Vol.2011-ARC-196 No.2
2011/7/27

# 自動メモ化プロセッサの入力値エントリ統合による高速化

小田遼亮[†1] 山田龍寛[†1] 池谷友基[†1]
津邑公暁[†1] 松尾啓志[†1] 中島康彦[†2]

我々は，計算再利用技術に基づく自動メモ化プロセッサを提案している．この自動メモ化プロセッサの実現のために必要となる再利用表は，エントリ幅の限られた汎用 CAM を用いるため，複数のエントリを用いてツリー構造を作る事で入力セット全体を記憶する．本稿では複数の入力値エントリを統合する事で効率的な入力値の格納を実現し，登録エントリ数と検索オーバヘッドの削減を図る手法を提案する．また，エントリ統合を適用した場合にツリー構造を正しく保つ手法についても提案する．SPEC CPU95 を用いてシミュレーションにより評価した結果，従来モデルでは最大 40.5%，平均 10.5%であったサイクル数削減率が，最大 50.0%，平均 16.4%まで向上することを確認した．

# Input Entry Integration for Auto-Memoization Processor

Ryosuke Oda,[†1] Tatsuhiro Yamada,[†1]
Tomoki Ikegaya,[†1] Tomoaki Tsumura,[†1]
Hiroshi Matsuo[†1] and Yasuhiko Nakashima[†2]

We have proposed an auto-memoization processor based on computation reuse. The table for registering inputs/outputs is implemented by a ternary CAM, and the input sequences are stored onto the table, being folded into tree forms. This paper proposes a new registration method for merging multiple input entries into a single entry. We also proposes a model for keeping tree structures correctly. The new model can efficiently store input values and can reduce the search cost. The result of the experiment with SPEC CPU95 suite benchmarks shows that the new method improves the maximum speedup from 40.5%to 50.0%, and the average speedup from 10.5%to 16.4%.

†1 名古屋工業大学
Nagoya Institute of Technology
†2 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

## 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ともない，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 やスーパスカラ等の命令レベル並列性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ともなってコア数も増大していくと考えられている．さて，これらの高速化手法は粒度の違いはあれど，いずれもプログラムの持つ並列性に着目したものである．

一方我々は，計算再利用技術に基づいた高速化手法である自動メモ化プロセッサ[1)2)] を提案している．並列化が，処理全体の総量自体は変化させず複数の単位処理を同時実行することにより高速化を図る手法であるのに対し，計算再利用は処理自体を省略することで高速化を図る手法であり，その着眼点は根本的に異なっている．自動メモ化プロセッサは，関数およびループを計算再利用可能な命令区間と見なし，実行時にその入出力を記憶しておくことで，再び同一命令区間を同一入力を用いて実行しようとした際に，その実行自体を省略する．計算再利用は並列化とは直交する概念であるため，並列化が有効でないプログラムでも効果が得られる可能性があり，並列化とも併用可能であるという利点がある．

この自動メモ化プロセッサは，入力の記憶に汎用 CAM(Content Addressable Memory) を用いる．エントリ幅の限られた CAM では，1 エントリで入力セット全体を記憶する事はできないため，複数のエントリを用いてツリー構造を作る事により入力セット全体を記憶する．

本稿では，従来の自動メモ化プロセッサが入力値を読み出した順にエントリ登録を行っていたのに対して，エントリの登録順を並び替える事によって，複数の入力値エントリを統合可能にする手法を提案する．これにより，登録エントリ数と検索コストの削減を図る．また，この手法を適用した場合に入力値のツリー構造を正しく保つ手法についても提案する．

## 2. 自動メモ化プロセッサ

本章では，本稿で取り扱う自動メモ化プロセッサについて，その高速化の方針，アーキテクチャの構成，動作を概説する．

### 2.1 再利用機構

再利用とは，プログラムの関数呼出しやループなどの命令区間において，その入力と出力の

Vol.2011-ARC-196 No.2
2011/7/27

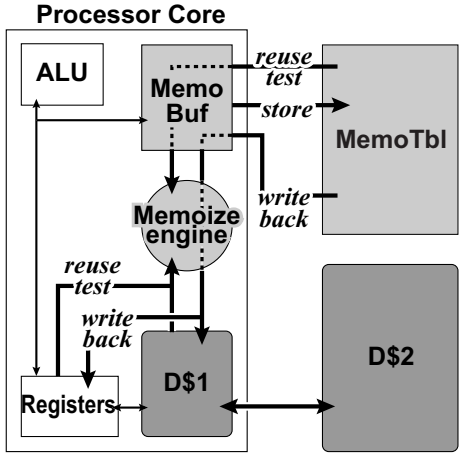



**図 1** 自動メモ化プロセッサの構成
Fig. 1 Structure of Auto-Memoization Processor.

組を記憶しておき，再び同じ入力によりその命令区間が実行されようとした場合に，過去の記憶された出力を利用する事で命令区間の実行自体を省略し，高速化を図る手法である．また，この手法を命令区間に適用する事をメモ化 (Memoization)[3] と呼ぶ．メモ化手法には，ハードウェアによるもの[4] やソフトウェアによるもの[5]，またその両方を利用したもの[6] など，様々なものが提案されている．

我々はハードウェアを用いて動的にメモ化を適用する事で，既存のバイナリを変更する事無く高速実行可能なプロセッサとして，自動メモ化プロセッサを提案している．自動メモ化プロセッサの構造の概略を**図 1** に示す．自動メモ化プロセッサは，コアの内部に一般的な CPU コアが持つ ALU，レジスタ (Reg)，1 次データキャッシュ (D$1) 等を持ち，コアの外部に 2 次データキャッシュ (D$2) を持つ．

