Vol.2014-ARC-212 No.11
2014/10/6

ARM型スーパスカラプロセッサにおける自動メモ化機構の実装と評価

柴田 裕貴[1]　津村 高範[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を提案している．自動メモ化プロセッサは，関数や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．我々はこれまで，この自動メモ化プロセッサのベースアーキテクチャとして，シングルイシューでパイプライン化されていない単純な SPARC アーキテクチャを採用し，良好な評価結果を確認してきた．しかし，現在市場に流通しているプロセッサにはこのような単純なアーキテクチャではなく，複雑なスーパスカラアーキテクチャが広く採用されているため，この評価結果が必ずしも実用性を保証できていない可能性がある．そこで本稿では，ARM 型スーパスカラプロセッサ上に自動メモ化機構を実装する場合に発生しうる問題を検討し，ARM 型自動メモ化プロセッサを設計した．その主要な設計上の工夫の 1 つとして，再利用を適用する際に発生するオーバヘッドによって起こるパイプラインストールを防ぐために，そのオーバヘッドと命令のパイプライン実行をオーバラップさせるようにした．SPEC CPU95 を用いてシミュレーションにより評価した結果，ARM 型自動メモ化プロセッサは既存の SPARC 型自動メモ化プロセッサとほぼ同等の性能向上を得られることが確認できた．

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単位の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][5] を提案している．自動メモ化プロセッサは，関数やループを計算再利用可能な命令区間とみなし，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．

我々はこれまで，自動メモ化プロセッサのベースアーキテクチャとして，シングルイシューでパイプライン化されていない単純な SPARC アーキテクチャを採用し，良好な評価結果を確認してきた．しかし，現在市場に流通しているプロセッサにはこのような単純なアーキテクチャではなく，複雑なスーパスカラアーキテクチャが広く採用されているため，この評価結果が必ずしも実用性を保証できていない可能性がある．そこで本稿ではまず，自動メモ化プロセッサのベースアーキテクチャとして，近年，モバイル用のプロセッサだけでなく，サーバ用のプロセッサにも用いられつつある ARM アーキテクチャに着目した．そして，その ARM アーキテクチャベースのスーパスカラプロセッサ上に自動メモ化機構を実装する場合に発生しうる問題を

[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

検討し，ARM 型自動メモ化プロセッサを設計，評価した．その主要な設計上の工夫の 1 つとして，再利用を適用する際に発生するオーバヘッドが原因で起こるパイプラインストールを防ぐために，そのオーバヘッドと命令のパイプライン実行をオーバラップさせるようにした．また，特定の命令を単純に監視するだけでは再利用対象区間である関数を特定することができないという問題を解決するために，関数呼び出しや関数復帰を監視するデコーダを実装した．さらに，命令区間のある入力がリタイアステージに至るまでに失われてしまうという問題を解決するために，パイプラインレジスタを拡張した．

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）**[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）**[4] を提案している．自動メモ化プロセッサは，実行時に動的に関数や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，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の概略図を**図 1** に示す．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（Memoization 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．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．

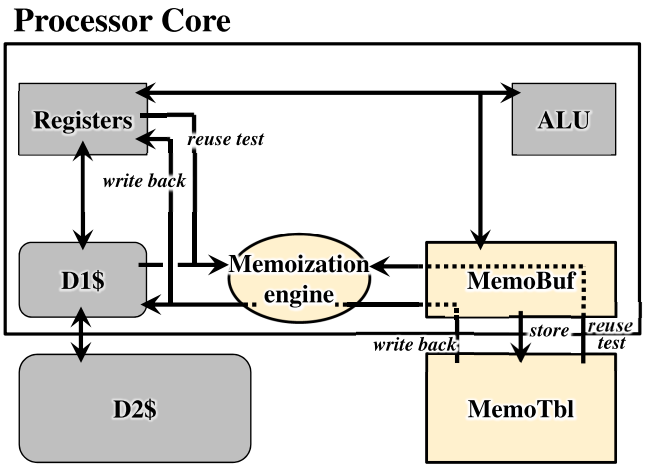


図 1 自動メモ化プロセッサの構成

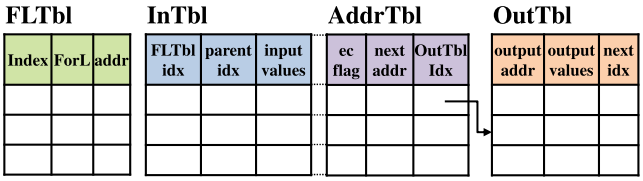
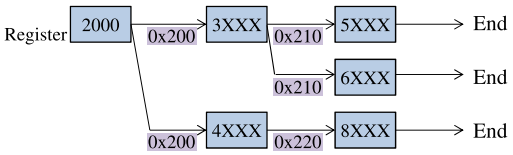


