Vol.2014-ARC-211 No.13
2014/7/28

実行バイナリの静的解析による自動メモ化プロセッサの高速化

津村 高範[1]　柴田 裕貴[1]　神村 和敬[1,†1]　津邑 公暁[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，再利用表の検索コストを削減するために，値が変化しない入力に対する一致比較を省略する手法を提案する．また，再利用の成功が見込めないループに対して，再利用の適用を中止する手法も併せて提案する．これらの手法では実行バイナリを静的解析することで得られる再利用対象区間の特徴を利用する．これは，分岐命令などの影響により，提案手法に用いる特徴を実行時に充分解析できないという問題があるからである．SPEC CPU95 を用いてシミュレーションにより評価した結果，通常通り命令を実行する場合と比較し，従来手法では最大 40.6 %，平均 11.9 %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 51.8 %，平均 16.5 %のサイクル数を削減し，有効性を確認した．

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が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．

これまでに提案されてきた自動メモ化プロセッサの高速化手法 [5]，[6] は，いずれも再利用対象となる命令区間を動的に解析することで抽出できる特徴を利用したものであった．しかし，プログラムの実行中に命令区間から抽出できる特徴は，動的に変化し得る実行パスに依存している．そのため，動的解析では限られた特徴しか抽出できない．さらに，ある特徴を持ち，その特徴に基づいた高速化手法を適用できる再利用対象区間であっても，特徴の解析後でなければ適用できないなど，手法の効果を得られる期間が限定的となってしまう．

そこで本稿では，実行対象プログラムを静的解析し，得られた特徴を利用することで，ループを再利用する際の検索オーバヘッドを削減し，自動メモ化プロセッサを高速化する手法を提案する．静的解析の場合，実行対象プログラム内の全ての実行パスを解析できるため，動的解析よりも多くの特徴を抽出できる．また，事前に各再利用対象区間

[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
[†1] 現在，株式会社デンソー
Presently with DENSO CORPORATION

の特徴を抽出しておくことができるため，プログラムの実行開始時から，その特徴を利用した高速化手法を適用できる．

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，再び同じ入力セット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セット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）**[7] 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．その上，ソフトウェアにより入出力を記憶し，再利用を適用するメモ化 [8] はオーバヘッドが大きく，フィボナッチ数を求めるプログラムなどの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，プログラムの実行時に動的に関数およびループを再利用可能な命令区間として検出し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に実行できる．なお，自動メモ化プロセッサは call 命令のターゲットから return 命令までの区間を関数，後方分岐命令のターゲットからその後方分岐命令までの区間をループとして検出する．

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，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），命令区間やその入出力セットを記憶しておく表である再利用表 **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再利用対象区間に進入する際，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．これを**再利用テスト**と呼ぶ．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，その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すことで，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．

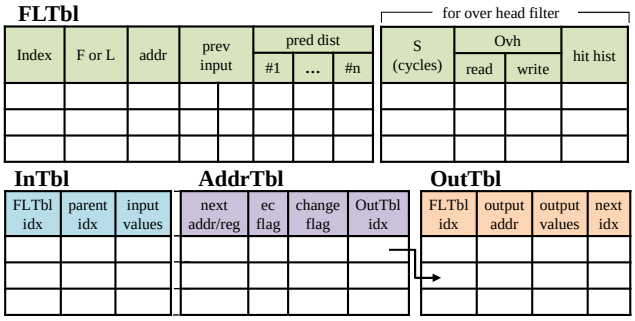

図 1　MemoTbl の構成

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，自動メモ化プロセッサは現在使用している MemoBuf の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．そのうち 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．なお，InTbl，AddrTbl，OutTbl をこれ以降まとめて入出力表と呼ぶ．

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），および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しているかを判別する．また，この入力値（input values）に加えて，命令区間の全入力パターンを木構造で管理するために親エントリのインデクス（parent idx）を持つ．なお，input values は 1 キャッシュブロック分の入出力を記憶し，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 は InTbl と同数のエントリを持ち，それらは InTbl の各エントリと 1 対 1 に対応している．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もしくはレジスタ番号（next addr/reg）を持つ．また，入力エントリの終端か否かを示すフラグ (ec flag) に加えて，next addr/reg が指すアドレスやレジスタに格納されている値に対し書き込みが発生したか否かを示すフラグ (change flag) 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），および出力値（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．

