修士論文

スーパスカラ型ARMをベースアーキテクチャとする自動メモ化プロセッサ

指導教員　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学大学院工学研究科
修士課程創成シミュレーション工学専攻
平成 20 年度入学　20413539 番

高木 伴彰

平成 22 年 2 月 4 日

スーパスカラ型 ARM をベースアーキテクチャとする自動メモ化プロセッサ

高木 伴彰

内容梗概

近年，科学技術計算だけでなく医療分野，工業製品の設計等といった様々な場で CPU には高度な計算処理能力が求められており，その高速化のために様々な研究が行われいる．動作クロックを向上させるためにパイプラインの多段化は進み，それに伴い増大するハザードによるペナルティを緩和するために命令レベル並列性についての研究がさかんに行われてきた．しかし，動作クロックの向上による消費電流の増大を無視できなくなり，CPU はマルチコア化の道を進むことになる．マルチコア CPU を有効に活用するためにスレッドレベル並列性の研究が行われてきたが，依然としてシングルタスクの高速化は非常に困難な状況である．

一方，これら並列化による CPU の高速化とはまったく異なったパラダイムである計算再利用という高速化手法が存在する．計算再利用の実行モデルの一つに自動メモ化プロセッサというものがある．自動メモ化プロセッサは過去に実行した関数やループの入出力を表に記憶しておき，後に同じ関数やループが過去と同じ入力で実行されようとする時に，過去の実行結果を再利用することにで高速化を図る．

この自動メモ化プロセッサの既存モデルにはいくつかの問題点が存在する．一つは、ベースアーキテクチャが単命令発行型プロセッサであるという点である．現在市場に流通しているプロセッサは通常，パイプライン化されている．そのため，自動メモ化プロセッサの現実的なパフォーマンスについて，既存モデルの評価結果だけでは十分とはいえない．また，既存モデルは SPARC アーキテクチャを採用しており，純粋な RISC である本アーキテクチャの各仕様はハードウェアによるメモ化を行う上で非常に適応的であった．よって，他のアーキテクチャで既存モデル同様ハードウェアによるメモ化を行うことが可能かどうかは未知である．以上の 2 つの問題点を解決するために，本研究では，スーパースカラ型 ARM プロセッサをベースアーキテクチャとした自動メモ化プロセッサを提案する．

本提案手法をサイクルベースのシミュレーションにより評価を行ったところ，SPEC CPU95 ベンチマークで通常実行時に比べて最大で 17%，平均で 2%の実行サイクル数削減を確認できた．

スーパスカラ型 ARM をベースアーキテクチャとする自動メモ化プロセッサ

目次

1 はじめに

現在までに，プログラムの実行を高速化する手法として，スーパースケーラのように命令レベル並列性 (Instruction-Level Parallelism：ILP) に着目したものが研究されてきた．しかしながら，プログラム自体に存在する ILP には限界があり，命令レベルの並列化を行うだけでは，プロセッサの性能向上が頭打ちになりつつある [1]．また，並列性を高めるための命令のスケジューリング制御部分はハードウェアに負担をかけている．この流れを受け，CPU がマルチコア化されるとともに，命令レベル並列性より粒度の荒い並列性であるスレッドレベル並列性 (Thread-Level Parallelism：TLP) が注目されることとなる．並列性の抽出にはマルチスレッドライブラリや，高度なコンパイラが用いられる．ライブラリを用いた場合，プログラム内の並列化できる部分を明示的にスレッドの形で書き下す必要があり，プログラマに負担がかかる．一方，コンパイラによる抽出はその精度に問題がある．よって TLP によるプログラム実行の高速化もまた難しい状況であるといえる [2].

これらはいずれもプログラム実行の並列化という方法で高速化を図るものであるが，本研究はそれとはまったく異なる計算再利用という手法に着目する．

計算再利用には，ハードウェアによるものとソフトウェアによるもの，またその双方によるものなど，様々なものが提案されている．専用のハードウェアを用いることによりバイナリの変更無しに，既存のプログラムを実行できる区間再利用のモデルに自動メモ化プロセッサというものが存在する．自動メモ化プロセッサは実行した関数の入出力を表に記録しておく．再び同関数が呼び出されたときにその入力と過去に実行した関数の入力とを比較し，一致すれば過去の出力を利用する [3].

この自動メモ化プロセッサの既存モデルにはいくつかの問題点が存在する．一つは、ベースアーキテクチャが単命令発行型プロセッサであるという点である．現在市場に流通しているプロセッサは通常，パイプライン化されている．そのため，自動メモ化プロセッサの実際的なパフォーマンスについて，既存モデルの評価結果だけでは十分とはいえない．また，既存モデルは SPARC アーキテクチャを採用しており，そのシンプルな命令セット，レジスタウィンドウ，ABI 等の各仕様はハードウェアによるメモ化を行う上で非常に都合が良い．逆説的に言うと，既存モデルは SPARC アーキテクチャに非常に依存した実装となっており，他のアーキテクチャで既存モデル同様ハードウェアによるメモ化を行うことが可能かどうかは未知である．以上の 2 つの問題点を解決するために，本研究では，スーパースカラ型 ARM プロセッサをベースアーキ

テクチャとした自動メモ化プロセッサを提案する．

以下，2 章では本研究が扱う自動メモ化プロセッサの動作モデルを概説する．3 章では，本論文の提案であるスーパースカラ型 ARM プロセッサをベースアーキテクチャとする自動メモ化プロセッサのモデルを述べ，4 章で提案手法の実装手法について詳細に説明する．5 章でその評価を行う．そして最後の 6 章において，結論を述べる．

2　自動メモ化プロセッサ

本章では，メモ化とそのハードウェア実装モデルの一つである自動メモ化プロセッサについて述べる．

2.1　メモ化

メモ化（Memoization）とは，関数呼び出しやループなどの命令区間において，その入力と出力の組を記憶しておくことである．後に，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，過去に記憶された出力を再利用することで，命令区間の実行自体を省略し，高速化を図ることができる．メモ化には，ハードウェアによるものやソフトウェアによるもの，またその双方を利用したものなど，様々なものが提案されている．

ソフトウェアによるメモ化は，Huang らが，コンパイラに機能を追加することで，再利用を実現する手法を提案している [4]．ソフトウェアメモ化は，メモ化可能な命令区間を柔軟に決めることができるが，メモ化によるオーバヘッドが大きく，このオーバヘッドを考慮したプログラミングが必要となる．一方で，ハードウェアによるメモ化は，柔軟に対象区間を決定することができないが，ソフトウェアメモ化に比べ，オーバヘッドは非常に小さい．ハードウェアメモ化は Sodani らが，単命令を対象とした汎用的な再利用を提案しており，その効果が確認されている [5]．

