社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－ARC－179 (19)
2008／8／7

複数スレッドを用いた自動メモ化プロセッサのオーバヘッド削減手法

神谷優志[†] 島崎裕介[†] 新美明仁[†]
津邑公暁[†] 松尾啓志[†] 中島康彦[††]

我々は，計算再利用技術に基づく高速化手法を用いた自動メモ化プロセッサを提案している．現在ではマルチコア CPU やメニーコア CPU が普及し，今後は 1 つの CPU が持つコア数が更に増加すると考えられる．このような現状を踏まえ，複数のコアを有効利用する方法の検討が必要である．そこで本稿では複数のコアを有効活用するため，従来と同様の動作をする再利用スレッドの他に，投機スレッドと通常実行スレッドを用いることで，計算再利用に要していたオーバヘッドを削減する手法を提案する．提案手法の効果を検証するため，SPEC CPU95 ベンチマークを用いて評価を行った結果，従来手法では最大 13%の高速化であったが，提案手法を用いることで 20%の高速化を実現した．

Decreasing the Reuse Overhead of Auto-Memoization Processor with Multithreading

Yushi Kamiya,[†] Yusuke Shimazaki,[†] Akihito Niimi,[†]
Tomoaki Tsumura,[†] Hiroshi Matsuo[†]
and Yasuhiko Nakashima[††]

We have proposed an auto-memoization processor. This processor automatically and dynamically memoizes functions and skips their execution. Multi-core and Many-core CPUs have become general in recent years, so a CPU will have more cores hereafter. Therefore it is necessary to consider how to use multi cores effectively. This paper describes a speedup technique for auto-memoization processor using multi-threading. Speculative thread and no-memoization thread reduce the reuse overhead. The result of the experiment with SPEC CPU95 suite benchmarks shows that proposing method improves the maximum speedup from 13% to 20%.

1. はじめに

消費電力や発熱量の増大から，マイクロプロセッサの動作周波数の向上は困難になってきている．このような状況から SIMD やスーパスカラなどの命令レベル並列性（ILP: Instruction Level Parallelism）に基づく高速化手法が注目されてきた．しかし，多くのプログラムは明示的な ILP を持たないことや，ILP を抽出できる場合でもメモリスループットなどの資源的制約があることから，これらの手法にも限界がある．

一方現在では，消費電力や発熱量の問題を解決しつつプロセッサあたりの処理能力向上を可能にするため，1 つの CPU に複数のコアを搭載したマルチコア CPU が普及しつつある．また 1 つの CPU に 64 個のコアを搭載した TILE64[1] などのメニーコア CPU も登場している．今後は半導体のプロセスルールの縮小に伴い，単一プロセッサ当たりのコア数が更に増加していくと考えられる．複数のコアの利用方法として，複数のプログラムを各コアに割り当てて並列に実行する手法以外にも，複数のコアを利用したプログラム当たりの高速化手法を検討する必要がある．

そこで，並列化されていないプログラムを複数のコアを用いて高速化する一般的な手法として，スレッドレベル並列性（TLP: Thread Level Parallelism）に着目してプログラムを複数スレッドに分割し，それぞれのコアに割り当てる方法が研究されている[2]．しかし，並列性を持たず TLP を抽出することが難しいプログラムも存在し，またユーザーが明示的に並列処理プログラムを記述することは，煩雑である場合が多い．多くのコアが存在する環境では，既存の手法を利用してもなお利用されていないコアが存在する状況も考えられる．このような状況下で省電力性よりも絶対性能を重視したい場合もある．

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，我々は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を用いた自動メモ化プロセッサを提案している[3]．メモ化（Memoization）[4]

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

とは，関数等の命令区間を計算再利用可能な形に変換する処理であり，命令の実行時に入出力セットの記憶および過去の入力セットとの比較を行うことで，同一入力による当該関数の再計算を省略し実行を高速化する．我々の提案する自動メモ化プロセッサは，既存のバイナリを変更することなく，ハードウェアを用いて動的に関数等の命令区間を検出し入出力の記憶・比較を行うことで，メモ化を自動的に適用する．計算再利用を行うためにはオーバヘッドが発生し，これまでの研究でこのオーバヘッドが大きいプログラムも存在することが明らかになっている．そこで本稿では複数のコアを利用して，計算再利用によるオーバヘッドを隠蔽する手法を提案する．これにより複数のコアを備えた自動メモ化プロセッサを設計する場合，従来活用されていなかったコアを有効利用して，高速化に寄与することが可能となる．