この MemoTbl から再利用機構がエントリを検索する際，まず現在の命令区間に対応する Index を FLTbl から検索する．次に InTbl を検索し，先ほど得られた Index と FLTbl idx が等しく，かつ input values が現在の入力と一致するエントリを取得する．そして，取得したエントリに対応する AddrTbl の next addr/reg が指すアドレスやレジスタに格納されている値を参照し，現在の入力値と一致比較する．これを入力エントリの終端を示す ec flag がセットされているエントリに達するまで繰り返す．しかし，AddrTbl の next addr/reg が指すアドレスやレジスタに格納されている値が一度も書き換えられていない場合，そのエントリに対する一致比較は本質的に不要である．そこで，自動メモ化プロセッサは書き込みが発生したか否かを示す change flag を確認することで，その不要な一致比較自体を省略する．

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，それぞれ固有の 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，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．

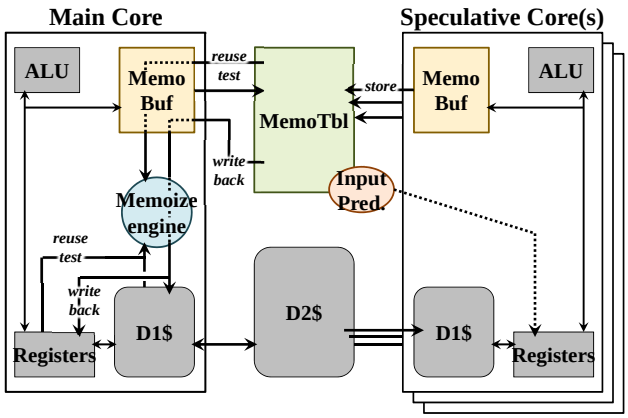

図 2 並列事前実行機構

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

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

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，

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

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，

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

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．

3. 実行バイナリの静的解析による高速化手法

1 章で述べたように，静的解析は動的解析に比べ，実行対象プログラムの特徴を多く抽出でき，さらにその特徴を利用した高速化手法を実行開始時から適用できる．本章では，この静的解析によって抽出した特徴に基づきループの再利用を支援する手法を提案する．

3.1 静的解析による命令区間の特徴抽出

一般的に，ループは連続して実行され，その入力の変化には特徴があると考えられる．この入力の変化の特徴はループの再利用に大きく影響する．例えば，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，入力値が単調に変化するという特徴を利用し，入力を予測して並列事前実行を適用することで，ループを再利用している．しかしループには，値が単調に変化する入力以外にも，値が全く変化しない入力や，値が不規則に変化する入力などがある．このような入力に対しても，既存の自動メモ化プロセッサは，単純に入力値が単調に変化すると仮定して，入力を予測しループに再利用を適用している．そのため，入力予測が成功しないループに対しても再利用テストを試行し続けてしまい，これに伴って発生する検索オーバヘッドは性能低下の要因となる．そこで，本稿では入力の特徴を利用してループの再利用を支援することを目指す．しかし，これらの入力の特徴は，条件分岐などで変化する実行パスごとに異なる可能性がある．そのため，動的解析の場合，ある実行パスに限った入力の特徴しか抽出できず，区間内全体で共通する入力の特徴を抽出することは困難である．一方で、静的解析の場合，ループ内で通り得る全ての実行パスの命令を一度に解析できるため，入力の変化の特徴を動的解析よりも正確に抽出できる．本稿では，ループ内で値が変化しない入力を持つループ，および値が不規則に変化する入力を持つループを検出し，それらのループの再利用を支援することで高速化を実現する．