本研究で扱う自動メモ化プロセッサは，ハードウェアにより自動的にメモ化を行う機構を備えている．自動メモ化プロセッサは，関数及びループの実行を省略することによりプログラムの実行の高速化を図る．

2.2　関数再利用

自動メモ化プロセッサにおける関数を対象としたメモ化の動作モデルを述べる．自動メモ化プロセッサはまず，関数実行時にその入力と出力を表に記憶する．そして，再び同じ関数が実行されるときにその入力を過去の入力と比較し，一致した時には対応

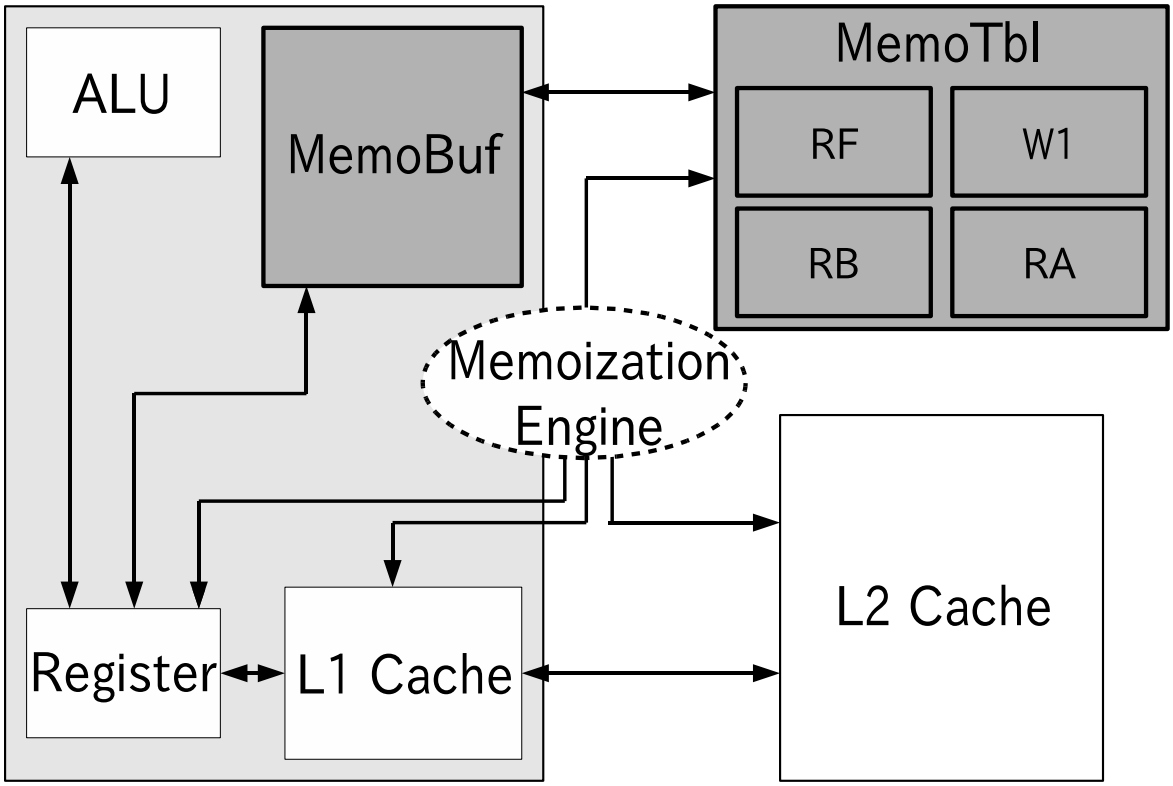


図 1: 自動メモ化プロセッサ構成図

する出力をレジスタ，メモリに書き戻し，当該関数の実行を省略する．

自動メモ化プロセッサの構成図を**図 1** に示す．MemoTbl は関数情報を記憶する RF，関数への入力セットを記憶する RB，関数への入力セットのアドレスを記憶する RA，出力を記憶する W1 の 4 つにより構成される．また，自動メモ化プロセッサは MemoTbl に加え，関数への入出力を一時的に保存しておくための MemoBuf を備えている．

(**図 2** 中 (1)) に示すように，関数再利用は call 命令から return 命令までの命令区間を再利用対象としてして扱う．ある関数への call 命令が検出された際，その関数の先頭アドレスが RF 上に存在するか検索を行う．関数の先頭アドレスが存在した場合には，入力の一致比較を行うために RB 上の比較開始地点を得る．RB と RA を用いた入力の一致比較 (再利用テスト) において，すべての入力が一致すれば，登録済みの出力が W1 から書き戻され，関数の実行は省略される．

もし RF に当該関数が見つからなかったりすべての入力が一致せず，再利用が行われなかった場合，関数を通常通りに実行し MemoBuf 上に関数の入出力 (引数格納レジスタ，及び局所変数以外のメモリ参照) を登録していく．当該関数の実行終了時，すなわち return 命令を検出した時，登録された内容を MemoTbl 本体である RF，RB，RA，

