Vol.2013-ARC-206 No.15
2013/8/1

再利用表パージアルゴリズムの改良による自動メモ化プロセッサのハードウェア削減手法

柴田 裕貴[1]　神村 和敬[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，およびこれに値予測に基づく投機マルチスレッド実行を組み合わせた並列事前実行機構を提案している．自動メモ化プロセッサは，関数とループを計算再利用の対象としており，実行時にその入出力を再利用表に記憶しておくことで，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，再利用表をより効率的に利用するためにループの特徴を利用した再利用表パージアルゴリズムを提案する．また，ループにおいて再利用が見込めない入出力の登録や検索を中止する機構も併せて提案する．これらにより，性能を低下させることなく再利用表の一部を構成している CAM を小容量で実装できるようになり，自動メモ化プロセッサの消費エネルギーを削減できると考えられる．SPEC CPU95 を用いてシミュレーションにより評価した結果，従来モデルにおいて，CAM の容量を 128KBytes とした場合には最大 40.5 %，平均 9.6 %，8KBytes とした場合には最大 24.9 %，平均 4.4 %であったサイクル削減率が，提案モデルでは，8KBytes で最大 40.6 %，平均 10.6 %となり，性能を低下させることなく，より小容量の CAM で実装できることを確認した．

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るようになった．また，近年では高い性能と低消費電力を両立させる観点から，SPARC T5[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っている．そして，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，コア数をさらに増大させたメニーコアプロセッサが一般化していくと予想されている．

これら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．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見なし，実行時にその入出力を再利用表に記憶しておくことで，同一命令区間を同一入力を用いて再び実行しようとした際に，その実行自体を省略する．

並列化が処理全体の総量は変化させず複数の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．計算再利用は並列化とは直行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．

また，我々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駆けて実行する投機実行コアを複数備える並列事前実行機構を提案している．これにより，イタレータ変数を入力の 1 つとして持つループに対しても計算再利用が適用可能となる．しかし，一般にイタレータ変数が単調変化するループにおいて，メインコアが実行したイタレーションに対応する入出力エントリは，今後再利用が見込めないエントリとなる．このため，再利用表に登録されている入出力エントリを等しく LRU に基づき追い出す従来の方式では，このようなエントリが再利用表に登録されたままになることで，再利用が見込める他の有益な入出力エントリが追い出されやすくなる．

そこで本稿では，再利用が見込めなくなったループの入出力エントリを優先的に再利用表から追い出す手法を提

[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

案する．また，そのような入出力エントリの登録や検索自体を中止する機構も併せて提案する．これらにより，再利用表の利用効率が向上することで，性能を低下させることなく再利用表を小容量化できる．なお文献 [5] では，メモ化のための機構によりプロセッサの消費エネルギーは約 15.4%増加するとされており，中でも再利用表を構成する CAM がその大きな要因となっている．そのため，より小容量の CAM で再利用表を実現可能とすることで，自動メモ化プロセッサの消費電力が大きく削減されると期待できる．

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の動作原理とその構成について概説する．

2.1 自動メモ化プロセッサの概要

計算再利用（Computation Reuse）とは，プログラムの関数呼び出しやループなどの命令区間において，その入力の組（入力セット）と出力の組（出力セット）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書き戻す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．また，この手法を命令区間に適用することを**メモ化（Memoization）**[6] 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，既存ロードモジュールやバイナリをそのまま高速化することはできない．

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ことで，既存のバイナリを変更することな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（Auto-Memoization Processor）**を提案している．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（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．

自動メモ化プロセッサは，実行時に動的に関数及びループを再利用可能な命令区間として検出し，その入出力を MemoTbl に記憶する．なお，関数は call 命令のジャンプ先から return 命令までの区間，ループは後方分岐命令から，その後方分岐命令のジャンプ先までの区間としている．

自動メモ化プロセッサは再利用対象区間に進入すると，MemoTbl を参照し現在の入力セットと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，入力セットに対応する出力セットはレジスタやキャッシュに書き戻され，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，入出力値を MemoBuf に格納し，実行終了時に MemoBuf の内容を MemoTbl に登録することで将来の再利用に備える．

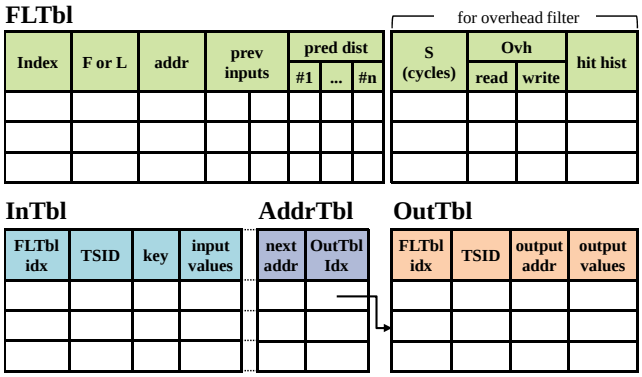

図 1 MemoTbl の構成

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．また，入れ子構造になった命令区間もメモ化対象とするために，MemoBuf は現在使用しているエントリをポインタで指しており，命令区間の検出時にそのポインタをインクリメントし，命令区間の実行終了時にデクリメントすることで入れ子構造を保持している．

