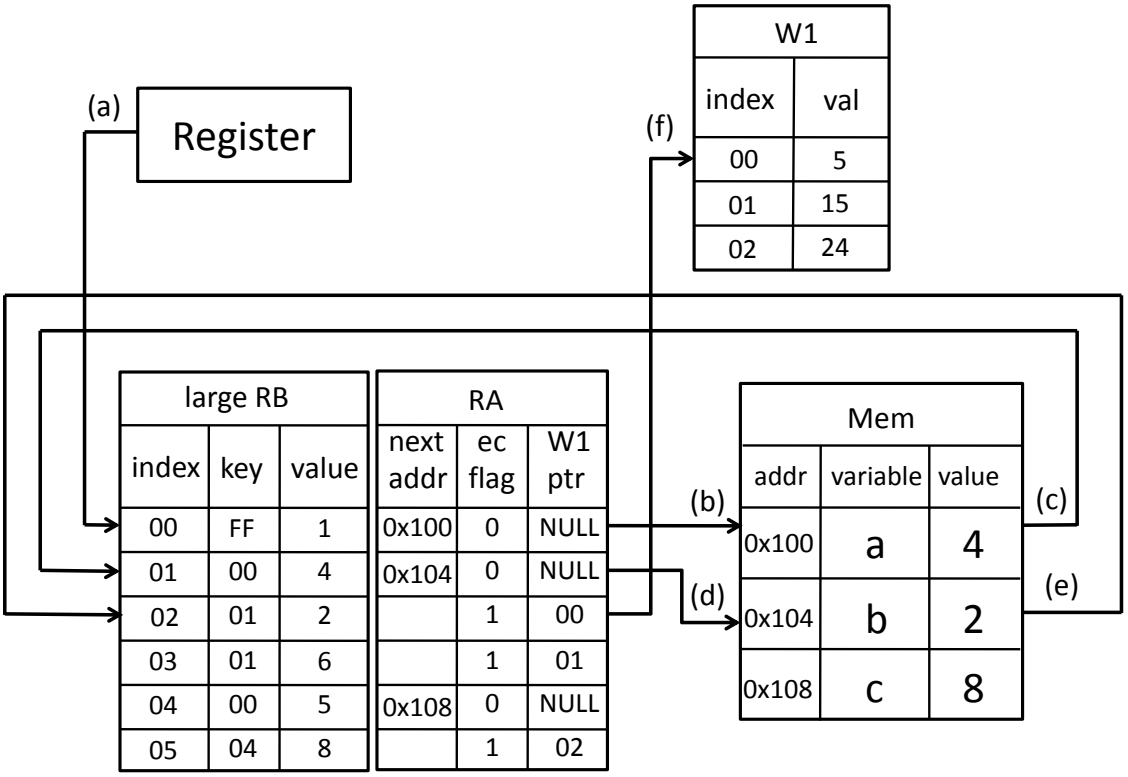

図 4: MemoTbl の検索手順

図 4 はどのように MemoTbl を検索するのかを示しており，この図を用いて，これらの入力が RB 中でどのように格納されるかを説明する．図 4 の RB 中の value フィールドは入力の値を，key フィールドは親エントリのインデクスを指す．図 3 を例にすると，input3 の親エントリは input2，input2 の親エントリは input1，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．また，RA 中の next addr フィールドは次に参照すべき入力のアドレスを格納しており，ec flag フィールドはエントリが入力値の組の最後のエントリであるか否かを示す．また，w1 ptr フィールドは，出力を記憶している W1 のエントリを指すポインタを格納する．

図 3 で見てきたように，関数の入力は多分木で構成されるが，RB のエントリが節に，RA のエントリが枝にそれぞれ対応する．図 3 の (A) の 3 つの入力は，それぞれエントリ 00，01，02 に格納されている．エントリ 00 の key 値は 1 番目のエントリであることを示す FF，value は入力の値の 1 である．エントリ 01 の key 値は親エントリ

のインデクスである 00，value は入力の値の 4 である．エントリ 02 も同様に格納されている．

ここで，サンプルプログラムの 8-14 行目を実行し，入出力が MemoTbl に登録された状態から，15 行目を実行し 16 行目の関数呼び出しに対して，どのように MemoTbl を検索するのかを図 4 を用いて説明する．最初のエントリであることを示す FF を key 値，引数が格納されているレジスタから得られる入力の値 1 を value として RB の検索を開始する (a)．RB のインデクス 00 で，エントリの持つ key の値と入力の値が，検索条件である key 値およびレジスタの値と一致し，RA 中でその一致行と同じインデクスを持つエントリから次に参照するアドレスを得る．そして得たアドレスを使って主記憶を参照する (b)．次に，一致した RB のインデクス 00 を key とし，参照先の値 4 を value として再度 RB 上のエントリを検索する (c)．RB のエントリ 01 の key 値と value に一致し，RA 中でその一致行と同じインデクスを持つエントリから次に参照するアドレスを得る．そして再び，得たアドレスを使って主記憶を参照する (d)．先ほどと同様に，一致した RB のインデクス 01 を key とし，参照先の値 2 を value として RB 上のエントリを検索すると，RB のエントリ 02 の key 値と value に一致する (e)．ここで RA の ec flag が 1 である (つまり，すべての入力の一致比較に成功した) ため，RA から該当する出力が格納されている W1 へのポインタを得る．そして，このポインタを用いて W1 を参照し，書き戻すべき出力が取り出される (f)．

2.3 再利用オーバヘッド

MemoTbl の検索や，再利用成功時に出力を MemoTbl から書き戻しを行う際にはオーバヘッドが発生する．ここで，MemoTbl を検索する際にかかるオーバヘッドを検索コストと呼び，MemoTbl から出力を書き戻す際にかかるオーバヘッドを書き戻しコストと呼ぶ．また，これら 2 つのオーバヘッドをまとめて，再利用オーバヘッドと呼ぶ．

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，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 に登録していることになり，MemoTbl が有効活用されない．そこで，自動メモ化プロセッサでは，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実行サイクル数を見積もり，計算再利用により効果を得られると判

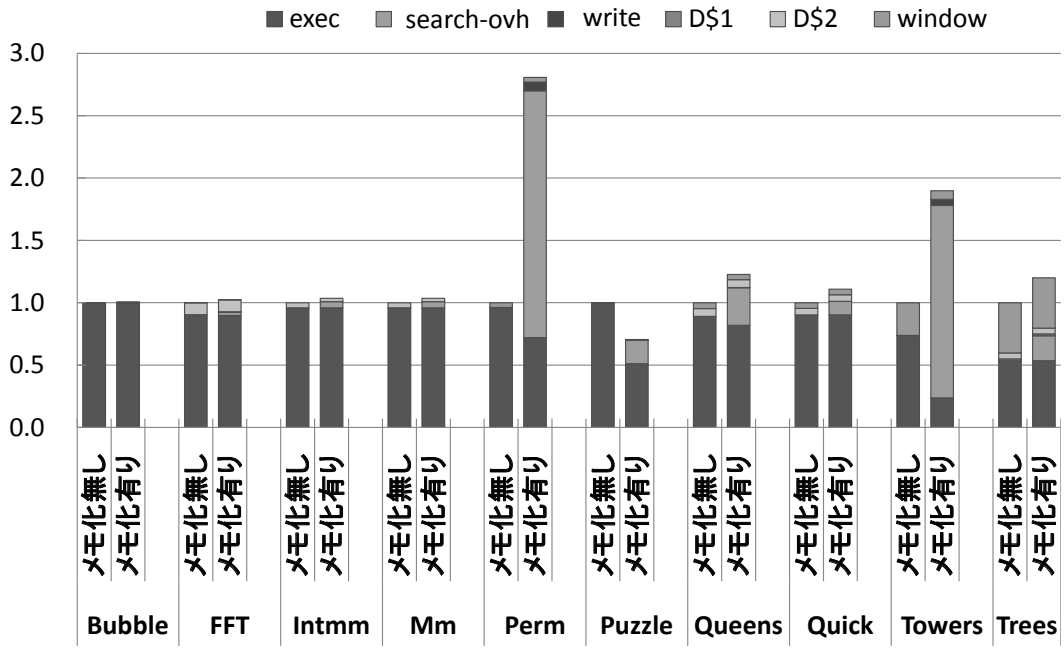

図 5: 既存手法での評価結果 (stanford)

断した命令区間に対してのみ再利用テストを行う．具体的には，命令区間の再利用により削減できるサイクル数と，その再利用に必要となるオーバヘッドについて概算を行う小さなハードウェアを MemoTbl に付加する．この機構をオーバヘッドフィルタと呼ぶ.