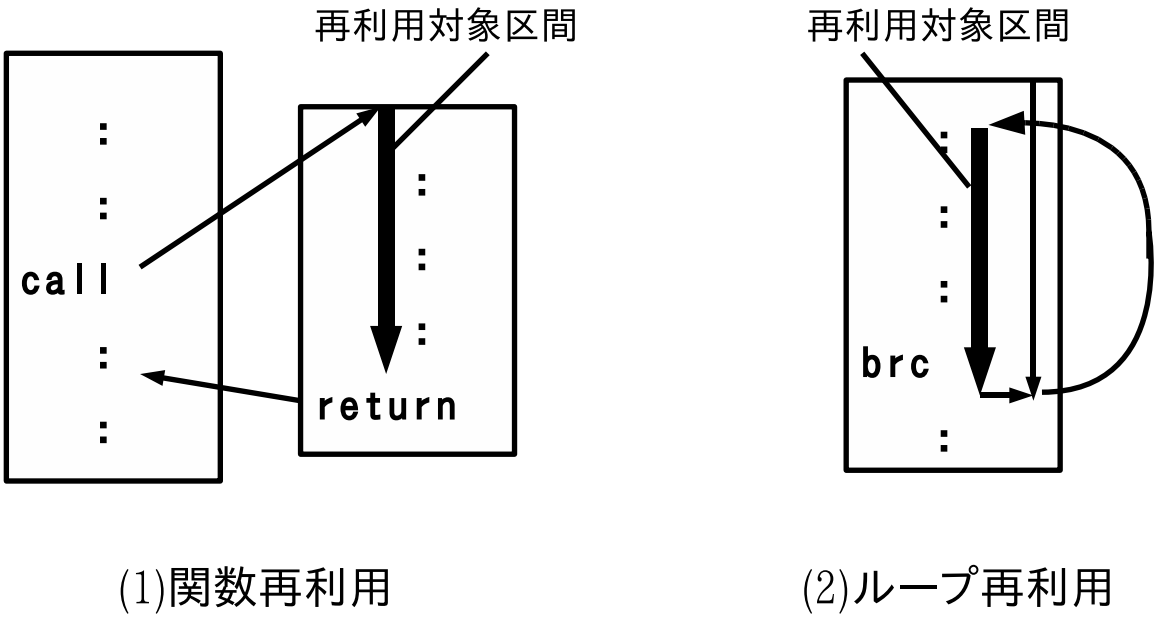


図 2: 関数再利用とループ再利用

W1 に登録する．

2.3 ループ再利用

次に，ループ区間を再利用対象とした場合を説明する．2.2 で述べたように，call 命令検出から return 命令検出までを再利用対象区間として認識することで，関数再利用は可能となっている．同様に，どの命令からどの命令までがループ区間であるかを認識することではじめてループ再利用が可能となる．ループは，一度後方分岐命令に到達し，後方分岐が成立した後に，再度同じ後方分岐命令に到達してはじめて，それまでの命令列がループを形成していたことがわかる (図 2 中 (2))．つまり，後方分岐命令の分岐先から，再度現れる同一の後方分岐命令までの入出力を MemoBuf に登録しておくことで，関数同様にループ再利用が可能となる．

前項で述べた通り，関数再利用に必要な入出力は，引数格納レジスタ及び当該関数の局所変数以外のメモリ参照である．一方，ループの再利用に必要な入出力は，ループ内での全てのレジスタ及びメモリ参照である．これは，関数と異なりループには局所変数が規定されないためである．

関数再利用では再利用が可能な場合，出力書き戻し後の復帰アドレスは必ず call 命令の次の命令 (厳密には遅延スロットがあれば，スロット分の命令後) のアドレスである．しかし，ループ再利用では，分岐方向 (taken or not taken) によって復帰アドレス

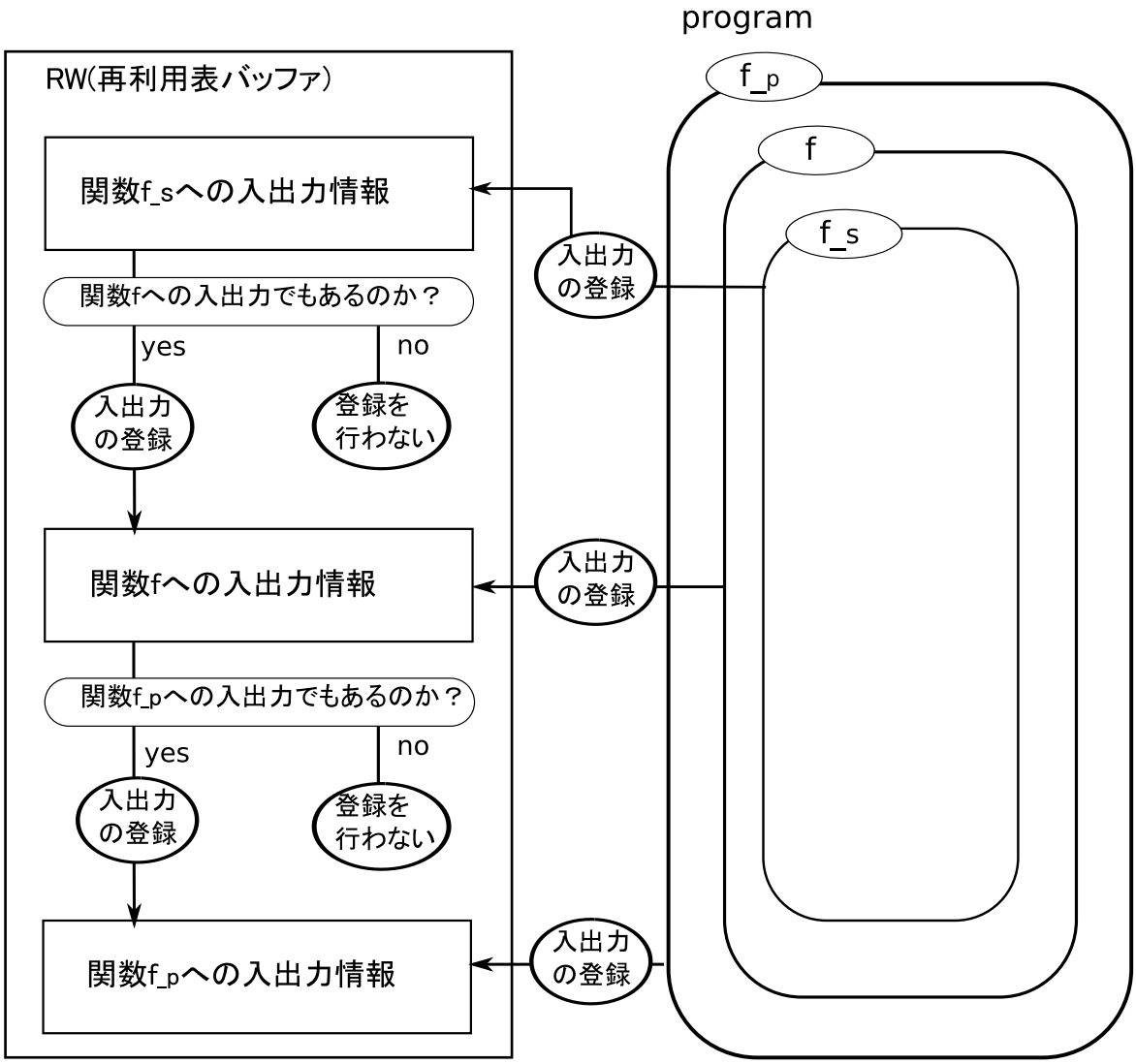


図 3: 多重再利用を実現する MemoBuf の構造

は異なる．そのため，再利用適用時に復帰アドレスを得るために，ループ毎に RB に分岐方向を記憶させておく必要がある．

2.4 多重再利用

一般的に，関数は関数を呼び出すことにより入れ子構造をとる．この入れ子構造にある関数を再利用可能とするような MemoTbl への登録の仕組みを多重再利用と呼ぶ．多重再利用時の登録の様子を以下に説明する．

図 3 は多重再利用時において、入れ子構造の関数の入出力がどのように MemoBuf に登録されるのかを示している．図 3 の右側半分は実行中のプログラムの関数の入れ