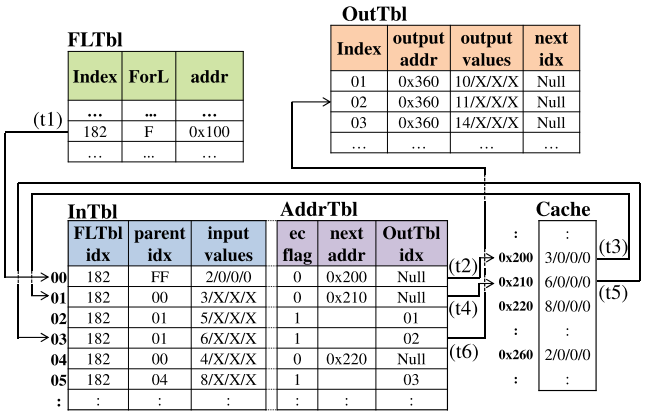
図 2 MemoTbl の構成

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 2** 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．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．



(a) 入力パターンの木構造



(b) MemoTbl の検索手順

図 3 入力パターンの木構造と MemoTbl の検索手順

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（ForL）と，命令区間の開始アドレス（addr）を用いる．

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 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（next addr）と入力エントリの終端か否か示すフラグ（ec flag）を持つ．また，入力エントリが終端である場合，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 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．

2.2 再利用機構の動作

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ことができる．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．

図 3(a) に，ある命令区間における入力セットの木構造の例を，**図 3(b)** にその入力セットを InTbl と AddrTbl へ実際に登録した時の格納例を示す．なお，図 3(a) のノードは命令区間の入力値に，エッジは次に参照するアドレスに，それぞれ対応する．また，入力値中の 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再利用テスト時に比較されない．なお，図 3(b) の InTbl の parent idx における FF は親エントリが存在しないこと，すなわち自身が入力パターンの木構造の根に相当するエントリであることを表している．

いま，命令区間の実行開始アドレスが検出されると，まず，FLTbl が当該命令区間の開始アドレスで検索される．これにより得られた FLTbl idx を持ち，input values が現在のレジスタ上の入力値と一致し，かつ parent idx が FF であるようなルートエントリが検索される (t1)．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 (t2)．そして，得られた値を input values として持ち，parent idx が 00 であるエントリを InTbl から検索する (t3)．以降，同様に検索を続ける (t4)(t5)．ここで，ライン 03 の ec flag が 1 であること，つまり入力セットの終端エントリに達したことが検出されるため，自動メモ化プロセッサは再利用テストに成功する．このとき，検索の終点となる AddrTbl エントリの 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 (t6)，読み出した出力値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ことができる．

3. スーパスカラ型自動メモ化プロセッサ

本章では，既存の SPARC 型自動メモ化プロセッサの問題点と ARM 型自動メモ化プロセッサのパイプライン実行の様子について述べる．

3.1 SPARC 型自動メモ化プロセッサの問題点

我々はこれまで，自動メモ化プロセッサのベースアーキテクチャとしてシングルイシューで，かつパイプライン化されていない SPARC アーキテクチャ [7] を採用し，良好な評価結果を確認してきた．しかし，この SPARC 型自動メモ化プロセッサには 2 つの問題点が存在する．

一つは，これまで確認されてきた自動メモ化プロセッサの評価結果が必ずしも実用性を保証できていない可能性が

ある点である．これは，現在市場に流通しているプロセッサには，SPARC 型自動メモ化プロセッサが採用している，シングルイシューで，かつパイプライン化されていない単純なアーキテクチャではなく，複雑なスーパスカラアーキテクチャが広く採用されているからである．そのため，市場のプロセッサ事情に即した，より実用的な自動メモ化プロセッサの性能を評価するために，スーパスカラプロセッサに自動メモ化機構を実装し，その性能を評価する必要がある．

もう一つは，SPARC 型自動メモ化プロセッサが，SPARC アーキテクチャに強く依存した実装になっているという点である．例えば，SPARC の命令セットや SPARC ABI，レジスタウィンドウの存在は，命令区間の検出など，メモ化を行う上で非常に都合が良い．つまり，SPARC 以外のアーキテクチャで，SPARC 型自動メモ化プロセッサと同様のハードウェアを用いたメモ化を行うことが可能かどうかは確認できていない．そこで，我々は自動メモ化プロセッサのベースアーキテクチャとして ARM アーキテクチャ [8] を採用し，ARM 型スーパスカラプロセッサ上に自動メモ化機構を実装することを試みた．ARM は，SPARC に比べ複雑な命令を多数持ち，近年，モバイル用のプロセッサだけでなくサーバ用のプロセッサにも用いられつつある [9]．そのため，もし ARM 型自動メモ化プロセッサが実装できれば，自動メモ化プロセッサの実用性を確認することができる．