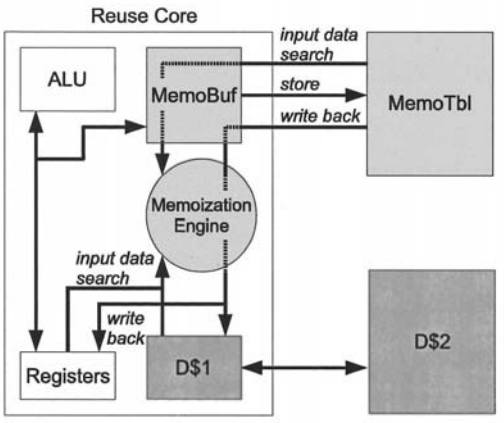

図 1 Structure of Auto-Memoization Processor

2. 自動メモ化プロセッサ

2.1 概要と構成

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記憶し，再度当該関数が過去と同じ入力で呼び出された時に実行を省略することで高速化を図る．具体的には関数の call 命令の検出から，return 命令を検出するまでに出現した入出力セットを表へと記憶する．その後再び同じ入力により当該区間を実行しようとした際には，表に記憶した出力を利用して，命令区間の実行を省略する．ここで入力とは関数の引数および関数内で参照される大域変数であり，出力とは関数の戻り値および関数内で書き換えられる大域変数である．入力，出力ともに関数内で扱う局所変数は含まない．我々の手法では，一般に OS がデータサイズおよびスタックサイズの上限を決定することを利用し，この上限および関数呼出が行われる直前のスタックポインタの値と変数アドレスとの関係から，局所変数を判別している．

自動メモ化プロセッサの構成を**図 1** に示す．自動メモ化プロセッサは関数の入出力を記憶するための表（**MemoTbl**）と MemoTbl へ書き込むためのデータを一時的に蓄えるバッファ（**MemoBuf**），そして計算再利用を行うためのこれら構成要素を管理する Memoization Engine を持つ．

再利用コアが命令区間の入出力を MemoTbl へ登録する際，サイズが大きく CPU コアからのアクセスに時間のかかる MemoTbl に対して参照を頻繁に行うと，そのオーバヘッドが大きくなる．このオーバヘッドを軽減するため，作業用の書き込みバッファとして MemoTbl に比べてサイズの小さい MemoBuf が用いられ，各命令区間の実行終了時に一括して MemoBuf の内容を MemoTbl へと登録する．紙面の都合上詳細は 3) にゆずるが，MemoBuf は再利用コアがこれまでに実行してきた関数の階層構造を保持しており，関数の内部でさらに関数を呼び出すような命令区間に対しても，メモ化を適用することができる．

MemoTbl は以下の複数の要素から構成される．

RF：命令区間の開始アドレスを記憶する表
RB：命令区間の入力セットを記憶する表
RA：命令区間の入力アドレスセットを記憶する表
W1：命令区間の出力を記憶する表

これら MemoTbl を構成する要素のうち，RF，RA，W1 は RAM で構成されている．また RB は連想検索が可能な CAM（Content Addressable Memory）で構成されており，入力セットの検索を高速に行うことができる．

一般に関数等の命令区間内においては，ある入力の値によって次に参照すべき入力アドレスが変化する．変数に主記憶アドレスが格納されている場合や条件分岐の存在がこの原因である．つまりある命令区間の全入力パターンは木構造で表わすことができ，関数の 1 入力セットはその木構造における 1 本のパスとして表現できる．

プログラムを実行し関数の call 命令を検出すると，レジスタやメモリに格納されている関数の入力値と MemoTbl に記憶した値との比較を行う．この動作を**入力値検索**と呼ぶ．入力値検索の結果，関数の入力セットが MemoTbl のあるエントリと完全に一致した場合は，当該エントリに対応する W1 エントリから出力セットを読み出し，再利用コアのレジスタやキャッシュに書き戻す．そして，入力値検索の対象であった関数の実行を省略し，後続命令区間の実行を開始する．

