卒業研究論文

計算再利用の部分的適用による
自動メモ化プロセッサの高速化

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 20 年度入学　20115049 番

神村 和敬

平成 24 年 2 月 8 日

計算再利用の部分的適用による自動メモ化プロセッサの高速化

神村 和敬

内容梗概

ゲート遅延に対する配線遅延の相対的な増大から，クロック周波数の向上だけではマイクロプロセッサの速度向上を見込めなくなってきた．また，集積回路の微細化に伴う消費電力や発熱量の増大からクロック周波数自体の向上も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 を持たないことから，これらの手法にも限界がある．そこで，計算再利用をハードウェアにより動的に適用する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，ある命令区間への入力に対応する出力を記憶しておき，その命令区間を過去と同一の入力で実行する場合に，過去に記憶しておいた出力を用いて，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再利用表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，現在の入力と過去に再利用表に記憶しておいた入力とを比較し，それらの値が完全に一致した場合，過去の出力を利用することで実行を省略する．そのため既存の自動メモ化プロセッサでは，この一致比較において入力値が一つでも異なる場合，再利用を行わない．これは，入力値が一つでも異なれば，その命令区間の実行結果は過去の実行結果と異なると考えられるためである．検索が失敗した場合には，再利用による実行の省略ができないばかりでなく，検索にかかったオーバヘッドが発生するため，実行サイクル数が増加してしまう．しかしながら，再利用が失敗した場合にも，いくつかの入力値は一致している可能性がある．

そこで本研究では，このことに着目し，再利用の対象とする命令区間を，既存の再利用対象区間の先頭から入力値が一致している範囲までの区間とする手法を提案する．これにより，既存の自動メモ化プロセッサでは検索に失敗する場合でも，部分的に命令の実行を省略することが可能となり，高速化を実現できると考えられる．

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 INT ベンチマークでシミュレーションにより評価した．その結果，通常通り命令を実行するのと比較し，従来手法では最大 24.0%，平均 4.9%のサイクル

数の削減であったのに対し，提案手法では最大25.5%，平均5.4%のサイクル数を削減し有効性を確認した．

計算再利用の部分的適用による自動メモ化プロセッサの高速化

目次

1 はじめに

これまで，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プロセッサの性能向上が難しくなってきた．このため，スーパースカラや SIMD 等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，クロック周波数の上昇に伴い，電力消費や発熱量は急激に増大する．この問題を解決しつつ，プロセッサあたりの処理能力を向上させるため，1 つの CPU に複数のコアを搭載したマルチコアプロセッサが広く普及している．このような背景から，複数コアを利用してプログラム全体のスループットを向上させる高速化手法として，スレッド並列性に着目してプログラムを複数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．

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，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が存在し，この計算再利用を用いた自動メモ化プロセッサに関する研究がこれまで行われている [1]．この自動メモ化プロセッサは，ハードウェアを用いることで既存のバイナリを変更することなく，動的に関数やループ等の命令区間を検出し，それら命令区間に対して計算再利用を自動的に適用する．これを実現するために，自動メモ化プロセッサは実行時に検出した命令区間の入出力を再利用表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，現在の入力と過去に再利用表に記憶しておいた入力とを比較し，それらの値が完全に一致した場合，過去の出力を利用することで実行を省略する．しかし，既存の自動メモ化プロセッサでは，この一致比較において入力値が一つでも異なる場合，再利用を行わない．これは，入力値が一つでも異なれば，その命令区間の実行結果は過去の実行結果と異なると考えられるためである．また，検索が失敗した場合には，再利用による実行の省略ができないばかりでなく，検索にかかったオーバヘッドが発生するため実行サイクル数が増加してしまう．しかしながら，再利用が失敗した場合にも，いくつかの入力値は一致している可能性がある．

そこで本研究では，このことに着目し，再利用の対象とする命令区間を，既存の再利用対象区間の先頭から入力値が一致している範囲までの区間とする手法を提案する．これにより，既存の自動メモ化プロセッサでは検索に失敗する場合でも，部分的に命令の実行を省略することが可能となり，高速化を実現できると考えられる．

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，入力が一致する範囲の命令区間の実行を省略することで，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概説する．

2.1　メモ化

計算再利用 (Computation Reuse) とは，主に関数などの命令区間に対してその入力と出力の組を実行時に記憶しておき，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，過去に記憶された出力を利用すること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．既知の入力値に対して再び同じ区間を実行する際に，正しい出力値を求めることができ，入力値さえ一致すれば実行結果を検証する必要がないことが特長に挙げられる．また，それら命令区間に計算再利用を適用することを**メモ化 (Memoization)**[2] と呼ぶ．このメモ化を用いた，ソフトウェア的な高速化手法は数多く提案されている [3, 4]．例えば森本ら [5] は，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を提案している．しかしながら，この手法はプログラマ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．また，プログラムを記述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．これらのソフトウェアによるメモ化はオーバヘッドが大きく，限られたプログラムでしか性能向上が得られない傾向がある．

2.2　自動メモ化プロセッサ

前節で述べたメモ化を既存のバイナリに対して，ハードウェアを用いる事で，動的に適用するプロセッサとして，**自動メモ化プロセッサ**が提案されている．本節ではこの自動メモ化プロセッサの構成と動作例を説明する．

2.2.1　再利用機構の構成

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一般的なプロセッサと同様に，コアの内部に ALU，レジスタ (Reg)，1 次データキャッシュ(D$1) 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，過去の入出力を記憶しておく表 (MemoTbl)，再利用機構を管理するための機構 (Reuse System)，MemoTbl への書き込みバッファ(MemoBuf)

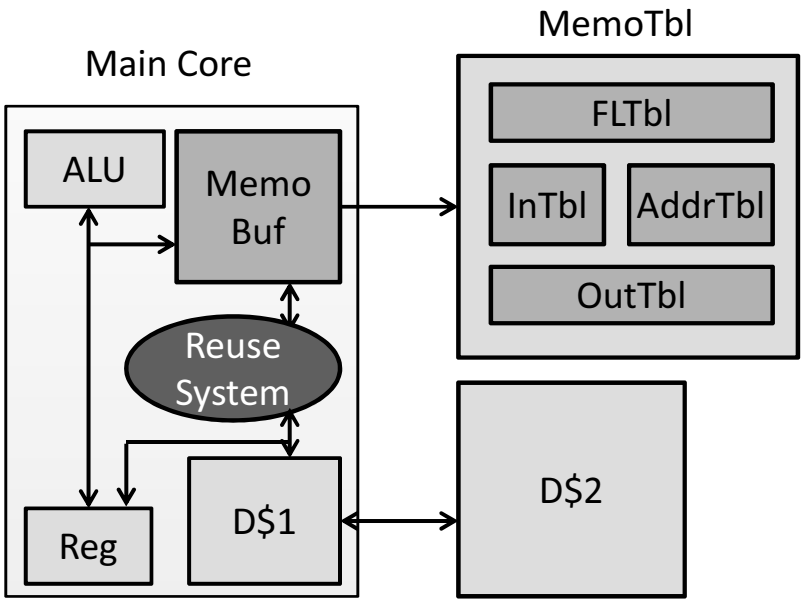

図 1: 自動メモ化プロセッサのハードウェア構成

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ため，入出力を検出する度に MemoTbl へアクセスする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ている．検出された入出力は MemoBuf に格納され，当該命令区間の実行終了時に MemoTbl へ一括して格納される．これにより，MemoTbl へのアクセス回数を減らしオーバヘッドを削減している．なお，関数に計算再利用を適用する場合，入力には関数の引数及び読み出された大域変数が含まれ，出力には関数の返り値及び書き込まれた大域変数が含まれる．また，ループに計算再利用を適用する場合，入力には読み出された大域変数及び局所変数が含まれ，出力には書き込まれた大域変数及び局所変数が含まれる．

次に，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 エントリが 1 入出力セットに対応する．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や関数の戻り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，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

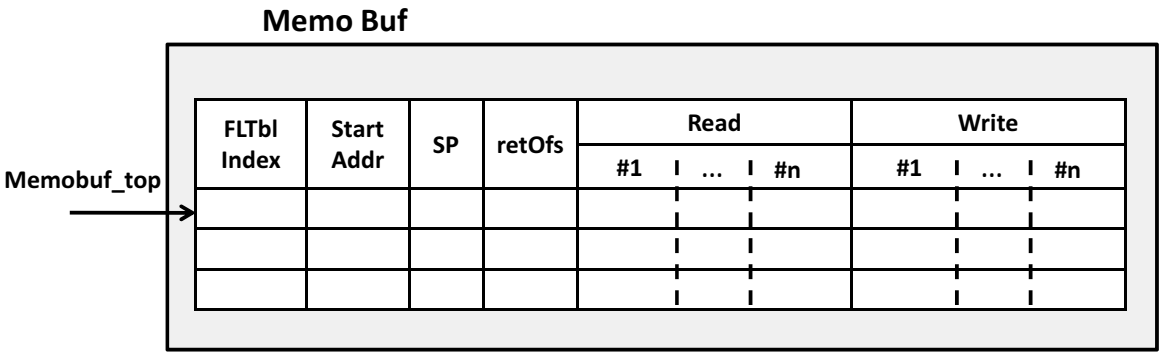

図 2: MemoBuf の構成

対象とするため，各 MemoBuf のエントリを番号の小さい方から順に使用し，ポインタ memobuf_top を用いて現在使用しているエントリを把握している．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．そして，return 命令により関数の呼び出し元へ戻った場合や，ループ区間の実行が終了した場合は，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 の値をデクリメントする．

次に MemoTbl の詳細な構成を**図 3** に示す．MemoTbl は，命令区間を記憶する **FLTbl**，入力値を記憶する **InTbl**，入力アドレスを記憶する **AddrTbl**，及び出力値を記憶する **OutTbl** の四つの表から構成されている．