既存の自動メモ化プロセッサでの，stanford ベンチマークの性能評価を**図 5** に示す. 左がメモ化無しで右がメモ化有における総実行サイクル数であり，メモ化無しの総実行サイクル数を 1 で正規化している.

凡例は左から順に，exec は命令の実行に要したサイクル数，search-ovh は MemoTbl の検索に要したサイクル数，write-ovh は MemoTbl から出力を書き戻す際に要したサイクル数，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ィンドウミスによって要したサイクル数である.

グラフ中の Perm，Queens，Towers のように，再利用による高速化よりも再利用オーバヘッドによる遅延が大きいと性能が悪化する．またグラフ中の Intmm，Mm，Quick，Trees のように，再利用が適用されず高速化されないプログラムは，再利用オーバヘッドにより性能が悪化する一方である．再利用が効くプログラムは，再利用オーバヘッ

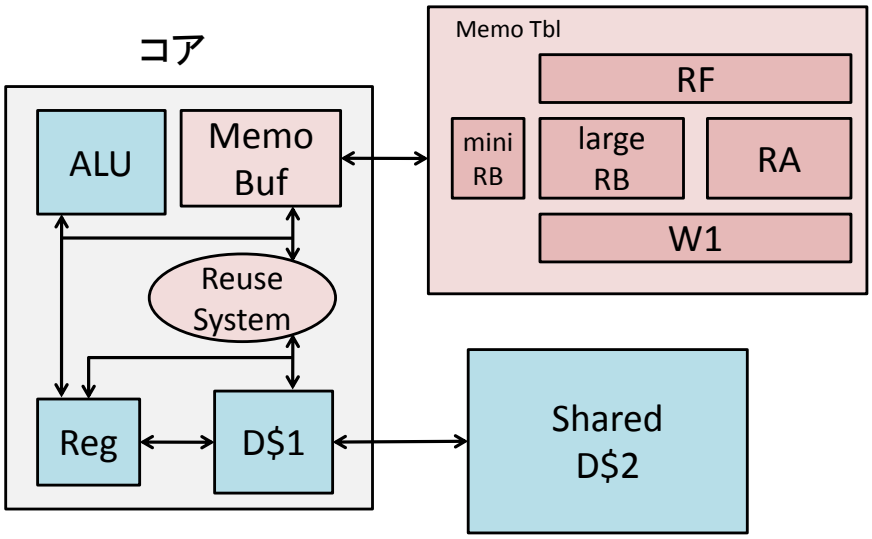

図 6: 提案手法でのアーキテクチャ図

ドの内訳として検索コストの方が書き戻しコストよりも大きい．また再利用が効かないプログラムは，出力の書き戻しが発生しないため，書き戻しコスト自体が発生していない．つまり，再利用オーバヘッドの中でも特に検索コストを削減することで，再利用の効くプログラム，効かないプログラム共に高速化が期待できる．

3　再利用表の多段化

本章では，再利用オーバヘッドのほとんどが検索コストであるということに着目し，MemoTbl 中の RB の多段化によって検索コストを削減する手法を提案する．

3.1　提案手法

従来までの自動メモ化プロセッサでは，再利用が有効となるためには，128kByte 程の大きさの RB が必要であった．しかし，RB をこの程度の大きさにすると，RB へのアクセスに時間を要し，検索コストが大きくなる．そこで，従来の大きい RB の他にアクセスコストの小さい RB を付加する．そして入力エントリの検索の際はまず小さい RB を検索し，成功しなければ大きい RB を検索するという風に，キャッシュのように検索を多段化することで検索コストの削減を図る．以下，従来の大きさの RB を largeRB，従来よりもサイズが小さく，アクセスコストの小さい RB を miniRB と呼ぶ．

従来の自動メモ化プロセッサに miniRB を追加した提案手法のハードウェア構成を，**図 6** に示す．miniRB は largeRB と同様に CAM で構成される．入力エントリの検索は

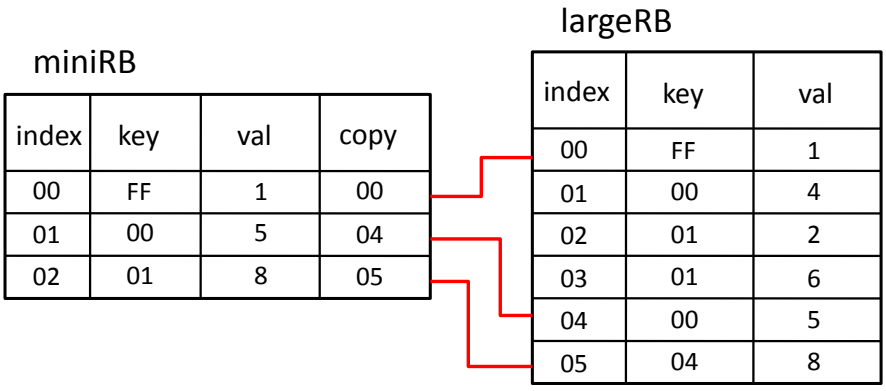

図 7: miniRB と largeRB の関係図

miniRB を検索してから largeRB を検索するという風に多段に行われる．RB と計算再利用を行うための機能以外は従来手法と全く同じものを用いる．また，MemoBuf から largeRB へのエントリの登録方法は変更せず，miniRB へ largeRB のエントリをコピーする機能を追加する．

miniRB 中のエントリを有効に使用するためには，miniRB で検索が一致する可能性の高いエントリを largeRB から miniRB にコピーする必要がある．そこで largeRB で検索を行った時にヒットしたエントリをコピーするようにする．また，新たなエントリを miniRB にコピーする際に空きエントリがない場合のエントリの削除には，LRU 方式をベースにする．これ以降エントリの削除をパージと呼ぶ．これらのコピーアルゴリズムとパージアルゴリズムについては，4 章の実装で述べる．

ここで，提案手法の RB の構成について詳しく説明する．miniRB と largeRB の関係を**図 7** に示す．図 7 は，largeRB 中のエントリ 00，04，05 が miniRB 中のエントリ 00，01，02 にコピーされた状態を表している．

miniRB は largeRB 中のいくつかのエントリを保持しており，largeRB 中にないエントリを miniRB が保持しているということはない．miniRB の各エントリは largeRB の各エントリが持つフィールドと miniRB に新しく追加した COPY フィールドを持っている．COPY フィールドは，コピー元の largeRB の index を示し，miniRB 中のエントリが largeRB 中のどのエントリからコピーしてきたのかが分かるようになっている．この COPY フィールドによって，largeRB と 1 対 1 で対応している RA へのアクセスや，miniRB 中での検索失敗時の，miniRB から largeRB への検索移行が可能になる．

また Key 値については，largeRB 中の key 値をそのまま miniRB 中にコピーするの

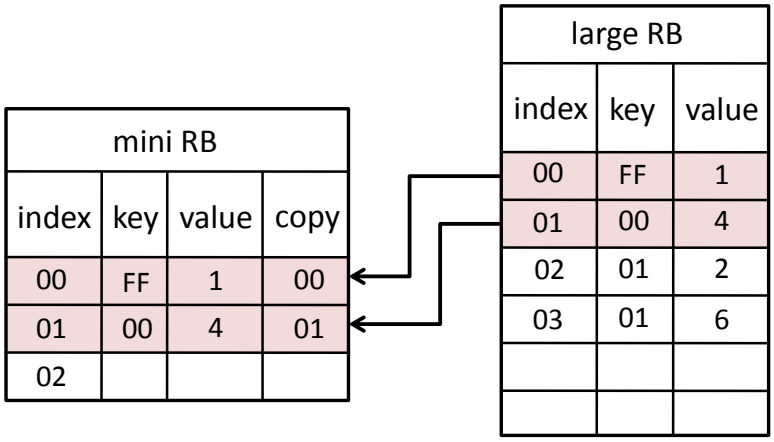

図 8: largeRB から miniRB へのコピー

ではなく，miniRB 中での親エントリのインデクスを指すように変更する．図 7 の例では，largeRB 中のエントリ 05 の親エントリはエントリ 04 であるため，エントリ 05 の key 値は 04 となっている．しかし，largeRB 中のエントリ 05 をコピーした miniRB 中のエントリ 02 の親エントリは miniRB 中のエントリ 01 であるためエントリ 02 の key 値は 01 となっている．