3.2 ループ内で値が変化しない入力

2.1 節で述べたように，自動メモ化プロセッサは再利用テスト時に，次に参照するアドレスの指す値が一度も書き換えられていない場合，そのアドレスに対応する入力の一致比較を省略する．しかし，レジスタに対する一致比較は省略しない．なぜなら，メモリに比べてレジスタはその値が頻繁に書き換わることから，レジスタに書き込みが発生するたびに change flag を操作するとその処理量が膨大に

```
1 00020000<loop1>:
2   20000 add %o1, 5, %o1
3   20004 inc %l2
4   20008 cmp %l2, 10
5   2000c bne 20000 <loop1>
6   20010 add %o1, %o3 ,%o1
7     :
8 00030000<loop2>:
9   30000 add %o1, %l2, %o1
10  30004 smul %l2, %l2, %o3
11  30008 inc %l2
12  3000c cmp %l2, 10
13  30010 bne 30000 <loop2>
14  30014 add %o1, %o3 ,%o1
```

図 3 ループのアセンブリコード

なってしまうからである，しかし，ループ内で値が変化しないレジスタ値を入力とするループも存在する．例えば，**図 3** に示すアセンブリコードの 0x20000 番地の命令から 0x20010 番地の命令までのループ区間があったとする．なお，0x2000c 番地の後方分岐命令の直後の命令までをループ区間とみなす理由は，自動メモ化プロセッサのベースアーキテクチャである SPARC では遅延スロットが採用されており，後方分岐命令の直後の命令もループイタレーションごとに実行されるためである．このループを解析すると，0x20010 番地の命令からレジスタ%o3 が入力であることが分かる．また，このループ内の全ての命令で%o3 に対して値が書き込まれないため，%o3 はループ内で値が変化しない入力であることが分かる．汎用ベンチマークプログラムである SPEC CPU95 の各ベンチマークプログラムを解析した結果，各ベンチーマークのループには，このようにループ内で値が変化しないレジスタ入力が平均 2 個程度含まれることが分かった．

そこで，静的解析によりループ内で値が変化しないレジスタを検出し，そのレジスタの値を記憶しているエントリに対する一致比較を省略する手法を提案する．この手法により，再利用テスト時に要する MemoTbl の検索オーバヘッドを削減できる．また，オーバヘッドフィルタ機構により再利用表への登録および再利用の適用が中止されていたループでも，検索オーバヘッドが削減されることで，再利用が適用されるようになる可能性もある．

3.3 値が不規則に変化する入力

2.2 節で述べたように，自動メモ化プロセッサは入力値予測が正しかった場合，ループのように入力が変化する場合でも，SpC が登録したエントリにより実行を省略できる．しかし，値が不規則に変化するような入力値を持つループの場合，入力値予測に失敗するため，再利用に成功しない．

例えば，図 3 に示すアセンブリコードの 0x30000 番地の命令から 0x30014 番地の命令までのループ区間の入力である%o1 は，イタレーションごとに変化するレジスタ%l2 および%o3 の値が加算される．ストライド予測は単調に変化する入力に対してのみ予測が成功するため，イタレーションごとに値が不規則に変化するこのような入力に対する予測は失敗する．

そこで，値が不規則に変化するような入力値を持つループに対して再利用の適用を中止する手法を，3.2 節で述べた手法と併せて提案する．これにより，再利用の失敗によるオーバヘッドの増加を防ぐことができる．

4. 実行バイナリの静的解析による高速化手法の実装

3 章で述べた提案手法を適用した場合，処理の流れはソフトウェアにより実行バイナリを静的解析する静的ステージと，解析結果を利用し自動メモ化プロセッサを高速化する動的ステージに分けられる．本章では，提案手法を実現するための各ステージの実装について説明する．

4.1 バイナリを静的解析するソフトウェア

自動メモ化プロセッサにおけるループの再利用を支援するために，ソフトウェアにより実行バイナリを静的解析することで，ループ内で値が変化しない入力を持つループ，および値が不規則に変化する入力を持つループを検出する．

提案手法の対象となるループを検出するためには，まず実行バイナリ中の各ループの位置を特定する必要がある．そのため，実行バイナリの全ての命令を解析し後方分岐命令を検出する．これは 2.1 節で述べたように，自動メモ化プロセッサは後方分岐命令のターゲットからその後方分岐命令までの区間をループとして検出するからである．後方分岐命令を検出するためには，分岐命令であり，かつそのターゲットアドレスの値がこの分岐命令のアドレスの値より小さい命令を検出すればよい．そして，後方分岐命令を検出したらそのターゲットアドレスをループの開始アドレスとし，その後方分岐命令の直後の命令のアドレスをループの終了アドレスとする．これにより，実行バイナリ中に存在する全てのループの位置を特定することができる．