また，自動メモ化プロセッサ独自の機構として，メモ化制御機構，再利用表 MemoTbl，及び MemoTbl への書き込みバッファ MemoBuf を持つ．MemoTbl は命令区間及びその入出力を記憶するための表であり，メモ化を実現するために必要となる．しかし，コアが命令区間の入出力を MemoTbl に登録する時，サイズの大きい MemoTbl に対して毎回参照を行うとオーバヘッドが大きくなってしまう．そのため，このオーバーヘッドを軽減するために，作業用の小さなバッファである MemoBuf をコアの内部に設けている．命令区間実行開始時には MemoTbl

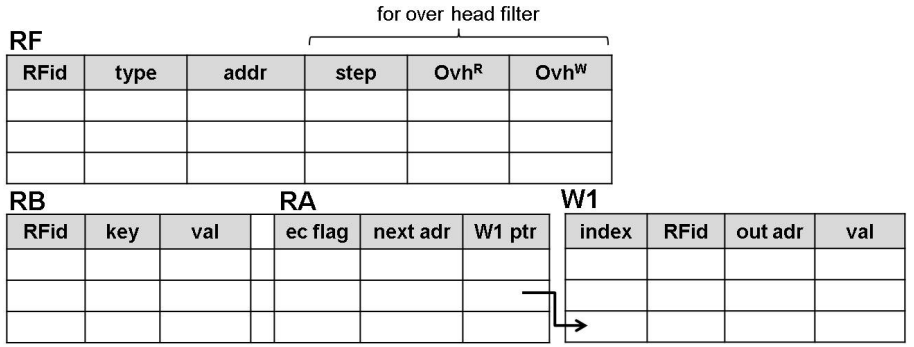



**図 2** MemoTbl の構造
Fig. 2 Structure of MemoTbl.

を参照し，過去の入力との一致比較を行う．一致するエントリが存在した場合，対応する出力が書き戻され，命令区間の実行は省略される．一致するエントリが存在しなかった場合，入出力を 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 MemoTbl に格納することで将来の再利用に備える．

MemoTbl は，命令区間の開始アドレスを記憶する RF，入力値を記憶する RB，入力アドレスを記憶する RA，そして出力値を記憶する W1 の 4 つの表から構成されている．この MemoTbl の詳細な構成を**図 2** に示す．

RF は，各再利用対象命令区間に対応する行を持っている．そして，他の各表のエントリが RF のエントリインデクスである RFid を用いて，そのエントリがどの命令区間に対応しているかを管理している．各命令区間の判別のためには，関数とループを判別するフラグ type，命令区間の開始アドレス addr を用いる．また，RF は後述するオーバヘッドフィルタにおいて使用される値も持つ．

RB は高速な連想検索が可能な汎用 3 値 CAM で構成される．これにより，入力一致比較時に MemoTbl を検索するオーバヘッドを小さく抑えることができる．ここで，CAM の各ラインのエントリ幅は限られているため，1 エントリで 1 入力セットを記憶する事はできない．そのため，1 エントリでは 1 キャッシュブロック分の入力値を記憶し，ツリー構造を作る事によって入力セット全体を記憶する．RB は，このツリー構造を管理するために用いる親エントリのインデクス key と，入力値 val を持つ．また，RB は 3 値 CAM を用いるため 0 と 1 に加え

てドントケアを記憶する事ができる．ここで，ドントケアはキャッシュブロック中の参照されなかったアドレスの入力値を表現するために用い，検索時その値は比較されない．

RA は入力値検索のために，次に参照するべきキャッシュブロックアドレス next adr を記憶する．RA は RB と同数のエントリを持ち，各エントリは 1 対 1 に対応している．また，RA はそのエントリが入力値セットの終端であるか否かを示すフラグ ec flag を持ち，終端エントリである場合出力を記憶している表である W1 のエントリを指すポインタ W1 ptr も持つ．

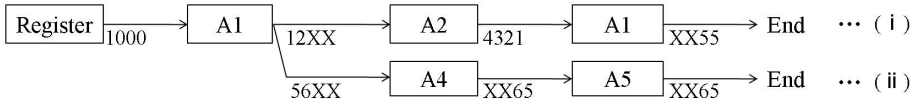
W1 は命令区間の出力先の主記憶アドレスと出力値を out adr と val に記憶する．全ての入力値が一致した場合，その入力に対応する出力を読み出しレジスタやメモリへ書き戻すことで，命令区間の実行を省略することができる．

### 2.2 入力値エントリ構造と検索

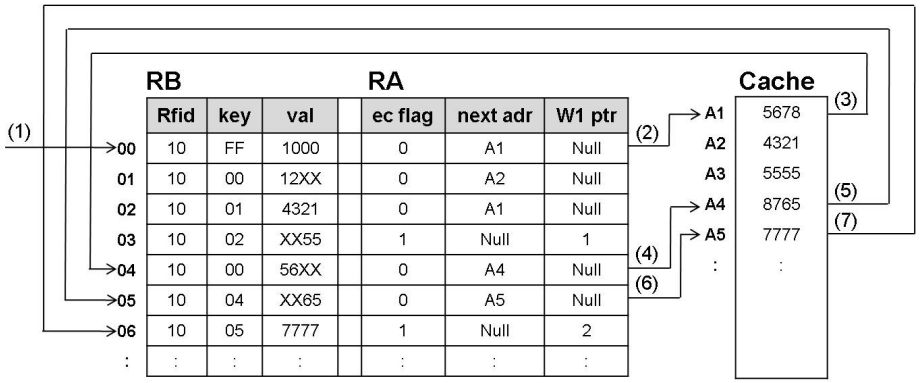
一般に命令区間内では，複数の入力が順に参照され使用される．しかし，同じ命令区間でも，ある入力の値が異なるとその次入力アドレスが変化する場合が存在する．これは，主記憶アドレス値自体が入力値として用いられる場合や，条件分岐の存在が原因である．このように，ある命令区間の入力アドレスの列はその入力値によって分岐してゆくため，その全入力パターンはツリー構造で表現する事ができる．このようなツリー構造で表現された入力パターンは，RB/RA を用いて記憶される．ここで，命令区間の入力値参照例をツリー構造により表現した概念図と，その入力を記憶している RB/RA の例を**図 3** に示す．