3.2 ARM 型自動メモ化プロセッサのパイプライン実行

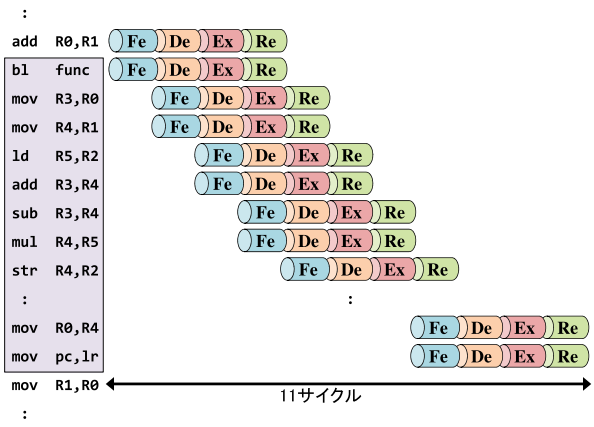
図 4 に，ARM 型自動メモ化プロセッサのパイプライン実行の様子を示す．以下，この ARM 型自動メモ化プロセッサの実行の様子を，メモ化を行わない場合，再利用テスト成功時，再利用テスト失敗時の 3 つに分けて説明する．

3.2.1 メモ化を行わない場合のパイプライン実行

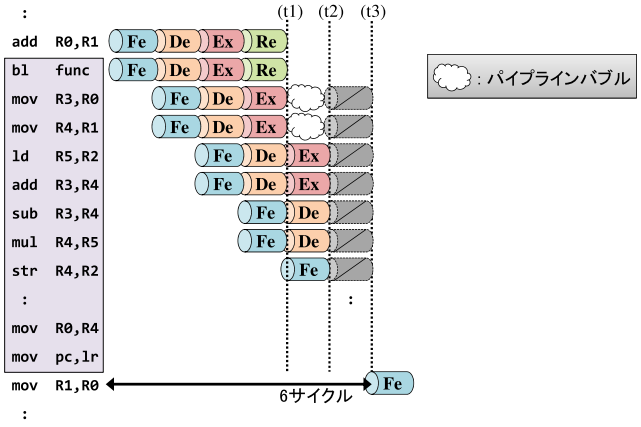
まず，メモ化を行わない場合の実行の様子を図 4(a) に示す．なお，説明を簡単化するために，ARM 型自動メモ化プロセッサのパイプラインは 2way で，各パイプラインは Fe (フェッチ)，De (デコード)，Ex (命令実行)，Re (リタイア) の 4 段構成をとり，各ステージは 1 サイクルで処理されるものとする．また，キャッシュミス等によりパイプラインがストールすることなく，理想的な状態で各ステージでの処理が実行されるものとする．この図において，`bl func` は関数呼び出しのための命令，`mov pc,lr` は関数復帰のための命令であり，`bl func` から `mov pc,lr` までの区間が関数である．この図では，ARM 型自動メモ化プロセッサはこの関数をストールすることなく実行しており，その実行には 11 サイクルを要している．

3.2.2 再利用テスト成功時のパイプライン実行

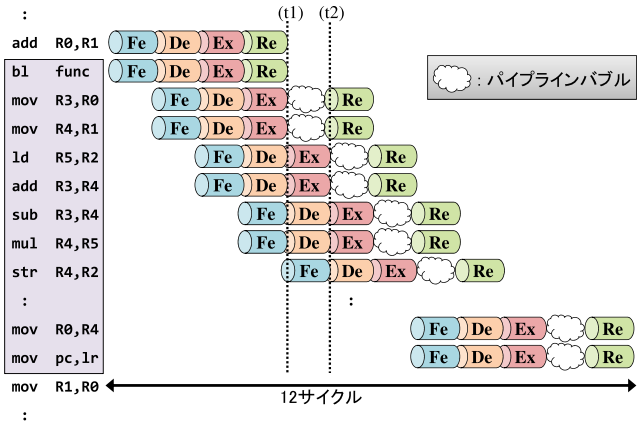
次に，再利用テスト成功時の実行の様子を図 4(b) に示す．ARM 型自動メモ化プロセッサは関数呼び出しのため



(a) メモ化を行わない場合のパイプライン実行の様子



(b) 再利用テスト成功時のパイプライン実行の様子



(c) 再利用テスト失敗時のパイプライン実行の様子

図 4 パイプライン実行の様子

の命令を検出すると，再利用を適用できるかどうかを確かめるために再利用テストを行う．この再利用テストを正しく行うためには，関数呼び出しのための命令より前の命令によるレジスタやキャッシュへの書き込みが完了している必要がある．ARM 型自動メモ化プロセッサでは，関数呼び出しのための命令がリタイアされていれば，それより以前の命令が終了しており，関数の入力となる値がレジスタやキャッシュに存在していることが保証される．そのため，再利用テストは関数呼び出しのための命令がリタイアされ