一方，関数の入力と一致するエントリが MemoTbl に存在しなかった場合，通常通り関数を呼び出し，命令区間を実行する．その際レジスタおよび主記憶への参照を入力，書き込みを出力として MemoBuf に記憶する．そして命令区間の終端である return 命令を検出すると，MemoBuf 内にこれまで蓄積してきた入出力セットを一括して MemoTbl に保存する．

2.2 入力値検索の手順

関数の検出時，call 命令により呼び出される関数の

開始アドレスを用いてRFを検索する．RFにより得られた始めに参照するRBエントリを初期エントリとして，MemoTblの検索を開始する，RBの各エントリは，次の入力アドレスを格納するRAエントリへのインデクスを保持している．RB内で入力の一致するエントリが存在した場合，対応するRAエントリから得た入力アドレスをもとにレジスタやメモリを参照し，次入力値を得る．この入力値を用いて再びRBを検索していく．以上の手順を繰り返し，関数の全入力が一致することを確認する．全入力が一致した場合，検索の終点となったRAエントリに格納されているW1へのインデクスを用いて，当該関数の出力セットをW1から得る．

MemoTblへのアクセスにはオーバヘッドが生じる．まず，ある関数に対して計算再利用が適用可能か否かをテストするための入力値検索を行う際には，RBやRAを参照して読み出したデータと，レジスタやメモリの内容と比較を行うためのオーバヘッドが生じる．また，入力値検索の成功時には，W1から再利用コアに対して関数の出力を書き戻すためのオーバヘッドが生じる．計算再利用を行う際に生じるこれらのオーバヘッドを，**再利用オーバヘッド**と呼ぶ．次章ではこの再利用オーバヘッドをマルチスレッドを用いて隠蔽する手法を提案する．

3. 再利用オーバヘッドの削減モデル

3.1 提案手法

本章では，入力値検索成功時に発生する再利用オーバヘッドと入力値検索失敗時に発生する再利用オーバヘッドのそれぞれに着目し，これらを隠蔽する手法について提案する．そのため，従来の自動メモ化プロセッサと同等の動作をする**再利用スレッド**に加え提案手法では，投機スレッド及び通常実行スレッドを使用する．

投機スレッドは，入力値検索成功時のオーバヘッドを隠蔽するため，当該関数に対する計算再利用が成功したものとして動作する．つまり，入力値検索の対象となる関数の入力に対応した出力は，入力値検索が終了する前にMemoTblから書き戻される．その後投機スレッドは書き戻された出力をもとに後続の命令区間を再利用スレッドと並行して実行し，再利用オーバヘッドの隠蔽を図る．

一方，入力値検索で対応する関数の入力セットがMemoTblに存在しなかった場合，再利用スレッドは入力値検索の対象区間を通常通り実行しなければならない．しかし入力値検索を行ったことによる再利用オーバヘッドは発生する．そこで**通常実行スレッド**は，入力値検索を行わずに再利用スレッドと並行して入力値検索の対象となっている命令区間を実行することでこのオーバヘッドを隠蔽する．

以上で述べた再利用スレッド，投機スレッド，通常実行スレッドを同時に実行するため，従来の自動メモ化プロセッサを3コア構成に拡張する．拡張後の自動メモ化プロセッサは，各スレッドを3つのコア間でプログラム実行中に動的に切り替えながら動作する．

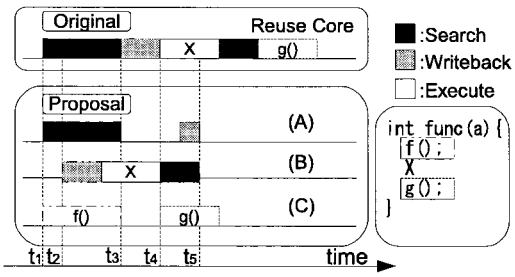

図2 Execution Model

3.2 動作モデル

拡張を加えた自動メモ化プロセッサの具体的な動作を**図2**を用いて説明する．ここでは説明の都合上，拡張後の自動メモ化プロセッサが持つ3つのコアに(A)，(B)，(C)と識別子を付し，プログラムの実行開始時には各コアがそれぞれ，再利用スレッド，投機スレッド，通常実行スレッドを実行しているものとする．