**図 3(a)** は入力値参照をツリー構造で表現した概念図であり，図中のノードは参照されるキャッシュブロックアドレス，エッジはその入力値を示している．ここでルートからリーフまでの経路の数が，入力セットのパターン数に対応する．また入力値中の X はドントケアであり，そのアドレスに対応する値は読み出されていない事を表す．これらの入力セットにおいて，レジスタの値が参照された時，その値は 1000 であり次にアドレス A1 が参照されている．そして入力セット (i) では，A1 の前半に対応するアドレスの入力値が 12 であり次に A2 が参照されている一方，入力セット (ii) では，この入力値が 56 であり次に A4 が参照されている．また，End はそれ以上の入力が存在しない事を示す．

ここで，図 3(a) の入力セット (i) では，A1 への参照を表現するノードが複数存在している．これらのノードの内 1 つは A1 中の前半のアドレスから読み出した入力値を記憶しており，もう 1 つは後半のアドレスから読み出した入力値を記憶している．これは入力値による分岐を考慮し，入力値を読み出した順にツリー構造で表現しているからである．一方で，同一のキャッシュブロック中に存在するアドレスからの，連続した入力参照は 1 つのノードで表現する．これは，複数の連続した入力を 1 つのエントリに記憶可能である場合，それらを 1 つの長い入力




(a) 入力値参照




(b) RB/RA への格納と検索

**図 3** 入力値参照と RB/RA への格納と検索
Fig. 3 Input reference and, store to and search for RB/RA.

とみなす事で，全入力パターンを矛盾なく表現できるためである．例えば，入力セット (i) 中の A2 への参照では，A2 中に存在するアドレスからの入力が連続して読み出されているため，それらの入力値が 1 つのノードで表現されている．

**図 3(b)** は (a) の入力値参照を RB/RA へ実際に登録した時の格納例を示している．この例では，RFid が 10 番である命令区間として登録されている．また，図 3(b) は命令区間への入力値が，図 3(a) の入力セット (ii) における入力値と一致する場合の MenoTbl 検索フローの例も示している．この例では，再利用区間の実行開始アドレスが検出された時に，val が引数と一致し，key が FF であるルートエントリが検索され，ライン 00 で該当するエントリが発見される (1)．次に，ライン 00 の RA エントリにおける next_adr が A1 を指しているため A1 の値を参照し (2)，val がその値と一致し，key がライン 00 であるエントリを検索する (3)．以降同様の手順で検索が行われ (4)(5)(6)(7)，ライン 06 で入力値の一致を確認し検索を終了する．

入力値検索はこのように動作し，エントリをたどるたびに CAM を走査するため，検索コストはたどるべきエントリの数に比例する．

### 2.3 オーバヘッドフィルタ機構

自動メモ化プロセッサは，計算再利用可能な命令区間の実行を省略する事で高速化を図る手法であるが，その際には MemoTbl を検索するコスト，および入力が一致したエントリに対応する出力値を MemoTbl からレジスタやメモリに書き戻すコストがオーバヘッドとして発生する．命令区間によっては，これらのオーバーヘッドが大きく，再利用を適用する事で性能が悪化してしまう場合がある．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ためにオーバヘッドフィルタ機構を備えている．この機構では，命令区間を再利用する事で削減できたサイクル数を $step$，その再利用の検索時に発生したオーバヘッドを $Ovh^R$，書き戻し時に発生したオーバヘッドを $Ovh^W$ として，これらの値から削減予測サイクル数を

$$step - Ovh^R - Ovh^W \quad (1)$$

と計算する．この削減予測サイクル数 (1) が負である時には再利用の適用によりサイクル数が増加してしまうと考えられるため，再利用を中止する事によって性能悪化を防ぐ．

## 3. 入力値エントリ統合手法

本章では，本稿で提案する入力値エントリ統合を行う再利用モデルについて説明する．

### 3.1 エントリ統合

前章で述べたように，既存手法では同一キャッシュブロック中のアドレスから連続して入力値を読み出した場合には，それらを 1 つの長い入力値とみなし，1 つのエントリに記憶する．しかし，読み出しが連続しない場合，読み出し順を考慮し，それらを複数のエントリに分けて記憶する．このため，格納効率の低いエントリが存在する場合がある．そこで，エントリの登録順を並べ替える事によって複数のエントリを統合する入力値エントリ統合手法を提案する．この手法では，登録エントリ数と検索コストの削減を図る．また，提案手法では検索時の入力値の参照順が変わってしまうため，そのツリー構造を正しく保持する手法についても提案する．

このエントリ統合手法について，**図 4** のサンプルプログラムが，**表 1** に示す入力セットを入力として実行される場合を例に述べる．ここで，サンプルプログラム中の配列 a は 1000 番値から順に確保され，配列 b は 2000 番地から順に確保されているとする．また，サンプルプログラムにはループと関数が存在しているが，ここでは関数の入力値登録についてのみ着目する．

