Vol.2012-ARC-201 No.16
2012/8/2

再利用対象区間の細分化による自動メモ化プロセッサの高速化

神村 和敬[1]　山田 龍寛[1]　小田 遼亮[1]　津邑 公暁[1]　松尾 啓志[1]　中島 康彦[2]

概要：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は，関数とループを再利用の対象としており，実行時にその入出力を記憶しておく事で，同一入力による同一命令区間の実行を省略する．本稿では，再利用対象区間において，途中までの入力一致による途中までの計算再利用を許す事で，命令区間の実行を部分的に省略する高速化手法を提案する．また，このような一部の区間に対する再利用からも十分な効果を得るため，再利用表を分割する事で再利用表の検索にかかるオーバヘッドを小さくする手法も提案する．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 40.6%，平均 10.6%であったサイクル削減率が，2 つの手法を併用する事により最大 55.1%，平均 22.8%まで向上する事を確認した．

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，微細化によるクロック周波数の向上によって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事で，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るようになった．また，近年は電力効率と性能向上を両立させる観点から，SPARC T4[1] や Opteron[2] などの，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，100 コア構成の TILE-Gx[3] が予定されているように，搭載コア数をさらに増大させたメニーコアプロセッサが今後一般化していくと予想されている．

これら既存のプロセッサ高速化手法は，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．これに対し我々は，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 [4] 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事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事で，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，また並列化とも併用可能であるという利点がある．

本稿では，再利用対象区間において，途中までの入力一致による途中までの計算再利用を許す事で，命令区間の実行を部分的に省略する高速化手法を提案する．これにより，既存の自動メモ化プロセッサでは再利用に失敗する場合でも，命令区間の実行を部分的に省略する事が可能となる．ただし，このような一部の区間では，計算量が軽微である場合が多く，再利用による効果を十分に得るためには，再利用に伴って発生するオーバヘッドを小さくする必要がある．そこで，再利用表を分割する事によって，再利用表の検索にかかるオーバヘッドを小さくする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 再利用機構の構成

計算再利用（Computation Reuse）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）**[5] と呼ぶ．メモ化は元来，高速化のためのプログラミングテクニックである．ただし，メモ化を適用するためには，プログラムを記述し直す必要があり，既存の

[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

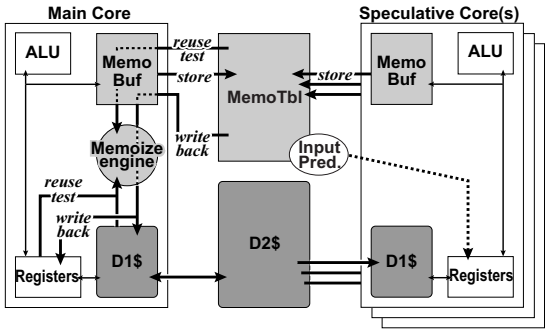

図 1　自動メモ化プロセッサの構成

ロードモジュールやバイナリをそのまま高速化する事はできない．その上，ソフトウェアによるメモ化 [6] 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．そこで，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**自動メモ化プロセッサ（Auto-Memoization Processor）**を提案している．自動メモ化プロセッサの構成の概略を**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（Registers），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（Memoize engine），再利用表（MemoTbl），および MemoTbl への書き込みバッファ（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．

なお，紙面の都合上，詳細は文献 [4] に譲るが，自動メモ化プロセッサは，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，値予測に基づいて同一命令区間をメインコアに先がけて実行する投機実行コアを複数備えている．これにより，イタレータ変数を入力のひとつとして扱うループイタレーションに対しても計算再利用が適用可能となる．

自動メモ化プロセッサは再利用対象命令区間に進入すると，MemoTbl を参照し現在の入力セットと MemoTbl に記憶されている過去の入力セットを比較する．もし，現在の入力セットが MemoTbl 上のいずれかの入力セットと一致する場合，Memoize engine は入力セットに対応する出力セットをレジスタやキャッシュに書き戻し，命令区間の実行を省略する．一方，現在の入力セットが MemoTbl 上のいずれの入力セットとも一致しない場合，自動メモ化プロセッサは当該命令区間を通常実行しながら，入出力を MemoBuf に格納し，実行終了時に MemoBuf の内容を MemoTbl に登録する事で将来の再利用に備える．