MemoTbl の詳細な構成を**図 1** に示す．MemoTbl は，命令区間を記憶する **FLTbl**，入力を記憶する **InTbl**，入力アドレスを記憶する **AddrTbl**，および出力を記憶する **OutTbl** の 4 つの表から構成される．なお，InTbl，AddrTbl，OutTbl をこれ以降まとめて入出力表と呼び，ループの入出力エントリをループエントリ，関数の入出力エントリを関数エントリと呼ぶ．

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 1 行が 1 命令区間に対応しており，その行番号 (Index) を各命令区間の識別番号とする．命令区間の識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，後述する並列事前実行の入力ストライド予測のために，直近の入力セット（prev inputs），各イタレーションの実行担当コア番号（pred dist）を持つ．

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

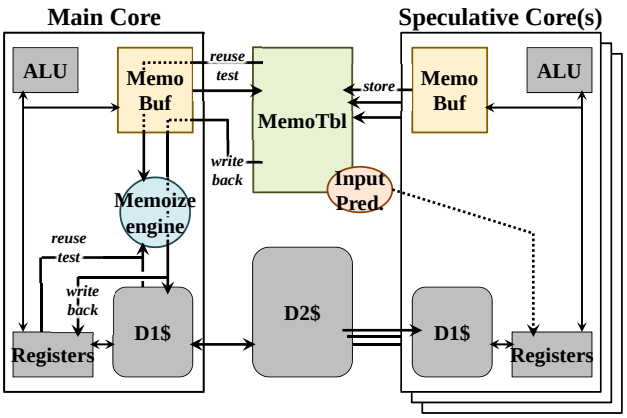

図 2　並列事前実行機構

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に入力値（input values）に加えて，親エントリのインデクス（key）を持つ．

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．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．

OutTbl の各行は FLTbl idx に加えて，命令区間の出力先のアドレス（output addr），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．

2.2　並列事前実行機構

自動メモ化プロセッサは計算再利用に基づく手法であり，ある命令区間を過去に完全に同一の入力セットで実行したことがある場合にのみ効果が得られる．よって，イタレータ変数を入力の一つとして持つループでは，全く効果が得られない．そこで，我々は計算再利用を行いながら実行を進めるメインコアの他に，値予測に基づいて同一命令区間をメインコアに先駆けて実行する投機実行コアを複数備える機構（**並列事前実行機構**）を提案している．以下この投機実行コアを **SpC**（**Speculative Core**）と呼ぶ．**図 2** に，並列事前実行機構の概要を示す．各 SpC は，それぞれ個別の MemoBuf と一次キャッシュを持ち，二次キャッシュは全コアで共有するものとする．メインコアが計算再利用対象区間を実行する際，SpC はこれに並行して，ストライド予測により算出した入力セットを用いて同一区間を投機実行する．そして，その投機実行に使用した入力セットおよび実行の結果得られた出力セットを，共有されている MemoTbl に登録する．値予測が正しかった場合，メインコアが次に実行しようとする命令区間は既に SpC により実行済みであり MemoTbl に結果が格納されているため，それを再利用することで実行を省略できる．また値予測が誤っていた場合も，メインコアは当該区間を通常実行するのみであるため，MemoTbl の検索コストは発生するものの，投機実行ミスに起因するオーバヘッドは発生しない．

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こと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 からレジスタやキャッシュに書き戻すコストが発生する．命令区間の中には，これらのオーバヘッドが大きく，再利用を適用することで却って性能が悪化してしまう区間がある．そこで，FLTbl では，各命令区間に対し一定期間における再利用の状況をシフトレジスタ（図 1 中 hit hist）を用いて記録し，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 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 S から，実際に削減できたサイクル数を

$$M \cdot (S - Ovh^R - Ovh^W) \qquad (1)$$

として計算する．なお 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．

また，再利用が行われなかった場合でも，MemoTbl の検索オーバヘッドは存在する．このオーバヘッドは，

$$(T - M) \cdot Ovh^R \qquad (2)$$

として計算できる．

ここで，式 (1) から式 (2) を引いたものを ***Gain*** とすると，

$$Gain = M \cdot (S - Ovh^W) - T \cdot Ovh^R \qquad (3)$$

となり，この $Gain$ が正値であれば発生したオーバヘッド (2) よりも，削減できたサイクル数 (1) が大きいため，再利用の効果があると判断する．そして，そのように判断した命令区間に対してのみ再利用表への登録および再利用の適用を行っている．