3.2 動作モデル

本節では 2.2 節で示したサンプルプログラムを実行したときの動作の様子を説明する．まず，10 行目を実行した時点では，largeRB，miniRB ともにエントリには何も登録されていない状態である．そのため，miniRB，largeRB で検索が失敗し，3 つの入力が largeRB 中のエントリ 00，01，02 に登録される．

続いて 11，12 行目を実行する．miniRB にはまだエントリがないため miniRB での検索は失敗する．largeRB のエントリ 00，01 にヒットするが，3 つ目の入力が largeRB 中にないため，largeRB 内での検索は終了し，3 つ目の入力が largeRB 中のエントリ 03 に登録される．ここで，largeRB 中のエントリ 00，01 にヒットしたため，これらのエントリを miniRB にコピーする．largeRB 中のエントリ 00，01 を miniRB 中のエントリ 00，01 にコピーしている様子を**図 8** に示す．コピーする際は key と miniRB 中のインデクスとを対応させ，COPY フィールドにコピー元の largeRB のインデクスを格納する．value の値は largeRB，miniRB で変わらないのでそのままコピーする．

続いて 13，14 行目を実行する．一つ目の入力である値 1 の引数 x が miniRB のエン

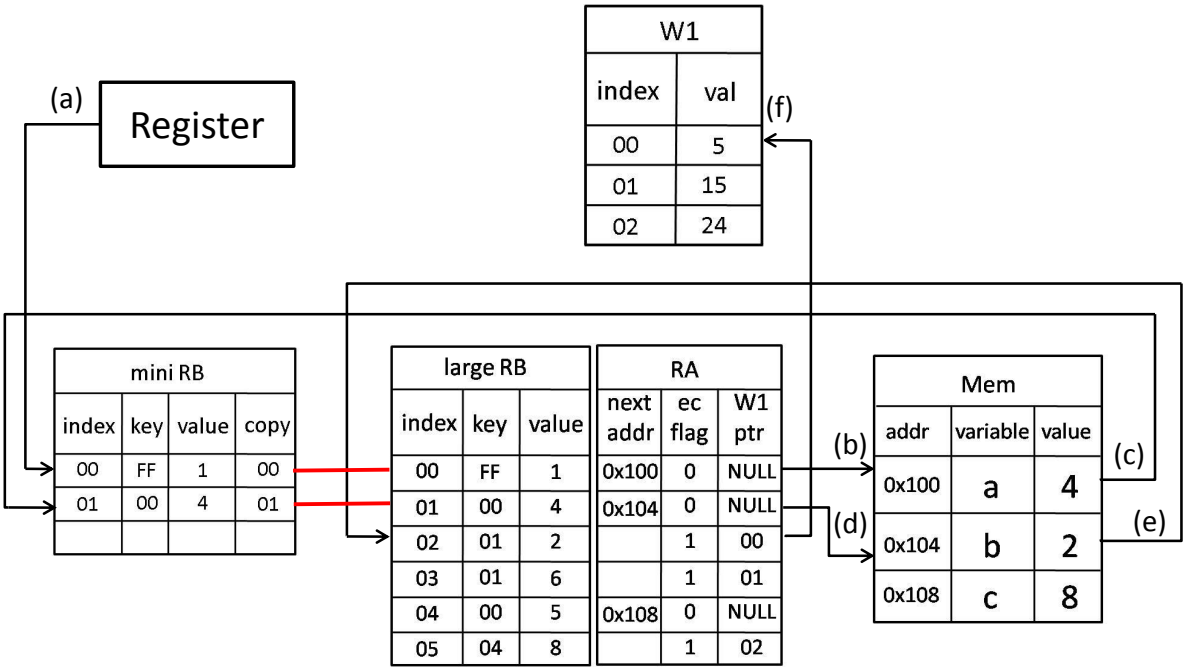

図 9: 提案手法における実行時の MemoTbl の検索手順

トリ 00 にヒットするが，二つ目の入力と三つ目の入力が miniRB にも largeRB にも登録されていないため，検索は失敗する．そして入出力が MemoTbl に登録される．この状態から，15 行目を実行し，16 行目の関数呼び出し時に発生する検索の様子を**図 9** に示す.

従来手法と同様に，まず key 値に FF と引数が格納されているレジスタから得られる入力の値 1 を value として検索を行う．提案手法では miniRB から検索を開始する．そこで miniRB 中のエントリ 00 において，エントリの持つ key 値と value の値が一致する (a)．ここで一致した miniRB のエントリの持つ COPY フィールドの値を用いて RA にアクセスし，次に参照すべきアドレスを得る．そして得たアドレスを用いて主記憶を参照する (b)．続いて，一致した miniRB のインデクス 00 を key とし，参照先の値 4 を value として miniRB 中のエントリを検索する．miniRB のエントリ 01 の key 値と value の値が一致する (c)(d)．次に，01 を key，6 を value として miniRB を検索するが見付からないので，ここから検索を largeRB に移行する．miniRB 中での key は largeRB 中の key とは異なるため，次に検索すべきエントリの largeRB 中における key 値を得る必要がある．そこで，最後に miniRB 中で一致したエントリ，つまりインデクス 01 の COPY フィールドの値 01 を largeRB 中で次に検索すべきエントリの key 値

	miniRB での検索回数	largeRB での検索回数	総検索コスト
従来手法	0 回	3 回	30 サイクル
提案手法	3 回	1 回	13 サイクル

表 1: 従来手法と提案手法との検索コストの比較

として用いる．01 を key 値，6 を value として largeRB を検索すると，largeRB 中のインデクス 02 で一致する (e)．そして，インデクス 03 の RA の ecflag が 1 なので検索は終了し，W1 へのポインタを得て，書き戻すべき出力が取り出される (f)．

以上のように，miniRB のみで検索が続行可能である間は，ヒットした miniRB エントリのインデクスを次の検索の key 値として用いればよい．また，miniRB 中で検索が失敗した際には，miniRB 中で最後にヒットしたエントリの COPY フィールドが largeRB 中の，次に検索すべきエントリの親エントリを指しているため，これを key 値として用いることで largeRB の検索へ移行できる．

3.3 期待される効果

largeRB のクロック周波数が CPU のクロック周波数の約 10 分の 1 程度だと仮定し，1 回の検索に 10 サイクルかかるとする．一方 miniRB は largeRB よりもサイズが小さいため largeRB より高速なクロックで動作させることが可能である．ここで仮に miniRB における 1 回の検索が 1 サイクルで実行可能だとする．

2.2 節のサンプルプログラムにおける，従来手法での検索コストと，提案手法での検索コストとの比較を**表 1** に示す．従来手法では miniRB がないため，3 つの入力全てを largeRB で検索している．そのため，miniRB での検索が 0 回，largeRB での検索が 3 回となり，総検索コストは 30 サイクルとなる．また，従来手法では 3 つ目の入力の検索で miniRB 中での検索が失敗したので miniRB での検索が 3 回，largeRB での検索が 1 回となり，総検索コストは 13 サイクルとなり，この例では 17 サイクルが検索オーバヘッドから削減できる．削減可能なオーバヘッドは，当然ながら miniRB のヒット率に影響を受け，miniRB で検索ヒットしない場合は逆に性能悪化を招くことになる．よって miniRB には使用頻度や使用される可能性の高いエントリが格納される必要がある．