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，どの命令区間に対応しているかを示すインデクス（FLTbl idx），その命令区間の実行開始時のスタックポインタ（SP），関数の戻りアドレスとループの終端アドレス（retOfs），命令区間の入力セット（Read）および出力セット（Write）を持つ．

```
1  int a = 3, b = 4, c = 8;
2  int calc(x){
3    int tmp = x + 1;
4    tmp = tmp + a;
5    if(tmp < 7)
6      tmp = tmp + b;
7    else
8      tmp = tmp + c;
9    return(tmp);
10 }
11 int main(void){
12   calc(2); /* x = 2, a = 3, b = 4 */
13   b = 5; calc(2); /* x = 2, a = 3, b = 5 */
14   a = 4; calc(2); /* x = 2, a = 4, c = 8 */
15   a = 3; calc(2); /* x = 2, a = 3, b = 5 */
16   return(0);
17 }
```

図 2　サンプルコード

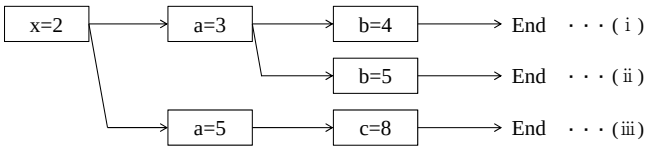

図 3　入力パターンの木構造

さて，一般に命令区間内では，複数の入力値が順に参照され使用される．しかし，同じ命令区間でも，その入力アドレスの列は分岐していく場合がある．例えば，条件分岐命令を実行した後，次に参照されるアドレスはその条件分岐命令の分岐結果によって変化してしまう．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いくため，その全入力パターンは木構造で表現する事ができる．そこで，自動メモ化プロセッサは，全入力パターンを木構造で表現し，MemoTbl に格納する．例えば，自動メモ化プロセッサが**図 2** に示すサンプルプログラムを実行する場合，関数 calc について**図 3** に示すような木構造を作る．なお，図 3 のノードは命令区間の入力値を，エッジは入力値と次に参照される入力値との対応関係を示しており，End はそれ以上の入力値が存在しない事を示す．また，入力セット (i)，(ii)，(iii) はそれぞれサンプルプログラムの 12 行目，13 行目，14 行目における関数呼び出しに対応する．この例において，入力セット (i) と入力セット (ii) では，変数 b が 3 番目に参照されるのに対して，入力セット (iii) では，変数 c が 3 番目に参照される．これは，2 番目に参照される変数 a の値が異なる事により，プログラムの 5 行目における条件分岐の結果が

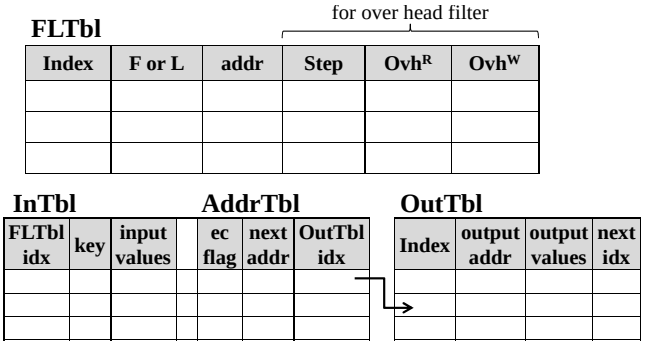

図 4　MemoTbl の構成

変化し，次入力アドレスが変化したためである．

次に**図 4** に MemoTbl の具体的な構成を示す．この MemoTbl は以下の 4 つの表から構成される．

FLTbl: 命令区間の開始アドレスを記憶する表

InTbl: 命令区間の入力を記憶する表

AddrTbl: 命令区間の入力アドレスを記憶する表

OutTbl: 命令区間の出力を記憶する表

FLTbl，AddrTbl，OutTbl は RAM で実装されており，InTbl は高速な連想検索が可能な汎用 3 値 CAM（Content Addressable Memory）で実装されている．

FLTbl は再利用対象となる各命令区間に対応する行を持っている．命令区間の判別には，関数とループを判別するフラグ（F or L）と，命令区間の開始アドレス（addr）を用いる．また，FLTbl は後述するオーバヘッドフィルタにおいて使用されるフィールドも持つ．

InTbl の各行は FLTbl の行番号 Index に対応する FLTbl idx を持ち，この値を用いてどの命令区間の入力値を記憶しているかを判別する．また，命令区間の全入力パターンを木構造で管理するため，入力値（input values）に加えて，親エントリのインデクス（key）を持つ．この input values は 1 キャッシュブロック分の入力値を記憶し，比較する必要のないアドレスの入力値についてはドントケアで表現される．