次に，再び実行バイナリの全ての命令を解析し，位置を特定した各ループの入出力の特徴を抽出する．なお，ループ内で値が変化しない入力レジスタを検出するためには，ループ内で値が読み出されるが書き込まれないレジスタを検出すればよい．一方，ループ内で値が不規則に変化する入力レジスタを検出するためには，ループ内で値が変化するレジスタを用いた演算の結果が書き込まれるレジスタを検出すればよい．しかし，値が変化するレジスタの値を用いた演算を検出することは困難であるため，本稿では，レジスタの値を用いた演算の結果が複数回書き込まれ，なおかつ入力となるレジスタをループ内で値が不規則に変化する入力レジスタとして検出する．

なお，SPARC-V8 では，1 つの関数内で使用できる整数レジスタは，グローバルレジスタ（GR），アウトプットレジスタ（OR），ローカルレジスタ（LR），インプットレジスタ（IR）がそれぞれ 8 個，浮動小数点レジスタは 32 個と決められている．そのため，各ループに対して，これら 64 個のレジスタの入出力情報を解析する．

次に，以上で述べた解析方法により，図 3 に示すループの入出力情報を解析する場合の具体的な動作を説明する．まず，後方分岐命令を検出し 0x20000 から 0x20010 までが loop1，0x30000 から 0x30014 までが loop2 であることが分かる．その後，再び実行バイナリを解析していき，0x20000 番地で loop1 を検出すると，各レジスタについて解析が開始される．このとき，0x20000 番地のオペコードを調べると add 命令であるとわかるため，続けてオペランドを調べ，入出力に用いられるレジスタを解析する．この add 命令の場合，%o1 を読み出して%o1 に書き込んでいるため，%o1 は入力，かつ出力であると判断できる．次に，0x20004 番地では%l2 を読み出して%l2 に書き込んでいるため，%l2 も入力，かつ出力であると判断できる．そして，0x20010 番地の命令では，%o3 を読み出しているため，%o3 は入力であると判断できる．さらに，この命令のアドレスとループの終了アドレスが一致するので，loop1 についての解析が終了する．この解析の結果，入力であり出力ではない%o3 は，ループ内で値が変化しない入力レジスタであることが分かる．なお，同様の手順で loop2 を解析すると，0x30000 番地で%o1 と%l2 を読み出して%o1 に書き込んでいるため，%o1 は入力，かつ出力であり，レジスタの値を用いた演算の結果が書き込まれたと判断できる．また，0x30014 番地で再び%o1 にレジスタの値%o3 を書き込んでいるため，%o1 はレジスタの値を用いた演算の結果が複数回書き込まれたと判断できる．以上の解析結果より，レジスタの値を用いた演算の結果が複数回書き込まれており，かつ入力である%o1 は，ループ内で不規則に変化する入力レジスタであると判断される．

ここで，解析情報を自動メモ化プロセッサが利用する方法として，解析情報ファイルを作成し，それを実行時に読み出す方法と，実行バイナリに解析情報を付加する方法の 2 通りが考えられる．前者の解析情報ファイルを作成する方法は，解析情報を格納するハードウェア領域が必要となる．一方で，後者の実行バイナリに解析情報を付加する方法は，解析情報を意味する特別な命令を実行バイナリに挿入する必要がある．本来なら，コンパイル時に計算再利用のための情報を埋め込む手法 [9],[10] が提案されているように，追加ハードウェアの不要な実行バイナリに解析情報を付加する手法がふさわしい．しかし，今回は実装が比較的容易である，解析情報ファイルを作成し，それを実行時