子構造を表しており，関数 f が関数 f_s を呼び出し，関数 f は関数 f_p に呼び出されている．さて，再利用は行われず，関数 f_s が通常実行される場合について考える．関数 f_s の処理が進むにつれ，MemoBuf には関数 f_s の入出力が登録されるが，関数 f_s の入出力の中で，大域変数や関数 f を呼び出した関数である f_p の局所変数の読み書きによるものは，関数 f の入出力でもあるため，関数 f の入出力としても登録する必要がある．つまり，関数の入れ子構造において，入れ子の内側の関数の入出力は入れ子の外側の関数の入出力となり得るということである．

このことから，多重再利用を実現するために MemoBuf はスタック構造をとっている．新たに関数が call される度に，MemoBuf にその関数の入出力を記録するためのエントリを push する．そして，return 命令で関数の実行が終わると pop し，その入出力内容を MemoTbl へ登録する．このように，関数呼び出しの入れ子構造をスタックで表現する．当該関数の入出力が検出された時，その関数の入出力の記録を担当する MemoBuf エントリに入出力は記録される．そして次に，該当 MemoBuf エントリより下に積まれている MemoBuf エントリ各々に対して，入出力が記録するべき入出力なのかの判断が行われ，該当エントリが管理する関数の入出力であると判断された場合，その入出力は記録される．

2.5 入力値の検出と SPARC ABI

ある関数 f を再利用するためには，f の実行時においてその入出力を記憶しておく必要がある．関数 f における大域変数，および関数 f を呼び出した関数の局所変数への参照を入力，書き込みを出力として表に記録する．自動メモ化プロセッサは実行するプログラムを SPARC ABI(Application Binary Interface) の規定に基づくと仮定することで，関数実行中におけるレジスタおよびメモリへのある参照，書き込みが，現在実行している関数に対する入出力であるのか，そうでないのかを判断することが可能となっている．以降，SPARC ABI に基づき，自動メモ化プロセッサがどのように関数の入出力を検出しているのかを説明する．

SPARC アーキテクチャ[6] では，レジスタウィンドウを規定している．レジスタウィンドウにより，関数の引数の受け渡しを主記憶を介さずに行うことが可能となる．レジスタウィンドウは**図 4** のような構成であり，以下の 4 種類のレジスタが基本構成要素となっている．なお，全ウィンドウから参照可能な global レジスタは構成上図中に記載していない．

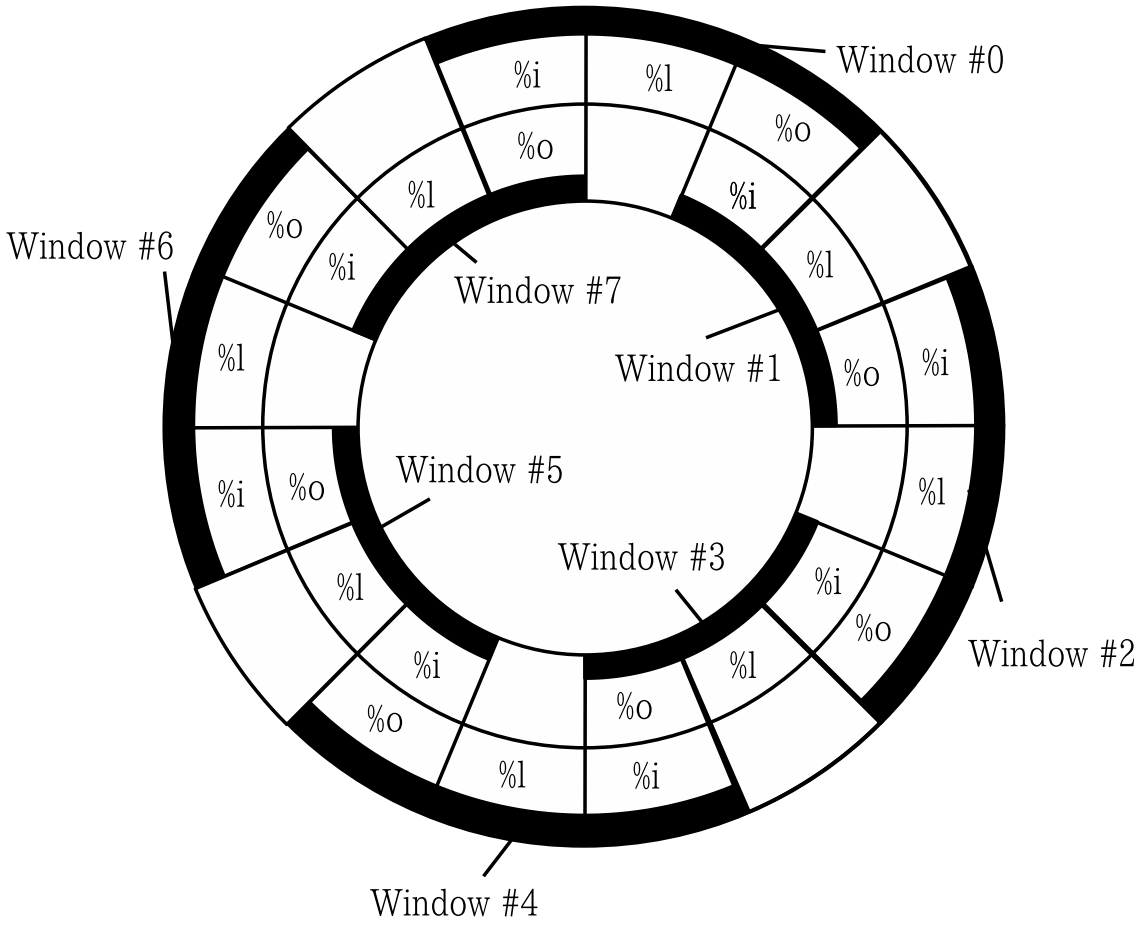


図 4: レジスタウィンドウ

global レジスタ (%g0-7)

どの関数からも常にアクセス可能なレジスタである．

out レジスタ (%o0-7)

%o0-5 は呼び出す関数への引数を受け渡すことに用いられたり，関数実行の作業用に用いられる．また%o0，%o1 は返り値を受け取るためにも用いられる．%o6(%sp) はスタックポインタとして用いられる．%o7 は関数呼び出しの際，戻り番地の格納のために用いられる．

local レジスタ (%l0-7)

局所変数および作業用に用いられる．

in レジスタ (%i0-7)

%i0-5 は関数が引数を受け取るときに用いられる．また%i0，%i1 は返り値の格納場所としても用いられる．%i6(%fp) はフレームポインタ，%i7 は当該関数からの