2.4　再利用表パージアルゴリズム

自動メモ化プロセッサは再利用表パージアルゴリズムとして，LRU に基づく仕組みと特定の命令区間の全エントリを追い出す仕組みの 2 つを備えている．

2.4.1　TSID パージ

TSID パージは LRU に基づいた追い出し手法である．自動メモ化プロセッサはリングカウンタを用いて時刻管理しており，入出力表に新しくエントリを登録する際やエントリが再利用された際に，それらエントリの TSID フィール

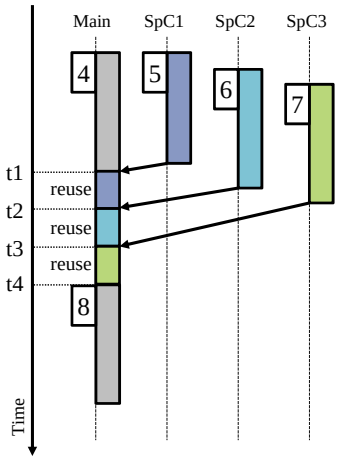

図 3　並列事前実行のタイミングチャート

ドにリングカウンタの現在の値を記憶する．そして，一定数のエントリが入出力表に登録されリングカウンタが更新されるたびに，更新後と同一のタイムスタンプを持つエントリを入出力表から追い出す．

2.4.2　FLID パージ

FLID パージは FLTbl や入出力表が溢れた場合に，特定の命令区間のエントリを選択して追い出す手法である．FLTbl が溢れた場合には，既に登録されている命令区間のうち，最近検索や登録が行われておらず再利用成功回数も低い命令区間のエントリを MemoTbl から全て追い出す．一方で，入出力表が溢れた場合には，登録中の当該命令区間に関する入出力表のエントリを全て追い出す．

3.　ループの特徴を利用した再利用表パージアルゴリズム

本章では，本稿で提案するループの特徴を利用した再利用表パージアルゴリズムについて説明する．また，併せて提案する再利用が見込めないループエントリの登録中止，検索中止についても説明する．

3.1　ループイタレーションパージ

一般にループのイタレータ変数は単調に変化することから，メインコアが実行したイタレーションに対応するエントリは今後再利用が見込めないエントリとなる．しかし，既存の再利用表パージアルゴリズムではこのような再利用が見込めないループエントリも他のエントリと同様に扱っている．そのため，このループエントリが再利用表に登録されたままになることで，再利用が見込める他の有益なエントリが追い出されやすくなる．そこで，この問題を解決するために，再利用が見込めなくなったループエントリを入出力表から追い出す手法を提案する．以降，この手法を**ループイタレーションパージ**と呼ぶ．

ループイタレーションパージを適用するタイミングを**図 3** を用いて説明する．図 3 は SpC を 3 基とし，イタレータ変数が単調増加するループに対してメインコアが入力値 4 に対応するイタレーションを実行し，SpC がそれぞれ入力値 5，6，7 に対応するイタレーションを投機実行するタイミングチャートを示したものである．まず，メインコアは時刻 t1 において，SpC1 が登録したエントリを再利用することで，入力値 5 に対応するイタレーションの実行を省略する．そして，時刻 t2 において直前に再利用した入力値 5 に対応するイタレーションのエントリは再利用が見込めなくなったため，ループイタレーションパージを適用してこのエントリを入出力表から追い出す．その後も同様に，時刻 t3 では入力値 6 に対応するイタレーションのエントリを，時刻 t4 では入力値 7 に対応するイタレーションのエントリを入出力表から追い出す．

3.2　ループイタレーション登録フィルタ

前節で述べたとおり，ループイタレーションパージは今後再利用が見込めなくなったループエントリを入出力表から追い出す手法であるが，ループエントリの中には入出力表に登録しても全く再利用が見込めないものがある．こうしたエントリとして，以下の 4 種が考えられる．

(A)　メインコアが登録するループエントリ
(B)　メインコアの実行までに投機実行が完了しなかった SpC が登録するループエントリ
(C)　入れ子構造になっているループの外側ループを SpC が投機実行した際に登録される内側ループのエントリ
(D)　メインコアが最後のループイタレーションの実行を終了した際にそのループのイタレーションを投機実行中であった SpC が登録するエントリ

まず (A) は，一般にループのイタレータ変数が単調変化するため 2 度と同じ入力セットで実行されることがなく，入出力表に登録しても再利用は見込めない．次に (B) は，メインコアが既に実行を開始したイタレーションに対するエントリであるため，(A) の理由と同様に入出力表に登録しても再利用は見込めない．次に (C) は，外側ループの再利用が失敗した場合に再利用される可能性がある．しかし，内側ループの入力が外側ループの入力に依存していないことは考えにくいため，入出力表に登録しても再利用が見込めないと考えられる．最後に (D) は，本来実行すべきイタレーション回数を超えたイタレーションのエントリであるため，入出力表に登録しても再利用は見込めない．