AddrTbl の各行は入力値検索のために，次に参照するべきアドレス（next addr）を持つ．この AddrTbl は InTbl と同数のエントリを持ち，各エントリは 1 対 1 に対応する．また，AddrTbl はそのエントリが入力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 OutTbl のエントリを指すインデクス（OutTbl idx）も持つ．

OutTbl の各行は命令区間の出力先のアドレス（output addr）と出力値（output values）を持つ．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 idx）を持つ．なお，この next idx を参照する事で出力セットの終端エントリであるか否かを判定できる．

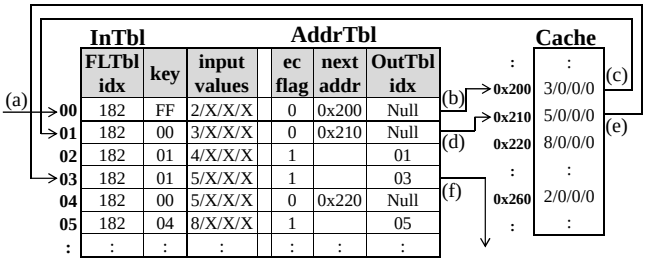

図 5　MemoTbl の検索手順

2.2　再利用機構の動作

図 5 は図 3 に示した入力パターンを InTbl と AddrTbl へ実際に登録した時の格納例を示している．図中の InTbl における X はドントケアであり，そのアドレスに対応する値は検索時に比較されない．また，図 5 は図 2 の 15 行目における関数呼び出し時の MemoTbl 検索フローも示している．いま，再利用区間の実行開始アドレスが検出されると，input values が現在のレジスタ上の入力値と一致し，かつ key が FF であるような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 0x200 番地を指しているため，そのアドレスに対応するキャッシュラインを参照する (b)．そして，得られた値を input values 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 03 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するため，入力の検索に成功する．この時，検索の終点となる AddrTbl エントリの OutTbl idx に格納されているインデクスが指す OutTbl エントリを参照し (f)，読み出した出力値をレジスタやキャッシュに書き戻す事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却って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，ある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

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

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，当該区間に対して再利用を中止する事によって性能悪化を防ぐ．

3. 再利用対象区間の細分化

本章では，本稿で提案する再利用対象区間の細分化を行う再利用モデルについて説明する．

3.1 計算再利用の部分的適用

自動メモ化プロセッサは，MemoTbl を検索する際に入力値が一つでも異なれば，出力値が過去と異なる可能性があるため，再利用を適用できない．その上，MemoTbl を検索するコストがオーバヘッドとして発生するため，実行サイクル数が増加してしまう．しかしながら，検索に失敗するような場合にも，途中までの入力が一致していれば，途中までの実行は同じであるため，そこまでの出力が一致する．例えば，図 2 のサンプルプログラムの場合，まず，12 行目で calc が呼び出される．この時，大域変数 b の値は 4 である．一方，13 行目で calc 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，13 行目における関数呼び出しは，検索に失敗する．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 とは異なる．しかし，この 2 回目の関数呼び出しにおいて，引数 x の値と大域変数 a の値はそれぞれ 1 回目と同じ値であるため，calc の 5 行目までの実行結果は一致する．このため，13 行目の関数呼び出し時では，局所変数 tmp に 6 を書き戻す事で，6 行目から実行を再開できる．また，引数 x の値のみが一致する場合には，局所変数 tmp に 3 を書き戻す事で，4 行目から実行を再開できる．このように，検索に失敗する場合には，それまでに一致した入力に応じて適切な出力を書き戻す事で，命令区間の実行を部分的に省略する事ができる．そこで，再利用対象区間において，途中までの入力一致による途中までの計算再利用を許す手法を提案する．これにより，関数やループを単位とした再利用に加え，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，命令区間の実行を部分的に省略する事が可能となり，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分の実行を省略する事を**部分再利用**と呼び，通常の計算再利用を**完全再利用**と呼ぶ．