たタイミングで行う (t1)．なお，再利用テストを行っている間はパイプラインをストールさせず，パイプライン中の命令をこの再利用テストとオーバラップして実行させることによって，再利用テストのオーバヘッドを隠蔽する．ただし，再利用テスト中のパイプライン実行では，リタイアステージでの処理を禁止している．これは，関数内の命令がリタイアされてしまうと，一致比較対象の入力値が上書きされてしまうことが起こりうるからである．このように一致比較対象の入力値が上書きされてしまうと，過去と入力が異なるにも関わらず一致比較に成功してしまい，不正な再利用が適用されてしまう可能性がある．

その後，この再利用テストに成功し，計算再利用を適用できることが判明すると，過去の実行結果をレジスタやキャッシュに書き戻すと同時に，パイプラインをフラッシュする (t2)．これは，既にパイプラインに投入されている命令は再利用対象区間である関数内の命令であり，無効化する必要があるためである．このとき，このパイプラインフラッシュによって，`mov R3,R0` という命令のフェッチ時から，パイプラインにはバブルが毎サイクル発生していたことになる．このパイプラインバブルには，計算再利用による性能向上を打ち消してしまう可能性がある．この問題の解決案については 5.2 節で考察する．

その後，関数復帰先の命令（`mov R1,R0`）をフェッチすることで，実行を継続する (t4)．再利用テストの成功により，図 4(b) の場合では，関数の実行に要したのは 6 サイクルのみであり，図 4(a) のメモ化を行わない場合と比べて，5 サイクル高速化できている．

3.2.3 再利用テスト失敗時のパイプライン実行

最後に，再利用テスト失敗時の実行の様子を**図 4(c)** に示す．再利用テスト成功時の場合と同様に，関数呼び出しのための命令がリタイアされたタイミングで再利用テストを行う (t1)．また同様に，再利用テストを行っている間，パイプライン中の命令はリタイアステージでの処理が禁止された状態で再利用テストとオーバラップして実行される．その後，この再利用テストに失敗し，計算再利用を適用できないことが判明すると，禁止していたリタイアステージでの処理が再開され，通常通り命令が実行される．この図の場合，関数の実行には 12 サイクル要しており，図 4(a) のメモ化を行わない場合と比べて，1 サイクル増加している．このサイクル数の増加は，再利用テストの間，リタイアステージでの処理が禁止されることにより，パイプライン中にバブルが発生してしまうことが原因である．

4. 実装

本章では，ARM 型自動メモ化プロセッサのベースアーキテクチャについて説明し，また，新たに必要となるハードウェアの拡張についても説明する．

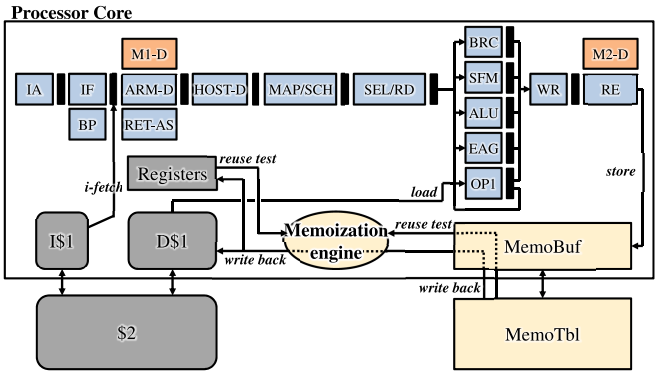


図 5　ARM 型自動メモ化プロセッサの構成

4.1 ベースアーキテクチャ

図 5 に ARM 型自動メモ化プロセッサのハードウェア構成を示す．ベースアーキテクチャには OROCHI[10] を採用した．OROCHI は VLIW 命令と ARM 命令を同時実行可能なスーパスカラプロセッサである．なお，自動メモ化機構を実装する上で，VLIW 命令用のユニット群は必要ないため無効化している．各パイプラインステージは以下のとおりである．

IA (Instruction Address)
次に実行すべき命令のアドレスを計算する．

IF (Instruction Fetch)
命令キャッシュから連続した複数命令をフェッチする．また，*g-share* による分岐予測も行う．

ARM-D (Arm-Instruction Decode)
ARM 命令を高速実行可能な内部命令に分解する．また，後述する関数検出デコーダ（M1-D）による，関数呼び出し，および関数復帰の検出も行う．

HOST-D (Host-Instruction Decode)
条件実行命令を実行ステージで処理できるように分解する．なお，条件実行命令とは，命令の実行に条件を付加する ARM 独自の命令である．

MAP/SCH (Register mapping/Schedule)
各命令のオペランドを，論理レジスタから，命令ウィンドウと物理レジスタを兼ねるリオーダバッファへマッピングする．また，同時に各命令のスケジューリングも行う．