FLTbl は，再利用対象命令区間を記憶する表であり，RAM で構成されている．各行は一つの命令区間に対応しており，メモ化のためのフィールドおよび後述するオーバヘッドフィルタのためのフィールドを持っている．メモ化のためのフィールドには，関数およびループの別（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．オーバヘッドフィルタのためのフィールドには，当該命令区間のサイクル数（Step），過去の再利用に要した入力検索および出力書き戻しオーバヘッド（Ovh^R, Ovh^W），過去の再利用ヒット履歴（Hit Hist）が保持される．

InTbl は，命令区間の入力値を記憶する表であり，CAM で構成されている．各行は FLTbl の行番号 (FLTbl index) 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，その次にアクセスされる入力値の

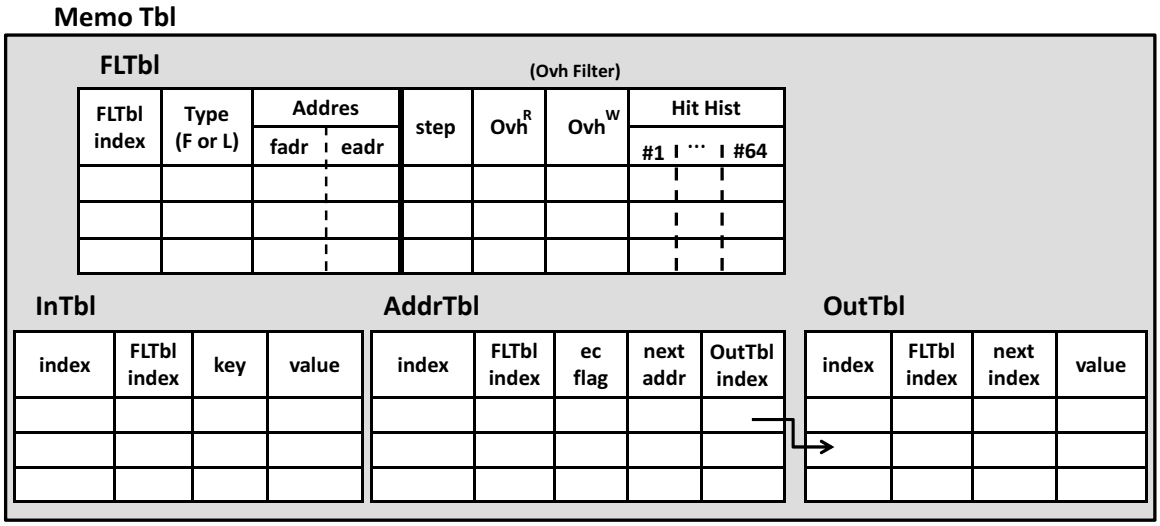

図 3: MemoTbl の構成

アドレスが変化する場合がある．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ため，全入力パターンはツリー構造で表現できる．そこで，このツリー構造を管理するため，InTbl の各エントリは，入力値 (value) に加えて，親エントリのインデクス (key) を記憶する．

AddrTbl は，入力アドレスのセットを記憶する表であり，RAM で構成されている．AddrTbl は InTbl 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．各行は次にアクセスするべきアドレス (next addr) を保持しており，入力セットの終端エントリか否かを保持するフラグ (ec flag) 及び FLTbl の行番号 (FLTbl index) を持つ．入力が完全に一致した場合には，その入力に対する出力セットを参照できるよう，入力セットの終端エントリは，対応する OutTbl エントリのインデクスを OutTbl index フィールドに保持している．

OutTbl は，命令区間の出力を記憶する表であり，RAM で構成されている．各行は命令区間の出力値 (value) 及び FLTbl の行番号 (FLTbl index) を保持している．また，出力セットの各エントリをリスト構造で管理するため，次に参照すべきエントリのインデクス (next index) を記憶する．

2.2.2 再利用機構の動作例

計算再利用は，MemoTbl へのエントリ登録操作と，MemoTbl を検索する操作によって実現される．また，この検索操作を再利用テスト呼ぶ．

```
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); /* x = 1, a = 4, b = 2, c = 8 */
11 :   b = 6;
12 :   func(1); /* x = 1, a = 4, b = 6, c = 8 */
13 :   a = 5;
14 :   func(1); /* x = 1, a = 5, b = 6, c = 8 */
15 :   a = 4,b = 2;
16 :   func(1); /* x = 1, a = 4, b = 2, c = 8 */
          ...
```

図 4: 計算再利用可能な命令区間を持つプログラム

エントリの登録

計算再利用の対象となる命令区間の実行が終了する際，命令区間の開始時点から MemoBuf に蓄えられてきた情報が，MemoTbl へ登録される．この登録動作を**図 4** に示すサンプルプログラムを用いて説明する．サンプルプログラム中には main 関数と func 関数が存在しているが，ここでは再利用対象命令区間として func 関数のみに着目する．

さて，関数の入力には，引数，大域変数，及びその関数を呼び出す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 が入力になり得る．このため，このサンプルプログラムが実行されると，入力エントリは**図 5** に示すような木構造を作る．ここで，図 5 の input1，input2，input3 は，関数 func が呼び出されてから，何番目に読み出された入力値であるかを示している．また，この木構造は多分木となっ

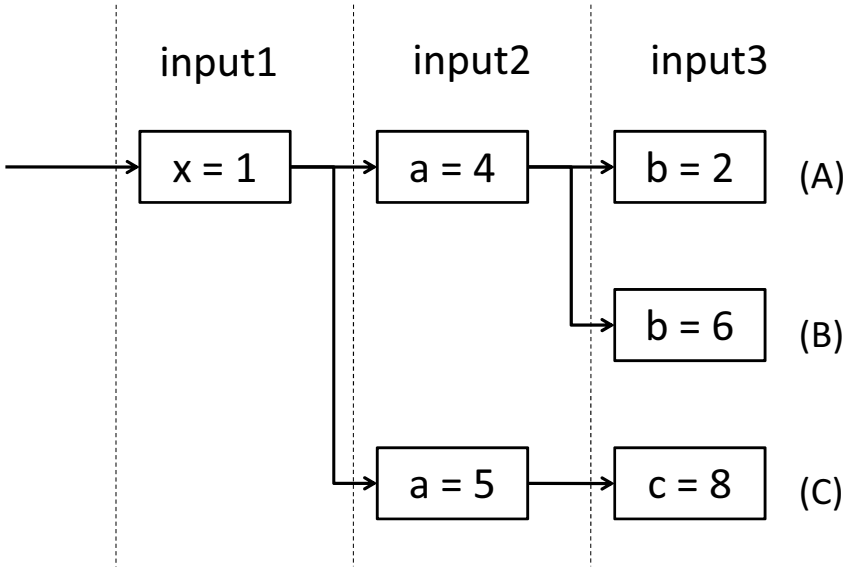

図 5: 入力エントリが成す木構造

ており，InTbl エントリが節に，AddrTbl エントリが枝に対応している．まず，図 4 の 10 行目において関数 func が 1 を引数として呼び出される場合の登録動作を説明する．この時の関数 func の入力値の候補は，引数と大域変数 a,b,c であるが，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 5(A) に示すように登録される．この時，変数 c は関数 func 内で参照されていないため入力とならず，そのエントリは登録されない．次に変数 b の値が変化し，12 行目において関数 func が呼び出された場合，図 5(B) に示すようにエントリが登録される．この呼び出しでは，先程と同様に 2,3,4 行目が実行される．ここで，引数 x 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 (A) と共有する．しかしながら，変数 b の値が (A) の時とは異なっているため，新しいエントリ b=6 が登録される．このように，同じ命令区間で入力値に共通する部分が存在する場合，途中までの入力エントリを共有することで，有限である InTbl テーブルを効率的に利用できる．そして 14 行目において関数 func が呼び出されたとき，入力候補の一つである変数 a が 13 行目の代入文で変化しているため，関数 func 内の条件分岐が不成立となる．このため，関数 func 内では 2,3,6 行目を実行することとなり，入力エントリは図 5(C) に示すように登録される．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．