既存手法では，入力値が読み出された順にエントリを登録していくため，サンプルプログラムが入力セット L，M，N を入力として実行されると，**図 5(a)** に示すエントリ構造が作られる．

```
1  : int func(int *a, int *b){
2  :   int i;
3  :   int c[3];
4  :   for(i=0; i<2; i++) {
5  :       c[i] = a[i];
6  :       c[i] = b[i];
7  :   }
8  :   c[i] = a[i];
9  :   return 0;
10 : }
11 :      ...
```

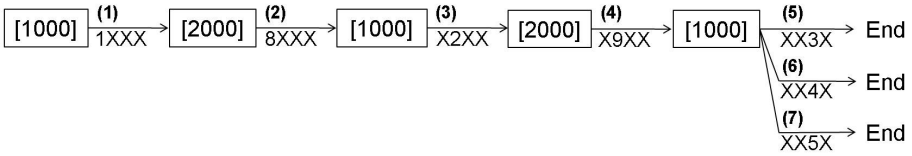
**図 4** サンプルプログラム
Fig. 4 sample program

**表 1** 入力例
Table 1 Example of input

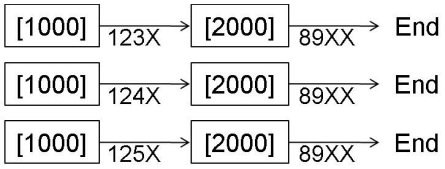
| | アドレス | 入力セット L | 入力セット M | 入力セット N |
|---|---|---|---|---|
| a[0] | 1000 | 1 | 1 | 1 |
| a[1] | 1004 | 2 | 2 | 2 |
| a[2] | 1008 | 3 | 4 | 5 |
| b[0] | 2000 | 8 | 8 | 8 |
| b[1] | 2004 | 9 | 9 | 9 |

入力セット L を入力としてサンプルプログラムが実行された場合には，5 行目において a[0] が，6 行目において b[0] が読み出され，その順にエントリが登録される (1)(2)．次のイタレーションにおいても a[1]，b[1] が順に読み出され同様にエントリが登録される (3)(4)．そして，8 行目では a[2] が読み出され，そのエントリが登録される (5)．このような順で入力値が読み出され MemoTbl へと登録されるため，各エントリには 1 つ以外の値がドントケアであるエントリが記憶される．また入力セット M，N が登録されると，これらの入力セットの間では a[2] の値のみが異なっているため，a[2] 以外のエントリが共有される (1)(2)(3)(4)．しかし，a[2] の入力値が異なるため，終端エントリは共有されず別の格納値を持つ (6)(7)．

一方，提案手法では入力値を読み出した順にエントリを登録していくが，読み出した値が他のエントリに統合可能であると判断すれば，その値を統合して登録する．そのため，サンプルプログラムが既存手法と同様の入力で実行されると，**図 5(b)** に示すエントリ構造が作られる．




(a) 既存手法




(b) 提案手法

**図 5** 入力値エントリ構造例
Fig. 5 Example of input entry structure

入力セット L を入力としてサンプルプログラムが実行された場合，まず a[0] と b[0] は先程と同様に登録される．しかし，a[1] を読み出した時に，このアドレスが a[0] と同一のキャッシュブロック中に存在する事から，a[0] のエントリに統合して a[1] を登録する．これにより，ルートエントリの 1004 番地用の領域に値が登録される．また，b[1] を読み出した時も同様に b[0] のエントリに統合して b[1] を登録する．続いて，8 行目で a[2] が読み出されると，これが a[0] および a[1] と同一のキャッシュブロック中に存在する事から，これらの値を統合して登録する．また入力セット M，N を入力としてサンプルプログラムが実行されると，入力セット L の場合と同様に入力値エントリ統合によって a[2] の値がルートエントリに統合して格納される．ここで a[2] の値は L，M，N 間で異なっているため，ルートエントリが共有されず，エントリ共有が全く起こらない．

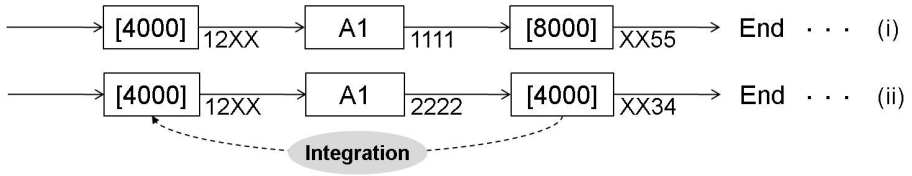
この登録例において，既存手法では全入力セットの登録に 7 エントリ使用するが，提案手法では 6 エントリで済み，登録エントリ数を削減する事ができている．しかし，提案手法では入力値の読み出し順を考慮しないため，ツリー構造を作りにくい場合が存在し，登録エントリ数が既存手法より増加してしまう可能性がある．例えば，図 4 のサンプルプログラムにおいて，a[2] のみ異なる入力セットが 5 つ以上登録された場合，提案手法の方が登録エントリ数が多くなってしまう．一方で，検索時にたどるべきエントリ数は過去にどのような入力セットがあっても変化せず，既存手法では 5 となり提案手法では 2 となる．このように，検索時にたどるべきエントリの数はツリー構造の共有と無関係であり，どのような場合にも入力値の統合により削減される．