SEL/RD (Select and Read)
命令が発行可能かどうかを調べ，依存関係のない命令を発行する．

IE (Instruction Execution)
ベースアーキテクチャは 5 並列の実行ステージを持つ．

BRC　分岐の taken/untaken を判定する．

SFM　シフト演算を処理する．また，積和補助演算も処理する．

ALU　加減算命令を処理する．

EAG　アドレス計算を処理する．また，積和補助演

算も処理する．

OP1　ロード・ストア命令を処理する．

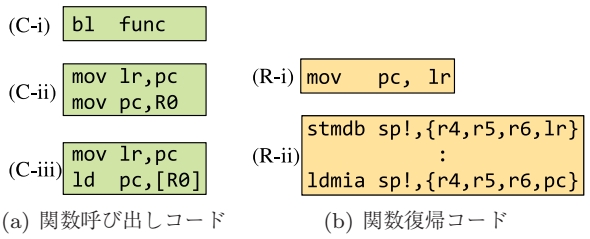
WR (WriteBack)

各実行ステージで処理された命令をリオーダバッファに書き込む．

RE (Retire)

先行命令が全て完了した命令をリオーダバッファから論理レジスタへ書き戻す．なお，分岐予測ミスなどの際に速やかに実行を再開するために，命令がコミットされる順番はプログラムの命令列と同じであることが保証されている．また，ARM-D ステージと同様に，後述する関数検出デコーダ（M2-D）による関数復帰の検出も行う．

上記の ARM-D ステージで述べたように，OROCHI は命令分解機構を取り入れており，ARM 命令を簡単な処理を行う内部命令へと変換，分解し，パイプラインの各ステージの処理量が均等になるようにしている．これにより，命令の効率的なパイプライン実行を可能としている．我々は，ARM 型自動メモ化プロセッサの汎用性が失われないようにするために，OROCHI 独自のこの命令分解機構に依存することなく，自動メモ化機構を実装した．



(a) 関数呼び出しコード　(b) 関数復帰コード

図 6　ARM バイナリにおける関数呼び出しコードと関数復帰コード

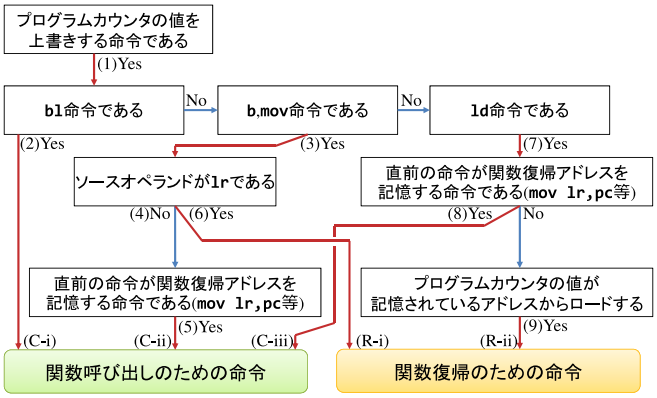


図 7　関数呼び出し，および関数復帰の検出フロー

4.2　メモ化のためのハードウェア拡張

本節では，ARM 型自動メモ化プロセッサを設計する上で必要なハードウェア拡張について説明する．

4.2.1　関数検出デコーダ

再利用対象区間である関数を特定するためには，関数呼び出し，および関数復帰を検知する必要がある．SPARC の ABI では関数呼び出しのための命令は `call`，関数復帰のための命令は `ret` と規定されている．そのため，既存の SPARC 型自動メモ化プロセッサは，これら二つの命令を監視することにより，関数を特定している．一方で，ARM の ABI では関数呼び出し，および関数復帰に特定の命令を使用すべきであるといった規定はない．ここで，**図 6** に，ARM バイナリにおける関数呼び出し，および関数復帰コードを示す．このように，ARM の関数呼び出し，および関数復帰コードは多岐にわたるため，SPARC 型自動メモ化プロセッサと同様に特定の命令を単純に監視するだけでは，関数を特定することはできない．そこで図 5 に示すように，関数呼び出し，および関数復帰を検出するための，関数検出デコーダ（M1-D，M2-D）をデコードステージとリタイアステージに追加する．