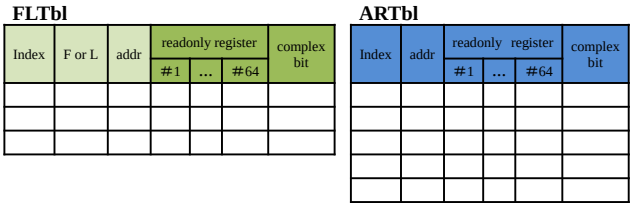

図 4　拡張後の MemoTbl

に読み出す方法によって静的解析情報を用いた手法の効果を確認する．

4.2　解析情報を利用し高速化するハードウェア

前節で述べた静的解析により得られた情報をプログラム実行時に利用するために，解析情報ファイルに出力された情報を記憶する専用表 **ARTbl** を MemoTbl に追加し，これに伴い FLTbl と MemoBuf を拡張した．拡張した MemoTbl を**図 4** に示す．ARTbl の各行は静的解析により検出した各ループに対応しており，ループを特定するためのループの終了アドレス (addr)，ループ内で値が変化しない入力レジスタ (readonly register)，およびループ内で値が不規則に変化する入力が存在するかどうかを示すビット (complex bit) を持つ．

また FLTbl には，ARTbl の readonly register と complex bit を保持するフィールドを追加する．これは，サイズが大きい ARTbl の情報をキャッシュしておくことで，MemoBuf が解析情報を容易に利用できるようにするためである．

そして，各コアの MemoBuf にも readonly register と complex bit を保持するフィールドを追加する．なお，これらのフィールドに保持される解析情報は，ループの各イタレーションの実行が開始される際に，当該ループに対応する FLTbl のエントリからコピーされる．これらのコピーされた解析情報は，入力セットを再利用表に登録する際に用いられる．

次に，以上で述べた追加ハードウェアを用いて，プログラムの実行を開始してから再利用表へ入出力を登録するまでの動作を説明する．まず，プログラムの実行が開始されると，命令の実行に並行して解析情報を ARTbl にコピーしていく．そして，命令区間を検出すると，FLTbl にその命令区間に対応するエントリを登録する．ここで，命令区間がループの場合，そのループに関する解析情報を ARTbl から FLTbl にコピーする．また，当該ループに対応するエントリを MemoBuf にも登録し，その後解析情報を FLTbl から MemoBuf にコピーする．このとき，complex bit が立っている場合，そのループに対して再利用を適用しないため，MemoBuf に入出力を登録しないようにする．その後，ループの実行が終わると，MemoBuf 内に記憶した当該ループのエントリを MemoTbl に登録する．このとき，入力となるレジスタの値が当該ループ内で変化する場合のみ，それらの AddrTbl エントリの change bit をセットしておく．これにより，再利用テスト時に change bit がセットされていない，つまりループ内で値が変化しない入力レジスタに対する一致比較を省略できる．

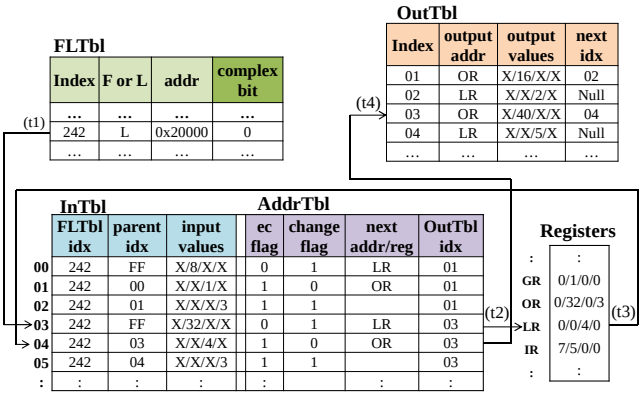

図 5　拡張後の MemoTbl 検索手順

なお，ループ内で値が変化しないと想定されているレジスタであっても，ループ外では値が変化する可能性がある．例えば，あるループでは値が変化しないが，そのループの外側のループで値が変化するような入力を含む多重ループを実行する場合が挙げられる．このようなループの場合，ループを抜けた後，再び同じループを実行する際，過去と入力が異なるにもかかわらず一致比較を省略してしまうことで，不正な再利用が適用されてしまう可能性がある．そこで，ループを抜けた際，FLTbl の readonly register を確認し，いずれかのレジスタに対応するビットがセットされている場合はそのループに関するエントリを入出力表から追い出すようにする．これにより不正な再利用が適用されてしまうことを防止する．