3.4 miniRB のサイズの検討

本節では検索コストを削減するにあたって，miniRB のサイズの検討を行う．ある関数が呼び出されたとき，従来手法において RB の検索に要した回数を n とし，RB での 1 エントリあたりの検索のコストを x とすると，従来手法における検索コスト *COST1* は

$$COST1 = xn$$

と表せる．

一方提案手法では，miniRB で全ての検索がヒットしない限り，miniRB での検索回数と largeRB での検索回数を足し合わせた，総検索回数は $n+1$ 回である．ここで，largeRB の 1 エントリあたりの検索コストを従来手法における RB での検索コストと同じ x にし，miniRB の 1 エントリあたりの検索コストを y とすると，提案モデルの検索コスト *COST2* は

$$\begin{aligned} COST2 &= ym + x \times (n+1-m) \\ &= ym + xn + x - xm \\ &= xn - (xm - x - ym) \end{aligned}$$

と表せる．ここで，xn は従来手法における検索コストであるので $(xm - x - ym)$ が提案手法で得られる高速分であると言える．よってこの値が 0 より大きければ従来手法より高速化可能であると言える．

$$\begin{aligned} xm - x - ym &\geq 0 \\ y &\leq \frac{x(m-1)}{m} \end{aligned}$$

miniRB における 1 エントリあたりの検索コストの上限は miniRB で検索する回数 m によって変わる．そこで，m の値によってどのように y が決まるかを**表 2** に示す．

表 2 を見て分かるように，miniRB でのヒット回数が 0 回のときは miniRB における 1 エントリあたりの検索コストが 0 以下となっている．つまり，miniRB における 1 エントリあたりの検索コストが 0 ということは有り得ないので，miniRB で 1 回もヒットしないと，検索オーバヘッドは確実に増加してしまう．しかし，miniRB で 1 回でもヒットすれば miniRB における検索コストが 5 サイクルまでなら，従来手法に比べて検索コストは減少する．つまり，miniRB 中でのヒット回数が増えれば，miniRB の 1 エントリあたりの検索コストの制約は緩くなり，miniRB のサイズを大きくすることが

miniRB での検索回数 m(回)	miniRB でのヒット回数 m - 1(回)	miniRB で 1 回あたりの検索コスト x(サイクル)
1	0	0 以下
2	1	5 以下
3	2	6 以下
4	3	7 以下

表 2: ヒット回数による検索コストの上限の変化

可能になってくる．miniRB のサイズを大きくすることが可能ならば，miniRB 中でのヒット回数も増え，より検索コストの削減が期待できる．

これらの考察は従来手法で，あるプログラムの全ての関数呼び出しにおいて RB 上でエントリがヒットしているということを前提としている．しかし，一般に全ての関数呼び出しに対して RB 上のエントリがヒットすることはない．5 章で検索コストを考慮しながら，最も適当な miniRB のサイズについて検証する．

4 実装モデル

4.1 実装の概略

提案手法を実現するために，従来の自動メモ化プロセッサに miniRB を追加し，miniRB を検索後に largeRB を検索するように実装を行う．大まかな検索の流れを**図 10** に示す．

入力エントリを検索するために，まず miniRB を検索する．miniRB 中で検索が成功した場合，従来どおり再利用を行う．miniRB 中で入力エントリの検索が失敗した場合，つまり入力組のうちのあるエントリが miniRB 中になかった場合，検索を largeRB に移行する．largeRB 中で検索が失敗した場合は従来どおり largeRB へ登録を行う．この時，largeRB へ登録したエントリの miniRB へのコピーは行わない．largeRB 中で検索が成功した場合は，従来どおり再利用を行い，largeRB 中でヒットしたエントリを miniRB にコピーする．

4.2 miniRB の実装

miniRB は largeRB 中のいくつかのエントリを持つ．そして largeRB のエントリのうち，より最近使用したエントリを miniRB が持つように実装を行う．本節では，そのために用いたコピーアルゴリズムと，パージアルゴリズムについて述べる．

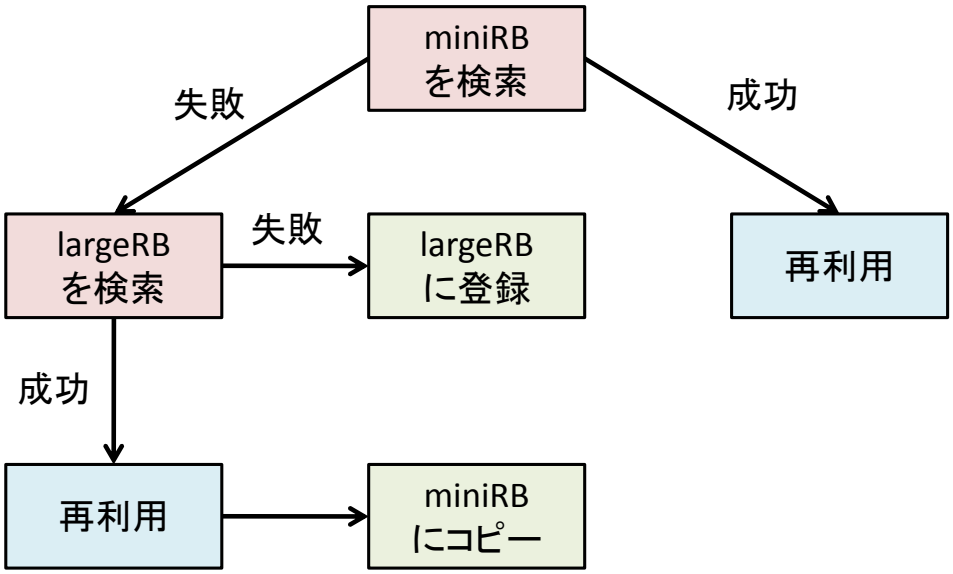

図 10: 検索の流れ

コピーアルゴリズム，パージアルゴリズム共に，より最近使用したエントリが miniRB に格納されるよう LRU 方式をベースとするアルゴリズムを採用する．LRU を実現するにあたって，コピーアルゴリズムは，簡単化された LRU と，より効率的な LRU との二つのアルゴリズムを実装し評価する．またパージアルゴリズムは，LRU だけに基づくものと，エントリ群を木としたときの深さを考慮するものとの，二つのアルゴリズムを実装し評価する．

4.2.1 miniRB のコピーアルゴリズム

largeRB 内で最近使用したエントリを miniRB にコピーするために，検索終了後に largeRB 内でヒットしたエントリを miniRB にコピーするように実装する．

検索後にコピーするアルゴリズムとして，

A largeRB でヒットし，かつ再利用が行われたときに，ヒットしたエントリをコピーする

B 再利用が行われる，行われないに関係なく largeRB 内でヒットしたエントリをコピーする

の，二つを考えた．それぞれのアルゴリズムについて詳しく説明する．

まず A のアルゴリズムについて説明する．例として，2.2 節で示したサンプルプログラムの後に例 2 のように続いたとする．例 2 の 18 行目で関数呼び出しが行われると，