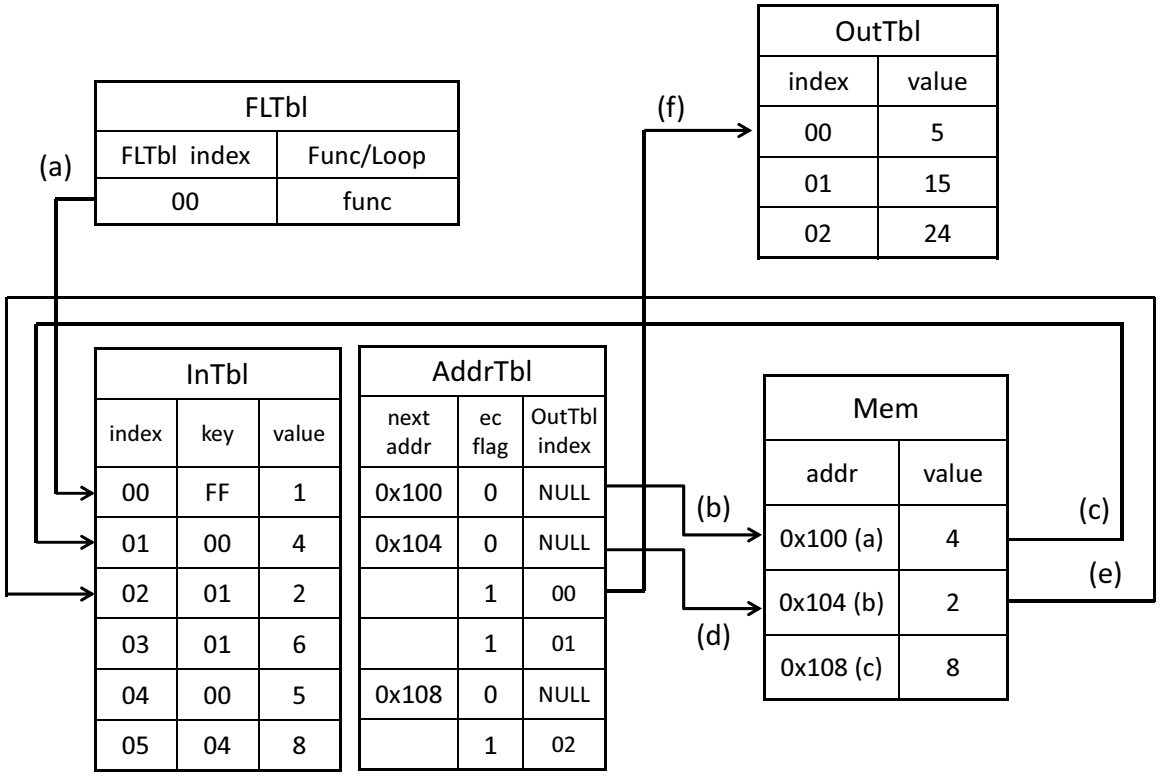

図 6: MemoTbl の検索手順

再利用テスト

計算再利用を適用するためには，現在の入力と過去に MemoTbl に記憶しておいた入力とを比較する必要がある．MemoTbl の検索手順を**図 6** に示す．ここで，図 6 中の MemoTbl は図 4 のサンプルプログラムを 14 行目まで実行した状態となっている．また，図 6 では図を簡略化するため，FLTbl は命令区間の名前とそれに対応する FLTbl index のみを記述し，InTbl,AddrTbl,OutTbl の FLTbl index は全て同じ値であるため省略している．

サンプルプログラムにおける 16 行目の func が呼び出されると，まず，FLTbl が関数名 func で検索される．これにより得られた FLTbl index を持ち，value 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る (a)．次に，該当するエントリがライン 00 で発見され，対応する AddrTbl の next addr が 0x100 番地を指しているため，主記憶のその番地を参照する (b)．そして，得られた値を value として持ち，key がライン 00 であるエントリを InTbl から検索する (c)．以降，同様に検索を続ける (d)(e)．

ここで，ライン 02 の ec flag が 1，つまり入力セットの終端エントリに達した事を検出したため，再利用テストが成功する．この時，検索の終点となった AddrTbl エントリの OutTbl index に格納されているポインタの指す OutTbl エントリを参照し，入力に対応する出力を読み出す (f)．そして，読み出した出力値をレジスタやメモリに書き戻すことで命令区間の実行を省略する事ができる．

2.3 オーバヘッド評価機構

自動メモ化プロセッサにおいて，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．まず，再利用適用時の再利用テストの際に，メインコアのレジスタや主記憶の値と MemoTbl に登録されている値とを比較するための検索オーバヘッドがある．この検索オーバヘッドは再利用テストの成功・失敗に関わらず発生する．また，再利用テストが成功した際に，出力を MemoTbl からレジスタやキャッシュへ書き戻すための書き戻しオーバヘッドがある．入力値の検索オーバヘッドと出力の書き戻しオーバヘッドをあわせて**再利用オーバヘッド (Reuse Overhead)** 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処理を終えることができる場合も存在する．その場合，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る．そして，計算再利用による効果が得られると判断した命令区間に対してのみ再利用テストや入出力セットの登録を行う．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 (Overhead Filter)** と呼ぶ．図 3 で示したように，オーバヘッドフィルタ機構は命令区間を記憶する表である FLTbl の拡張として備えられている．FLTbl では，各命令区間に対して一定期間における再利用の状況をシフトレジスタ (図 3 中の Hit Hist.) を用いて記録し，これを用いてそれぞれの命令区間の再利用適応度を算出している．

ある命令区間について，最近の一定回数 T 回の再利用試行における再利用成功回数 M は上記シフトレジスタから得られる．この値と，当該命令区間の過去の省略サイク

ル数 C から，実際に削減できたサイクル数を

$$M \cdot (C - Ovh^R - Ovh^W) \tag{1}$$

として計算する．Ovh^R，Ovh^W はそれぞれ，過去の履歴より概算した，当該命令区間の MemoTbl 検索オーバヘッド，および MemoTbl からキャッシュ等への書き戻しオーバヘッドである．なお，C，Ovh^R，Ovh^W は，図 3 に示した FLTbl の Step フィールド，および Ovh read/write フィールドからそれぞれ取得される．

また，再利用が適用できなかった場合でも，MemoTbl の検索オーバヘッドは存在する．このオーバヘッドは，

$$(T - M) \cdot Ovh^R \tag{2}$$

として計算できる．

ここで，発生したオーバヘッド (2) よりも，削減できたステップ数 (1) が大きいような命令区間は，再利用の効果が得られると考えられる．よって，この命令区間の再利用適応度を

$$M \cdot (C - Ovh^W) - T \cdot Ovh^R \tag{3}$$

として計算する．ここで，(3) の値が正である場合，再利用の効果があると判断できる．そこで，MemoTbl に小さなハードウェアを付加することによってこれを計算し，再利用の効果が得られると判断された命令区間に対してのみ MemoTbl への登録及び再利用している．

3 提案

自動メモ化プロセッサでは，再利用可能な命令区間に対する入力値が，過去の入力値と完全に一致した場合に限り実行を省略できる．このため，入力値が一つでも異なる場合には実行を省略する事ができない．そこで，入力の一部が異なるような場合にも，入力が一致した区間に限定して部分的に実行を省略することにより，高速化を図る手法を提案する．

3.1 部分再利用

自動メモ化プロセッサでは，再利用テストにおいて入力値が一つでも異なる場合，出力値が過去と異なる可能性があるため，再利用を行わない．これは，入力値が一つでも異なれば，その命令区間の実行結果は過去の実行結果と異なると考えられるためで

```
0  : int a, b;
1  : int func2(x){
2  :   int tmp = x + 1;
3  :   tmp = tmp + a;
4  :   tmp = tmp + b;
5  :   return(tmp);
6  : }
7  :
8  :int main(void){
9  :   a = 3, b = 4;
10 :   func2(2); /* x = 2, a = 3, b = 4 */
11 :   b = 5;
12 :   func2(2); /* x = 2, a = 3, b = 5 */
13 :   func2(2); /* x = 2, a = 3, b = 5 */
14 :   return(0);
15 : }
```

図 7: 部分再利用可能な命令区間を持つプログラム

ある．検索が失敗した場合には，再利用による実行の省略ができないばかりでなく，検索オーバヘッドが発生するため実行サイクル数が増加してしまう．しかしながら，再利用が失敗した場合にも，いくつかの入力値は一致している可能性がある．このことに着目し，再利用の対象とする命令区間を，既存の再利用対象区間の先頭から，入力値が一致している範囲までの区間とする手法を提案する．これにより，通常の計算再利用に加え，既存の自動メモ化プロセッサでは検索に失敗する場合でも，部分的に命令の実行を省略することが可能となり，高速化を実現できると考えられる．以下，提案手法を用いて命令区間の一部だけを省略する事を**部分再利用 (Partial Computation Reuse)** と呼び，通常の計算再利用を**完全再利用 (Full Computation Reuse)** と呼ぶ.

この部分再利用手法の概要について，**図 7** に示すサンプルプログラムを用いて説明する．このプログラムが実行されると，まず，10 行目で func2 が呼び出される．この時，大域変数 b の値は 4 である．一方，12 行目で func2 が呼び出される時，大域変数 b の値は 5 となっている．このため，12 行目における関数呼び出しは，完全再利用に

失敗する．実際に，2 回目の関数呼び出しの返り値は 11 であり，1 回目の返り値である 10 とは異なっている．しかし，この 2 回の関数呼び出しにおいて，引数 x の値と大域変数 a の値は同じ値である．このため，func2 の 2，3 行目までの実行結果は一致する．したがって，2 回目の関数呼び出し時に，再利用対象命令区間の範囲を 3 行目までに限定すれば，その実行結果を部分的に再利用することが可能である．

部分再利用を適用するには，命令区間の途中から実行を再開するため，実行を省略する命令区間で書き換えられる値と，実行を再開するプログラムカウンタの情報が必要である．一方，完全再利用ではその特性を活かすことによって，再利用に必要な情報量を最適化している．例えば，関数に完全再利用を適用する場合では，呼び出した関数の実行は全て省略され，呼び出し元から実行を再開する．このため，実行を再開するプログラムカウンタとして，戻り番地のみ記憶すれば良い．また，呼び出した関数の局所変数は関数終了時に破棄されるため，この局所変数の出力を MemoTbl へ登録する必要がない．図 7 の 13 行目を例とした場合，func2 の局所変数である tmp の値は書き戻す必要がない．また，ループに完全再利用を適用する場合，命令区間の終端の後方分岐命令の結果によって，命令区間の開始時点もしくは次の命令から実行を再開する．したがって，MemoTbl に命令区間の終端アドレスと後方分岐命令の結果を記憶し，その分岐命令の結果に応じて，実行を再開するプログラムカウンタを選択している．また，ループの出力に関しては，再利用適用後も同一関数を実行中であるため，書き換えられた変数の値を全て登録している．これに対し，関数に部分再利用を適用する場合には，呼び出した関数の途中から実行を再開するため，呼び出した関数の局所変数を含めた出力の全てと実行を再開するプログラムカウンタを MemoTbl へ記憶する必要がある．また，ループに部分再利用を適用する場合，同様にループ内の途中から実行を再開するため，実行を再開するプログラムカウンタと書き換えられた変数の値を MemoTbl へ記憶する必要がある．

3.2 動作モデル

本節では，部分再利用が可能な場合の具体的な検出方法について説明する．過去に実行した事のある再利用可能な命令区間が再び実行される場合に，部分再利用を適用するためには，それを適用する範囲の入力値が現在の入力と一致することを確認しなければならない．なお，部分再利用と完全再利用の両方が適用可能であった場合，完全再利用の方が削減サイクル数が大きくなる可能性が高いため，完全再利用を適用するべきである．このような再利用範囲の選択は，完全再利用のための一致比較と部分