戻り番地の格納のために用いられる．

各ウィンドウは%i，%l，%o から成り，%i は隣接の%o，%o は隣接の%i と同一となっている．

図 4 において実行中のプログラムが#0 のウィンドウを使用しているとする．非リーフ関数 (関数内部で関数の呼び出しがある関数) が呼び出されたとき，save 命令が実行され使用するウィンドウは#1 となり，関数の呼び出し前に使用していた%o が%i として扱われる事となる．関数の呼び出し前に使用していた%i と%l が使用不可能になるが，新たにウィンドウ#1 内の%l と#2 の%i でもある%o が使用可能となる．このように関数が呼び出される毎にプログラムがアクセス可能なレジスタ群はスライドしてゆく．

さて，呼び出された関数は引数を#2 の%i(#1 の%o) を介して受け取る．つまり，非リーフ関数が呼び出された時，%i への書き込みがまだ行われていない状態で%i からの読み出しが発生した場合，これを関数の入力と判断できる．また，%i0，%i1 への書き込みは出力と判断できる．一方，リーフ関数が呼び出されたときは，save 命令は実行されないため使用ウィンドウの変更ははない．そして，リーフ関数に与えられるレジスタは#0 の%o のみとなる．結果，%o への書き込みがまだ行われていない状態で%o からの読み出しが発生した場合，これを関数の入力と判断できる．また，%o0，%o1 への書き込みを出力と判断できる．

このように，レジスタを介した引数の受け渡しの場合は，関数の入出力の検出を容易に行うことができる．しかし，レジスタを介した引数の受け渡しは最大 6 つまでしか行えない一方，関数によっては 7 つ以上の引数の受け渡しが存在する．7 番目以降の引数はレジスタではなく，スタックを介して受け渡しされる．

次に，主記憶参照時における関数の入出力の検出について説明する．SPARC ABI に従って書かれたプログラムは以下の条件を満たす．

- %sp 以上の領域のうち，%sp+0〜63 はレジスタ待避領域，%sp+68〜91 は引数待避領域である．
- 構造体を返す場合，%sp+64〜67 に暗黙的引数として構造体へのポインタを格納する．
- 7 つ目以上の明示的引数は，%sp+92 以上の領域に置かれる．
- OS がスタックサイズの上限として定める LIMIT 未満の領域に大域変数は置かれる．

- %sp は，LIMIT 以下になることはなく，LIMIT〜%sp の領域は無効である．

これらの条件から，次のように，主記憶参照を伴う関数の入出力を検出できる．主記憶参照される関数の入出力は大域変数と当該関数を呼び出した関数の局所変数との二つに分けられる．ここで，現在実行中の関数を f，関数 f の呼び出し元の関数を f_p とする．まず，大域変数による入出力についてであるが，LIMIT 未満のアドレスへの読み書きが発生した時に，これを関数 f の入出力と判断できる．次に，関数 f_p の局所変数による入出力についてだが，一般に，関数 f の実行中において，%sp+92 が f の局所変数なのか f_p の局所変数なのかの区別がつかない．このため，引数を 7 つ以上検出した関数呼び出しは再利用の対象外とし，関数 f 呼び出し前の%sp+92 を記憶しておく．関数 f 実行時において，主記憶参照が記憶しておいた%sp+92 未満の場合は，関数 f の局所変数であるため関数の入出力と判断しない．一方，主記憶参照が%sp+92 以上ならば，関数 f_p の局所変数であるため，関数の入出力と判断する．

2.6 入力値の検索

関数内には往々にして分岐が存在し，各分岐先ではそれぞれで次に参照する入力のアドレスが違う場合がある．分岐先を決定する値が関数の入力であった時，次に参照する入力のアドレスはそれより前の入力に依存しているといえる．このように，関数の入力は，その入力の値により次に参照する入力のアドレスが決定されることが多い．これを考慮に入れた MemoTbl における関数の入力の記憶構造 (RB，RA) について述べる．

RB は入力を記録する機構であり，RA は次に参照すべき入力の主記憶上のアドレスを記録している．RB の各行は RA の各行と一対一に対応している．RB は親エントリを表す key と，関数の入力とその入力に対するマスクを持つ．そして，対応する RA は次に比較すべき入力のアドレスまたは，出力エントリを指し示す W1 へのポインタを持つ．再利用テストが成功した時に，このポインタで参照される W1 の内容が出力としてレジスタ，メモリに書き戻される．

図 5 で入力の一致比較がどのように行われるのかを示す．key 値の FF は最初の入力であることを示しており，これに設定した key 値と引数が格納されたレジスタの値で RB の連想検索を開始する．RB のインデックス 00 において，エントリの持つ key の値と入力の値が検索条件である key 値およびレジスタの値と一致し，同行の RA から次に参照するアドレスを得る．一致した RB のインデックスを key とし，参照先の値