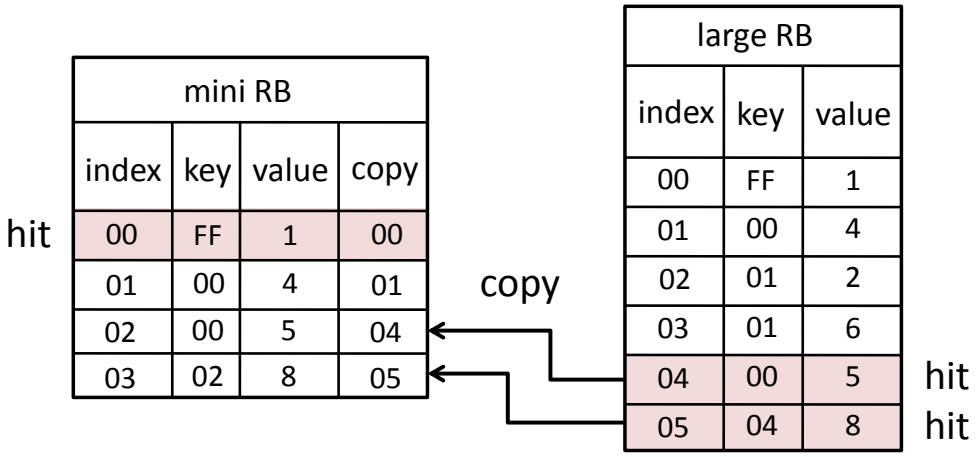

図 11: コピーアルゴリズム A

例 2

```
17 :   a = 5;
18 :   func(1);
```

2.2 節のサンプルプログラム中の 2，3，5，6 が実行されるので，この関数呼び出しで，値が 1 の引数 x，値が 5 の大域変数 a，値が 8 の大域変数 c の 3 つの入力が検索される．miniRB 中のエントリ 00 で引数 x がヒットし，値 5 の大域変数 a が miniRB 中にないため，largeRB に検索が移行する．そして値 5 の大域変数 a が largeRB 中のエントリ 04，値 8 の大域変数 c がエントリ 05 でヒットする．3 つの入力全てがヒットしたため検索が成功し再利用が行われる．largeRB でヒットした 2 つのエントリは miniRB 中にはないエントリのため，エントリ 04 と 05 が largeRB から miniRB にコピーされる．このコピーの様子を**図 11** に示す．このアルゴリズムでは，再利用されたエントリ群のみをコピーするので，実装が容易である．しかし，再利用が全く効かないプログラムでは，miniRB にエントリのコピーが全く行われないため，miniRB を付加することでの検索コストの削減が望めない．

次に B のアルゴリズムについて説明する．B のアルゴリズムは，A のアルゴリズム同様に，largeRB 内で検索がヒットし再利用が行われたエントリをコピーする．また，largeRB 中の検索で途中のエントリで検索失敗し，再利用されなかった場合でもエントリをコピーするアルゴリズムである．

例として，2.2 節で示したサンプルプログラムの後に例 3 のように続いたとする．例

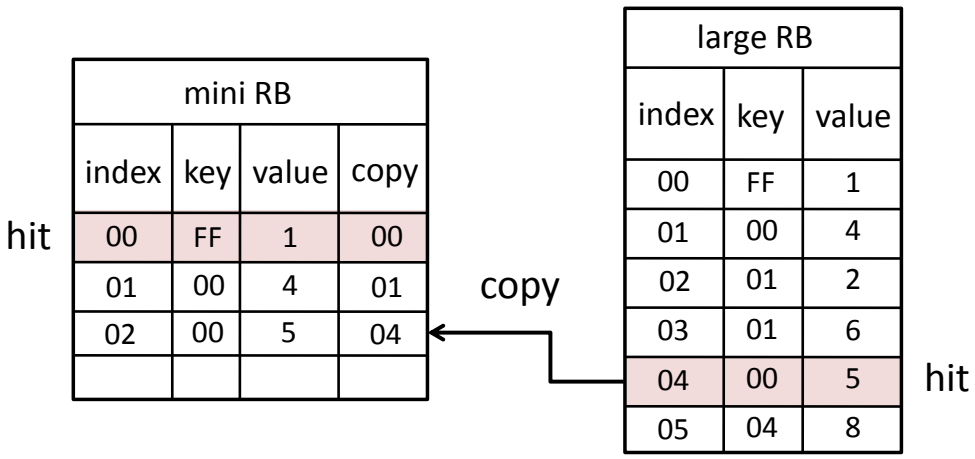

図 12: コピーアルゴリズム B

例 3

```
17 :   c = 10;
18 :   func(1);
```

3 の 18 行目で関数呼び出しが行われると，2.2 節のサンプルプログラム中の 2，3，5，6 行目が実行されるので，この関数呼び出しで，値が 1 の引数 x，値が 5 の大域変数 a，値が 10 の大域変数 c の 3 つの入力が検索される．miniRB 中のエントリ 00 で引数 x がヒットするが，値 5 の大域変数 a が miniRB 中にないため，largeRB に検索を移行する．そして値 5 の大域変数 a が largeRB 中のエントリ 04 でヒットする．しかし値 8 の大域変数 10 が largeRB 中にないため，検索は失敗し終了する．largeRB でヒットしたエントリは miniRB 中にはないエントリのため，エントリ 04 が largeRB から miniRB にコピーされる．このコピーの様子を**図 12** に示す．

A のアルゴリズムでは，再利用が全く行われないプログラムでは miniRB 中にエントリが全くコピーされないため，miniRB のヒット率が向上することがなく，検索コストの削減は望めなかった．しかし，B のアルゴリズムでは再利用が効かないプログラムでも，miniRB 中へエントリがコピーされて，miniRB 中でヒットする可能性が生まれるため，検索コストの削減が期待できる．

4.2.2 miniRB のパージアルゴリズム

miniRB 内に空きエントリがなくなった場合，miniRB 内で最も過去に使ったエントリをパージするために，エントリ毎にタイムスタンプを設け，そのタイムスタンプが

一番古いものをパージするように実装する．なお，以下の 2 つのパージアルゴリズムを考えた．

A　タイムスタンプが古いものをパージするアルゴリズム

B　タイムスタンプとエントリの深さを考慮するアルゴリズム

エントリ毎に設けたタイムスタンプは largeRB から miniRB にコピーされてきた時に 0 で初期化し，他のエントリがコピーされる毎にインクリメントする．miniRB 中でエントリの検索が成功したとき，そのエントリのタイムスタンプを再び 0 にする．また，largeRB からコピーされるエントリ群の親エントリが miniRB 中に存在する場合，コピーされるエントリ群同様に miniRB 中の親エントリも 0 にする．こうすることで，親が子より古いタイムスタンプを持つことがないようにできる．これによりエントリのパージの際には，エントリがなす木構造のうち部分木だけが削除され，木構造全体に矛盾が生じることはない．

パージを行うタイミングは，largeRB からエントリをコピーしようとする際，エントリをコピーするための空きエントリがない場合である．miniRB 中のタイムスタンプのうちタイムスタンプが最大のエントリ群をパージする．A のパージアルゴリズムは以上のような流れでパージを行う．

次に，エントリ郡が木構造を成していることに注目し，タイムスタンプだけでなくエントリの深さも考慮するアルゴリズムを考える．図 3 で示した入力ツリーの検索において，miniRB 中で input1 のエントリ 1 がヒットしなければ，それ以降 miniRB でヒットすることはない．しかし，input1 のエントリ 1 がヒットすればそれ以降ヒットする可能性は残る．それは，input2 のエントリ 4 と 5 にも同じことが言える．input2 のエントリ 4 または 5 が miniRB でヒットしなければそれ以降 miniRB でヒットすることがない．つまり，エントリの深さが小さいエントリを miniRB 中に残すことで，miniRB のヒット率を向上させられる可能性がある．

B のパージアルゴリズムでは，ルートエントリから数えて何番目のエントリであるかを示す Depth フィールドを miniRB に設ける．この Depth フィールドの値はルートエントリでは 0 とし，それ以降ルートエントリから離れるにつれて Depth フィールドの値をインクリメントする．図 3 のエントリ群を例とすると，input1 のエントリの Depth フィールドは 0 を，input2 のエントリの Depth フィールドは 1 を，input3 のエントリの Depth フィールドは 2 を保持する．

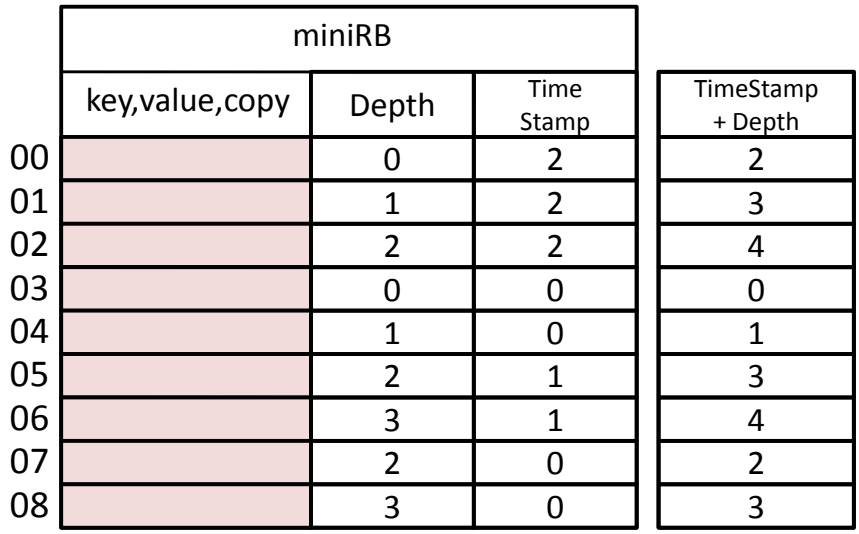

図 13: パージアルゴリズム

ここで，パージエントリの選択基準として，以下の計算式を定義し．この計算式から得られた値が大きいエントリから順にパージするものとする．

$$mT + nD$$

Tはエントリのタイムスタンプを示し，Dはエントリの深さを示す．m及びnは係数である．これらの係数の値を変動させることで，タイムスタンプとエントリの深さのどちらを重視するかを決めることができる．例えば，$n = 0$ ならばエントリの深さは全く考慮されない．また，$m < n$ ならばエントリの深さを重視して選択する．