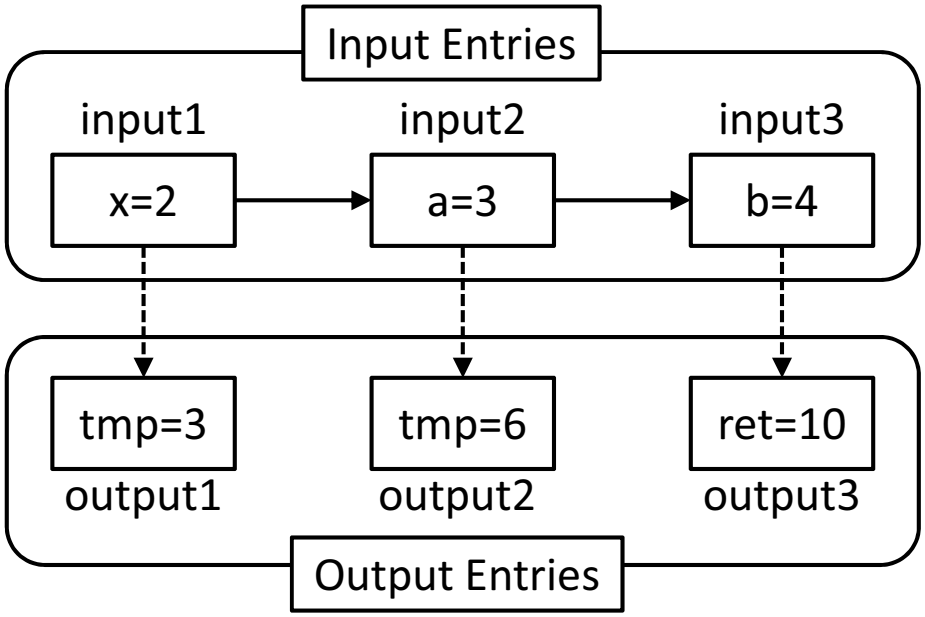

図 8: 部分再利用のエントリ構造

再利用のための一致比較の 2 回に分けて再利用テストを行うことで容易に実現することができる．しかしながら，この手法を用いると，既存手法よりも検索回数が増加してしまう．そのため，検索オーバヘッドが大きくなり，かえって性能が低下してしまう可能性がある．そこで，既存の自動メモ化プロセッサでは，読み出した順に入力を一致比較している点に着目する．この事から，ある入力値が一致した場合，その命令区間の先頭から次の入力値が読み出されるまでの範囲の命令区間は，過去の実行結果と一致する命令区間であることを保証できる．そこで，それぞれの入力エントリに対して，次の入力が読み出されるまでの実行結果を部分再利用の出力として登録する．そして，再利用テストで入力値が全て一致した場合には完全再利用の出力を適用し，再利用テストが失敗した場合には，それまでに一致した入力に対応する部分再利用の出力を適用することで，検索オーバヘッドを増大させることなく提案手法を実現できる．

ここで，図 7 のサンプルプログラム中の 10 行目の func2 が呼ばれた場合，提案手法ではどのようなエントリが登録されるかを**図 8** に示す．図中には三つの出力エントリ ret = 10，tmp = 6，tmp = 3 が存在している．また，図中の実線の矢印は入力値と次に読み出される入力値の対応関係を示しており，点線の矢印は入力値と出力値の対応関係を示す．ret = 10 は完全再利用の適用時に書き戻される出力エントリであり，x = 2，a = 3，b = 4 の全ての値が，入力値と一致した場合に書き戻される．一方で tmp = 3，

tmp = 6 は部分再利用の適用時に書き戻されるエントリである．ここで，tmp = 6 は，x = 2，a = 3 が一致した場合に書き戻され，tmp = 3 は x = 2 のみが一致した場合に書き戻される．

3.3 提案手法の効果と問題点

提案手法を用いる事により，以下にあげる効果と問題が生じると考えられる．

3.3.1 提案手法による効果

部分再利用を適用することによって，計算再利用の適用範囲が増加するため，命令実行サイクル数が削減されると考えられる．部分再利用を適用する事によって削減する事ができるサイクル数は，部分再利用によって削減された実行サイクル数を C^P，その再利用の検索時に発生するオーバヘッドを Ovh^R，書き戻し時に発生するオーバヘッドを Ovh^W とを用いて

$$C^P - Ovh^R - Ovh^W \tag{4}$$

として計算される．(4) の値が正である場合には，部分再利用を適用する事で，通常に実行するよりも性能が向上する．また，部分再利用が成功する場合には既存手法では検索が失敗し完全再利用が適用できない．そのため，部分再利用時に発生する検索オーバヘッドは，既存手法でも発生していたものと同じである．このことから，提案手法は，既存手法と比較して

$$C^P - Ovh^W \tag{5}$$

サイクル削減される事になる．よって，(5) の値が正であれば，部分再利用を適用した方が，既存手法よりも性能を向上させる事ができる．

3.3.2 提案手法の問題点

部分再利用には，いくつか問題点も存在する．まず一つ目の問題点として，部分再利用を適用する範囲が元の命令区間の先頭から入力値が一致している範囲までの区間に限定されることが挙げられる．このため，入力値が一致する命令区間が元の命令区間の後半部分にあるような場合でも，前半の入力値が一致しなければ，部分再利用を適用することはできない．これは 3.2 節で述べたように，部分再利用と完全再利用の成否を同時に検索するため，入力を読み出した順に比較していることが原因である．しかし，この制約によって部分再利用が適用できなくなるような場合とは，命令区間の後半部分に前半部分と依存関係がなく，なおかつその範囲の命令区間の入力値が一致するという極めて限定的な状況である．そのため，このような部分再利用に対応して

いないことによる性能の低下は軽微であると考える．

二つ目の問題点として，入れ子関係にある関数において，外側の関数に部分再利用を適用した場合，内側の関数から実行を再開することが難しいことが挙げられる．入れ子構造の内側の関数における主記憶参照は，外側の関数の入力にも含まれることになる．このため，部分再利用を適用する事によって，入れ子構造の内側の関数の途中までの実行を省略する場合，実行を再開するプログラムカウンタと出力を書き戻す事に加え，コールスタックの状態も書き換えなければならない．しかし，コールスタックを書き換えようとした場合，同じ関数でも呼び出し時のコールスタックの状態に応じて相対的にアドレスが変化するため，相対アドレスの計算が必要になる．その上，この相対アドレスの計算は入れ子構造の深さに依存して複雑化するため，実現は非常に困難である．そこで提案手法では，部分再利用を適用する関数と実行が再開される関数が同一である場合のみ，部分再利用の出力エントリを登録する．これにより，入れ子構造の内側の関数から実行が開始されることがなくなるため，コールスタックを書き換える必要はない．

三つ目の問題点として，ハードウェアの複雑化が挙げられる．具体的な追加ハードウェアについて詳しくは 4 章で説明するが，提案手法を実現するには，例えば，3.1 節で述べたように，実行を再開するプログラムカウンタや局所変数の出力を記憶するため，MemoTbl を拡張する必要がある．このように，提案手法を実現するために必要な領域を拡張するため，ハードウェア量が増大する．また，3.2 節で述べたように，提案手法では各入力エントリに対して部分再利用の出力を登録し，検索失敗時に一致した入力に対応する出力を書き戻す．したがって，MemoTbl の登録や検索時の動作が複雑化するため，その動作を実現するハードウェアも複雑化すると考えられる．

最後に四つ目の問題点として，登録される出力エントリ数の増大が挙げられる．既存手法では一つの入力パターンに対して，完全再利用のための一つの出力パターンが対応していた．しかし提案手法では 3.2 節で述べたように，一つの入力パターンに対して完全再利用以外にも，部分再利用のための複数の出力パターンが対応する上，局所変数の出力も登録する．このため，増加した出力パターンの数に応じて新たに出力エントリを登録する必要がある．そして，登録される出力エントリが増大すれば OutTbl のテーブルを圧迫することになり，性能が低下する恐れがある．本研究では，この出力エントリ数が増大したことによる性能低下を，MemoTbl の OutTbl 容量を増加させることによって軽減する．

4 実装

再利用テストで検索に失敗した場合に，入力が一致した命令区間を部分的に再利用するためには，自動メモ化プロセッサの拡張が必要となる．そこで本章では，提案する機構の実装について述べる．

4.1 実装の概略

提案手法を実現するには，まず部分再利用に成功した際に書き戻す出力を MemoTbl へ登録しておく必要がある．この出力には，書き込みがあった大域変数や返り値に加え，局所変数や実行を再開するプログラムカウンタといった情報も必要になる．そこで，これらの追加情報を記憶する領域を MemoTbl に追加する．また，部分再利用に用いる出力エントリを再利用区間の実行途中に登録するように，登録時の動作を変更する．そして，部分再利用と完全再利用の成否を同時に検索するように，検索時の動作を変更する．また，最終的にはこの提案手法を関数とループの両方に対応させる予定であるが，まずは関数に適用することによりその効果を確認する．

4.2 ハードウェア拡張

追加情報を記憶するために拡張した MemoBuf と MemoTbl を**図 9** に示す．MemoBuf の Read と Write の各エントリ及び MemoTbl の AddrTbl と OutTbl に対して新たな領域を拡張する．