### 3.2 ツリー構造の保持

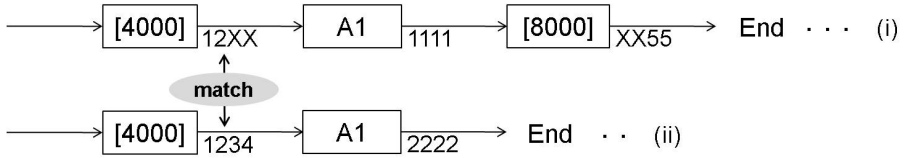
既存手法では入力値エントリを一意に特定するために，入力値を読み出した順にエントリを登録する．しかし，提案手法では入力値の読み出し順を考慮せずに複数のエントリを統合するため，ドントケアの位置の異なるエントリが存在する事になり，CAM 検索時に誤ったエントリの一致を検出してしまう可能性がある．入力値エントリが複数の入力セット間で共有可能であるかどうかを検索する際に誤った一致検索が起こると，誤ったツリー構造が作られてしまう．これにより不正な再利用が適用され，正しい実行結果が得られなくなってしまう可能性がある．このような誤ったツリー構造を作らないため，共有可能エントリ検索時には，CAM による検出が正しい検出であるかどうかをドントケアの位置を比較する事で確認する．

誤った一致の検出は，**図 6(a)** の例に示すようなエントリ統合が原因となって起こる．この例では，A1 中の値が条件分岐において読み出される事で，以降に入力アドレス列の変化が起きている．この条件分岐の結果，入力セット (i) ではエントリ統合が行われないが，入力セット (ii) ではエントリ統合が行われる．これにより，**図 6(b)** に示すように 4000 番のキャッシュブロックの値を記憶するエントリ間でドントケアの位置が異なるエントリが登録される．ここで，入力セット (ii) を MemoTbl に登録する際，入力セット (i) が既に MemoTbl に登録されていたとすると，これらの入力セット間でエントリが共有可能であるかどうかが検索される．この時 (ii) における 4000 番のキャッシュブロックへの格納値は 1234 であり，(i) の格納値は 12XX であるため，1234 は 12XX と一致すると判断される．これにより，4000 番のエントリが共有されてしまうと，**図 6(c)** に示すような誤ったツリー構造が作られてしまう．このようなツリー構造が作られると，4000 番のキャッシュブロックの後半の値が比較されずに再利用が適用される事があるため，誤った再利用が適用される可能性がある．

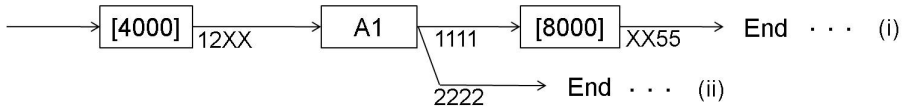
このように，誤った一致が検出されたエントリ間ではドントケアの位置が異なっている．そこで，提案手法では CAM の検索によって得られたエントリの入力値と，現在の入力値のドントケアの位置を比較する事により，正しい一致であるかどうかを確認する．この比較により，正しい一致では無いと判断された場合，**図 6(d)** に示すようにエントリの共有を行わず，新しい経路 (ii) を作成する．その結果，4000 番のキャッシュブロックへの格納値として 12XX を持つエントリと 1234 を持つエントリが MemoTbl に登録される事になる．この時 1234 を入力として持つエントリが検索されると，12XX を持つエントリと 1234 を持つエントリの両方



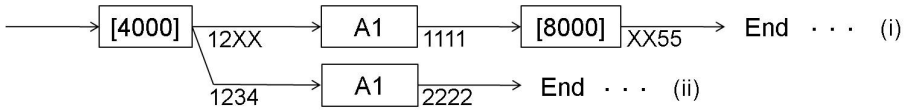

(a) 条件分岐をまたぐ入力値統合




(b) 誤った一致検出




(c) 誤ったツリー構造




(d) 修正されたツリー構造

**図 6** 誤ったツリー構造の修正
Fig. 6 Revision of false tree structure.

が一致するため，マルチマッチが発生する．本稿では，マルチマッチが起こった場合，それらのエントリの中から CAM 上で一番小さなアドレスを持つエントリが返されるよう，CAM のデコーダを構成する．これによって，マルチマッチが起こった場合，マッチした複数のエントリの中からいずれかのエントリが検出される事になる．共有可能エントリの検索時には，このように検出されたエントリに対しても同様にドントケア位置を比較し，正しい一致であればエントリを共有し，誤った一致であればエントリを共有しない．

再利用適用時の入力値検索の際に誤った検索一致が起こった場合，その入力セットに対して検索が続けられる事になる．これにより，以降の入力も比較される事になるため，誤った検索一致を引き起こす原因となった分岐において参照される値もいずれ比較され，その比較の際には不一致と判断される．また，マルチマッチが起こった場合には，マッチした複数の経路の中からいずれかの経路が選択され，それが正しい経路であれば再利用が成功し，誤った経路であれば再利用が失敗する．このように，再利用適用時の検索において，誤った検索一致やマルチマッチが起こったとしても誤った再利用が適用される事は無い．そのため，再利用適用時に特別な追加比較を行う必要が無く，追加のオーバーヘッドは発生しない．しかし，MemoTbl にエントリが登録されているにも関わらず，再利用に失敗する場合があるため，性能が低下してしまう可能性はある．

## 4. 実　　装

前章までは主記憶アドレスの入出力を例に述べてきたが，レジスタも再利用区間への入出力になる．そのため，自動メモ化プロセッサでは，参照された各種レジスタ値も入出力として登録する．本章では，これまでに述べてきた入力値エントリ統合の実現に必要となる実装について述べる．