以上のことから再利用が見込めない (A)~(D) の 4 種のループエントリの入出力表への登録を中止する機構を提案する．ただし，(D) は do-while 文のように後方分岐命令でメインコアがループを終了する場合にのみ対応することとする．これは，それ以外のループの終了をハードウェアで検出することは困難なためである．コンパイラを改良することであらゆるループに対応可能であるが，既存バイナリに変更を加えることになってしまうため，本稿では対応し

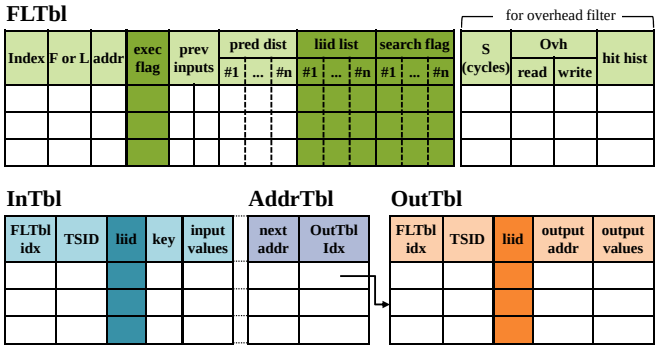

図 4　拡張後の MemoTbl の構成

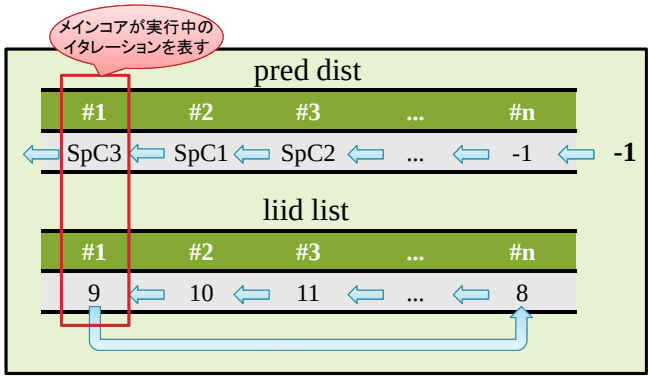

図 5　pred dist と liid list の構成

ないこととする．以降，再利用が見込めないループエントリの登録を中止するこの機構のことを**ループイタレーション登録フィルタ**と呼ぶ．この機構とループイタレーションパージにより，再利用が見込めないループエントリが入出力表に登録されなくなり，再利用表の利用効率を向上させることができる．

3.3　ループイタレーション検索フィルタ

既存の自動メモ化プロセッサでは，メインコアが実行するループに関する実行結果が入出力表に登録されていなくても，再利用表を検索する．しかし，これまで述べてきたように，ループの再利用はメインコアが将来実行するイタレーションの実行結果が事前に SpC により登録されている場合にのみ成功する．このため，メインコアが実行するループの実行結果が入出力表に登録されていない場合に再利用表を検索することは無駄である．

そこで，このような場合に再利用表の検索を中止する機構を提案する．以降，この機構のことを**ループイタレーション検索フィルタ**と呼ぶ．これにより，ループにおいて再利用率を低下させることなく検索コストを削減することができると考えられる．

4.　実装

本章では，提案手法を実現するために必要となるハードウェア拡張と動作モデルについて説明する．

4.1　ループイタレーションパージのための拡張

前章で述べた提案手法のために拡張した MemoTbl を**図 4** に示す．まず本節では，3.1 節で述べたループイタレーションパージのための拡張について説明する．

ループイタレーションパージを実現するためには，イタレーションからループエントリを特定できる必要がある．自動メモ化プロセッサは，他のコアが実行しているイタレーションと同一のイタレーションの入力値を SpC に与えてしまうのを回避するために，各コアが実行しているイタレーションを管理する必要がある．そのため，メインコアが実行しているイタレーションを基点として，そこから数個先までのイタレーションの実行担当コア番号を記憶しておくフィールド（pred dist）を FLTbl に設けている．これにより，SpC はループエントリ登録時に，投機実行終了したイタレーションがメインコアが実行しているイタレーションより何回先のイタレーションであるのかを知ることができる．そこで，そのイタレーションを他のイタレーションと区別するための固有の値を記憶しておくフィールド（liid list）を FLTbl に設ける．以降，このイタレーションを区別する liid list の値のことをイタレーション ID と呼ぶ．これにより，pred dist のフィールドに対応するイタレーションを liid list のイタレーション ID でそれぞれ区別することができるようになる．そして，InTbl と OutTbl にもこのイタレーション ID を記憶するフィールド（liid）を追加することで，メインコアは自身が実行したループイタレーションに対応するループエントリをイタレーション ID から特定し，追い出すことができる．

