RC-009

自動メモ化プロセッサにおける再利用オーバヘッド削減

神谷優志† 島崎裕介† 新美明仁†
津邑公暁† 松尾啓志†

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．マルチコア CPU が一般的となった現在，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，複数コアを有効活用するため，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，メモ化無しに比べ，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最大 22%の高速化を実現した．

Decreasing the Reuse Overhead of Auto-Memoization Processor

Yushi Kamiya,† Yusuke Shimazaki,† Akihito Niimi,†
Tomoaki Tsumura† and Hiroshi Matsuo†

We have proposed an auto-memoization processor. This processor automatically and dynamically memoizes functions and skips their execution. Multi-core CPUs have become general in recent years, so it is necessary to consider the way of using multi-core 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 22%.

1. はじめに

CPU の消費電力や発熱量の増大から，動作周波数の向上が難しくなってきている．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 が一般的となってきている．しかし，ここで問題となるのは多くのコアを利用したプログラムの高速化である．シングルスレッドプログラムを複数のコアを用いて高速化[1] するためには，スレッドレベルの並列性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必要がある．しかし，スレッドレベルの並列性には限界があるため，割り当てるコアの数にも限界がある．

これに対し我々は，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[2]．**メモ化 (Memoization)**[3] 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，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．本稿では既存の自動メモ化プロセッサを拡張し，計算再利用を行うために必要となるオーバヘッドを複数コアを用いて隠蔽することで，処理の高速化を図る手法を提案する．

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再利用表へ記憶する．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，再利用表に記憶した出力を利用し，命令区間の実行を省略する．ここで入力とは関数の引

† 名古屋工業大学
Nagoya Institute of Technology

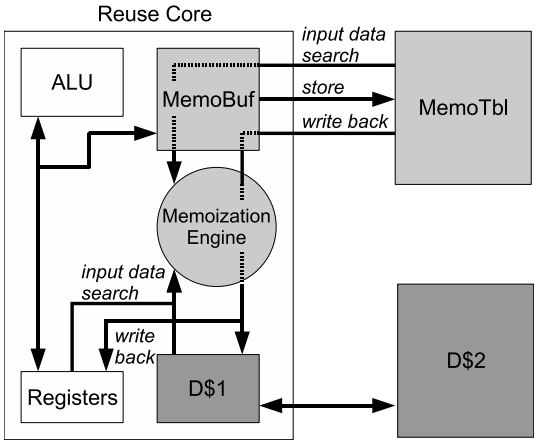

図 1 Structure of Auto-Memoization Processor

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．

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 に対して頻繁に参照を行うと，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 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．

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．一方で 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．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 エントリから出力データを読み出して再利用コアのレジスタやメモリに書き込む．そして，入力値検索の対象であった関数の実行を省略し，後続する命令区間の実行を開始する．

2.2 入力値検索の手順

関数の検出時，その開始アドレスである call 命令のアドレスを用いて RF を検索する．RF により得られた関数の開始アドレスを基に RB の検索を開始する，RB の各エントリは，次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 内で入力が一致するエントリが存在した場合，RA から得た入力アドレスを用いて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 を検索する．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．全入力が一致すると，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 を参照する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．

3. 再利用オーバヘッドの削減手法

3.1 動作モデル

既存の自動メモ化プロセッサの再利用オーバヘッドを削減するため，従来の再利用コアに加え，合計 3 つのコアを持つように拡張する．各コアは再利用スレッド，投機スレッド，通常実行スレッドをプログラム実行中に動的に切り替えて動作する．

一般的にメモ化の対象となる関数の実行では，関数の入力値は複数存在する．そのため，関数の入力値が格納されたレジスタやメモリの内容と MemoTbl の RB エントリとを複数回比較する必要がある．そこで当該関数に対する計算再利用が成功したものとして，当該関数に対応する出力を MemoTbl から書き戻し，後続の命令区間を投機的に実行する投機スレッドを再利用スレッドと並行動作させ，再利用オーバヘッドの隠蔽を図る．一方，入力値検索の結果，対応する入力セットが MemoTbl に存在しなかった場合，再利用スレッドは対象区間を通常通り実行しなければならず，このときの再利用オーバヘッドは無駄となる．そこで計算再利用を行わずに対象命令区間を実行する通常実行コアを用い，入力値検索失敗時のコストを隠蔽する．

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

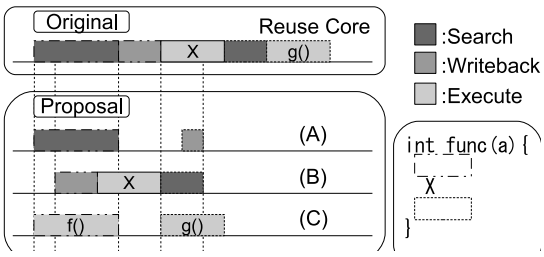

図 2 Execution Model

には各コアがそれぞれ，再利用スレッド，投機スレッド，通常実行スレッドの役割を持つものとする．

(A) が時刻 t_1 で関数 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とレジスタセット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を MemoTbl から読み出し，後続区間 X を実行し始める．

投機スレッドが命令区間 X 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得る必要がある．このため，MemoTbl に拡張を加え，RA の各エントリに**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．

時刻 t_3 で (A) が入力の完全一致を検出すると，再利用の投機的実行が成功であったと分かる．この時点で，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) との間で実行スレッドを入れ替える．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．