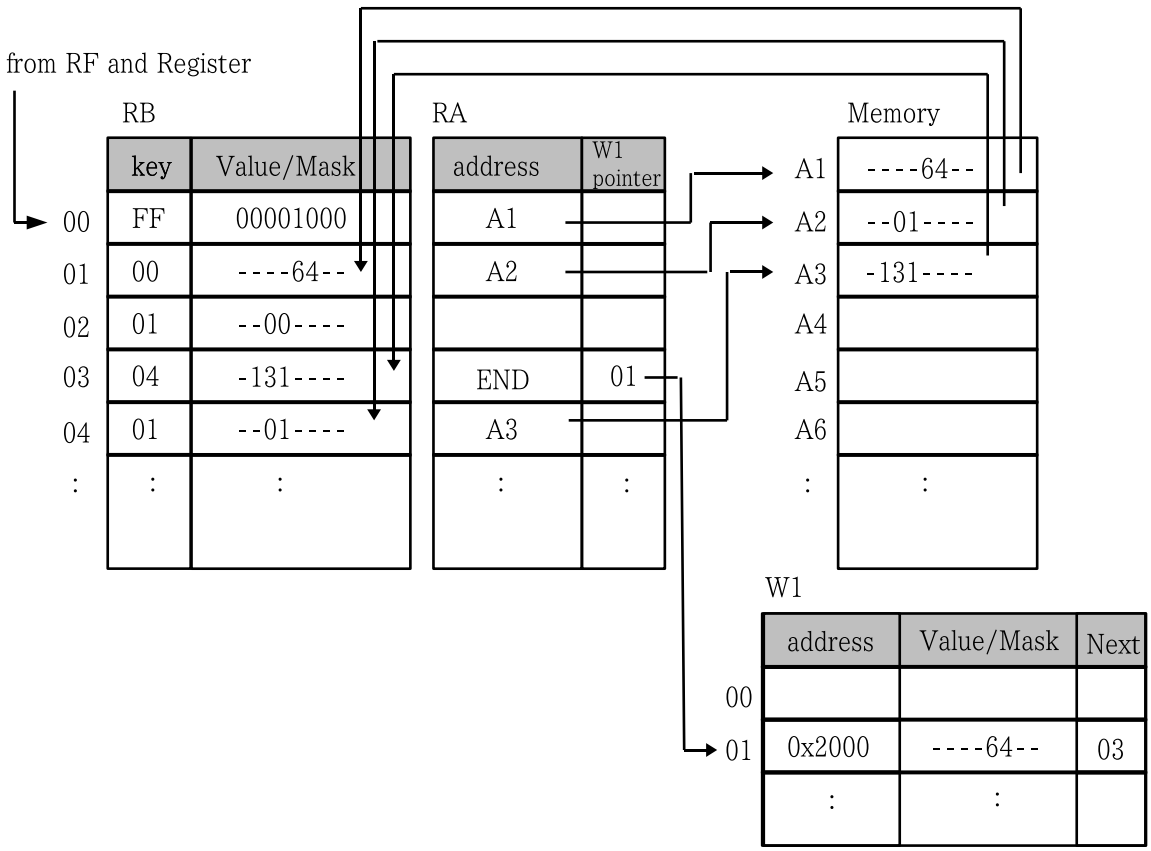


図 5: MemoTbl の検索手順

と一致する RB 上のエントリを連想検索する．これを繰り返し，RA から得られるアドレスがない (つまり，すべての入力の一致比較に成功した) 時，RA から該当する出力が格納されている W1 へのポインタを得る．そして，このポインタを用いて W1 を参照し，書き戻すべき出力が取り出される．

このような入出力格納方式を取っているのには主に 2 つ理由がある．1 つは，RB を CAM(Contents Addressable Memory) で構成していることに関係する．CAM は高速な連想検索が可能であるため本手法に用いられている．一方で，その 1 エントリの幅に制限があるためすべての入力を CAM の 1 エントリに納めることは不可能であり，分割して納める必要があった．2 つめは，RB のエントリを効率的に用いるためである．ある関数への入力情報が，MemoTbl に登録されている同関数への入力情報と途中まで同じであった場合，本格納方式では同じ部分に関しては重複登録を避けることができる．

2.7 オーバーヘッドフィルタ

さて，入力値の検索は，高速に連想検索可能な CAM を用いているといえども，そのオーバヘッドは大きい．また，再利用に関するオーバヘッドは入力値の検索に要する時間だけでなく，RB からキャッシュへの出力の書き戻しにかかる時間も存在する．そのため，再利用によって得られる効果が小さいような短い命令区間については，削減されるサイクル数よりも，再利用オーバヘッドの方が大きいという場合が存在する．このような命令区間に対しては，計算再利用を適用しないようにするため，自動メモ化プロセッサには再利用オーバヘッドを動的に評価し，それに基づいて入力値検索の実施を決定する機構が備わっている．

自動メモ化プロセッサは直近の一定期間 T における RB へのエントリの登録回数 N を記録している．これらの値と当該命令区間の過去の省略ステップ数 S から，実際に削減できたステップ数は

$$N \times (S - Ovh^R - Ovh^W) \tag{1}$$

として計算できる．なお，Ovh^R,Ovh^W はそれぞれ，過去の履歴より概算した，入力値の検索によるオーバヘッドと，RB からキャッシュ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，入力値の検索によるオーバヘッドは存在する．このオーバヘッドは，

$$(T - N) \times Ovh^R \tag{2}$$

として計算できる．

このとき，発生したオーバヘッド式 (2) よりも削減できたステップ数式 (1) が大きいような命令区間は，再利用の効果が得られると考えられる．そこで，RF にいくつかのカウンタを付加することによってこれらを記録し，それをもって上記のオーバヘッド式を計算し，効果が得られないと判断された命令区間は再利用の対象とせず，RB への登録を行わない．

3 提案

3.1 既存モデルの問題点と提案

自動メモ化プロセッサの既存モデルにはいくつかの問題点が存在する．一つは、ベースアーキテクチャが単命令発行型プロセッサであるという点である．現在市場に流通しているプロセッサは，動作周波数を向上させるために，通常，パイプライン化されている．一般的に，単命令発行型プロセッサに比べてパイプラインプロセッサは複雑

なハードウェア構成をとる．そのため，現在メインストリームにあるプロセッサに自動メモ化機構を搭載した場合，既存モデルの評価結果が示す通りのパフォーマンスが得られるかどうかについては疑問が残る．また，既存モデルはSPARC アーキテクチャを採用しており，純粋な RISC である SPARC のシンプルな命令セット，レジスタウィンドウ，関数呼び出し規約を含む ABI 等の各仕様は，ハードウェアによるメモ化を行う上で非常に都合が良い．逆に言うと，自動メモ化プロセッサは SPARC アーキテクチャに非常に依存した実装となっており，他のアーキテクチャで，既存モデルと同様のハードウェアによるメモ化を行うこと可能かどうかは未知である．

そこで，スーパースカラ型プロセッサをベースアーキテクチャとするメモ化プロセッサのモデルを提案する．これによって，市場のプロセッサ事情に適合した，より現実的な自動メモ化プロセッサのパフォーマンスの評価が可能となる．また，命令セットアーキテクチャには ARM[7] を採用する．ARM は，SPARC 同様 RISC に分類されるが，SPARC に比べ複雑な命令を多数持つ．ARM を対象として自動メモ化プロセッサに基づいたハードウェアメモ化手法を構築できれば，より汎用的な自動メモ化プロセッサのモデルを構築できたといえる．

3.2 提案する自動メモ化プロセッサモデル

3.2.1 提案するメモ化の動作モデル

最初に，提案するスーパスカラ型 ARM をベースアーキテクチャとするメモ化プロセッサの動作モデルについて述べる．本提案では関数のみを対象としたメモ化を扱う．

図 6 に説明に用いるプログラム例とそれが実行された時のパイプラインの様子を示す．プログラム中の四角で囲ってある部分は func 内の命令列である．なお，説明の簡単化のため，図例に示すベースアーキテクチャのパイプラインは，IF(命令フェッチ)，ID(デコード)，EXE(命令実行)，RE(リタイア) のシンプルな構成を取っている．また，キャッシュミス等によりパイプラインが stall することがない，理想的な実行時の様子となっている．

ではまず，提案するモデルの再利用成功時の動作を説明する．再利用が可能かどうかを調べるために行う入力値の検索は，既存モデル同様に関数呼び出し命令が検知された時に開始される．パイプラインプロセッサをベースアーキテクチャとしている本提案モデルでは，リタイアステージにて関数呼び出し命令がコミットされた時点で入力値の検索は行われる．入力値の検索とは，論理レジスタ，及びキャッシュの値が該当関数を過去に実行した時と同じものであるかどうかを比較することである．この比較