pred dist と liid list の詳細な構成を**図 5** に示す．この図において，pred dist の左端のフィールドにはメインコアが実行中のイタレーションの実行担当コア番号が記憶されており，メインコアがイタレーションの実行を終了したとき，および再利用に成功し実行を省略したときに，各フィールドに記憶されている実行担当コア番号が左に 1 つシフトされる．また，前述した通り pred dist が管理しているイタレーションに対応するイタレーション ID を liid list は記憶している．そのため，liid list も pred dist と同様のタイミングでイタレーション ID を左に 1 つシフトすることとする．なお，シフト後の右端のフィールドはシフト前の右端のフィールドに対応するイタレーションの次のイタレーションを表すことになり，それはまだどのコアからも投機実行されていない．そのため，pred dist の右端のフィールドには実行担当コアが存在しないことを表す値が記憶される．この図ではその値を −1 としている．一方で，liid list はイタレーション ID をできる限り少ないビット数で表すために，右端のフィールドには左端のフィールドの追い出したイタレーション ID をセットし，値を使い回すようにする．

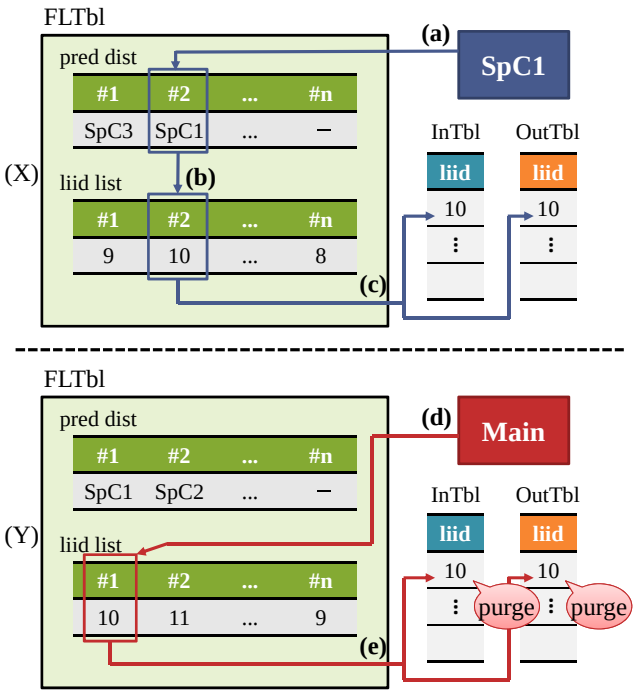

図 6　ループエントリの登録，追い出し手順

次に，ループエントリの登録および追い出し手順を示した図 6 を用いて，ループイタレーションパージの動作モデルについて説明する．なお，図 6 の (X) は SpC1 がループエントリを登録する際の MemoTbl の状態を表し，(Y) はその SpC1 が登録したループエントリをメインコアが追い出す際の MemoTbl の状態を表している．まず，SpC1 はループエントリ登録時に，実行したイタレーションのイタレーション ID を調べるために，pred dist の自身のコア番号が記憶されているフィールドを検索する (a)．そして，そのイタレーションに対応するイタレーション ID を liid list から取得し (b)，入出力表に登録するループエントリの InTbl と OutTbl の liid にその値を記憶する (c)．これにより，イタレーション ID からループエントリを特定できるようになる．その後メインコアが，イタレーション ID が 9 であるイタレーションの実行を終了したとする．このとき，前述した通り，pred dist，liid list の各フィールドは 1 つ左にシフトされ，(Y) の状態になる．その後，メインコアは SpC1 が投機実行したイタレーションを再利用したとする．このとき，メインコアは自身が再利用したイタレーションのループエントリを追い出すために，そのイタレーション ID を liid list の左端のフィールドから取得する (d)．そして，入出力表に登録されているループエントリのイタレーション ID と取得したイタレーション ID とを比較して (e)，一致しているならそのループエントリは今後再利用が見込めないと判定し，入出力表から追い出す．

4.2　ループイタレーション登録フィルタのための拡張

本節では 3.2 節で示した (A)~(D) の 4 種のループエントリについてそれぞれどのようにして入出力表への登録を中止するのかについて説明する．

まず，(A) の「メインコアが登録するループエントリ」の登録を中止するためには，実行した命令区間が関数，ループのどちらであるかを判定する必要がある．そこで，メインコアは入出力表にエントリを登録する時に FLTbl の関数とループを判別するフラグである F or L を確認することで，実行した命令区間がループであれば，そのエントリの登録を中止する．