MemoTbl への入力値の登録には 2.1 節で述べたように作業用のバッファとして MemoBuf を用いる．ここで，MemoBuf の詳細な構成を**図 7** に示す．MemoBuf は複数のエントリを持ち，1 エントリが 1 入出力セットに対応する．各エントリは，該当する命令区間を記憶する RFid，命令区間の実行開始時のスタックポインタ SP，関数の戻り値とループの終端アドレスを記憶する FOfs，命令区間の入力セットを記憶する Read，および出力セットを記憶する Write のフィールドからなる．これらのフィールドでも MemoTbl と同様にドントケアを記憶する必要があるため，全ビットに対してマスクビットを用意する事によりこれを実現している．

命令区間の実行中に入力が読み出されると，Write フィールドを検索し，その後 Read フィールドを検索することにより，既に入出力として登録されているかどうかを調査する．これは，既に入力として登録されている値は登録する必要が無く，出力として登録されている値は入力ではないため登録する必要が無いためである．主記憶アクセス値の登録を調査する場合には，Read および Write フィールド中の各入力エントリが，同一キャッシュブロックに存在するかどうか確認し，存在する場合にはマスクビットを比較する事で同一アドレスへの入力であるか確認する．また各種のレジスタ値を調査する場合には，その種類により同一エントリ上に存在する事を確認した後，マスクビットを比較する．

Vol.2011-ARC-196 No.2
2011/7/27

| RFid | SP | FOfs | Read | | | Write | | |
|---|---|---|---|---|---|---|---|---|
| | | | #1 | … | #n | #1 | … | #n |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |

**図 7** MemoBuf の構造
Fig. 7 Structure of Memobuf.

ここで，エントリを統合するためには，同一エントリへ複数の値を格納可能である事を判定する機構が必要となる．既存手法においても入力値による分岐に関わらないような場合は複数の入力値を同一のエントリへと登録している．そのため，この判定機構は既存の自動メモ化プロセッサにも存在する．そこで，提案手法における格納可否の判定にもこの機構を用いる．この判定機構では，レジスタアクセスの場合，レジスタの種類を比較し同じ種類のレジスタであれば同一のエントリに統合可能であると判断している．また，主記憶アクセスの場合，エントリの下位 3 ビットをマスクして一致比較をすることで，同一のキャッシュブロック中にある事を確認できた場合には統合可能であると判断している．

既存手法では Read フィールド検索時，複数の入力値を同一のエントリに格納可能である事を検出すると，入力順が変わらない場合に限り同一のエントリへと入力値を格納する．これについて**図 8** を用いて述べる．図 8 の Read フィールドには，引数 ARG，主記憶 8000 番地，global レジスタ GR，主記憶 4000 番地の入力が順に登録されている．このような場合，8004 番地からの読み出しが起こると，Traditional で示すように Read フィールドが#1 から順に検索され，#2 で同一キャッシュブロックのエントリが発見されマスクビットが比較される．これらのマスクビットは一致しないため同一キャッシュブロック中の別のアドレスの入力であるとわかる．しかし，このエントリは終端エントリでないため，入力順を考慮してエントリ統合は行われず，以降のフィールドが検索される．ここで，以降のフィールドには同一キャッシュブロックに対するエントリが存在しないため，空エントリである#5 に入力を登録する．

一方，提案手法では Read フィールドにおいて複数の入力値を同一のエントリに格納可能であると検出した場合には，入力順が変わるような場合にもそのエントリに値を統合して登録する．提案手法では 8004 番地からの読み出しが起こると，Proposal で示すように Read フィールドが検索され，既存手法と同様に#2 において同一キャッシュブロック中の別のアドレスからの入力を発見する．そこで，8004 番地からの入力を#2 に統合して登録する．

また，3.2 節で述べたように，エントリ登録時に誤った一致検索を行ってしまうと，誤った再利用を適用してしまう可能性がある．そこで提案手法では，CAM の検索により得られたエントリの入力値と，現在の入力値エントリのドントケアの位置の比較を行うハードウェアが必要となる．3 値 CAM におけるドントケアは，一般的にマスクビットを用いる事で表現されているため，ドントケア位置を比較するためには MemoBuf と MemoTbl のマスクビットを比較すれば良い．このため，マスクビットを比較するためのハードウェアを自動メモ化プロセッサに追加する．なお，エントリ登録時の一致比較は通常実行と並行して行われるため，これにより通常実行がストールする事は無い．

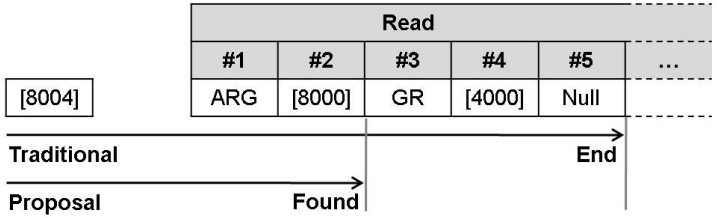



**図 8** Read フィールド検索
Fig. 8 Search to Read Field

## 5. 評　　価

以上で述べた実装を既存の自動メモ化プロセッサシミュレータに対して行った．また，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サイクルベースシミュレーションによる評価を行った．

### 5.1 評価環境

評価には，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．評価に用いたパラメータを**表 2** に示す．なお，キャッシュや命令レイテンシは SPARC64-III[7] を参考とした．また，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[8] を参考にし，サイズは 32 バイト幅 ×4k 行の 128kByte とした．また，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ーヘッドを見積もっている．

### 5.2 評価結果

提案手法の有効性を確かめるため，汎用ベンチマークプログラムである SPEC CPU95 ベンチマークについて平均登録エントリ数，平均検索コストおよび実行サイクル数を評価した．

**表 2** 評価環境
Table 2 Simulation parameters.

| | |
|---|---|
| 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 |
| MemoTbl サイズ | 32bytes × 4K 行 (128KBytes) |
| レジスタ ⇔ CAM | 9cycles/32bytes |
| メモリ ⇔ CAM | 10cycles/32bytes |
| CAM ⇒ レジスタ or メモリ | 1cycles/32bytes |