一方，再利用の投機的実行に失敗する場合もある．命令の入力は木構造を成して MemoTbl に登録されているため，予測ポインタの内容は入力が完全一致したときに用いる W1 へのインデクスとは必ずしも同じでない．このため，入力の一部が一致した際，投機コアが後続区間の実行を開始しても，投機コアからの出力と本来の関数の出力とが異なることがある．このとき，再利用の投機的実行は失敗となる．

時刻 t_3 で (B) が再利用の投機的実行に成功した後，引き続きプログラムを実行し，再利用スレッドを実行している (B) が時刻 l で関数 g() に対する call 命令を検出すると，(B) は入力値検索を開始する．このとき，時刻 t_1 で f() を呼び出した時と同様，(C) は入力値検索を行わず，検索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) と (C) との間でスレッドを入れ替える．これにより，(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検索対象となっている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を隠蔽できる．

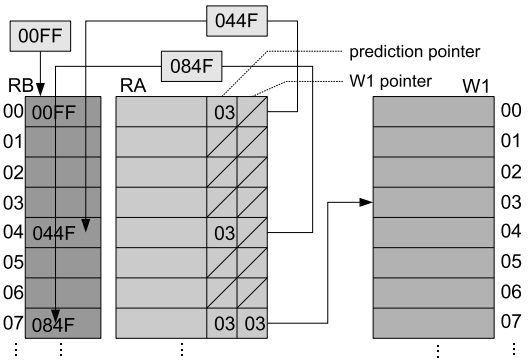

図 3 Prediction pointer

3.2 実　　装

以上で述べた動作モデルを実現するため，既存の自動メモ化プロセッサに対し，拡張が必要となる． 再利用の投機的実行の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やレジスタセットをコピーするためのバスを持つが，全て同じ構造となる．また，2 次データキャッシュや MemoBuf は各コアで共有している．

関数の入力値が 00FF，044F，084F であった場合の検索の様子を**図 3** に示す．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記憶しておく．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も，最も最近参照された入力セットに対応する出力セットを記憶している W1 エントリを知ることができる．

4. 評　　価

4.1 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[4] を参考にした．また MemoTbl の RB を構成している CAM は，MOSAID 社の DC182888[5] の構成を参考にし，検索オーバヘッドを見積もった．コア間でのレジスタコピーに要する

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
CAM サイズ	128 kB
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
コア間のレジスタコピー	2 cycle/8 本

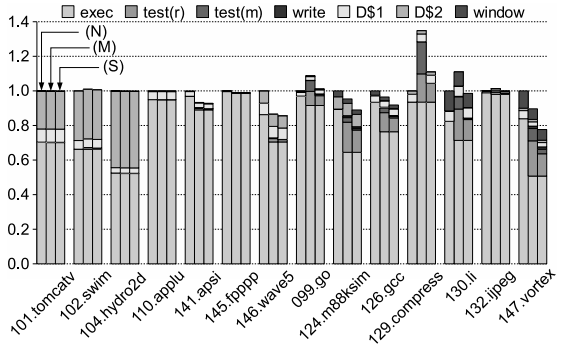

図 4 実行サイクル数 (SPEC CPU95)

オーバヘッドは，32bit レジスタ 8 本をコピーするのに 2 サイクルを要するものとして見積もった．

4.2 SPEC CPU95

SPEC CPU95 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．**図 4** に結果を示す．結果のグラフは左から順にメモ化を行わない場合 (N)，メモ化を行う場合 (M)，従来手法に複数コア使用の拡張を加えた本提案手法 (S) のグラフである．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)，レジスタと MemoTbl との比較 (test(r))，メモリと MemoTbl との比較 (test(m))，MemoTbl から得た出力のレジスタやメモリへの書き込み (write)，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．

099.go，129.compress，130.li では，(M) の方が (N) よりも総サイクル数が増大している．一方，提案手法 (S) を用いることで，(M) では再利用オーバヘッドによって (N) よりも総サイクル数が増大していたプログラムに対しても，再利用オーバヘッドを隠蔽することができた．099.go や 129.compress では (M) よりも総サイクル数を削減し，130.li では (N) よりもさらに総サイクル数を削減できた．これは，(M) では入力値検索に失敗したとき，そのオーバヘッド分だけ余分に実行サイクル数を要していたが，通常実行コアによりこのオーバヘッドを隠蔽できたためである．

また (S) では，124.m88ksim，126.gcc，147.vortex といったプログラムにおいて，(M) よりも大きく総サイクル数を削減できた．この理由として，もともと再利用オーバヘッドの総サイクル数に占める割合が比較的大きく，本提案手法による効果がより現れやすかったことが考えられる．提案手法 (S) では従来のメモ化 (M) に比べ平均で 5%，最大で 12%のサイクル数を削減できた．

5. おわりに

本稿では，複数のコアを有効活用するため，既存の自動メモ化プロセッサに対し拡張を加え，MemoTbl 参照コスト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高速化を実現した．

今後の課題として，予測ポインタの精度の向上が考えられる．現在の実装では，投機スレッドが動作を開始する際に用いる予測ポインタを，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、1 つの RA エントリが複数の予測ポインタを持ち，投機スレッドを複数用いて並列に計算再利用の投機的実行ができるよう改良を行いたい．

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) 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).
2) 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).
3) Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).
4) HAL Computer Systems/Fujitsu: *SPARC64-III User's Guide* (1998).
5) MOSAID Technologies Inc.: *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edition (2003).