このようにして登録されたエントリに対してどのように検索操作が行われるのかを**図 5** を用いて説明する．なお，図 5 は図 3 に示したアセンブリコードの loop1 において，そのループのイタレータに対応するレジスタ %l2 の値が 3 の時に開始されるイタレーションまでをメインコアが実行し，%l2 の値が 4 の時に開始されるイタレーションを SpC が実行した状態を示している．また，図中の InTbl の input values における X はドントケアを表しており，そのアドレスに対応する値は検索時に比較されない．そして，図中の InTbl の parent idx における FF は親エントリが存在しないことを表している．まずメインコアが，%l2 の値が 4 の時に開始されるイタレーションの実行を開始しようとすると，FLTbl から loop1 の開始アドレスである 0x20000 を addr に持つエントリを検索する．このとき，該当するエントリに complex bit がセットされておらず，このループに対して再利用を適用するため，続けて当該エントリの Index と同じ FLTbl idx を持ち，かつ parent idx が FF であるようなルートエントリを InTbl から検索する．すると，該当するエントリがライン 03 で発見される (t1)．次に，

表 1　評価環境

MemoBuf	64 KBytes
MemoTbl CAM	128 KBytes
MemoTbl small CAM	8 KBytes
Comparison (register and CAM)	9 cycles/32 Bytes
Comparison (Cache and CAM)	10 cycles/32 Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle/32 Bytes
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

ライン 03 の change flag は 1 であるため，next addr/reg に対応する入力値に対して一致比較する必要がある．そのため，next addr/reg が指す LR のレジスタセットを参照する (t2)．そして，得られた値を input values として持ち，parent idx が 03 であるエントリを InTbl から検索する (t3)．ここで，ライン 04 の change flag が 0，つまり next addr/reg に対応する入力値の一致比較が省略できることが判明する．さらに，当該ラインの入力エントリが終端か否かを示す ec flag の値が 1 であるため，再利用テストに成功し，これ以上一致比較の必要がないことが判明する．このとき，当該ラインの OutTbl idx に登録されているインデクスが指す OutTbl エントリを参照し (t4)，エントリに登録された出力値を，レジスタやキャッシュに書き戻すことで，自動メモ化プロセッサは入力レジスタ%o3 に対する一致比較を省略した上で loop1 の%l2 の値が 4 の時に開始されるイタレーションの実行を省略することができる．これにより，再利用テストにかかるオーバヘッドを削減することができる．

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[11] を参考とした．MemoTbl 内の InTbl に用いる CAM の構成は MOSAID 社の DC18288[12] を参考にし，サイズは 32Bytes 幅 ×4K 行の 128KBytes 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．また今回は，静的解析情報を用いた手法の効果を確認するため，解析情報を利用する高速化手法を最大限に利用した場合の実行サイクル数を評価する．そこで，汎用ベンチマークプログラムである SPEC CPU95 のいずれのプログラムにおいても，そのプログラム中のループの解析情報を全て利用できるようにするために，ARTbl のサイズを 18K エントリとした．

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**図 6** に示す．図では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おり，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
(R) ループ内で値が変化しないレジスタに対する一致比較省略モデル
(C) (R) + 値が不規則に変化する入力を持つループに対する再利用の適用中止モデル

が要した総実行サイクル数を表しており，メモ化を行わないモデル (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 1 次および 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, (R) の結果より, 101.tomcatv, 102.swim, 103.su2cor などの多くのプログラムにおいて，(M) の結果に比べて性能が向上していることが分かる．中でも，124.m88ksim, 134.perl，107.mgrid，125.turb3d の 4 つのプログラムでは，read が削減されている．このことから，ループ内で値が変化しない入力レジスタに対する一致比較を省略することで，MemoTbl の検索にかかるオーバヘッドを期待通り抑えられることを確認できた．一方，129.compress，101.tomcatv，102.swim，103.su2cor，104.hydro2d，110.applu，141.apsi，146.wave5 の 8 つのプログラムでは，exec が削減されている．このことから，提案手法により MemoTbl の検索にかかるオーバヘッドが小さくなる事で，オーバヘッドフィルタ機構によって再利用の適用が中止されていたループに対して，再利用が適用できるようになり，命令サイクル数を