再利用スレッドを実行している(A)が時刻t_1でf()に対するcall命令を検出し，入力値検索を開始すると同時に，(B)および(C)は(A)からプログラムカウンタの内容を専用バスを利用してコピーする．同時刻に通常実行スレッドを実行する(C)は入力値検索を行わず，入力値検索の対象となっている関数f()の実行を開始する．(A)が入力値検索を行い，時刻t_2で入力の一部が一致したことを確認すると，投機スレッドを実行している(B)は当該関数に対応する出力のうち一つを選択してMemoTblから読み出し，後続区間Xを実行し始める．

時刻t_3で(A)が入力の完全一致を確認すると，(B)が行っている再利用の投機的実行が成功したか否かを判定する．この判定は投機スレッドが出力を読み出す際に用いたW1へのインデクスと，入力値検索の終点となったRAエントリが持つW1へのインデクスの値とを比較することで行われる．これらの値が等しい場合，再利用の投機的実行が成功となり，再利用スレッドを実行中の(A)と投機スレッドを実行中の(B)との間で実行するスレッドを入れ替える．同時に通常実行スレッドを実行していた(C)の計算結果はsquashされる．投機スレッドや通常実行スレッドは再利用スレッドとは独立して動作するため，再利用スレッドのMemoTblやメモリへのアクセスを妨げることはない．

ただし入力値検索に成功しても，再利用の投機的実行が失敗する場合もある．それは投機スレッドが用いたW1へのインデクスと，入力値検索終了後に実際に参照するW1へのインデクスの値が等しくない場合である．このとき，(A)は従来手法と同様に動作し，(B)や(C)の計算結果はsquashされ，スレッドの切り替えは発生しない．なお，再利用の投機的実行の失敗は

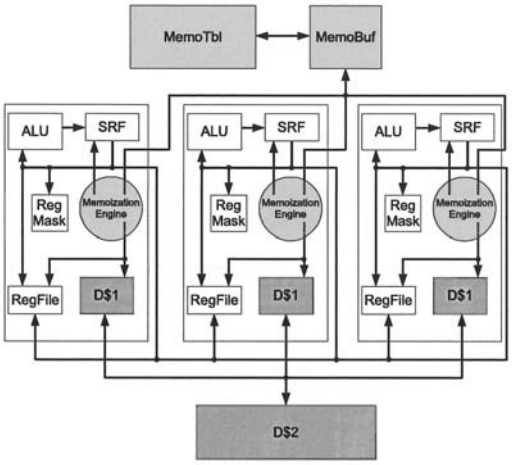

図 3 Structure of Proposed Auto-Memoization Processor

(A) から検出されないため，(B) が投機スレッドを実行したことによるオーバヘッドは無い．

時刻 t_3 で (B) が再利用の投機的実行に成功した後，時刻 t_4 で関数 g() に対する call 命令を検出すると，入力値検索を開始する．このとき，時刻 t_1 の時と同様，(C) は入力値検索を行わず，入力値検索の対象となっている関数 g() の実行を開始する．時刻 t_4 で (B) が入力値検索に失敗したことを検出すると，(B) は通常実行スレッドを実行している (C) との間でスレッドを切り替える．以降 (C) は今後再利用スレッドを実行し，(B) は通常実行スレッドを実行することになる．通常実行スレッドは，(B) が入力値検索を行っている間も先行して入力値検索の対象区間の実行を行っているため，(B) が入力値検索に失敗したことによる再利用オーバヘッドは隠蔽される．

なお本提案手法では，投機スレッドや通常実行スレッドといった投機的な実行を行うスレッドが，call 命令により新たに関数を呼び出したり return 命令により現在実行している関数からその呼び出し元へ戻ることを不可能としている．これは保持すべき入出力の増大により，MemoBuf の容量が増加することを避けるためである．

4. 実　　装

4.1 アーキテクチャ

前節で述べた提案手法とその動作モデルを実現するため，従来の自動メモ化プロセッサに対して拡張を行う．拡張後の構成を，**図 3** に示す．

再利用の投機的実行成功時や入力値検索の失敗時には各コアが実行するスレッドを切り替える必要がある．このため 3 つのコアはコア間でプログラムカウンタの値やレジスタデータをコピーするためのバスを持つ．なお各コアは全て同じ構造であり，2 次データキャッシュや MemoBuf，MemoTbl は各コアで共有している．各コアの ALU の出力は自身のレジスタファイル (RegFile) だけでなく，他のコアのレジスタファイルとも接続されており，あるコアが行ったレジスタ書き込み内容を他のコアのレジスタファイルにも同時に書き込みできる仕組みとなっている．