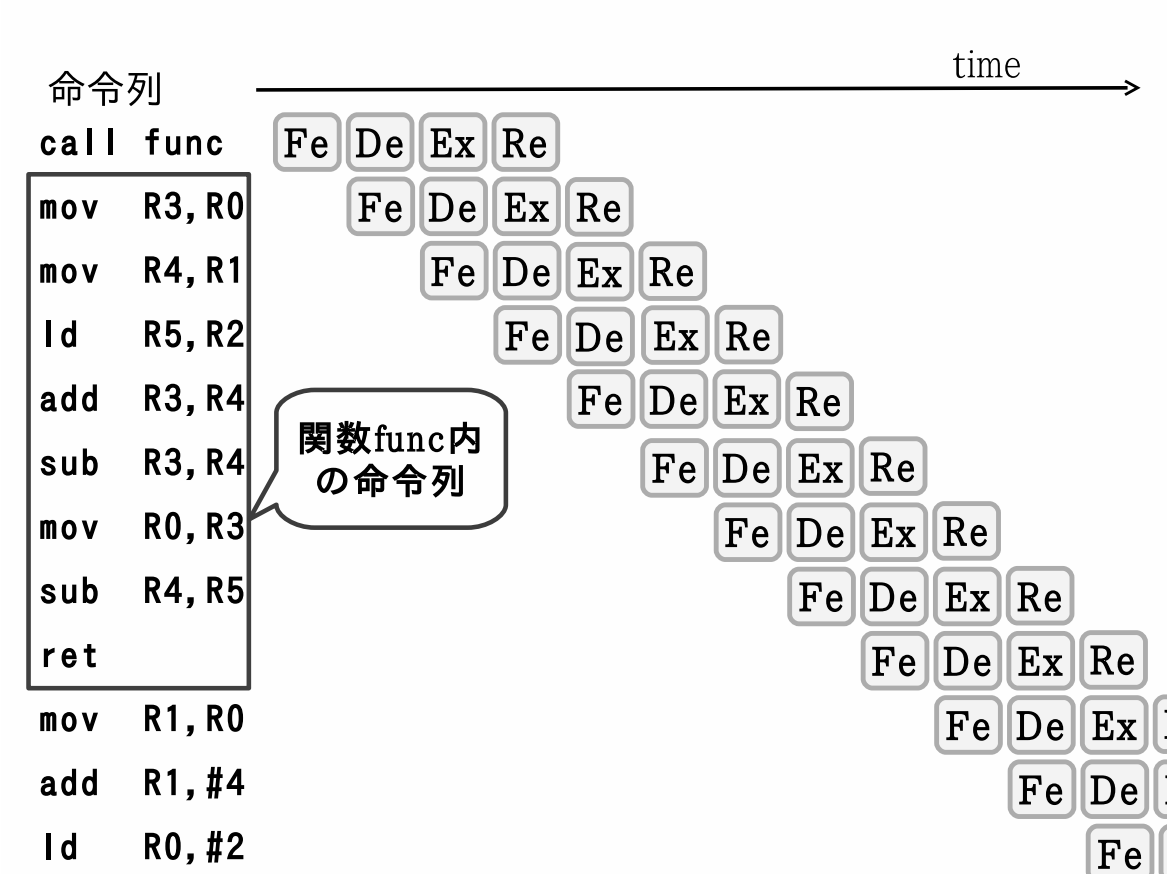


図 6: 動作モデル:通常実行時

を正しく行うには，関数呼び出し命令より前の命令列による論理レジスタやキャッシュへの書き込みが既に終了されている必要がある．関数呼び出し命令がコミットされていれば，それより以前の命令が終了していることが保証されるので，入力値の検索は関数呼び出し命令がリタイアされたタイミングで行う．

図 7 に提案するモデルの再利用成功時の様子を示す．先ほど述べたように，関数呼び出し命令がリタイアされたタイミングで，入力値の検索が開始される (図 7 中 (1))．その後，入力値が過去の入力値と全て一致するのが確かめられ，再利用が適用可能であることが判明する (図 7 中 (2))．関数の実行を省略するために，過去の関数実行時出力をレジスタ，キャッシュに書き戻すのだが，その前にパイプラインのフラッシュを行う (図 7 中 (2)-(3))．すでにパイプラインに投入されている命令列は，再利用の対象である関数内の命令列であり，無効化する必要があるためである．パイプラインのフラッシュ後，あらためて出力を書き戻し (図 7 中 (3))，その後，後続命令をフェッチしてくる (図 7 中 (4))．このパイプラインのフラッシュによりバブルが発生することになる．