3.2 再利用表の水平分割

前節で述べた部分再利用により，命令区間の一部を再利用する事が可能になる．ただし，この部分再利用は完全再利用と比べて 1 度に省略できる実行サイクル数が少ないため，再利用に伴なって発生するオーバヘッドの影響を受けやすい．しかしながら，入力の検索に用いる InTbl は汎用 3 値 CAM で構成されているため，CAM のアクセスレイテンシを小さくすることで，InTbl の検索にかかるオーバヘッドを小さくできると考えられる．そこで，CAM のアクセスレイテンシがその総容量に依存する事に着目し [7]，より小容量で高速な CAM を複数台用いて InTbl を構成する事によって，検索にかかるオーバヘッドを小さくする手法を提案する．ここで，小容量 CAM の台数を N とすると，幅が 32Bytes，深さが 4K 行の既存の InTbl は，幅が 32Bytes，深さが $4K/N$ 行の N 個の InTbl に水平分割される．また，AddrTbl は InTbl と 1 対 1 に対応するため，InTbl と共に分割される．これ以降，InTbl と AddrTbl をまとめて入力表と呼ぶ．

<table>
<tr><th rowspan="2">FLTbl idx</th><th rowspan="2">SP</th><th rowspan="2">retOfs</th><th colspan="5">Read</th><th colspan="3">Write</th></tr>
<tr><th colspan="4">#1</th><th>…</th><th colspan="2">#1</th><th>…</th></tr>
<tr><th>val</th><th>part</th><th>PC</th><th>Out</th><th>…</th><th>val</th><th>local</th><th>…</th></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

図 6 拡張後の MemoBuf の構成

この分割された入力表を効率良く使用するためには，一部の入力表に登録が集中しエントリが追い出されてしまうのを防ぐため，各入力表にエントリを可能な限り分散させて登録する必要がある．そこで，新しいエントリを登録する際には，保持しているエントリ数が最も少ない入力表にエントリを登録するようにする．また，自身の親エントリのインデクスである key が同じエントリは同一の入力表に登録されるようにする．これにより，各入力値を検索する際には，その親エントリにより指定された入力表のみ検索する事で再利用の可否を判断できる．

さて，2.3 節で述べたように，再利用を適用する事で性能が悪化すると考えられる命令区間は，オーバヘッドフィルタによって再利用の適用を中止されている．しかし，入力表の分割により，MemoTbl の検索にかかるオーバヘッドが減少すると，再利用を適用する事で性能が向上する命令区間が増加すると考えられる．このため，入力表を分割する事によって，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，再利用回数が増加するという効果も期待できる．

4. 実装

本章では，前章で述べた再利用モデルを実現するために必要となる実装について説明する．

4.1 拡張したハードウェア

部分再利用手法と再利用表分割手法を実現するために拡張した MemoBuf と MemoTbl をそれぞれ**図 6** と**図 7** に示す．以下，提案手法を実現するための拡張を部分再利用手法と再利用表分割手法に分けて説明する．

4.1.1 部分再利用手法のための拡張

部分再利用手法を実現するためには，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できるようにする必要がある．このため，実行を再開するプログラムカウンタ（PC）を MemoBuf の Read と AddrTbl の各入力エントリにそれぞれ追加する．ただし，この途中までの実行結