次に (B) の「メインコアの実行までに投機実行が完了しなかった SpC が登録するループエントリ」の登録を中止するためには，投機実行したイタレーションをメインコアが実行中であるか，もしくは既に実行終了しているかを判定する必要がある．そこで SpC は，ループエントリ登録時に pred dist の左端のフィールドに自身のコア番号が登録されているか，もしくは pred dist のいずれのフィールドにも自身のコア番号が登録されていなければ，メインコアは既にそのイタレーションを実行中，または実行終了したと判定し，そのエントリの登録を中止する．

次に (C) の「入れ子構造になっているループの外側ループを SpC が投機実行する際に登録される内側ループのエントリ」の登録を中止するためには，SpC が外側ループと内側ループを区別できる必要がある．そこで，命令区間の入れ子構造を保持しながらエントリを記憶している MemoBuf に，投機実行を開始したループであるか否かを判定するためのフラグ（start flag）を設ける．まず，SpC はループの投機実行を開始する際に，対応する MemoBuf のエントリの start flag をセットする．その後，内側ループに進入したとすると，現在の MemoBuf の使用エントリを指しているポインタがインクリメントされる．このとき，その内側ループは投機実行を開始した区間ではないため，そのエントリの start flag はセットしない．そして，SpC はループエントリを登録する際に，MemoBuf の現在の使用エントリの start flag を確認する．このとき，start flag がセットされている場合は，そのループは投機実行を開始したループであると分かるため，入出力表にエントリを登録する．一方で，start flag がセットされていない場合は，そのループは内側ループであると分かるため，エントリの登録を中止する．

最後に (D) の「メインコアが最後のループイタレーションの実行を終了した際にそのループのイタレーションを投機実行中であった SpC が登録するエントリ」の登録を中止させるためには，SpC がエントリを登録しようとしているループを，メインコアがいまだ実行中であるか否かを判定する必要がある．そこで，FLTbl にそれを判定するためのフラグ（exec flag）を設け，メインコアはループ実行開始時にそのループに対応する exec flag をセットする．なお，3.2 節で述べたように後方分岐命令でループ実行を終

了する場合にのみこのループエントリの登録を中止する．そこで，メインコアはループの終了アドレスで not taken となった時に実行中のループを終了するとし，セットされている exec flag をクリアする．そして，SpC はループエントリ登録時に，実行したループ区間の exec flag を確認し，クリアされているならばそのループエントリの登録を中止する．

4.3 ループイタレーション検索フィルタのための拡張

3.3 節で述べたループイタレーション検索フィルタを実現するためには，各イタレーションに対応するループエントリが入出力表に登録されているか否かを判定する必要がある．そこで，それを判定するフラグ（search flag）を FLTbl に設ける．SpC はループエントリ登録終了時に，投機実行したそのイタレーションに対応する search flag をセットする．そして，メインコアがイタレーションの実行を終了するたびに search flag は，pred dist や liid list と同様に各フィールドに記憶されているフラグを 1 つ左にシフトし，右端のフィールドをクリアする．これにより，メインコアが MemoTbl を検索する際に，search flag の左端のフィールドがセットされている場合は，メインコアが実行を開始するイタレーションのループエントリが既に SpC によって入出力表に登録されていると分かるため，その検索を続ける．一方で，search flag の左端のフィールドがクリアされている場合は，SpC はそのイタレーションの投機実行を完了していないと分かるため，MemoTbl の検索を中止する．

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に，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り評価を行った．

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 基に，3 基の SpC 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[7] を参考とした．MemoTbl 内の InTbl に用いる CAM の構成は MOSAID 社の DC18288[8] を参考にし，サイズは 32Bytes 幅 ×4K 行の 128KBytes とした．また，提案手法で使用する小容量 CAM の構成は 32Bytes 幅 ×256 行の 8KBytes とし，それにしたがって AddrTbl，OutTbl の深さを 4k 行から CAM の深さと同じ 256 行とした．なお，プロセッサのクロック周波数は 128KBytes の CAM のクロック周波数の 10 倍と仮定して検索オーバヘッドを見積もっている．ここで，

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
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 サイクル数削減率 (SPEC CPU95)

	Mean	Max
(M_{128K})	9.6 %	40.5 %
(M_{8K})	4.4 %	24.9 %
(P_{8K})	8.3 %	39.8 %
(C_{8K})	10.6 %	40.6 %

CAM のアクセスレイテンシがその総容量に依存する [9] ことに着目すると，8KBytes の小容量 CAM のアクセスレイテンシは 128KBytes の CAM のアクセスレイテンシよりも小さいと考えられる．しかし本稿では，記憶できるエントリ数の違いによる性能を比較するために同じアクセスレイテンシであるとした．