ここで，**図 13** を例に二つのパージアルゴリズムを用いてパージを行う様子を具体的に述べる．A のパージアルゴリズムではタイムスタンプのみ用いてパージを行うため，タイムスタンプが最大のエントリ群である，エントリ 00，01，02 の三つのエントリがパージされる．また，B のパージアルゴリズムではタイムスタンプと Depth を用いてエントリが削除される．例えば，選択基準として用いる計算式の n と m が共に 1 であるとすると，タイムスタンプと Depth の加算結果が最大のエントリ群である，エントリ 02，06 がパージされる．A のパージアルゴリズムではエントリの深さを考慮していないため，ルートエントリであるエントリ 00 や，ルートエントリから二つ目のエントリである 01 もパージされてしまうが，B のアルゴリズムではそれらのエントリは削除されなくなる．

D1 Cache 容量	32KByte
ラインサイズ	32Byte
ウェイ数	4way
レイテンシ	2cycle
Cache ミスペナルティ	10cycle
共有 D2 Cache 容量	2MByte
ラインサイズ	32Byte
ウェイ数	4way
レイテンシ	10cycle
Cache ミスペナルティ	100cycle
Register Window 数	4set
Window ミスペナルティ	20cycle/set
largeRB サイズ	32Byte × 4K 行 (128KByte)
miniRB サイズ	32Byte × 1K 行 (32KByte)
レジスタ ⇔ largeRB	9cycle/32Byte
メモリ ⇔ largeRB	10cycle/32Byte
レジスタ ⇔ miniRB	1cycle/32Byte
メモリ ⇔ miniRB	2cycle/32Byte
CAM ⇒ レジスタ or メモリ	1cycle/32Byte

表 3: 評価環境

5 評価

提案手法の有効性を示すため，SPEC CPU95 ベンチマークを用いて評価と考察を行った．

5.1 評価環境

評価には単命令発行の SPARC V8 シュミレータを用いた．シミュレーションで用いた評価環境を**表 3** に示す．なお，キャッシュや命令レイテンシは SPARC64-III[6] を参考にした．SPEC CPU95 ベンチマークプログラムを gcc-3.0.2 (-O2 -msupersparc) によりコンパイルし，スタティックリンクにより生成したロードモジュールを用いた．largeRB に用いる CAM の構成は，ルネサステクノロジ社の R8A20410BG[7] を参考に

している．再利用テストのコストとして，レジスタと largeRB を 32bytes 比較するのに 9 サイクル，メモリと largeRB を 32bytes 比較するのに 10 サイクルを要するものとする．またレジスタと miniRB を 32bytes 比較するのに 1 サイクル，メモリと miniRB を 32 ビット比較するのに 2 サイクルを要するものとする．現在の一般的なアーキテクチャでは，CPU コア内部のクロック速度は，外部のメモリバスのクロック速度の 10 倍程度で動作している．そのため，レジスタやメモリと，CPU コア外部にある再利用表との比較に要するコストとして，上記のコスト設定は現実的である．

5.2 評価結果

評価モデルとして，以下のものについて測定した．

- (N) : メモ化を行わない場合
- (M) : メモ化のみを行う場合 (従来手法)
- (S-AA): メモ化に提案手法を組み合わせ，
  コピーアルゴリズム A，パージアルゴリズム A を用いた場合
- (S-BA) : メモ化に提案手法を組み合わせ，
  コピーアルゴリズム B，パージアルゴリズム A を用いた場合
- (S-BB) : メモ化に提案手法を組み合わせ，
  コピーアルゴリズム B，パージアルゴリズム B を用いた場合
- (M-V) : メモ化にオーバヘッドフィルタを動作させた場合
- (S-AA-V) : メモ化に提案手法を組み合わせ，コピーアルゴリズム A，
  パージアルゴリズム A を用い，オーバヘッドフィルタを動作させた場合
- (S-BA-V) : メモ化に提案手法を組み合わせ，コピーアルゴリズム B，
  パージアルゴリズム A を用い，オーバヘッドフィルタを動作させた場合
- (S-BB-V) : メモ化に提案手法を組み合わせ，コピーアルゴリズム B，
  パージアルゴリズム B を用い，オーバヘッドフィルタを動作させた場合

5.2.1 コピーアルゴリズムの検証

まず，2 つのコピーアルゴリズムによる，提案手法の効果の違いを検証した．パージアルゴリズムはどちらも A とし，オーバヘッドフィルタを動作させた場合と，動作させない場合の 2 種類について評価を行った．オーバヘッドフィルタを動作させずに評価を行った SPEC CPU95 ベンチマークの結果を**図 14** に示す．評価グラフは左から順に，メモ化無し (N)，従来手法 (M)，コピーアルゴリズム A を用いた (S-AA)，コピーアルゴリズム B を用いた (S-BA) である．

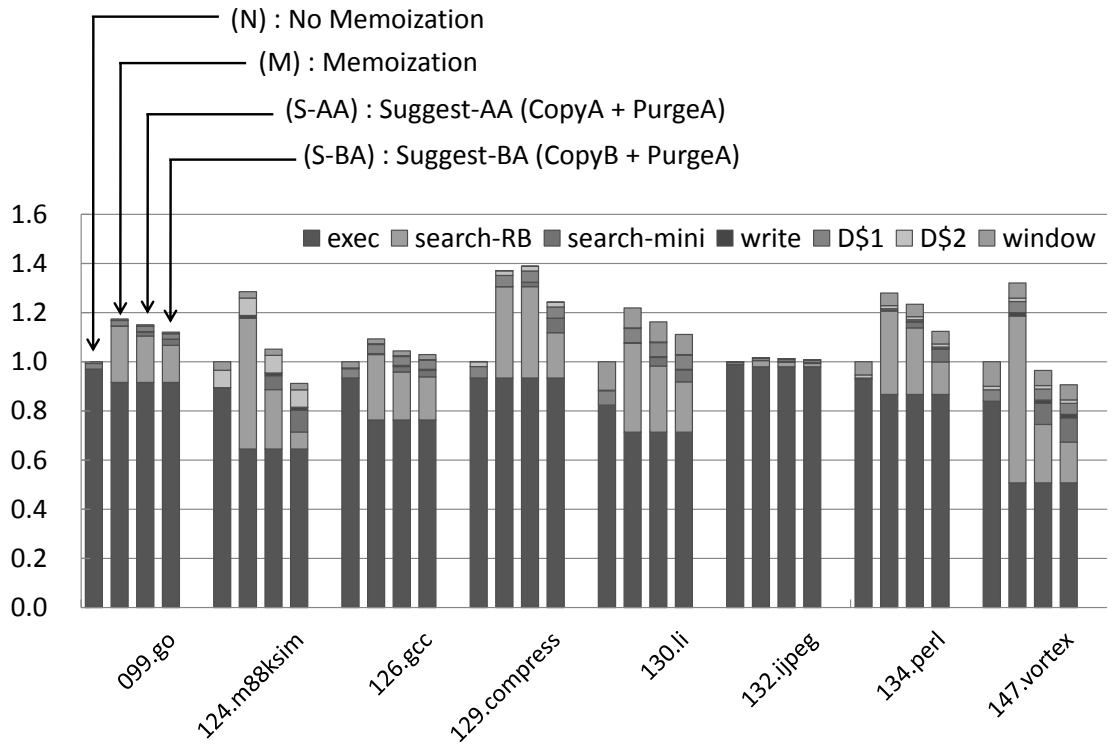

図 14: 評価結果 1

グラフは各モデルの総実行サイクル数を表しており，それぞれメモ化を行わない場合 (N) の総実行サイクル数を 1 として正規化した．凡例は順に，exec は命令の実行に要したサイクル数，search-RB は largeRB の検索に要したサイクル数，search-mini は miniRB の検索に要したサイクル数，write は MemoTbl から出力を書き戻す際に要したサイクル数，D$1 は 1 次データキャッシュミスペナルティのサイクル数，D$2 は 2 次データキャッシュミスペナルティのサイクル数，window は SPARC アーキテクチャ特有のレジスタウィンドウミスによって要したサイクル数を示している．