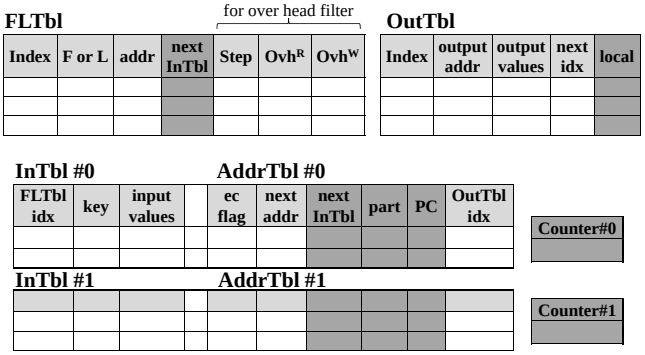

図 7　拡張後の MemoTbl の構成

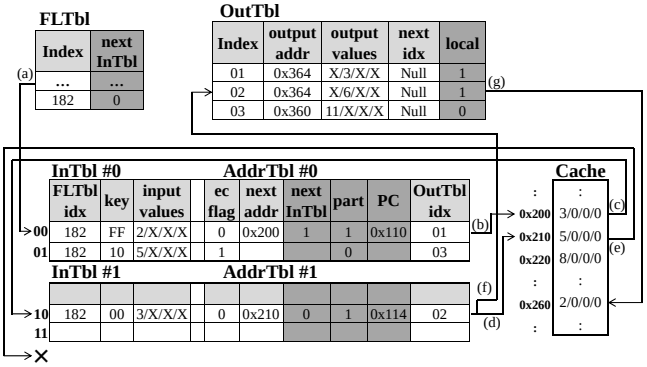

図 8　拡張後の MemoTbl 検索手順

果には，その関数の局所変数への書き込みも含まれる．このため，完全再利用では不要であった局所変数に対する出力も，部分再利用のために記憶する必要がある．そこで，各出力エントリが大域変数か局所変数かを判別する 1bit のフラグ（Local）を，MemoBuf の Write と OutTbl の各出力エントリにそれぞれ追加する．なお，この途中までの実行結果とは，関数の実行が開始されてから次の入力値が読み込まれるまでに発生した書き込みである．そこで，新しい入力値が登録される毎に，MemoBuf の Write を部分再利用の出力として OutTbl に登録するようにする．一方で，入力エントリについては，命令区間の終了時に一括で登録するようにする．このため，命令区間の終了までの間 OutTbl へのインデクスを保持しておくフィールド（Out）を MemoBuf の Read の各入力エントリに追加する．AddrTbl も同様に OutTbl へのインデクスを保持しておく必要があるが，AddrTbl には，このインデクスを格納するフィールドが既に存在する．このフィールドは完全再利用の出力のルートエントリを指すために使用されるもので，従来モデルでは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．そこで，入力表にエントリを登録する際には，このフィールドに部分再利用の出力を指すインデクスを格納するようにする．ただし，部分再利用のエントリは対応する入力エントリより先にテーブルから追い出される可能性があるため，各入力エントリに対応する部分再利用の出力エントリが登録されているか否かを判断する 1bit のフラグ（Part）を MemoBuf の Read と AddrTbl の各入力エントリにそれぞれ追加する．

4.1.2　再利用表分割手法のための拡張

再利用表分割手法を実現するためには，入力エントリの木構造を維持しつつ，各入力表に入力エントリを登録する必要がある．また，入力エントリを検索するコストを抑えるため，各入力エントリの子エントリは同一の入力表に登録されるようにし，どの InTbl を検索するべきかを示すフィールド（next InTbl）を FLTbl と AddrTbl にそれぞれ追加する．また，入力表毎に登録エントリ数が大きく異なってしまうのを抑えるため，各入力表はそれぞれ自身の登録エントリ数を保持するカウンタ（Counter）を持つ．

さて，部分再利用手法では各入力値にそれぞれ対応する実行結果が存在する．このため，OutTbl に登録される出力エントリ数が増加し，OutTbl の空き容量が不足する事により，性能が低下する恐れがある．そこで，本稿では関数のみを部分再利用の対象にする事で，出力エントリ数の増加自体をある程度抑制する．なおループに関しては，部分再利用によって省略できる計算量が僅かであるため，この制約による性能の悪化はほとんど無いと考えられる．さらに，OutTbl の容量を増大させる事で，出力エントリ数の増加による性能低下を軽減させる．この OutTbl は RAM で実装されているため，CAM で実装されている InTbl の容量を増大させるよりも，コストの増加は深刻ではない．OutTbl の具体的な大きさについては，5 章で述べる．

4.2　登録時の動作

提案手法では，関数を実行する際，既存手法と同様に入出力を MemoBuf に登録する．この時，関数の局所変数に対する書き込みが発生すると，Local フラグをセットしてその出力値を MemoBuf に登録する．そして，入力値が新たに登録される毎に MemoBuf の Write を OutTbl へ登録し，その出力エントリの先頭エントリのインデクスを MemoBuf の Read の当該エントリに記憶する．また，同時にその時の PC の値が，部分再利用適用後に実行を再開する PC となるため，同様にその値を MemoBuf の Read の同一エントリに記憶する．その後，命令区間の実行終了時に，MemoBuf の Read の内容と Local フラグがセットされていない Write の内容を MemoTbl へ一括で登録する．この時，各入力表の登録エントリ数を記憶するカウンタを参照し，現時点で登録エントリ数が最も少ない InTbl にエントリを登録する．そして，その InTbl の番号を親エントリの AddrTbl の next InTbl に記憶する．