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 7** に示す． 図では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おり，それぞれ左から順に

(N) メモ化を行わないモデル
(M_{128K}) CAM 容量 128KBytes の従来モデル
(M_{8K}) CAM 容量 8KBytes の従来モデル
(P_{8K}) (M_{8K}) にループイタレーションパージとループイタレーション登録フィルタを適用したモデル
(C_{8K}) (P_{8K}) にループイタレーション検索フィルタを適用したモデル

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

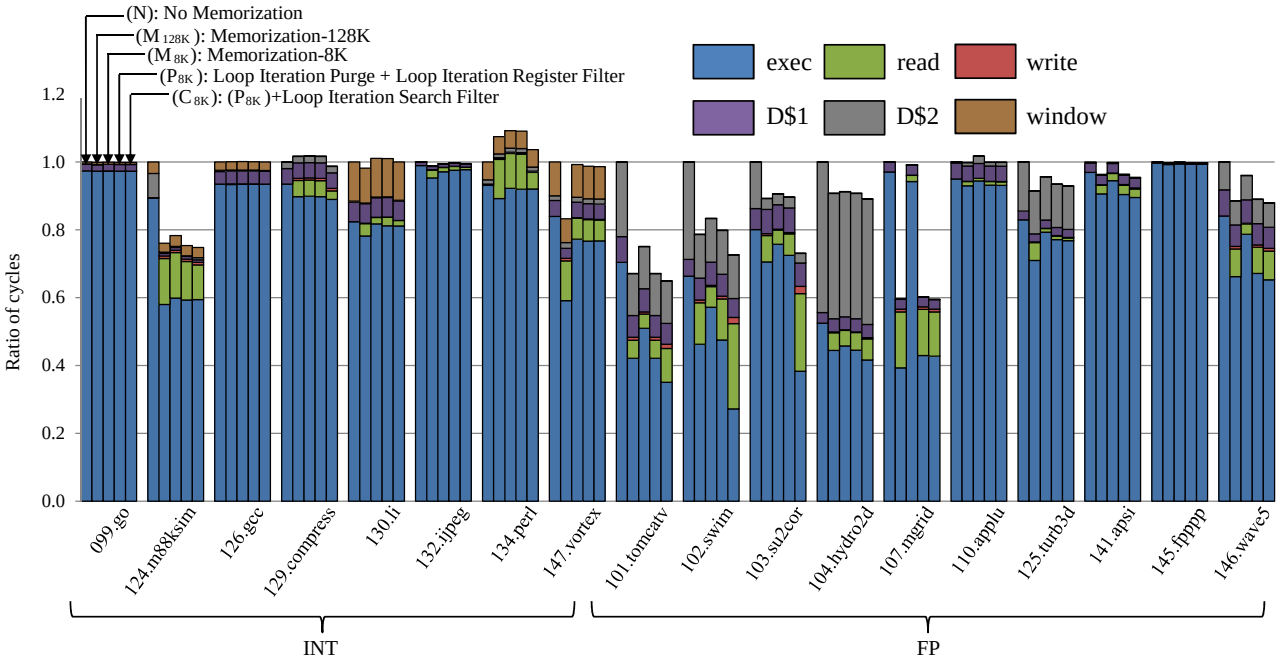

図 7 実行サイクル数

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 1 次と 2 次データキャッシュミスペナルティ，window はレジスタウインドウミスペナルティである．

まず，(M_{8K}) において，147.vortex，101.tomcatv，107.mgrid 等の多くのプログラムでは，性能低下が生じていることが分かる．これは，小容量 CAM を用いた場合，CAM に記憶できるエントリ数が減少し，再利用が見込める有益なエントリが追い出されやすくなることで，再利用を適用できる命令区間が少なくなってしまったためである．

一方 (P_{8K}) では，101.tomcatv，107.mgrid 等の多くのプログラムで，(M_{128K}) とほぼ同等の性能が得られていることが分かる．このことから，多くのプログラムでは，再利用が見込めないループエントリの追い出しやその登録を中止することで，再利用表の利用効率が向上することで，小容量 CAM を用いた場合の性能低下を抑制できることが分かった．

しかし，(P_{8K}) において，130.li，134.perl，147.vortex，125.turb3d の 4 つプログラムでは，性能低下をあまり抑制できていないことが分かる．まず，130.li と 147.vortex の 2 つのプログラムでは，登録されるエントリの大部分が関数エントリであったために，提案手法による効果が得られなかった．次に，134.perl，125.turb3d の 2 つのプログラムには，1 入力セットあたりのエントリ数が大きく，かつエントリが再利用されるまでの期間が長い関数があった．このような関数では，CAM が小容量しかない場合，たとえその関数エントリで CAM を占有した場合でも，再利用される前にそのエントリが TSID パージにより追い出されてしまう．そのため，ループイタレーションパージやループイタレーション登録フィルタにより再利用が見込めないループエントリを再利用表から排除しても，このような関数は再利用されず，性能低下を抑制できなかった．