部分再利用が行われる場合，3.1 節で述べたように，関数の途中までの実行結果を書き戻し，その次の命令から実行を再開する．このため，実行を再開するプログラムカウンタ (PC) を記憶しなければならない．この PC の値は，部分再利用を適用する範囲に応じて変化するため，一致する入力値が増える毎に，異なる PC の値を記憶する必要がある．したがって，MemoBuf の Read と MemoTbl の AddrTbl のそれぞれの入力エントリに PC の値を記憶する領域を追加する．また，3.3.2 項で述べたように，部分再利用を適用する関数と実行が再開される関数が同一である場合のみ，部分再利用の出力エントリを登録する．そのため，各入力エントリ全てに部分再利用の情報が記憶されているとは限らない．そこで，当該入力エントリに部分再利用の情報が記憶されているか否かを示す 1bit のフラグ (part) を，MemoBuf の Read 及び MemoTbl の AddrTbl の各入力エントリにそれぞれ追加する．また，次の節で詳しく述べるが，本研究では部分再利用の出力エントリを，新しい入力が MemoBuf に登録される毎に，MemoTbl

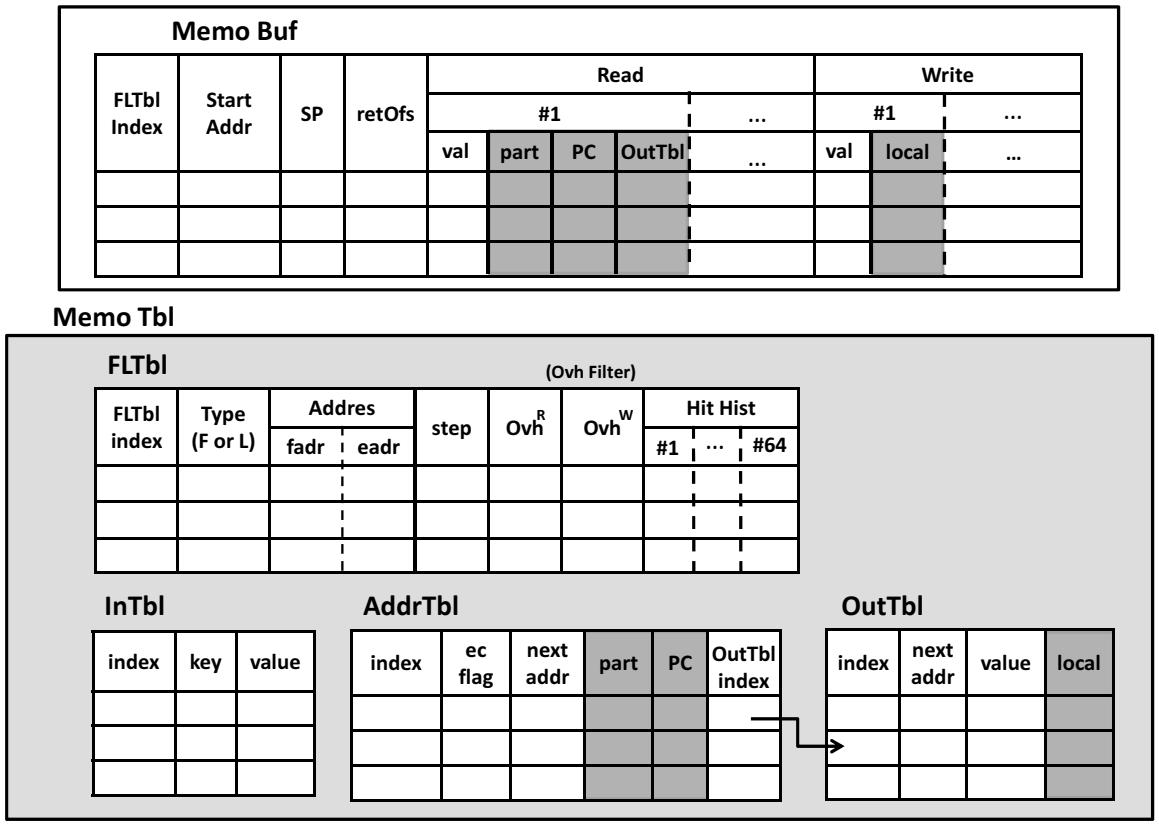

図 9: 拡張後の MemoBuf と MemoTbl

の OutTbl へ登録する．しかし，入力エントリは命令区間の終了時に一括で登録するため，部分再利用の出力エントリを指すポインタを MemoBuf 内で記憶する必要がある．したがって，MemoBuf の Read の各入力エントリに対して，MemoTbl の OutTbl へのポインタ (OutTbl) を格納する領域を追加する．一方 MemoTbl の AddrTbl には，OutTbl へのポインタを格納する領域が既に存在する．この領域は既存モデルでは完全再利用の出力を指すために用いられており，入力セットの終端のエントリ以外では使用されていないため，部分再利用の出力を指すポインタをこの領域に格納すれば良い．なお，3.1 節で述べたように，部分再利用の出力には局所変数も含まれるため，大域変数と同様に MemoTbl へ登録する．しかし，関数の完全再利用には局所変数の出力値は不要であるため，MemoTbl の出力エントリが局所変数なのか大域変数なのかを見分ける必要がある．したがって，出力エントリが局所変数であることを示す 1bit のフラグ (local) を，MemoBuf の Write と MemoTbl の OutTbl の出力エントリにそれぞれ追加する．

4.3 実行モデル

本節では，提案手法における MemoTbl へのエントリ登録操作と MemoTbl の検索操作について説明する．

4.3.1 登録時の動作

既存の自動メモ化プロセッサでは，2.2 節で述べたように命令区間の実行終了時に MemoBuf に蓄えられた入出力を一括で MemoTbl に登録する．そのため，提案手法でも同様に完全再利用の出力エントリを命令区間の実行終了時に MemoTbl に登録する．ここで，部分再利用の出力エントリを完全再利用の出力エントリと同じタイミングで登録するか，異なるタイミングで登録するかの 2 通りの実装方法がある．まず，部分再利用の出力エントリを完全再利用と同じタイミングで登録するように実装する方法である．この場合，計算再利用を適用する範囲に応じて，同一の変数の値が書き換えられる可能性を考慮する必要がある．このため，MemoBuf に同じ変数に対する複数の出力値を記憶する必要がある．また，それら各出力値がどの入力に対応するのか記憶する必要がある．このため，この方法ではハードウェアコストが大きい上，動作が複雑になってしまう．

そこで，入力が新たに出現するタイミングで部分再利用の出力エントリを登録することにより，部分再利用のための登録を実現する．3.2 節で述べたように，部分再利用を適用する範囲は命令区間の先頭から次の入力が読み出されるまでの範囲である．このため，ある入力に対応する出力は次の新しい入力値が MemoBuf に登録される時に確定し，その時の MemoBuf の Write が部分再利用の出力となる．したがって，次の入力値が登録されるタイミングで MemoBuf の Write を MemoTbl の OutTbl へ登録し，出力セットの先頭エントリへのポインタを MemoBuf の Read の w1 に記憶する．また，同時にその時の PC の値が部分再利用適用後の PC の値であると確定するため，その値を MemoBuf の Read の同一エントリに記憶する．そして，部分再利用のための登録を終えた後で，通常通り新しい入力値を MemoBuf に登録する．このように新しい入力値が MemoBuf に登録されるたびに部分再利用の出力エントリを，また命令区間の実行終了時に入力エントリと完全再利用の出力エントリを MemoTbl に登録する．これにより，既存手法と比べて MemoTbl へのアクセス回数が増加するが，メインコアによる命令実行と並行して登録できるため，既存モデルと同様にコストを考慮する必要がない．

ここで，図 7 のサンプルプログラム中の 10 行目の func2 が呼ばれた場合，提案手法ではどのように MemoTbl へ登録されるのか説明する．**図 10** はこの登録動作におい