4.3　検索時の動作

MemoTbl の検索手順を**図 8** に示す．なお，図 8 は図 2 のサンプルプログラムを 12 行目まで実行した状態を示している．

さて，サンプルプログラムの 13 行目において calc が呼

び出されると，まず，182 を FLTbl idx として持ち，ルートエントリを示す FF を key として持つようなエントリが，FLTbl の next InTbl で指定されている InTbl#0 から検索され，ライン 00 のエントリが発見される (a)．ここで，対応する AddrTbl エントリの Part が 1 であるため，このラインの PC と OutTbl idx の情報を使えば部分再利用を適用できるとわかる．そのため，入力値の検索に失敗するまでの間このライン番号 00 を記憶しておく．その後，既存の自動メモ化プロセッサと同様にキャッシュから得られた値を用いて，AddrTbl の next InTbl で指定されている InTbl#1 が検索され，ライン 10 のエントリが発見される (b)(c)．この時，先程と同様に Part フラグが立っているため，ここまでの入力に対応する区間も部分再利用可能である事がわかる．従って，より多くの入力値が一致したライン 10 の出力の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ため，記憶しているライン番号を 00 から 10 に更新する．その後，次の入力値である b の値が一致しないため，完全再利用に失敗する (d)(e)．

この時，既存手法では通常実行を開始するのに対し，提案手法ではこれまで保持していたライン 10 の情報を用いて部分再利用を適用する．そのために，まず，AddrTbl エントリの OutTbl idx に格納されているインデクスが指す OutTbl エントリを参照し，当該命令区間の出力を読み出す (f)(g)．次に，読み出した出力値をレジスタやキャッシュに書き戻し，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 0x114 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事で，部分的に命令区間の実行を省略する．

5. 評価

以上で述べた拡張を既存の自動メモ化プロセッサシミュレータに対して実装した．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 1** に示す．なお，全てのモデルはメインコア 1 つに，3 つの投機実行コアを加えた合計 4 コア構成とし，キャッシュや命令レイテンシは SPARC64-III[8] を参考とした．また，MemoTbl 内の InTbl に用いる CAM の構成は MOSAID 社の DC182888[9] を参考にし，サイズは 32Bytes 幅 ×4K 行の 128KBytes とした．一方，再利用表分割手法で使用する小容量 CAM の構成は eSilicon 社の eFlexCAM[10] を参考にし，サイズは 32Bytes 幅 ×256 行の 8KBytes とした．入力表を分割するモデルではこの CAM を 16 台用いて InTbl を構成した．また，プロセッサのクロック周波数が既存手法や提案手法の CAM のクロック周波数のそれぞれ 10 倍，4 倍と仮定して検索オーバーヘッドを見積もっている．そして，4.1 節で述べたように，登録される出力エントリ数が増加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．

表 1 評価環境

MemoBuf	64 kBytes
MemoTbl CAM	128 kBytes
MemoTbl small CAM	8 kBytes
Comparison (register and CAM)	9 cycles/32Bytes
Comparison (Cache and CAM)	10 cycles/32Bytes
Comparison (register and small CAM)	3 cycles/32Bytes
Comparison (Cache and small CAM)	4 cycles/32Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle/32Bytes
D1 cache	32 KBytes
line size	32 Bytes
ways	4 ways
latency	2 cycles
miss penalty	10 cycles
D2 cache	2 MBytes
line size	32 Bytes
ways	4 ways
latency	10 cycles
miss penalty	100 cycles
Register windows	4 sets
miss penalty	20 cycles/set

表 2 削減サイクル数率 (SPEC CPU95)

	Mean	Max
(M) 従来モデル	**10.6%**	**40.6%**
(P) 部分再利用モデル	11.0%	40.7%
(S) 再利用表分割モデル	22.5%	55.1%
(C) 併用モデル	**22.8%**	**55.1%**

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 を用いて実行サイクル数を評価した．この結果を**表 2** および**図 9** に示す．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，それぞれは

(N) メモ化を行わないモデル
(M) 従来モデル
(P) 部分再利用モデル
(S) 再利用表分割モデル
(C) 部分再利用手法と再利用表分割手法を併用するモデル

が要した総実行サイクル数を表しており，メモ化なしモデル (N) を 1 として正規化している．また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 は MemoTbl との比較に要したサイクル数（検索オーバヘッド），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（書き戻しオーバヘッド），D$1 および D$2 は