図 7 にこれらデコーダの関数呼び出し，および関数復帰検出フローを示す．まず，M1-D はプログラムカウンタの値を上書きする命令を監視する (1)．検知したプログラムカウンタ上書き命令の種類によって，その後の関数呼び出し，および関数復帰検出フローが 3 つのパターンに分けられる．まず，その検知した命令が `bl` 命令である場合 (2)，ARM 型自動メモ化プロセッサはその命令を関数呼び出しのための命令として検出する (C-i)．次に，検知した命令が `b` 命令，または `mov` 命令である場合 (3)，その命令のソースオペランドを調べる．ソースオペランドが関数復帰アドレスを保持するリンクレジスタ（`lr`）以外であるならば (4)，次は直前の命令を調べる．その直前の命令が関数復帰アドレスを退避する命令（`mov lr, pc` 等）である場合 (5)，検知した命令を関数呼び出しのための命令として検出する (C-ii)．一方で，ソースオペランドが関数復帰アドレスを保持するリンクレジスタであるならば (6)，検知した命令を関数復帰のための命令として検出する (R-i)．最後に，検知した命令が `ld` 命令である場合 (7)，直前の命令を調べる．その直前の命令が関数復帰アドレスを退避する命令であるならば (8)，検知した命令を関数呼び出しのための命令として検出する (C-iii)．なお，関数プロローグにおいてスタックに退避された関数復帰アドレスが，関数エピローグにおいてロード命令により直接プログラムカウンタに上書きされ，関数復帰が行われる場合がある．このような場合にもそのロード命令を関数復帰のための命令として検出するために，M2-D は関数プロローグでプログラムカウンタの値が退避されたアドレスを記憶する．そして，そのアドレスから値を読み出すロード命令を検知したとき (9)，そのロード命令を関数復帰のための命令として検出する (R-ii)．以上のように関数呼び出し，および関数復帰のための命令を検出することで，再利用対象区間である関数を特定することができる．

4.2.2 パイプラインレジスタの拡張

2 章で述べたように，自動メモ化プロセッサは計算再利用を関数に適用するために，MemoBuf や MemoTbl に各関数の入出力を登録する必要がある．ここで，関数の入出力とは，関数内で読み書きされた値とアドレスの組のことである．ARM 型自動メモ化プロセッサでは，関数の入出力を登録するタイミングはリタイアステージを想定している．これは，リタイアステージよりも前のステージで関数の入出力を登録しようとすると，実行された命令が分岐予測ミスなどによりフラッシュされた場合に，フラッシュされた命令に対応する MemoBuf のエントリを無効化する必要があり，その無効化のための操作が大きなオーバヘッドとなるからである．しかし，リタイアステージで関数の入出力を登録する場合にも問題はある．それは，関数の入力となるロード命令のソースオペランドの値がリタイアステージよりも前で失われてしまうために，リタイアステージで関数の全ての入力を MemoBuf に登録することができないという問題である．

ここで，関数の入力となるロード命令のソースオペランドの値がリタイアステージよりも前で失われてしまう例を，`ld R0, [R1]` という命令が実行される場合で示し，その解決方法を説明する．`ld R0, [R1]` というロード命令が実行される場合，関数の入力となるものは `R0` にロードされる値と `R1` が指しているアドレスである．さて，この命令が実行ステージの OP1 ユニットで処理されるとき，論理レジスタ `R0` に対応する物理レジスタに対し `R1` が指しているアドレスに格納されている値がロードされる．その後，この命令がリタイアされるとき，その物理レジスタの値が論理レジスタ `R0` に書き込まれる．このとき，このロード命令のリタイア処理には，ソースオペランドである `R1` に関する情報は必要ないため，リオーダバッファからその情報が失われてしまっている．そのため，このままでは `R1` が指しているアドレスを入力として MemoBuf に登録することができない．そこで，入力となるロード命令のソースオペランドの値をリタイアステージまで伝播させるために，実行ステージ以降のパイプラインレジスタでも，ソースオペランドを保持し，次段に渡すように，パイプラインレジスタを拡張する．これにより，実行ステージ以降で失われてしまうソースオペランドの値をこのパイプラインレジスタを通して後段まで伝播させることで，関数の入出力をリタイアステージで MemoBuf に登録できるようにする．

5. 評価

以上で述べた ARM 型自動メモ化プロセッサのための実装を ARM 型スーパスカラプロセッサのシミュレータに実装した．また，ベンチマークプログラムを用いてその性能を評価した．

表 1　シミュレータ諸元

MemoBuf	64 KBytes
MemoTbl CAM	256 KBytes
Comparison (register and CAM)	1 cycle / 64Bytes
Comparison (Cache and CAM)	2 cycles / 64Bytes
Writeback (MemoTbl to Reg. / Cache)	1 cycle / 64Bytes
L1 I-cache	16 KBytes
line size	64 Bytes
ways	4 ways
miss penalty	8 cycles
L1 D-cache	32 KBytes
line size	64 Bytes
ways	4 ways
miss penalty	8 cycles
L2 cache	2 MBytes
line size	64 Bytes
ways	4 ways
miss penalty	40 cycles
pipeline stage	
IA (Instruction Address)	1 insn / cycle
IF (Instruction Fetch)	2 insns / cycle
ARM-D (ARM-Instruction Decode)	4 μop. / cycle
HOST-D (Host-Instruction Decode)	4 μop. / cycle
MAP/SCH (Register Mapping/Schedule)	4 μop. / cycle
SEL/RD (Select and Read)	4 μop. / cycle
IE (Instruction Execution)	1 μop. / cycle
WR (Writeback)	1 μop. / cycle
RE (Retire)	4 μop. / cycle
Reorder Buffer	32 entries