次に，再利用表の利用効率を向上させるだけでなく，再利用が見込めないループエントリの検索を中止する機構を組み合わせた (C_{8K}) において，全てのプログラムで (P_{8K}) と同等か，もしくはそれ以上の性能が得られていることが分かる．中でも，124.m88ksim，129.compress，130.li，134.perl は read が削減されていることが分かる．これは，メインコアが実行するイタレーションのエントリが，SpC によって登録されている場合にのみ MemoTbl を検索することで，無駄な検索を減らすことができたためである．

また，129.compress，101.tomcatv，102.swim，103.su2cor，104.hydro2d，125.turb3d，141.apsi，146.wave5 では exec を削減できていることが分かる．この理由を考察するために，その削減幅が最も大きい 103.su2cor に対して各ループの再利用回数と再利用成功率の変化を調査した．この結果を**表 3** に示す．なお，ループの数は膨大であるため，ここでは実行サイクル数を削減している割合の高いループのみ示した．各ループについて，開始アドレス（addr），命令実行サイクル数（cycles），(P_{8K}) および (C_{8K}) における，その命令区間の再利用回数（reuse），および再利用成功率（rate）を示している．

この表より，各ループの再利用回数と再利用成功率が大幅に増加していることが確認できる．(P_{8K}) ではこれらのループにおいて，ある一定回数の再利用試行における再利用成功率が低く，2.3 節で述べたようにオーバヘッドフィ

表 3　ループの再利用回数とその成功率 (103.su2cor)

		(P_{8K})		(C_{8K})	
addr	cycles	reuse	rate	reuse	rate
0x1c094	573	0	0%	11022087	86.8%
0x1e3cc	151	10733895	76.2%	20849659	99.3%
0x1d6d4	186	4992035	75.4%	22983500	99.5%
0x1d0ec	196	4457037	75.6%	9506568	99.5%
0x1ce08	186	2587456	76.2%	9503552	99.5%

ルタによって再利用を中止させられてしまう期間があった．しかし (C_{8K}) では，ループイタレーション検索フィルタによって再利用される可能性のあるタイミングでのみ再利用試行がなされるため，その再利用成功率は高くなる．そのため，これらのループがオーバヘッドフィルタによって再利用を中止させられてしまうことがなくなり，再利用回数が増加した．この結果から，ループイタレーション検索フィルタにより，無駄な検索を減らすことで検索コストが削減されるだけでなく，ループでの再利用回数が増加することで，命令サイクル数も削減されることが分かった．

結果をまとめると，SPEC CPU95 では (N) に比べ (M_{128K}) では最大 40.5%，平均 9.6%，(M_{8K}) では最大 24.9 %，平均 4.4 %のサイクル数の削減だったのに対し，(C_{8K}) では，最大 40.6 %，平均 10.6 %となった．このことから提案手法により，小容量 CAM で高い性能を達成できることを確認した．

6. おわりに

本稿では，再利用表をより効率的に利用するために，一般にループのイタレータ変数が単調に変化することを利用して，再利用が見込めなくなったループエントリを再利用表から追い出す手法を提案した．また，再利用が見込めないループエントリの登録や検索を中止するための機構も併せて提案した．

SPEC CPU95 ベンチマークを用いて評価を行った結果，従来モデルにおいて，CAM の容量を 128KBytes とした場合には最大 40.5 %，平均 9.6 %，8KBytes とした場合には最大 24.9 %，平均 4.4 %であった実行サイクル削減率が，提案モデルでは 8KBytes で最大 40.6 %，平均 10.6 %となり，性能を低下させることなくより小容量の CAM で実装できることを確認した．

今後の課題としては，ループエントリだけでなく関数エントリについても適切な追い出しをする手法を検討し，さらなる再利用表の効率的な利用を目指すことが挙げられる．また，CAM の小容量化による自動メモ化プロセッサの具体的な消費エネルギーの変化を検証することも今後の課題である．

参考文献

[1] Feehrer, J., Jairath, S., Loewenstein, P., Sivaramakrishnan, R., Smentek, D., Turullols, S. and Vahidsafa, A.: The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets, *IEEE Micro*, Vol. 33, No. 2, pp. 48–57 (online), DOI: 10.1109/MM.2013.49 (2013).

[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] 島崎裕介，津邑公暁，中島　浩，松尾啓志，中島康彦：自動メモ化プロセッサにおける消費エネルギー制御，情報処理学会論文誌 コンピューティングシステム（ACS），Vol. 1, No. 2, pp. 1–11 (2008).

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

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

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

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