まず，平均登録エントリ数と平均検索コストを評価した結果を**図 9** に示す．ここで，平均登録エントリ数とは，登録された RB エントリの総数を登録された入力セットの総数で割った値であり，平均検索コストとは，入力セットを検索する時にたどる RB エントリ数の総数を登録された入力セットの総数で割った値である．図 9 中では各ベンチマークプログラムの結果を 2 本のグラフで示しているが，それぞれ左から順に，エントリ統合手法を適用した場合の平均登録エントリ数，平均検索コストを示している．なお，各計測値は既存手法における値を 1 とする正規化を行っている．

図 9 のグラフより，多くのプログラムにおいて平均登録エントリ数と平均検索コストの双方が大きく削減できた事が分かる．特に 125 では，平均登録エントリ数を 73.7%削減でき，平均検索コストを 69.5%削減できている．一方，107 における平均登録エントリ数と平均検索コストは大幅に増加している事が分かる．そこで，107 について詳細な調査を行った所，検索コストが削減される事で，既存手法では登録されなかった非常に検索コストの高い再利用区間が MemoTbl へと登録されるようになっていた．これにより，平均登録エントリ数や検索コストの平均値が大幅に増加してしまっている事が分かった．また，平均登録エントリ数削減率と平均検索コスト削減率は同程度である場合が多いが，124，147，107，141 では平均検索コストの方が大きく削減されている．特に，147 では検索コストは既存手法よりも削減できているが，平均登録エントリ数は既存手法よりも僅かに増加してしまっていた．これは 3 章で述べたように，提案手法を適用した事によりエントリの共有が起こりにくくなった事で，平均登録エントリ数が増加してしまった事が原因であると考えられる．このような場合，平均検索コスト減少による高速化が平均登録エントリ数の増加による低速化を上回ると，提案手法適用により性能が向上する．

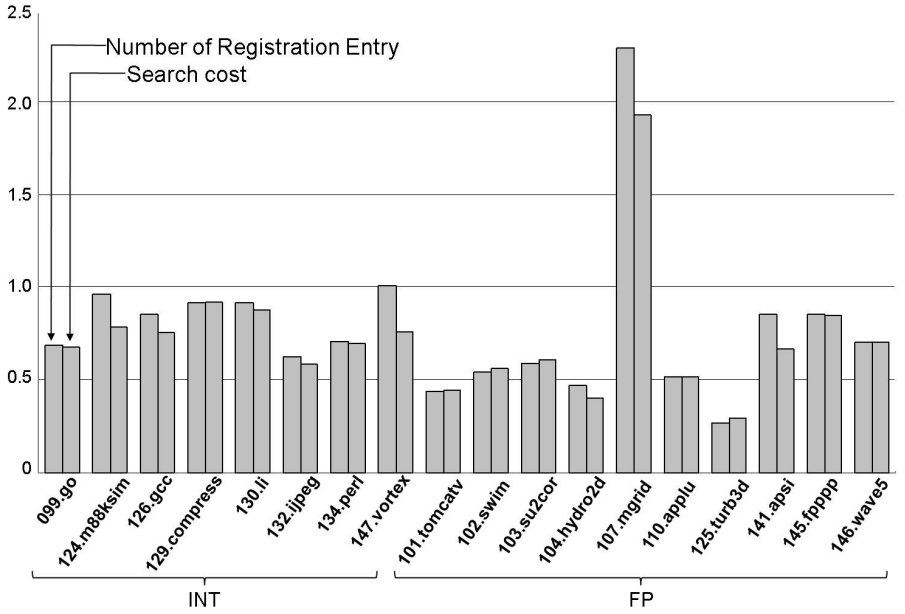



**図 9** 平均登録エントリ数と平均検索コスト
Fig. 9 Average store entry number and Average searching Cost.

続いて，実行サイクル数についての評価を行った．この結果を**表 3** および**図 10** に示す．図 10 中で各ベンチマークプログラムの結果を 3 本のグラフで表しているが，それぞれ左から順に

**(N)** メモ化を行わないモデル
**(M)** 既存モデル
**(I)** 入力値統合モデル

となっている．グラフは各モデルの総実行サイクル数を表しており，それぞれメモ化なし (N) を 1 として正規化した．また図 10 の凡例はサイクル数の内訳を示しており，exec は命令サ

**表 3** 削減サイクル数率 (SPEC CPU95)
Table 3 Reduced execution cycles. (SPEC CPU95)

| | Mean | Max |
|---|---|---|
| (M) 既存モデル | **10.5%** | **40.5%** |
| (I) 提案モデル | **16.4%** | **50.0%** |