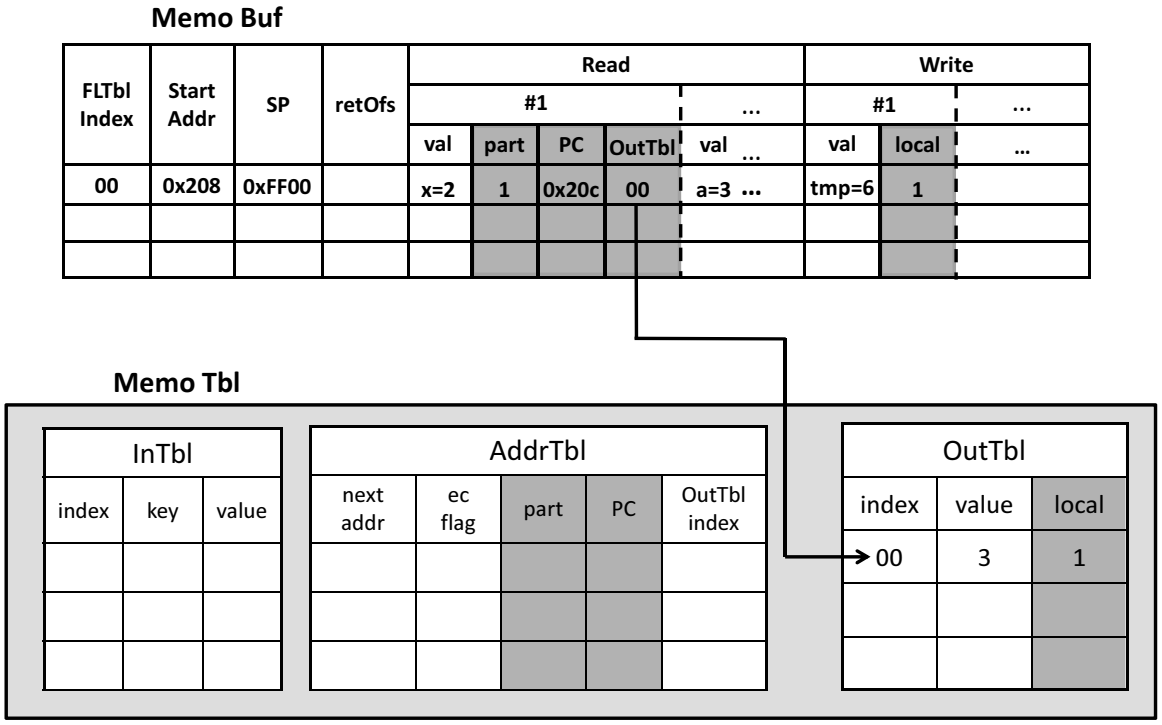

図 10: 実行途中の MemoBuf と MemoTbl の様子

て，初めて部分再利用の出力が登録された時の MemoBuf と MemoTbl の状態を示している．なお，説明の簡略化のため，FLTbl を省略している．まず，10 行目で func2 が呼び出され，その FLTbl の index である 00 を MemoBuf の FLTbl index に記憶する．また，この func2 は 0x208 番地から実行が始まり，その時のスタックポインタ (SP) の値は 0xFF00 であると仮定する．この 0x208 を MemoBuf の StartAddr に，0xFF00 を MemoBuf の SP にそれぞれ記憶する．func2 の実行が始まり，2 行目の命令で x の値が読み出されるため，MemoBuf の Read に入力値 $x = 2$ を記憶する．また，tmp は func2 の局所変数であるため，この命令の実行結果である $tmp = 3$ は不必要な情報として既存手法では登録されなかった．しかし，提案手法では部分再利用によって命令区間の実行途中から再開する場合もあるため，この値を MemoBuf の Write に記憶する必要がある．また，局所変数と大域変数を見分けるため，$tmp = 3$ は local フラグを立てて記憶する．次の 3 行目の命令の実行に移り，まず tmp の値が読み出されるが，この tmp は既に MemoBuf の Write に記憶されているため，新たな入力ではない．一方 a については，MemoBuf の Read にも Write にも記憶されていないため，新しい入力となる．この時，入力値 $x = 2$ によって部分再利用できる範囲が確定するため，MemoBuf

の Write に登録されている情報，つまり tmp = 3 を，部分再利用の出力エントリとして MemoTbl の OutTbl に登録する．そして，その先頭エントリの index である 00 を MemoBuf の x = 2 のエントリにおける w1 領域に記憶する．また，現在の PC の値を，部分再利用適用後に実行を再開するプログラムカウンタとして，x = 2 における PC 領域へ記憶する．最後に，部分再利用の情報が登録されていることを示すフラグを x = 2 に記憶する．その後，通常通り入力値 a = 3 を MemoBuf の Read に登録し，出力値 tmp = 6 を MemoBuf の Write の tmp = 3 のエントリに上書きする．この時，memoTbl は図 10 の状態になる．以下同様に次の 4 行目の命令では，b は新たな入力であるため，tmp = 6 を MemoTbl の OutTbl へ登録し，部分再利用のための情報を MemoBuf の a = 3 のエントリに記憶する．そして，入力値 b = 4 を MemoBuf の Read に登録し，出力値 tmp = 10 を MemoBuf の Write に上書きする．次の 5 行目で，return 命令を検出することで再利用区間の終了であることを判別できる．よって，返り値 ret = 10 を MemoBuf の Write に登録し，その後 MemoBuf の入出力を全て MemoTbl に登録する．この時，MemoBuf の Write の local フラグが立ってるエントリは完全再利用には不要であるため，MemoTbl には登録しない．

4.3.2 検索時の動作

次に，提案手法での検索時の動作について説明する．MemoTbl の検索手順を**図 11** に示す．ここで，図 11 は図 7 のサンプルプログラムを 10 行目まで実行した状態となっている．

サンプルプログラムにおける 12 行目の func2 が呼び出されると，まず，既存手法と同様に FLTbl から得られた FLTbl index とルートエントリを示す FF を用いて InTbl が検索され，ライン 00 のエントリが発見される (a)．ここで，対応する AddrTbl エントリには，部分再利用が可能であることを示す part フラグが立っているため，この PC 領域と OutTbl index 領域の情報を使えば部分再利用を適用できるとわかる．しかし，入力が一致したため部分再利用を適用せず検索を続行する．この時，部分再利用の情報を再利用テストに失敗するまで保持するため，このライン番号の PC と OutTbl index を MemoBuf の PC 領域と OutTbl 領域に記憶する．再び既存手法と同様に検索が進み，InTbl の検索でライン 01 のエントリが発見される (b)(c)．すると，先程と同様に part フラグが立っているため，このラインの情報でも部分再利用を適用できるとわかる．ここで，ライン 01 の出力はライン 00 とライン 01 の両方の入力が一致する範囲の出力であるため，ライン 00 よりライン 01 の情報を用いて部分再利用を適用する方が，より多くの命令実行サイクル数を削減できると考えられる．このため，MemoBuf に記憶し

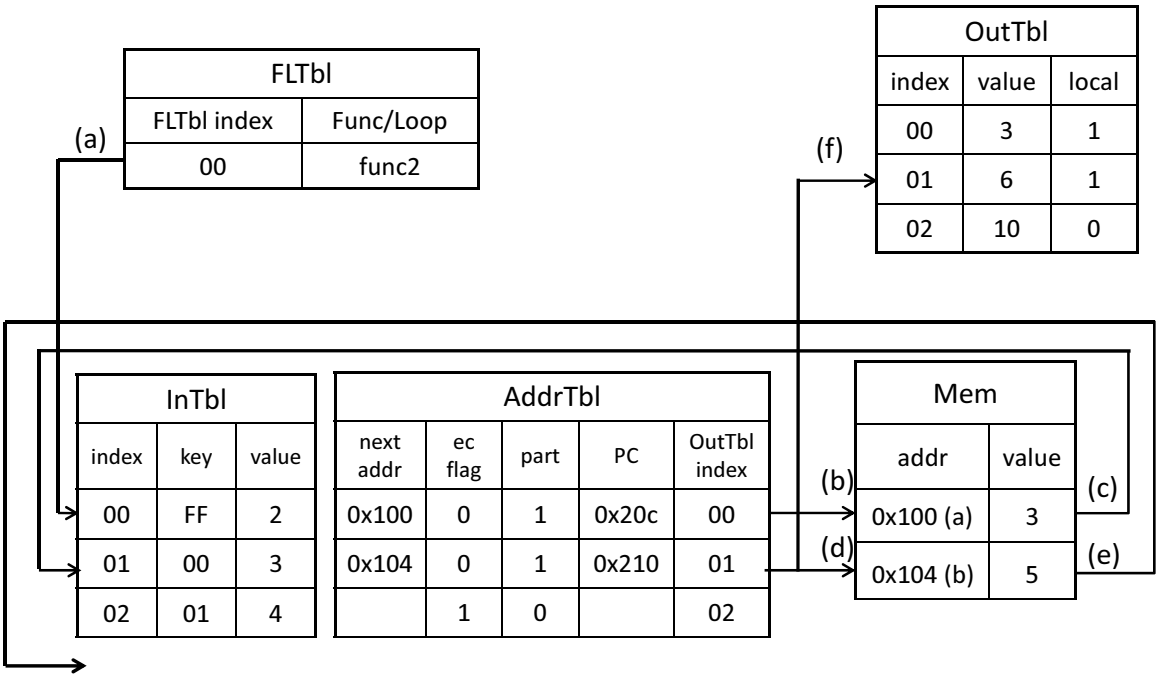

図 11: 拡張後の MemoTbl 検索手順

ている部分再利用の情報をライン 00 の情報からライン 01 の情報に更新する．その後，再び既存手法と同様に検索が進むが，次の入力となる b の値が一致しないため，再利用テストに失敗する (d)(e)．この時，これまで MemoBuf に記憶していたライン 01 の情報を用いて部分再利用を適用する．まず，AddrTbl エントリの OutTbl index に格納されているポインタの指す OutTbl エントリを参照し，当該命令区間の出力を読み出す (f)．次に，読み出した出力値をレジスタやメモリに書き戻し，それと同時に，現在の PC の値を AddrTbl エントリの PC に格納されている 0x20c に書き換える．そして，書き換えた PC の値から実行を再開することで，命令区間の実行を省略する．

4.4 部分再利用を考慮したオーバヘッド評価機構

自動メモ化プロセッサは 2.3 節で述べたように，それぞれの命令区間に対して計算再利用により効果が得られるかを判断するオーバヘッドフィルタ機構を備えている．このオーバヘッドフィルタは完全再利用の成功回数とその削減サイクル数，及び検索失敗回数とそのオーバヘッドを用いて命令区間の再利用適応度を算出している．しかし，部分再利用が可能になったことで，3.3.1 項で述べたような効果があり，各命令区間の

再利用適応度に変化が生じると考えられる．そこで，部分再利用の成功回数とその効果を計測し，オーバヘッドフィルタに反映させることによって，この再利用適応度を算出する精度の向上を図る．

まずは，入出力エントリの登録時の判定方法について説明する．これは，完全再利用時に利益が得られるかどうかが基準であり，完全再利用に成功しても利益が得られないようなエントリの登録を中止する．したがって，部分再利用が可能になったことによる完全再利用時の利益の増減は無いため，この基準を変更する必要はない．