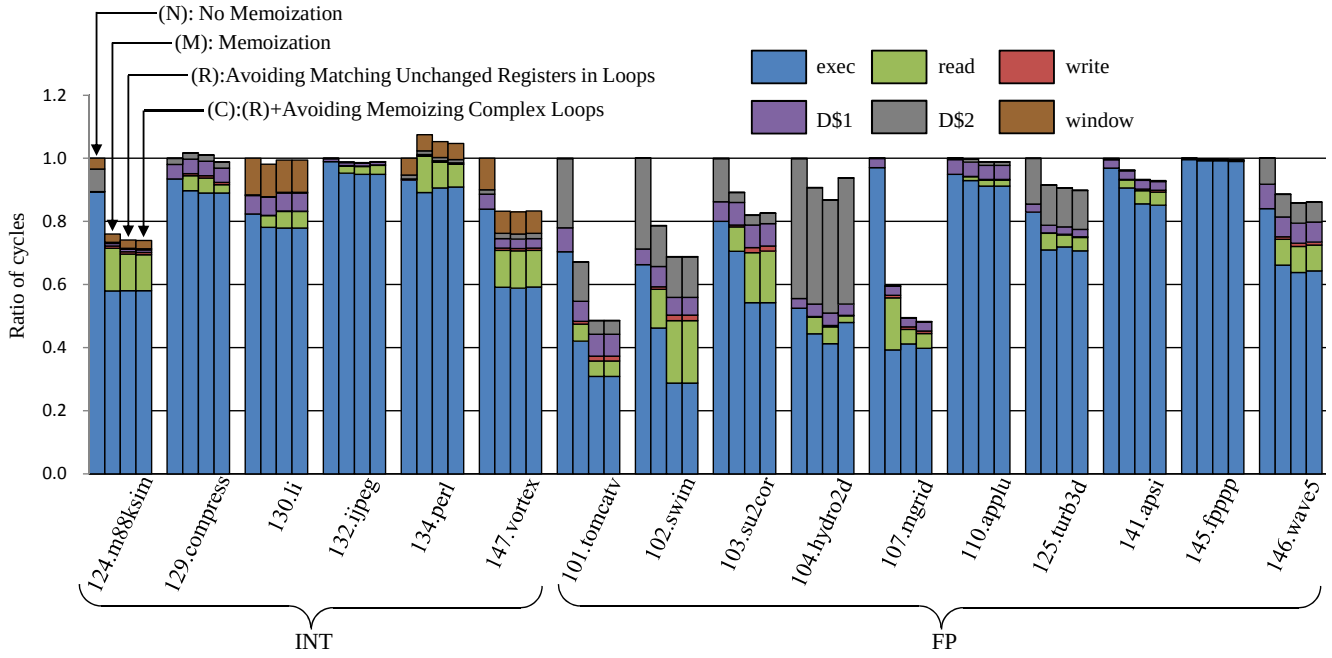

図 6　評価結果

削減することができた．しかし，それ以外のプログラムでは，性能向上があまり得られていない．特に，130.li は (M) に比べて read が増加してしまっている．これは，提案手法により 1 回の再利用テストにかかるオーバヘッドが削減されることで，オーバヘッドフィルタ機構によって再利用の適用が中止されなくなり，再利用テストを実施したものの，再利用には成功せず，再利用の失敗によるオーバヘッドが増加したためと考えられる．