多くのプログラムにおいて，従来手法 (M) よりもコピーアルゴリズム A を用いた (S-AA) は性能が向上している．また，全てのプログラムにおいて，コピーアルゴリズム A を用いた (S-AA) よりも，コピーアルゴリズム B を用いた (S-BA) の方がサイクル数を削減できた．しかし，メモ化なし (N) に比べて高速化できているプログラムは 124.m88ksim と 147.vortex だけである．

次に，オーバヘッドフィルタを動作させた場合の評価結果を**図 15** に示す．評価グラフは左から順に，メモ化無 (N)，従来手法にオーバヘッドフィルタを動作させた (M-V)，コピーアルゴリズム A にオーバヘッドフィルタを動作させた (S-AA-V)，コピーアル

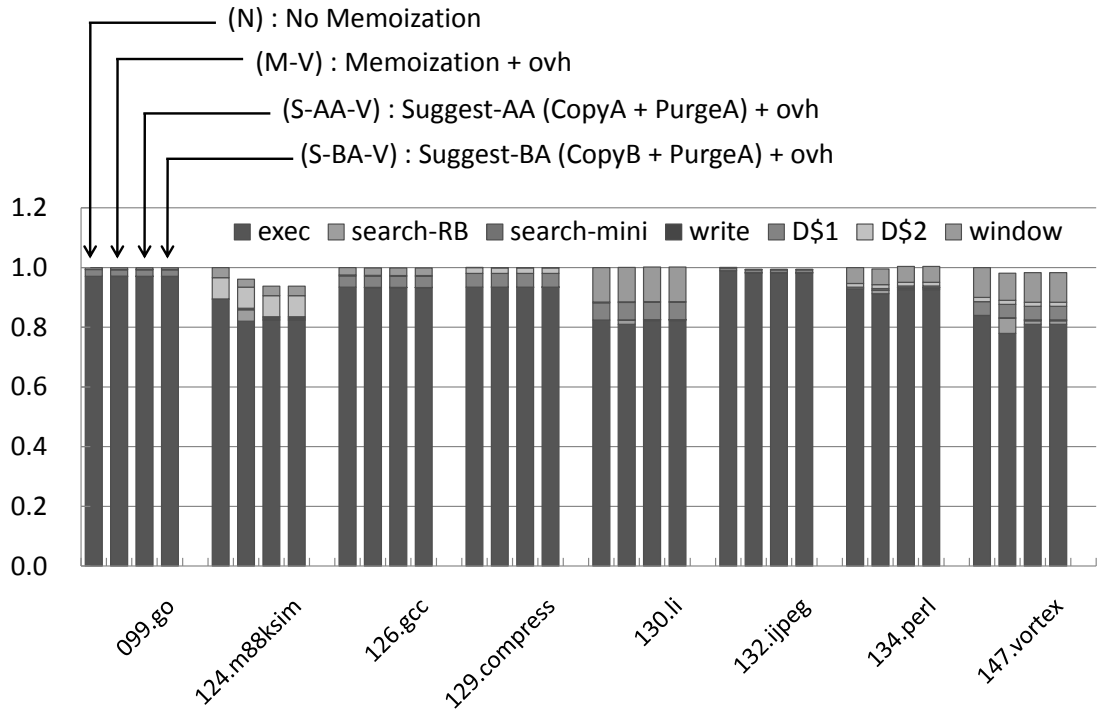

図 15: 評価結果 2

ゴリズム B にオーバヘッドフィルタを動作させた (S-BA-V) である．

こちらではコピーアルゴリズムの違いによる提案手法の効果の違いはあまり見られなかった．オーバヘッドフィルタを動作させた場合，従来手法 (M-V) のオーバヘッド自体が少なくなるため提案手法による再利用オーバヘッドの大幅な削減はできていないが，メモ化なし (N) より性能が悪化しているプログラムはない．特に 124.m88ksim は従来手法 (M-V) による再利用オーバヘッドのほとんどを削減できている．一方，(S-AA), (S-BA) では高速化できていた 147.vortex は，提案手法により削減できたサイクル数の分だけ exec が増加してしまっており，従来手法 (M-V) と同等なサイクル数になっている．

4.2.1 項で，コピーアルゴリズム B は再利用の効果がないプログラムでも，提案手法における効果が期待できると述べた．実際，129.compress のように再利用による高速化が望めないプログラムは，コピーアルゴリズム A を用いた場合，従来手法より miniRB での検索オーバヘッドの分だけ性能が悪化しており，コピーアルゴリズム B を用いた場合は従来手法より性能が向上している．しかし今回の評価を通じて，再利用の効果があるプログラムにおいてもコピーアルゴリズム A を用いた場合より，コピーアルゴ

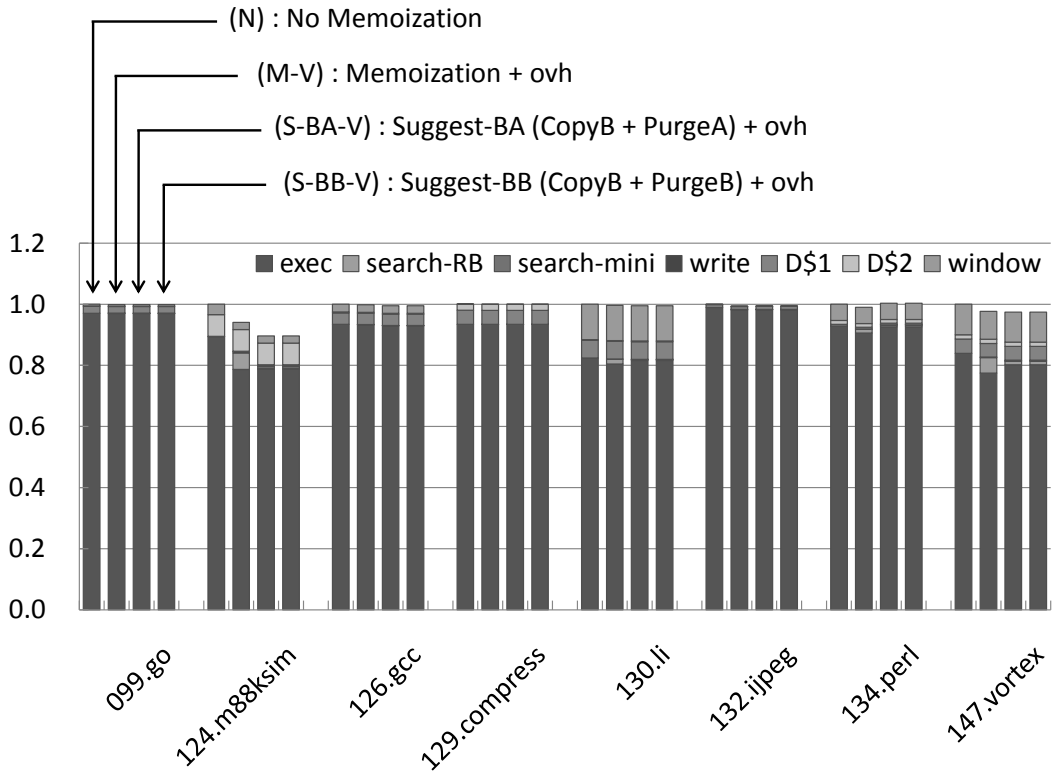

図 16: 評価結果 3

リズム B を用いた場合の方が検索オーバヘッドを削減できることが分かった．これは，再利用の効果があるプログラムにも再利用の効果がない関数があり，再利用の効果がないプログラムと同じような現象が起きていると考えられる．