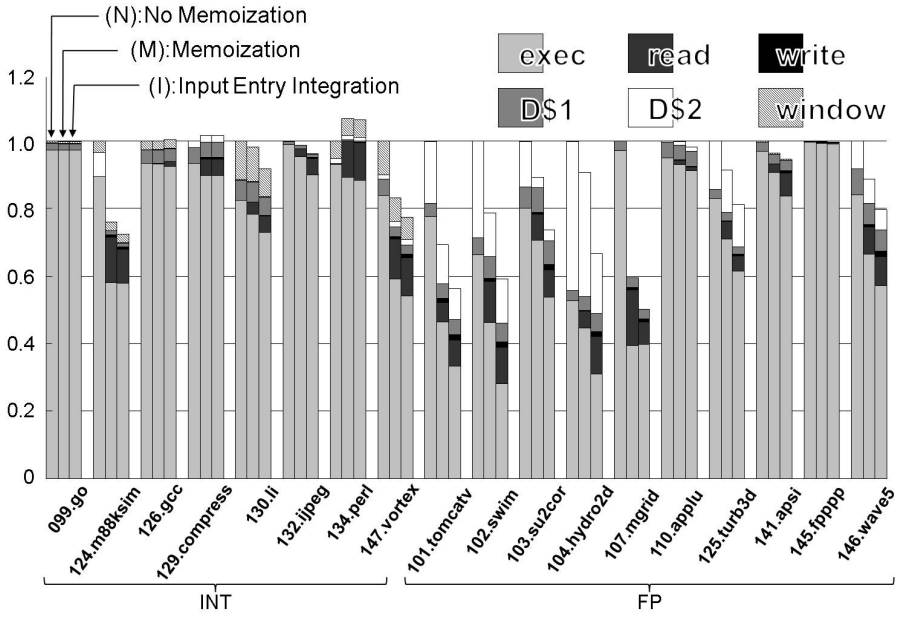



**図 10** 実行サイクル数
Fig. 10 Ratio of cycles.

イクル数，read は MemoTbl との比較に要したサイクル数 (検索オーバーヘッド)，write は MemoTbl の出力をレジスタやメモリに書き込む際に要したサイクル数 (書き戻しコスト)，D$1 および D$2 は 1 次と 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，平均登録エントリ数と平均検索コストが大幅に増加してしまった 107 についてサイクル数をみると，その read は削減されている．これは，提案手法を適用する事によって，新たに MemoTbl へと登録されるようになった検索コストの高い再利用区間が原因で検索コストの平均値は高くなったが，他の再利用区間の検索コストが減少しているためである．しかしながら，その exec は増加してしまっている．これは，平均登録エントリ数の多い再利用区間が登録される事により，これまで再利用が成功していた区間が，MemoTbl から追い出されてしまった事が原因であると考えられる．

**表 4** 関数区間 0x4e218 の詳細
Table 4 detail of 0x4e218.

| | 既存モデル (M) | 提案モデル (I) |
|---|---|---|
| 平均登録エントリ数 | 6.2 | 8.0 |
| 平均検索コスト | 10.0 | 8.0 |
| 再利用中止回数 | 1317472 | 961174 |
| 再利用成功回数 | 1992350 | 2062569 |

また，平均登録エントリ数のみが既存手法よりも増加していた 147 について見ると，提案手法を適用する事により総実行サイクル数と exec の双方が削減されている．この理由を確認するために，147 について詳細な調査を行ったところ開始アドレスが 0x4e218 である関数区間でオーバーヘッドフィルタによる再利用中止回数の減少および再利用成功回数の増加を確認した．この関数に関する情報を**表 4** にまとめた．この関数では，提案手法を適用する事で平均登録エントリ数が増加してしまっているが，平均検索コストは削減されている．また，提案手法適用によりオーバーヘッドフィルタによる再利用中止回数が減少し，再利用成功回数が増加している事も確認できた．

一方で，再利用率が向上する事によってかえって性能が悪化する場合も存在し，126 では read の増加によって総実行サイクルが増加してしまっている．このような性能悪化はオーバーヘッドフィルタの精度の低さが原因であると考えられる．

また，平均検索コストを削減する事ができる場合にも再利用率の向上が図れない場合がある．元々再利用が成功しにくいアクセスパターンを持つプログラムである 099 や 145 では，提案手法の適用による変化は見られなかった．また，既存手法においてオーバーヘッドフィルタによる再利用中止が少なかった 124 では，read を削減する事ができたが，exec にはほぼ変化がなく再利用率は向上していなかった．

しかしながら，多くのプログラムでは検索コスト削減に伴う再利用率が向上による，実行サイクル数の削減が確認できた．

Vol.2011-ARC-196 No.2
2011/7/27

## 6. おわりに

本稿では，従来の自動メモ化プロセッサの高速化手法として，入力値エントリの統合手法を提案した．提案手法の有効性を確認するため，SPEC CPU95 ベンチマークを用いて評価を行った結果，登録エントリ数と検索コストを削減できる事を確認した．また，従来モデルでは最大 40.5%，平均 10.5%であったサイクル数削減率が，最大 50.0%，平均 16.4%まで向上することを確認し，提案手法の有効性を確認できた．

本研究の今後の課題としては，できるだけツリー構造を作らない登録を行う手法による更なる高速化の検討が挙げられる．これは，登録エントリ数が増加した場合であっても，検索コストが削減されていれば高速化が可能であるという事がわかったためである．つまり，登録エントリ数が増加したとしても，検索コストが削減できるようなエントリ構成を目指す事により，更なる高速化の可能性があると考えられる．

## 参 考 文 献

1) 池谷友基，津邑公暁，松尾啓志，中島康彦：複数イタレーションの一括再利用による並列事前実行の高速化，情報処理学会論文誌 コンピューティングシステム（ACS），Vol.3, No.3, pp.31–43 (2010).
2) 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).
3) Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).
4) González, A., Tubella, J. and Molina, C.: Trace-Level Reuse, *Proc. International Conference on Parallel Processing*, pp.30–37 (1999).
5) Huang, J. and Lilja, D.J.: Exploiting Basic Block Value Locality with Block Reuse, *Proc. 5th International Symposium on High-Performance Computer Architecture*, pp.106–114 (1999).
6) Connors, D.A. and Hwu, W.W.: Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results, *32nd MICRO* (1999).
7) HAL Computer Systems/Fujitsu: *SPARC64-III User's Guide* (1998).
8) ルネサステクノロジ：ニュースリリース: R8A20410BG (2009).