5.1 評価環境

ARM 型自動メモ化プロセッサの評価に用いたシミュレータの仕様を**表 1** に示す．MemoTbl 内の InTbl に用いる CAM の構成は MOSAID 社の DC18288[11] を参考にし，サイズは 64Bytes 幅 ×4K 行の 256KBytes とした．なお，ループに計算再利用を適用するための機構が未実装であるため，本稿では関数を対象とした計算再利用のみで評価した．

5.2 評価結果

ARM 型自動メモ化プロセッサが既存の SPARC 型自動メモ化プロセッサの評価結果が示す通りのパフォーマンスを得られるかどうかについて検証するために，以下の 2 つのプロセッサを評価した．

(N)　メモ化を行わないプロセッサ（baseline）
(A)　ARM 型自動メモ化プロセッサ

ここで，メモ化を行わないプロセッサ (N) とは，自動メモ化機構を実装していない ARM 型スーパスカラプロセッサのことである．ベンチマークプログラムには，SPEC CPU INT（train）のプログラムを gcc-4.1.1（-O2 -msoft-float -march=armv4）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．

表 2　実行命令数の削減率

099. go	124. m88ksim	129. compress	130. li	132. ijpeg	134. perl	147. vortex
6.4%	29.0%	1.7%	10.8%	1.6%	7.5%	39.2%

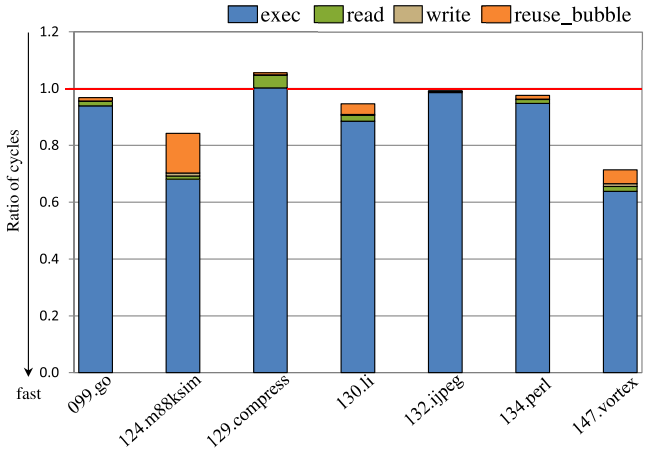


図 8　実行サイクル数比（SPEC CPU95 INT）

まず，ARM 型自動メモ化プロセッサ (A) における実行命令数の削減率を**表 2** に示す．この結果から，計算再利用により，関数の実行を省略できたことによって，多くのベンチマークプログラムで，実行命令数が削減されていることが分かる．プログラム全体では，平均で 13.7%となり，実行サイクル数の削減や，IPC の向上が期待できる．

次に，実行サイクル数を評価した．その結果を，**図 8** に示す．この図は，各ベンチマークプログラムの結果をグラフで示しており，ARM 型自動メモ化プロセッサ (A) が要したサイクル数を表している．なお，メモ化を行わないプロセッサ (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 は命令の実行に要したサイクル数，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），reuse_bubble は 3.2.2 項で述べたように再利用テストの成功によって発生するパイプラインバブルによるペナルティである．なお，exec は，1 次，2 次キャッシュミスペナルティ，および再利用テストを開始するためにキャッシュコントローラがビジー状態からアイドル状態に遷移するまで待機していたサイクル数も含んでいる．

評価の結果をまとめると，ARM 型自動メモ化プロセッサ (A) ではサイクル数削減率が，平均 7.9%，最大 28.6%となった．この削減率は，SPARC 型自動メモ化プロセッサとほとんど同じである．つまり，自動メモ化プロセッサはベースアーキテクチャをスーパスカラプロセッサとした場合でも，計算再利用による性能向上を十分に得られることがわかった．

各ベンチマークの結果について見てみると，124.m88ksim

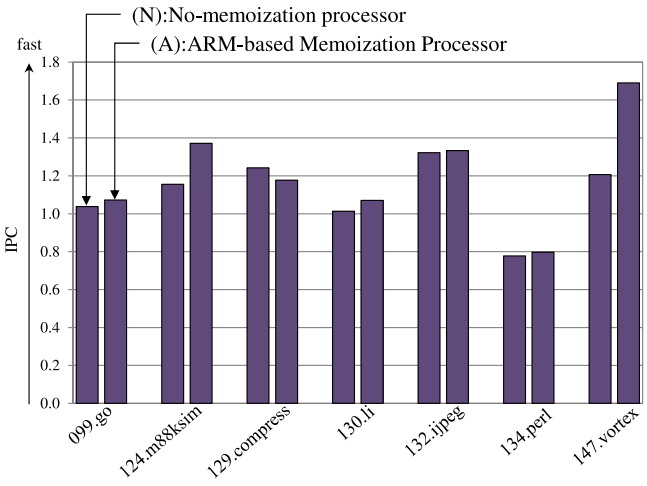


図 9　IPC (SPEC CPU95 INT)