次に，部分再利用の出力エントリ登録時の判定方法について説明する．再利用テストで部分再利用を適用する場合と，それを適用せずに検索失敗する場合を比較すると，どちらも同じ検索オーバヘッドがかかる．このため，3.3.1 項で述べたように，部分再利用は削減サイクルが書き戻しオーバヘッドを上回れば高速化する．したがって，部分再利用による省略サイクル数を C^P，書き戻しオーバヘッドを $POvh^W$ とすると，部分再利用により削減できたサイクル数を

$$C^P - POvh^W \tag{6}$$

として計算できる．この値が正値であれば部分再利用の出力エントリの登録を行い，負値であれば登録を中止する．

最後に，検索時の判定方法について説明する．検索時にはある命令区間について，最近の一定回数 T 回の再利用試行によるサイクル削減効果を基準としている．よってこのサイクル削減効果を，過去 T 回における完全再利用，部分再利用，検索失敗の回数に 1 回当たりの削減効果をかけることによって算出する．過去 T 回の再利用試行における，完全再利用成功回数 M と省略サイクル数 C とすると，完全再利用による削減サイクル数は (1) と同じになる．同様にこの過去 T 回の再利用試行における，部分再利用成功回数 P と省略サイクル数 C^P，書き戻しオーバヘッド $POvh^W$ から部分再利用による削減サイクル数を

$$P \cdot (C^P - Ovh^R - POvh^W) \tag{7}$$

として計算する．また，過去 T 回の再利用試行における，検索失敗によるペナルティを

$$(T - M - P) \cdot Ovh^R \tag{8}$$

として計算できる．

ここで，発生したオーバヘッド (8) よりも，削減できたステップ数 (1)，(7) が大き

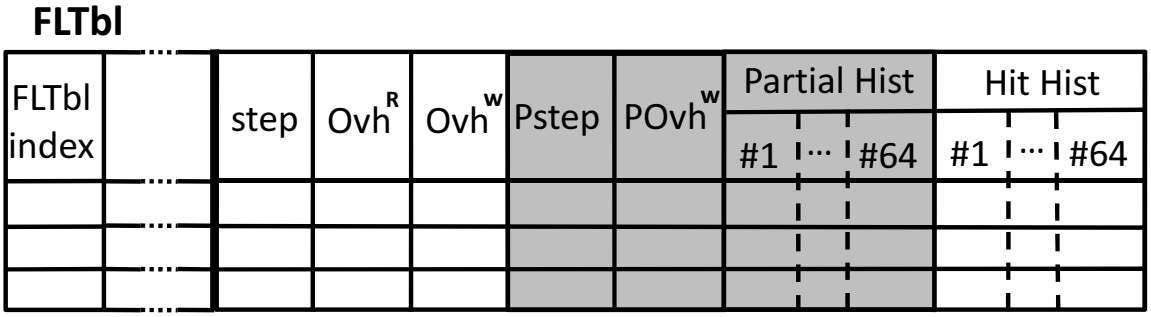

図 12: 拡張後の FLTbl

いような命令区間は，再利用の効果が得られると考えられる．よって，この命令区間の再利用適応度を

$$M \cdot (C - Ovh^W) + P \cdot (C^P - POvh^W) - T \cdot Ovh^R \tag{9}$$

として計算する．ここで，(9) の値が正である場合，その命令区間は再利用の効果があると判断され，再利用テストの対象となる．

さて，このオーバヘッドフィルタの改良によって，性能向上が得られる場合に限り部分再利用の出力エントリが登録されるようになる．また，改良前のオーバヘッドフィルタでは部分再利用による利益が判定に反映されず，再利用テストが中止されていた命令区間に対してでも，改良後のオーバヘッドフィルタではその命令区間は再利用の効果があると判定され，再利用テストが続行される場合もある．これによる再利用テストの回数の増加に伴い，再利用成功回数が増加し，性能が向上すると考えられる．

このように，より高い精度で命令区間の再利用適応度を計算するためには，部分再利用の成功回数や 1 回当たりの削減サイクルなどの追加情報を計測しておく必要がある．したがって，各命令区間の情報を記憶する表である FLTbl を拡張し，これらの追加情報を記憶する．**図 12** に拡張後の FLTbl を示す．まず，部分再利用による効果を計測するため，部分再利用を適用した時の削減サイクル数 ($Pstep$) と書き戻しオーバヘッド ($POvh^W$) を記憶する領域を拡張する．ここで，部分再利用時の検索オーバヘッドは検索失敗時と同じ値になるため，新たに記憶する必要はない．また，部分再利用の成功回数 ($Partial\ Hist$) を計測するため，完全再利用と同一のシフトレジスタを追加する．

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
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	1cycle/32bytes

5 評価

本提案手法を実現するため，既存の自動メモ化プロセッサに対して，関数に対する部分再利用を追加実装した．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評価し，その結果について考察した．

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 1** に示す．なお，キャッシュアクセスレイテンシや命令レイテンシは SPARC64-III[6] を，MemoTbl 内の InTbl に用いる CAM の構成は MOSAID 社の DC182888[7] を，それぞれ参考にしている．また，評価対象のプログラムには，汎用ベンチマークプログラムである SPEC CPU95 INT を用いた．

また，入力値検索のコストとして，レジスタと CAM を 32 バイト比較するのに 9 サイクル，メモリと CAM を 32 バイト比較するのに 10 サイクルを要するものとした．現

表 2: 削減サイクル数率 (SPEC CPU95 INT)

	Mean	Max	
(M)	4.9%	24.0%	(124.m88ksim)
(P)	5.3%	25.5%	(124.m88ksim)
(O)	5.4%	25.5%	(124.m88ksim)

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，外部のメモリパ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 MemoTbl との比較に要するサイクル数は現実的な値となっている．また，3.3.2 項で述べたように，登録される出力エントリ数が増大すると考えられるため，MemoTbl の OutTbl 容量を 4K 行から 32K 行に増加させている．

5.2 評価結果

SPEC CPU95 INT (train) の 8 つのプログラムを 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．評価結果を**図 13** および**表 2** に示す．

図 13 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，read_ovh はレジスタやキャッシュと InTbl(CAM) との一致比較オーバヘッド，write_ovh 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，cache1_mis，cache2_mis，window_mis はそれぞれ 1 次/2 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．なお，提案モデルについては，関数の部分再利用を実装したものと，それに加えて 4.4 節で述べたような，オーバヘッドフィルタを改良したものについて評価を行った．図 13 中で各ベンチマークプログラムの結果を 4 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) 従来モデル
(P) 関数の部分再利用を行うモデル
(O) オーバヘッドフィルタを改良したモデル

が要したサイクル数を表している．なお，各サイクル数は (N) を 1 とする正規化を行っている．

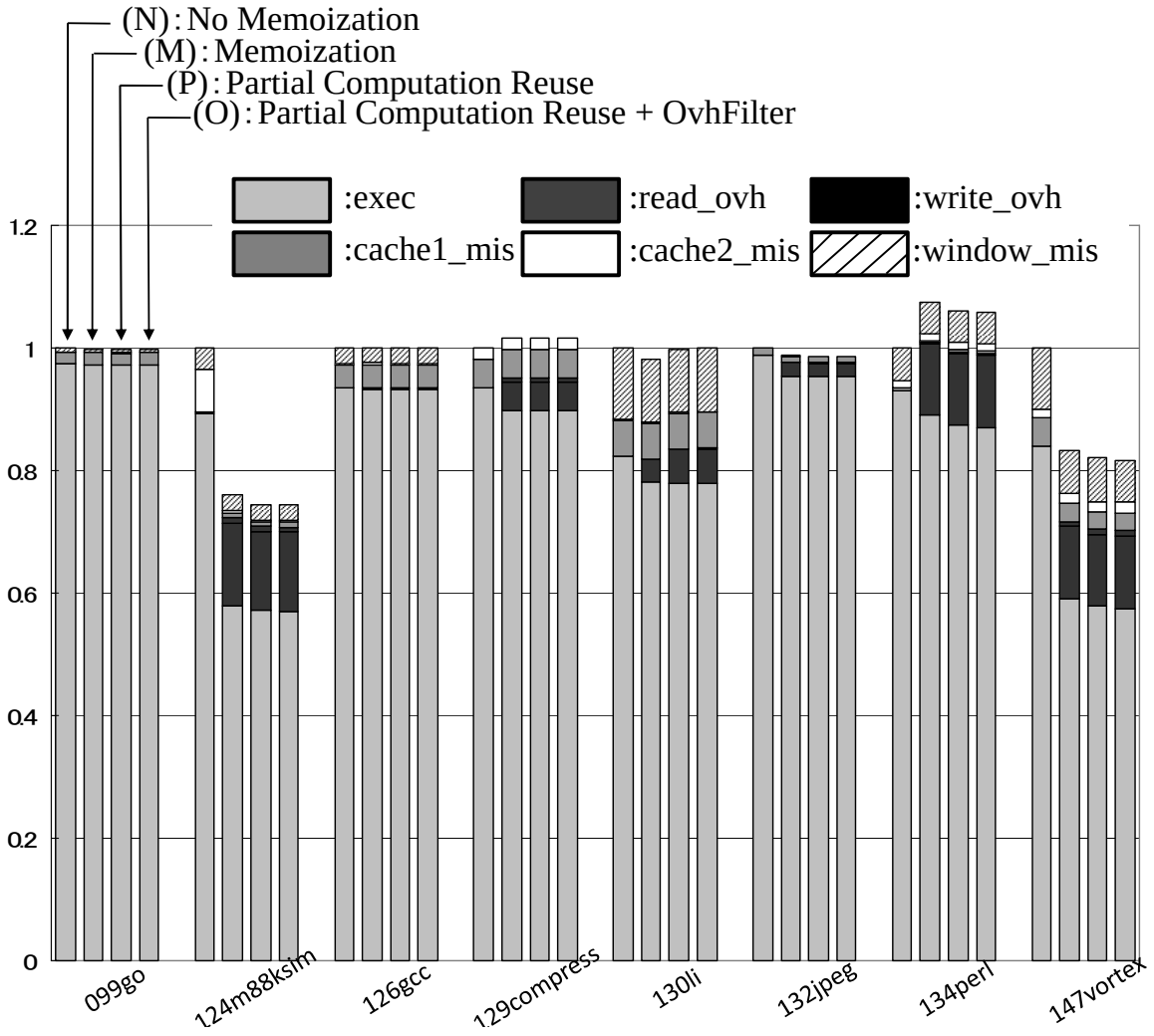

図 13: 実行サイクル数比（SPEC CPU95 INT）

まず，(P) の結果より，124.m88ksim，134.perl，147.vortex の三つのプログラムで exec を削減することができ，性能が向上している．特に 124.m88ksim に関しては，検索コストも削減されていることがわかる．しかし，130li は検索コストが増加したことにより，性能が悪化している．他の四つのプログラムでは，部分再利用があまり行われず，性能が変化していない．