次に，(C) の結果を見ると，129.compress，134.perl，107,mgrid，125.turb3d，141.apsi の 5 つのプログラムでは，(R) に比べ性能が向上していることが分かる．特に 129.compress では，exec を増加させることなく read を削減できている．このことから，値が不規則に変化する入力レジスタを持つループ対して再利用の適用を中止することで，再利用テストに成功する見込みのない無駄な検索を削減できることを確認できた．一方で，124.m88ksim，101.tomcatv，102.swim などのプログラムでは，(R) の結果に比べ性能があまり変化していないことが分かる．これは，これらのプログラムでは，値が不規則に変化する入力レジスタを持つループを検索してしまうような機会が，(R) においてあまり存在しなかったためである．さらに，104.hydro2d では，(R) の結果に比べると exec が増加し，性能が悪化してしまっている．この原因を調査したところ，ある特定のループにおいて，(M) や (R) ではストライド予測による入力予測が成功し再利用できていたものの，(C) ではこのループを値が不規則に変化するループと判断してしまったことにより，一度も再利用できていなかった．そこで，このループの構成命令を調べたところ，条件分岐を多く含んでいることが分かった．今回の解析では，ループ内の全ての命令から入出力の特徴を抽出した．つまり，ループ内の全ての命令を実行する場合は，ある入力値が不規則に変化することでストライド予測に失敗し，再利用の成功が見込めない．しかし，条件分岐などで一部の命令しか実行しない場合は，入力値が単調に変化しストライド予測に成功するループであったと考えられる．加えて，このループの分岐先が偏っており，ある実行パスの命令が多く実行され，なおかつその実行パスにおける入力に対してはストライド予測が成功するため，(M) や (R) では再利用に成功したと考えられる．しかし，(C) では再利用を適用しないため，(M) や (R) に比べ性能が悪化してしまった．

結果をまとめると，SPEC CPU95 では (N) に比べ，(M) では最大 40.6 %，平均 11.9 %のサイクル数の削減だったのに対し，(R) では最大 51.4 %，平均 16.7 %，(C) では最大 51.8 %，平均 16.5 %のサイクル数の削減に成功した．

6. おわりに

本稿では，プログラムの実行前にバイナリを静的解析することで，ループ内で値が変化しない入力を持つループと値が不規則に変化する入力を持つループを特定し，その情報を用いてループの再利用を支援することで，自動メモ化プロセッサを高速化する手法を提案した．提案手法により，ループ内で値が変化しないレジスタ入力の一致比較を省略することで，再利用テスト時に要する MemoTbl の検索コストを削減した．また，値が不規則に変化する入力を持つループに対して再利用の適用を中止することで，再利用が見込めないループに対する不要な検索を削減した．SPEC CPU95 ベンチマークを用いて評価を行った結果，従来モデルでは最大 40.6 %，平均 11.9 %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 51.8 %，平均 16.5 %のサイクル数を削減し，有効性を確認した．

今後の課題としては，値が不規則に変化する入力を持つループの解析精度を向上させるが挙げられる．これにより，値が不規則に変化する入力を持つループに対するさらなる性能向上が可能になる．また，解析情報を自動メモ化プロセッサが利用する方法として，解析情報ファイルを作成するのではなく，実行バイナリに付加する方法を実装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] 池谷友基，津邑公暁，松尾啓志，中島康彦：複数イタレーションの一括再利用による並列事前実行の高速化，情報処理学会論文誌 コンピューティングシステム（ACS），Vol. 3, No. 3, pp. 31–43 (2010).

[6] Shibata, Y., Kamimura, K., Tsumura, T., Matsuo, H. and Nakashima, Y.: CAM Size Reduction Method for Auto-Memoization Processor by considering Characteristics of Loops, *Proc. 1st Int'l Workshop on Computer Systems and Architectures (CSA'13)*, pp. 378–384 (2013).

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

[8] Huang, J. and Lilja, D. J.: Exploiting Basic Block Value Locality with Block Reuse, *Proc. 5th Int'l Symp. on High-Performance Computer Architecture (HPCA-5)*, pp. 106–114 (1999).

[9] Huang, J. and Lilja, D. J.: Exploring Sub-Block Value Reuse for Superscalar Processors, *Proc. 2000 Int'l Conf. on Parallel Architectures and Compilation Techniques(PACT'00)*, pp. 100–107 (2000).

[10] Connors, D. A., Hunter, H. C., Cheng, B. and Hwu, W. W.: Hardware Support for Dynamic Activation of Compiler-Directed Computation Reuse, *Proc. 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-9)*, pp. 222–233 (2000).

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

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