本提案手法では 3 つのコアが動的にスレッドを切り替えながら動作するため，各コア間でレジスタデータや主記憶データの一貫性を保証する必要がある．このため各コアはコア間でレジスタ内容を通信し，命令の実行に必要なレジスタデータやプログラムカウンタの値を得る．この時，各スレッドの動作開始時に再利用スレッドから必要なレジスタデータを全てコピーすることは，オーバヘッドの面で現実的でない．そこで，コア間で効率的にレジスタ内容を通信し，必要なコピーをできるだけ少なく抑える機構を追加した．各コアに追加した機構として，**SRF (Speculative Register File)** と **RegMask** がある．SRF は各コアが持つレジスタファイルとほぼ同様の容量，構成をしており，投機スレッドや通常実行スレッドが投機的に行ったレジスタ書き込みを RegFile に行う代わりに SRF に行う．SRF は投機的なデータを保持しているため，スレッドの切り替えが発生せず，投機スレッドや通常実行スレッドの計算結果が squash される場合は，SRF に書き込んだデータも削除される．また RegMask は，各レジスタアドレスのデータに対し，読み出すべき場所を表すためのマスクを持つ．

4.2 レジスタ及び主記憶データの一貫性

以上で述べた機構を用いて，レジスタ及び主記憶データの一貫性を保証する仕組みについて述べる．各コアは命令を実行する際，実行に必要なレジスタアドレスに対応した RegMask を参照し，当該レジスタアドレスのデータ格納場所を知る．データ格納場所には，自身の RegFile 又はいずれかのコアの SRF が考えられる．もし他のコアが当該レジスタアドレスのデータを保持していた場合，その時点で全てのコアの命令実行を中断する．そしてデータを保持しているコアの SRF からレジスタデータを得ることで，その一貫性を保証する．

次に主記憶データの一貫性を保証する仕組みについて述べる．あるスレッドが投機的な実行に失敗したとき，そのスレッドが投機的に行った主記憶書き込みの内容は不要となる．したがって，実際に主記憶に対して値を書き込んだ場合，投機的な実行が失敗した際，各コアで共有している主記憶や 2 次データキャッシュの内容に矛盾が生じる．この問題を解決するため，投機スレッド及び通常実行スレッドの実行中には，主記憶やキャッシュに値を書き込む代わりに，MemoBuf を投機的な主記憶書き込みのバッファとして使用し，主記憶の値に矛盾が生じることを防ぐ．

4.3 予測ポインタ

投機スレッドが命令の実行を開始するためには，再利用スレッドが行っている入力値検索の対象区間に対応した出力を MemoTbl から得る必要がある．このた

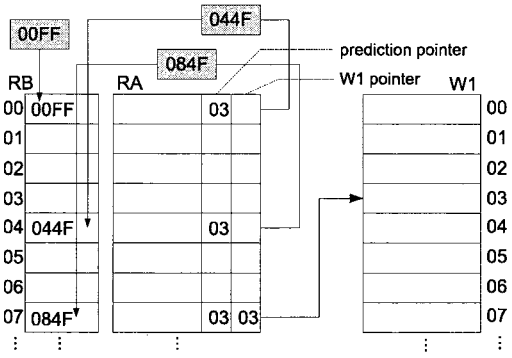

図 4 Prediction pointer

め，MemoTbl の各 RA エントリに**予測ポインタ**を追加する．予測ポインタは再利用スレッドが入力値検索を行い，入力が部分的に一致したことを確認したとき，投機スレッドが W1 から対応する出力を読み出すために参照するインデクスである．関数の入力は木構造を成して MemoTbl に登録されているため，入力が部分的に一致した時点では検索の終点となる RA エントリを一意に特定できない．これを解決するため，予測ポインタは入力値検索を行っている関数の入力に対応する W1 へのインデクスの候補を保持する．この候補は MemoTbl へ関数の入出力セットを登録した際に登録され，計算再利用に成功し関数の入力に対応する出力を読み出した際に更新される．