次に，(O) の結果を (P) と比較すると，134.perl，147.vortex の二つのプログラムで exec を削減することができている．これは，改良前のオーバヘッドフィルタでは部分再利用による利益が判定に反映されず，再利用テストが中止されていた命令区間に対

してでも，改良後のオーバヘッドフィルタではその命令区間は再利用の効果があると判定され，再利用テストが続行された事により，再利用成功回数が増加したからである．しかし，再利用テスト回数が増加したことによって，検索コストも増加しており，性能向上はわずかなものとなっている．また，124.m88ksim はわずかに exec を削減することができたが，検索コストの増加がそれを打ち消してしまい，性能が変化していない．130li に関しては，検索コストの増加が exec の削減を上回ってしまい，わずかに性能が悪化している．これは，再利用テストを続けても，それによる再利用の成功回数がほとんど増加しなかったためである．他の四つのプログラムでは，元々部分再利用があまり行われていないため，それを考慮した所でほとんど影響がなく，性能が変化していない．

結果をまとめると，メモ化なし (N) に比べ，従来モデルである (M) では最大で 24.0%，平均で 4.9%のサイクル数の削減だったのに対し，提案モデルである (O) では最大 25.5%，平均 5.4%のサイクル数を削減できた．

5.3 考察

提案モデル (P) では関数に部分再利用を適用したことによって，三つのプログラムでサイクル数が削減できた．これは，部分再利用を適用することにより，従来モデルでは再利用できなかった部分の実行を省略することができ，exec が削減されたからである．

次に，提案モデルで 124.m88ksim の検索コストが削減されたことについて**図 14** を用いて説明する．図 14 は 124.m88ksim の killtime 関数のアセンブリコードを示す．このプログラムの 0x31834 の bpos 命令は分岐命令であり，0x31814 から 0x31838 までの範囲はループである．このため，killtime 関数に部分再利用を適用し，0x3189c までの実行を省略した場合，その範囲にループが含まれる事になる．従来モデルではこの場合，完全再利用に失敗するため，killtime 関数を通常通り実行する．そして，0x31814 から 0x31838 までの範囲のループに対して再利用テストを行うため，ループを再利用するための検索コストがかかっていた．一方提案モデルでは，部分再利用によって 0x3189c から実行が再開されるため，従来モデルでかかっていたループの検索コストが削減された．

一方，130li で検索コストが増加したことについては，部分再利用によって上記と同様に一括で再利用した事が影響したと考えられる．従来モデルでは，上記で例として挙げたような命令区間は再利用テストに失敗するため，通常通り実行される．そし

```
000317f0<killtime>:
   317f0: sethi  %hi(0x236800), %o1
     …
   31814: ld  [ %o2 + %o4 ], %o0
     …
   31834: bpos  31814 <killtime+0x24>
   31838: add  %o2, 4, %o2
     …
   3189c: ld  [ %o2 + %o4 ], %o1
     …
   318c4: retl
   318c8: nop
```

図 14: 124.m88ksim の killtime 関数のアセンブリコード

て，命令区間の内側に含まれるループ区間が実行され，そのループの入出力エントリが MemoTbl に登録される．この MemoTbl の容量は有限であるため，ループの入出力エントリが登録されたことによって，既に MemoTbl に登録されていた他のエントリが追い出される．これに対して，提案モデルにおいて内側にループを含んだ命令区間が部分再利用された場合，内側のループは実行されないため，ループの入出力エントリは MemoTbl に登録されない．このため，従来モデルではこの時 MemoTbl から追い出されていたエントリが，提案モデルでは追い出されない．これにより，従来モデルでは再利用テストの対象とならなかった命令区間に対しても，提案モデルでは再利用テストが行われる可能性がある．この再利用テストはその成否に関わらず，検索コストを増加させる．また，他のプログラムでは，部分再利用されることがあまり無かったため，性能が変化していない．これについては，既存の自動メモ化プロセッサでは検索オーバヘッドが大きく，再利用による性能向上が得られない命令区間が多いため，オーバヘッドフィルタによって再利用の登録や検索が限定されていることが原因である．したがって，検索コストを減らす他の自動メモ化プロセッサ高速化手法と組み合わせることによって，相乗効果が得られると考える．

一方，(P) と (O) を比較すると，これら二つの総実行サイクル数はほとんど変わらず，部分再利用による効果を考慮した新たなオーバヘッドフィルタを適用しても期待

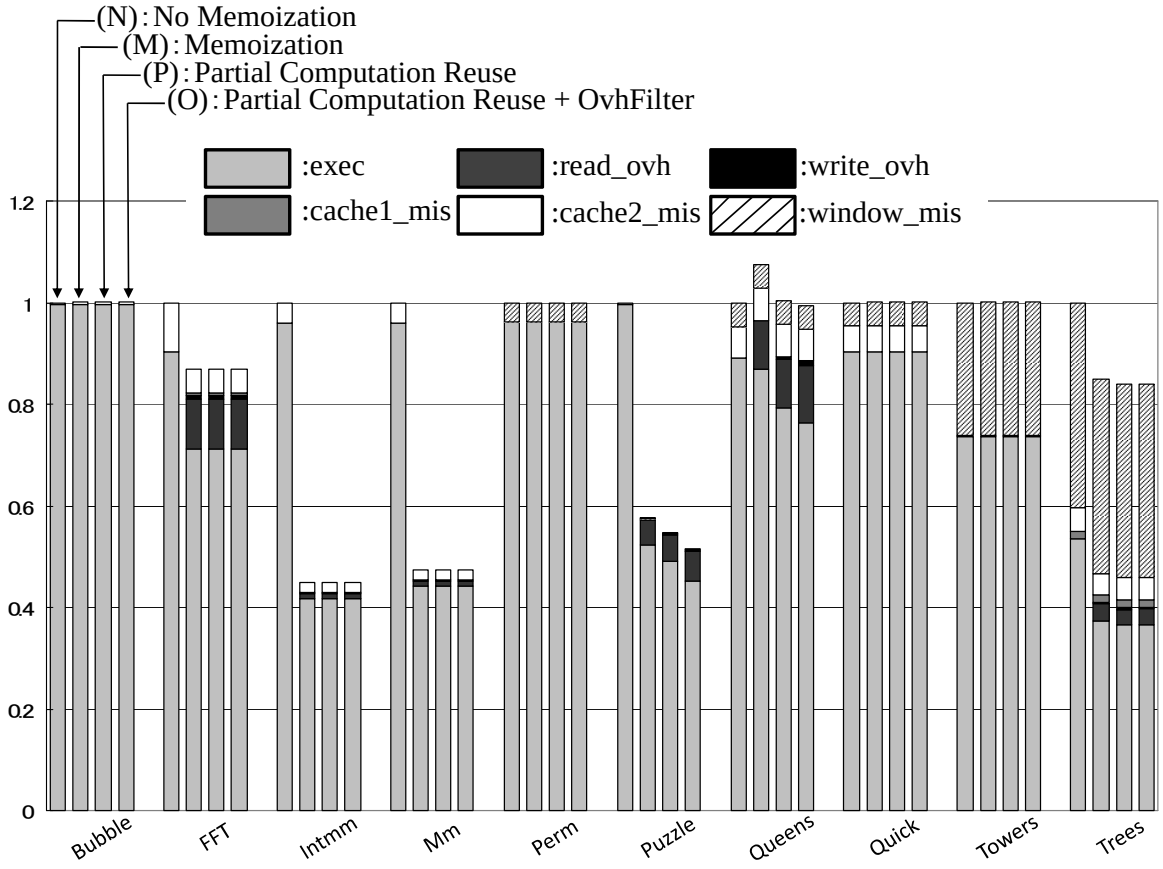

図 15: 実行サイクル数比（Stanford）

するような効果が得られなかった．そこで，あらゆるプログラムにおいても効果が無いのかを調べるために，これらの手法について Stanford ベンチマークでも評価を行った．その結果を**図 15** に示す．図 15 からわかるように (O) では Puzzle と Queens の二つのプログラムで (P) に比べて多くのサイクル数を削減できている．これは，改良前のオーバヘッドフィルタでは部分再利用による利益が判定に反映されず，再利用テストが中止されていた命令区間に対しても，改良後のオーバヘッドフィルタではその命令区間は再利用の効果があると判定され，再利用テストが続行された事により，再利用成功回数が増加したからである．その結果，(O) は (P) と比較して Puzzle で 3.0%，Queens で 1.1%性能が向上しており，オーバヘッドフィルタを改良したことによる効果が得られた．

6　おわりに

本研究では，従来の自動メモ化プロセッサの更なる高速化手法として，再利用の対象とする命令区間を既存の再利用対象区間の先頭から入力値が一致している範囲までの区間とすることで，既存の自動メモ化プロセッサでは検索に失敗する場合でも部分的に命令の実行を省略する手法を提案した．また，この手法に対して，より適切なオーバヘッド評価機構を提案することによって，更なる高速化を図った．SPEC CPU95 INT ベンチマークを用いて評価した結果，通常通りに命令を実行するのと比較して，従来手法では最大で 24.0%，平均で 4.9%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 25.5%，平均で 5.4%のサイクル数の削減となり、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，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，ループの部分再利用を実装することである．本研究では，関数の部分再利用しか検証しておらず，もう一つの再利用対象であるループへ部分再利用を適用することで，新しい知見が得られると考えられる．長期的な課題としては，本研究における提案手法を他の自動メモ化プロセッサ高速化手法と組み合わせることが挙げられる．本研究における提案手法は，自動メモ化プロセッサの再利用オーバヘッドや再利用表の大きさによる制限を解消する他の手法とは着眼点が異なっているため，こうした自動メモ化プロセッサ高速化手法との融合が可能だと考えられる．特に，検索オーバヘッドを削減する手法と組み合わせることによって相乗効果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，松井俊浩准教授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，松井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] 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).

[4] 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).

[5] 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).

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

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