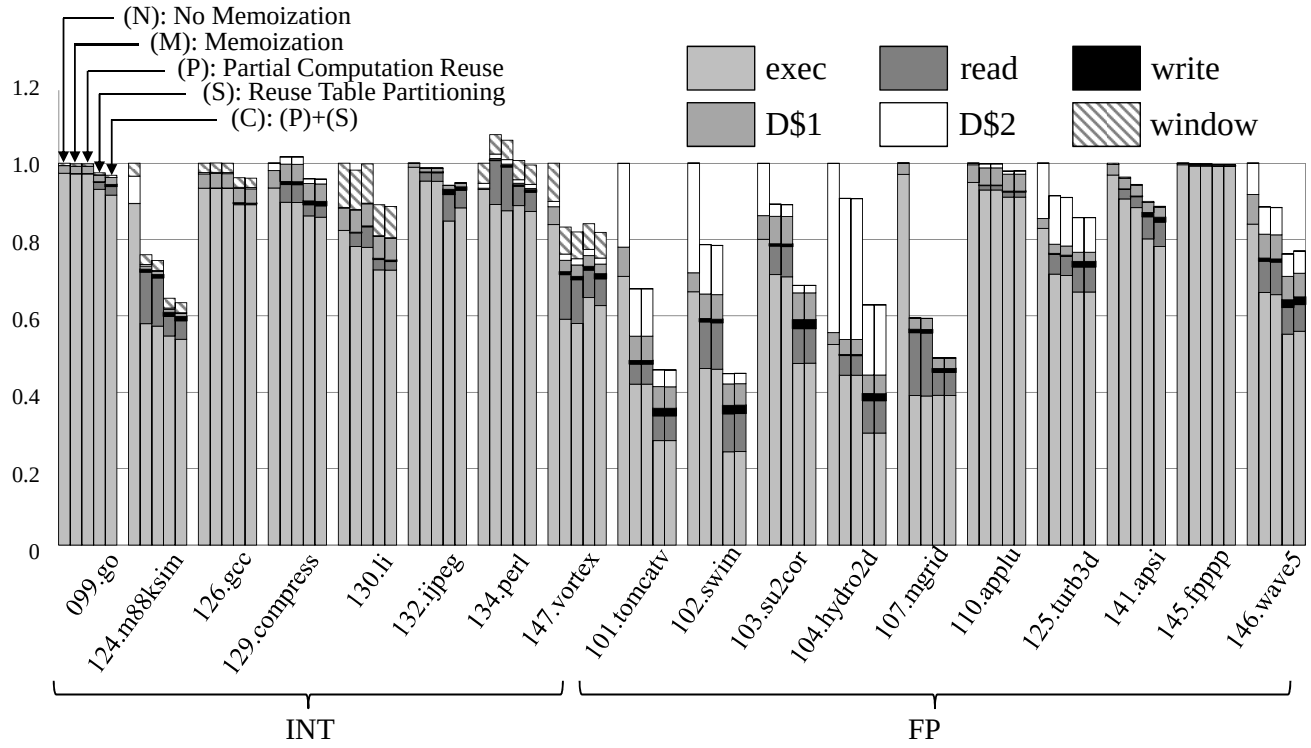

図 9　実行サイクル数

```
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 10　killtime 関数のアセンブリコード

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，(P) の結果より，124.m88ksim，134.perl，147.vortex，141apsi の 4 つのプログラムにおいて，exec が削減されている事が分かる．これは，(M) では再利用する事ができなかった命令区間に対して，(P) では部分再利用が適用され，その命令区間の実行を省略する事ができたからである．

なお，124.m88ksim に関しては，exec だけではなく read も削減されている事が分かる．**図 10** は 124.m88ksim で最も部分再利用が適用された関数である killtime 関数のアセンブリコードである．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．従来モデルでは，killtime 関数を完全再利用するための検索に失敗し，0x317f0 から通常実行される．そして，0x31814 から 0x31838 までの範囲のループを再利用するために MemoTbl を検索するため，その検索コストがかかっていた．一方 (P) では，0x3189c までの入力が一致し部分再利用が適用されるため，ループの実行は省略される．これにより，ループの再利用に伴って発生していた検索コストが削減された．

一方，124.m88ksim，134.perl，147.vortex，141apsi 以外のプログラムでは，性能向上があまり得られなかった．これは，オーバヘッドフィルタによって再利用が中止されてしまう関数が多く，部分再利用があまり適用されなかったからである．

次に (S) の結果を見ると，124.m88ksim，134.perl，107.mgrid の 3 つのプログラムにおいて，read が削減されている．このことから，入力表を分割することで InTbl の検索にかかるオーバヘッドが期待通り抑えられることを確認できた．また，多くのプログラムにおいて，exec も大幅に削減されている．このことから，MemoTbl の検索にかかるオーバヘッドが小さくなった事で，オーバヘッドフィルタによる再利用中止回数が減少し，再利用が適用される命令区間が増加するという期待通りの効果が得られている事も確認できた．一方，147.vortex では exec が増加し性能が低下している．これは，再利用の対象となる命令区間が増加し，MemoTbl に登録されるエントリ数が増加した事によって，(M) では再利用されていた命令区間のエントリが，再利用さ

れる前に MemoTbl から追い出されてしまったためだと考えられる．なお，ベンチマーク全体において，入力表の分割により入力表の利用効率が低下すると考えられるにも関わらず，その影響をあまり受けていないことが結果から見て取れる．このことから，登録エントリの少ない入力表へ優先的に新しいエントリを登録するという単純な方針でも，入力表の効率的利用に対し一定の効果があげられることが確認できた．

最後に (C) の結果を (P)，(S) と比較すると，099.go，124.m88ksim，134.perl，147.vortex，141.aspi の 5 つのプログラムにおいて，exec が削減されている事が分かる．これらのうち 099.go 以外の 4 つのプログラムは (P) でも性能が向上しており，部分再利用手法と再利用表分割手法を併用する事により，更なる高速化を達成できている事がわかる．また，099.go については，再利用表分割手法によって，再利用の対象となる命令区間が増加したため，これに伴ってより多くの関数に対して部分再利用が適用されるようになり，性能が向上した．このことから，再利用に伴うオーバヘッドを小さくする事で，部分再利用自体の効果をより増大させる事を確認できた．