次に予測ポインタの動作について述べる．**図 4** は関数の入力値が順に 00FF，044F，084F である場合に MemoTbl を検索する様子を示している．MemoTbl の RB や RA を検索する際，どのエントリを検索に用いたかを RA に記憶していく．そして入力値検索に成功し，出力を MemoTbl の W1 から読み出す際に，終点の RA エントリが保持している W1 へのインデクスを，これまで記憶してきた全ての RA エントリが持つ予測ポインタに対して書き込む．これにより投機スレッドは，再利用スレッドが入力値検索を行っている途中であっても，入力セットに対応する出力セットを予測して読み出し，後続区間の実行を開始することができる．

5. 評　　価

5.1 評価環境

以上で述べた拡張を自動メモ化プロセッサシミュレータに実装し，評価を行った．シミュレータは単命令発行の SPARC V8 をベースとしている．シミュレーション時のパラメータを**表 1** に示す．なお、キャッシュや命令レイテンシは SPARC64-III[5] を参考にした．なお MemoTbl の RB を構成している CAM は，MOSAID 社の DC182888[6] の構成を参考にし，検索オーバヘッドを見積もった．また文献 7) を参考にし，32bit のレジスタデータをコア間でコピーするのに要するオーバヘッドは，1 サイクルを要するものとして見積もった．

表 1 シミュレーションパラメータ

D1 Cache 容量	32 KByte
ラインサイズ	32 Byte
ウェイ数	4 way
レイテンシ	2 cycle
ミス ペナルティ	10 cycle
D2 Cache 容量	2 MByte
ラインサイズ	32 Byte
ウェイ数	4 way
レイテンシ	10 cycle
ミス ペナルティ	100 cycle
Register Window 数	4 set
Window ミス ペナルティ	20 cycle/set
MemoBuf. サイズ	312 kByte
CAM サイズ	128 kByte
レジスタ ⇔ CAM 比較	9 cycle/32byte
メモリ ⇔ CAM 比較	10 cycle/32byte
CAM ⇒ レジスタ，メモリ書き込み	1 cycle/32byte
SRF サイズ	442 Byte
コア間のレジスタデータコピー	1 cycle/32bit

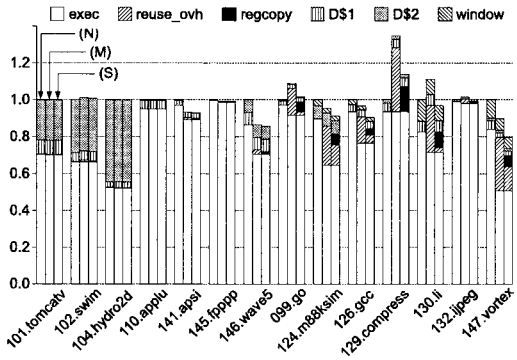

図 5 実行サイクル数（SPEC CPU95）

表 2 SPEC CPU95 の結果

	平均総サイクル数	最大削減サイクル数
(M) 従来手法	+1.2%	13.5% (146.wave5)
(S) 提案手法	-4.6%	20.0% (147.vortex)

5.2 SPEC CPU95

SPEC CPU95 を gcc-3.0.2(-O2 -mcpu=supersparc) 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．**図 5** 及び**表 2** に結果を示す．各ベンチマークプログラムにおいて、棒グラフは左から順にメモ化を行わない場合 (N)，従来手法によるメモ化を行う場合 (M)，従来手法を複数のコアを利用できるように拡張した本提案手法 (S) の総実行サイクル数である．それぞれ (N) を 1 として正規化した．凡例は左から順に命令の実行（exec），再利用オーバヘッド（reuse_ovh），コア間でのレジスタコピー（regcopy），一次データキャッシュミス（D$1），二次データキャッシュミス（D$2），レジスタウインドウミス（window）に要したサイクル数である．

まず 099.go, 129.compress, 130.li では，従来手法によるメモ化 (M) の方がメモ化を行わない場合 (N) よりも総サイクル数が増大している．一方提案手法 (S) では，(M) に比べ総サイクル数を削減することができた．しかし 099.go と 129.compress では (N) と比較すると総サイクル数が増加している．これらのプログラムは，ほぼ全ての入力値検索が失敗し，計算再利用を適用できる区間が非常に限られているという特徴を持つ．(S) では通常実行スレッドにより入力値検索の失敗に伴うオーバヘッドは隠蔽されるものの，オーバヘッド regcopy が発生する．そのため，(S) は (N) よりも総サイクル数が増加したと考えられる．その一方で 130.li ではある程度メモ化が適用できる区間が存在するが，再利用オーバヘッドも大きかったため，(M) は (N) に比べ総サイクル数が増大していた．しかし (S) では再利用オーバヘッドを隠蔽できたことで，(N) に比べ，総サイクル数を 3%削減できた．