において，reuse_bubble がサイクル数の約 20%も占めており，計算再利用による性能向上を計算再利用適用時に発生するパイプラインバブルによって打ち消してしまっていることがわかる．そこで，この問題を解決する手法について考察する．

ARM 型自動メモ化プロセッサでは，再利用テストの間，常に関数内の命令を実行する．これにより，再利用テストに失敗した場合は，当該関数の通常実行にすぐに復帰することができる．しかし，再利用テストに成功した場合は，再利用テストとオーバラップして実行していた関数内の命令をパイプラインからフラッシュしなければならず，再利用テストの間に実行した命令は全て無駄になってしまう．そこで，分岐予測器のように，再利用テストの成功，失敗を予測する機構を実装し，成功すると予測したときは，再利用テストの間，関数の復帰先命令以降を実行させるようにする方法が考えられる．なお，関数呼び出しのための命令は ARM-D ステージで検出できるため，再利用テストの成功，失敗をこの時点で予測するようにする．そして，実際に再利用テストに成功した場合は，パイプラインをフラッシュせずに，命令の実行を継続することができる．これにより，再利用テストの予測に成功した場合は，パイプラインバブルによる性能低下を抑制することができると考えられる．

最後に，IPC を評価した．その結果を，**図 9** に示す．図では各ベンチマークプログラムの結果を 2 本のグラフで示しており，左はメモ化を行わないプロセッサ (N) の IPC を，右は ARM 型自動メモ化プロセッサ (A) の IPC を示している．なお，多数の内部命令に分解される複雑な命令と，単純な加算命令のように分解されない命令を同じ 1 命令として評価することはできないため，分解後の内部命令の IPC を評価した．また，自動メモ化プロセッサは命令の実行自体を省略しつつ，メモ化を行わない場合と同じ結果を出力するため，実際に実行された命令数で IPC を算出し

てしまうと，計算再利用による性能向上が正しく評価できない．そこで本稿では，実行が省略された命令も含め，本来実行すべきであった命令数を用いて，IPC を算出した．この図より，124.m88ksim や 147.vortex などの多くのプログラムでは，計算再利用により IPC が向上していることが分かる．しかし，129.compress では IPC が低下してしまっている．これは，129.compress では，表 2 や図 8 を見て分かるように，計算再利用を適用できた関数がほとんど存在しておらず，検索オーバヘッドのみが増加してしまったためである．そのため今後は，計算再利用を適用しても効果の得られない命令区間に対するメモ化の適用を中止する機構を実装していく必要がある．なお，この機構は既に SPARC 型自動メモ化プロセッサには実装済みであり，我々はこの機構が検索オーバヘッドの増加による性能低下を抑制できることを確認している [5].

以上の 3 つの評価結果より，ARM 型自動メモ化プロセッサは既存の SPARC 型自動メモ化プロセッサと同様に計算再利用による性能向上を十分に得られることが確認できた．

6. おわりに

本稿では，自動メモ化プロセッサの実用的なパフォーマンスを評価するために，ARM 型スーパスカラプロセッサに自動メモ化機構を実装した．シミュレーションによる評価の結果，ARM 型自動メモ化プロセッサは既存の SPARC 型自動メモ化プロセッサと同様に計算再利用による性能向上を十分に得られることが確認できた．また，ARM 型スーパスカラプロセッサに自動メモ化機構を搭載する上で必要なハードウェアの拡張に関する知見を得ることができた．

今後の課題は，計算再利用適用時に発生するパイプラインバブルによる性能低下を抑制する手法を実装することである．また，ループに計算再利用を適用するための機構を実装することや，計算再利用を適用しても効果の得られない命令区間に対してメモ化の適用を中止する機構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] Kamimura, K., Oda, R., Yamada, T., Tsumura, T., Matsuo, H. and Nakashima, Y.: A Speed-Up Technique for an Auto-Memoization Processor by Reusing Partial Results of Instruction Regions, *Proc. 3rd Int'l. Conf. on Networking and Computing (ICNC'12)*, pp. 49–57 (2012).

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

[7] Sun Microsystems: *UltraSPARC III Cu User's Manual* (2002).

[8] ARM Limited: *"ARM Architecture Reference Manual"ARM DDI 0100E* (2000).

[9] AMD Corporation: *AMD Opteron A1100 Processor* (2014).

[10] Shimada, H., Shimada, T., Tabata, T., Kitamura, T., Kojima, T., Nakashima, Y. and Kise, K.: Outline of OROCHI: A Multiple Instruction Set Executable SMT Processor, *Proc. 2007 Innovative architecture for future generation high-performance processors and systems (IWIA'2007)*, pp. 110–117 (2007).

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