結果をまとめると，SPEC CPU95 ではメモ化なし (N) に比べ，従来モデル (M) では最大 40.6%，平均 10.6%のサイクル数の削減だったのに対し，部分再利用手法と再利用表分割手法を併用する提案モデル (C) では，最大 55.1%，平均 22.8%のサイクル数の削減に成功した．

6. おわりに

本稿では，自動メモ化プロセッサの高速化手法として，再利用対象区間において，途中までの入力一致による途中までの計算再利用を許す手法と再利用表を分割する事で検索オーバヘッドを小さくする手法をそれぞれ提案した．また，これら 2 つの手法を併用する事で更なる高速化を図った．

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，従来モデルでは最大 40.6%，平均 10.6%であったサイクル削減率が，最大 55.1%，平均 22.8%まで向上する事を確認した．

今後の課題としては，まず，検索オーバヘッドの更なる削減が挙げられる．本稿でのモデルは検索オーバヘッド削減による効果が大きく，他の検索オーバヘッド削減モデル [11] と融合することで，更なる性能向上が期待できる．また，本稿では比較的単純なアーキテクチャを想定して評価を行ったが，より複雑な環境を想定したシミュレーションを行うとともに，命令レベル並列性に基づく高速化手法とメモ化とを組み合わせた手法を探っていくことも今後の課題である．

参考文献

[1] Shah, M., Golla, R., Grohoski, G., Jordan, P., Barreh, J., Brooks, J., Greenberg, M., Levinsky, G., Luttrell, M., Olson, C., Samoail, Z., Smittle, M. and Ziaja, T.: Sparc T4: A Dynamically Threaded Server-on-a-Chip, *IEEE Micro*, Vol. 32, No. 2, pp. 8–19 (online), DOI: 10.1109/MM.2012.1 (2012).

[2] Conway, P. and Hughes, B.: The AMD Opteron Northbridge Architecture, *IEEE Micro*, Vol. 27, No. 2, pp. 10–21 (online), DOI: 10.1109/MM.2007.43 (2007).

[3] Tilera Corporation: *TILE-Gx Processor Family Product Brief* (2009).

[4] 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).

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

[6] 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).

[7] Agrawal, B. and Sherwood, T.: Ternary CAM Power and Delay Model:Extensions and Uses, *IEEE VLSI*, Vol. 16, No. 5, pp. 554–564 (2008).

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

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

[10] eSilicon Corporation: *HiSilicon Licenses eSilicon's 40nm Silicon-Proven TCAMs for High-Performance Network Chips* (2011).

[11] Oda, R., Yamada, T., Ikegaya, T., Tsumura, T., Matsuo, H. and Nakashima, Y.: Input Entry Integration for an Auto-Memoization Processor, *Proc. 2nd Int'l Conf. on Networking and Computing (ICNC'11)*, pp. 179–185 (2011).