次に 146.wave5 では，(M) は (N) に比べ最大の総サイクル数削減を実現しているが，(S) による高速化はほとんど実現できていない．この理由として 146.wave5 は一つの命令区間は比較的大きいが，一つの関数当たりの入力の数が少ない上，プログラム全体を通して関数呼び出しを行う回数も少ない．そのため，再利用オーバヘッドの総サイクル数に占める割合は極僅かとなり，再利用オーバヘッドの削減を目的とした本提案手法の恩恵を殆ど得られなかったためと考えられる．また，これと同様の傾向が 141.apsi でも現れている．

また 124.m88ksim, 126.gcc, 147.vortex といった (M) でも比較的高速化を実現しているプログラムでは，(S) により更なる高速化を実現した．これらのプログラムでは，メモ化を適用できる区間が多いため，入力値検索の発生する頻度も高く，再利用オーバヘッドの総サイクル数に占める割合は比較的大きい．そのため，本提案手法による効果がより現れやすかったと考えられる．

まとめると提案手法 (S) では SPEC CPU95 の 10 個のベンチマークプログラムについてメモ化無し (N) と従来手法 (M) の両方に対して高速化を実現した．(M) では (N) に比べ平均で 1.2%総サイクル数が増大していたものが (S) では平均で 4.6%の削減に成功した．また，(M) と比較して削減できた総サイクル数が最大となった 147.vortex では，(M) は (N) に比べ 10%の総サイクル数削減であったが，(S) により 20%の総サイクル数の削減に成功した．一般的なマルチコア環境では，並列性を持たないプログラムを高速化することは難しいが，メモ化による高速化は可能なものも存在する．本提案手法はそのようなプログラムに対し，少しでも高速化を行いたい場合に有効な手法であることが分かった．

6. おわりに

本稿では，複数のコアを用いた高速化手法として，既存の自動メモ化プロセッサを拡張し，再利用オーバヘッドを削減する手法を提案した．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ところ，多くのプログラムにおいて従来手法によるメモ化に比べ高速化を実現した．

今後の課題として，入力値検索の対象となっている関数の正しい出力をより高い精度で得るために，予測ポインタの実装方式を改良することが考えられる．現在の実装では，予測ポインタを最も最近参照したエントリ情報に基づいて登録している．この予測ポインタに登録する W1 へのインデクスを選択するためのアルゴリズムを改良し，予測ポインタの精度を改善したい．また関数の入力が MemoTbl に記憶している値と異なっていてもその出力は一致している場合，入力値検索は失敗となるが，投機スレッドの実行していた計算結果は正しい．これは関数にメモ化を適用してその実行を省略した際，その後の命令実行には関数の出力のみを使用しており，入力を使用することはないためである．このような場合，現在の実装では再利用の投機的実行を失敗としているが，投機的実行を成功として命令の実行を続けることのできるモデルを考案することも考えられる．

謝辞 本研究の一部は，文部科学省科学研究費補助金（萌芽研究 18650005，若手研究 (B) 19700041），(財) 栢森情報科学振興財団研究助成金，(財) 堀情報科学振興財団研究助成金による．

参考文献

1) Tilera Corporation: *Product Brief: TILE64 Processor* (2007).
2) 大津金光，小野喬史，横田隆史，馬場敬信：バイナリレベルマルチスレッド化コード生成方法とその評価，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム，Vol.42, No.SIG 1(HPS 6), pp.70–80 (2003).
3) Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. of Parallel and Distributed Computing and Networks*, pp.245–250 (2007).
4) Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).
5) HAL Computer Systems/Fujitsu: *SPARC64-III User's Guide* (1998).
6) MOSAID Technologies Inc.: *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edition (2003).
7) Barli, N., Tashiro, D., Iwama, C., Sakai, S. and Tanaka, H.: A Register Communication Mechanism for Speculative Multithreading Chip Multiprocessors, *In Proc. of the SACSIS 2003*, pp.275–282 (2003).