結果をまとめると，SPEC CPU95 ベンチマークでメモ化無し (N) に比べ，従来手法 (M) では最大で 3.9 %，平均で 0.9 %のサイクル数の削減だったのに対し，提案手法では最大で 6.3 %，平均で 1.1 %のサイクル数を削減できた．

5.2.2 パージアルゴリズムの検証

5.2.1 項で示した結果より，コピーアルゴリズム B はコピーアルゴリズム A より同等もしくは良い結果をもたらすことが分かった．そのため，パージアルゴリズムによる提案手法の効果の違いを検証するにあたっては，コピーアルゴリズムを B に固定し，オーバヘッドフィルタを動作させて評価を行った．

評価結果を**図 16** に示す．評価グラフは左から順に，メモ化無し (N)，従来手法にオーバヘッドフィルタを動作させた (M-V)，パージアルゴリズム A にオーバヘッドフィルタを動作させた (S-BA-V)，コピーアルゴリズム B にオーバヘッドフィルタを動作させた (S-BB-V) である．(S-BB-V) のグラフは，4.2.2 で示した計算式の n を 0，m を 1 と

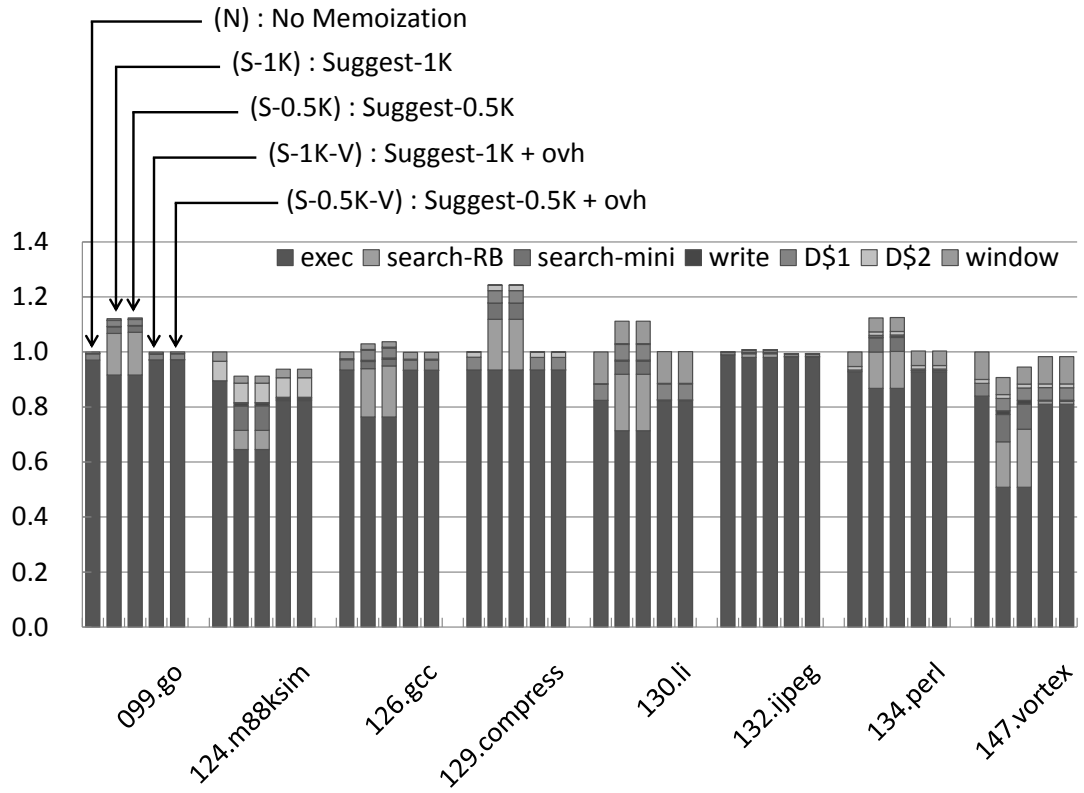

図 17: 評価結果 4

している．

グラフは図 15 のグラフと同様，各モデルの総実行サイクル数を表しており，それぞれメモ化を行わない場合 (N) の総実行サイクル数を 1 として正規化している．凡例も図 15 のグラフと同様である．

すべてのプログラムにおいて 2 つのパージアルゴリズムによる提案手法の効果の違いは見られなかった．その理由としてコピーアルゴリズム A，パージアルゴリズム B を用いた提案手法 (S-BA-V) によって，従来手法において発生していた検索コストのほとんどを削減できていることが挙げられる．

5.2.3 miniRB のサイズの検証

次に，miniRB のサイズの検証を行う．これまでに行ってきた評価は全て，miniRB のエントリ数を 1K，つまり miniRB のサイズを 128KByte に固定して行ってきた．ここでは，エントリ数が 0.5K，サイズが 64KByte の miniRB を用いた場合，どのように性能が変化するか，評価を行った．

評価結果を**図 17** に示す．評価グラフは左から順に，メモ化無 (N)，miniRB のエン

トリ数を 1K とした提案手法 (S-1K)，miniRB のエントリ数を 0.5K とした提案手法 (S-0.5K)，miniRB のエントリ数を 1K とした提案手法にオーバヘッドフィルタを動作させたもの (S-1K-V)，miniRB のエントリ数を 0.5K とした提案手法にオーバヘッドフィルタを動作させたもの (S-0.5K-V) である．提案手法のグラフは，全てコピーアルゴリズム B，パージアルゴリズム A を用いている．miniRB の比較コストは，miniRB のエントリ数が 0.5K の場合も，今までと同様にレジスタとの比較に 1cycle，メモリとの比較に 2cycle かかるものとして評価を行った．

147.vortex のオーバヘッドフィルタを動作させなかった場合を除いて，miniRB のエントリ数を 0.5K にすることによる，大きな性能の悪化は見られなかった．最も性能が悪化している 147.vortex のオーバヘッドフィルタを動作させなかった場合は，4 %性能が悪化ししまっている．しかし，今回の評価では，1K 及び 0.5K で同じ比較コストを想定したが，実際は 0.5K の CAM の方がレイテンシは低いこと．また，今回再利用オーバヘッドの変化を確認するためにオーバベッドフィルタを無効にしたモデルの評価も行ったが，本来のモデルはオーバヘッドフィルタを有効にしたものであることを考慮すると，この性能悪化は大きな問題でないと考えられる．以上の評価結果から，miniRB のエントリ数をより現実的な 0.5K 行と想定した場合においても，本モデルは有効であると言える．

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の更なる高速化として，再利用表を，今日の一般的なキャッシュのように多段化することで，検索コストを削減する手法を提案した．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，メモ化無しに比べ，従来手法では最大で 3.9 %，平均で 0.9 %のサイクル数の削減だったのに対し，提案手法では最大で 6.3 %，平均で 1.1 %のサイクル数を削減できた．

本研究の今後の課題として，複数コアの利用やループ再利用の適用など，既存の自動メモ化プロセッサの機能に対応することが挙げられる．本研究における提案手法は 1 コアでの動作しか検証しておらず，評価結果は既存の複数コアを用いた自動メモ化プロセッサの性能には及ばなかった．また，ループ再利用を適用可能な自動メモ化プロセッサに対しても同様であった．そこで，これらの既存の研究に，再利用表の多段化を組み込むことで，新しい知見が得られると考えられる．

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．とくに津邑先生には本研究を進めるにあたって，終始御指導をいただきました．

本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室およびの齋藤研究室の皆様に深く感謝致します．中でも，神谷優志氏には本研究を進めるにあたって，終始ご支援して頂きました．また，高木伴彰氏，池谷友基氏にも様々な相談や検討を通じご支援をして頂きました．ここに感謝致します．

参考文献

[1] 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).

[2] 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).

[3] 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).

[4] 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).

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

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

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