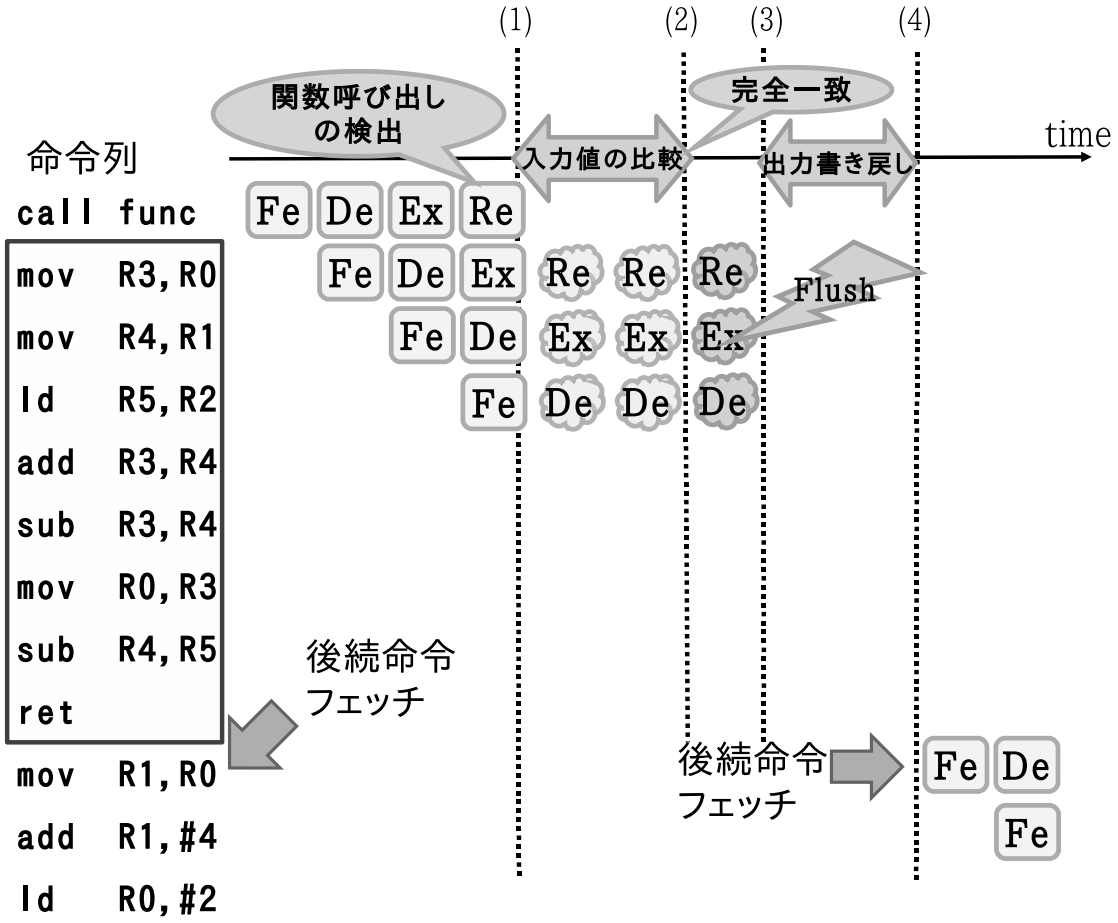


図 7: 動作モデル:再利用成功時

再利用により削減可能なステップ数が大きければ，このバブルのペナルティによる性能低下は相対的に無視できると考えられる．

次に，提案するモデルの再利用失敗時の動作の様子を示す．**図 8** がその様子であり，実行するプログラム例は再利用成功時の例と同じである．まず，先程の例と同様に，Retire ステージにて関数呼び出し命令がコミットされるのを検知し，入力値の検索が始まる (図 8 中 (1))．レジスタ，キャッシュの値と RB 内の値が一致せず，再利用の適用が不可能なことが判明すると，その後は通常通り再び命令の実行を継続する (図 8 中 (2))．

さて，高速化を達成するに当たって，図 8 からもわかる通り，入力値の検索によるオーバヘッドが問題となる．既存モデルでもこのオーバヘッドは問題視されており，オーバヘッドフィルタ (2.7 節) 等の対策が講じられてきた．ベースアーキテクチャをスーパースカラ型プロセッサとしている本提案モデルでは，入力値の検索とパイプラ

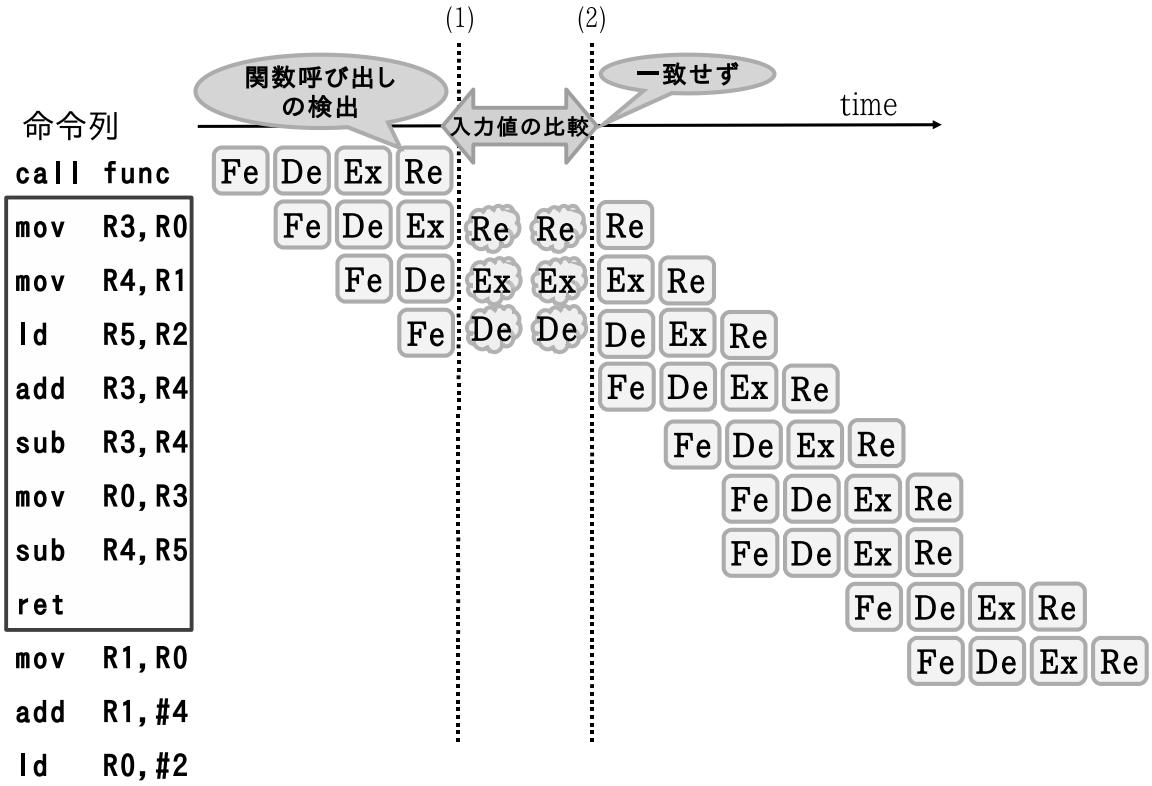


図 8: 動作モデル:再利用失敗時

インの実行を並行に行うことでこのオーバヘッドを削減できると考えられる．

図 9 に入力値の検索によるオーバヘッド削減モデルの動作例を示す．このように，入力値の検索中に，関数内の命令列を並行に実行することで，再利用失敗時におけるオーバヘッドの削減を行うことができる (図 9 中 (1)-(2))．しかしながら，図 9 は理想的な動作の様子である．実際は，入力値検索のための Load 要求とパイプラインで実行される命令からの Load 要求との衝突など考慮するべき点が多々あり，完全にオーバヘッドを削減することはできない．詳細は 4.2 節に譲る．

3.2.2 SPARC ABI と ARM ABI がメモ化に与える影響

自動メモ化機構実装対象であるアーキテクチャの ABI が定める関数呼び出し規約は，ハードウェアによるメモ化モデルに大きな影響を与える．既存モデルがベースアーキテクチャとして採用している SPARC の ABI と提案モデルが採用する ARM の ABI 間にある差について説明する．**図 10** に SPARC ABI が定める関数スタックフレームの構造を示す．図は `FuncA` が `FuncB` を呼び出し，復帰するまでの関数スタックフレーム構造の移り変わりを表している．SPARC ABI が定める関数呼び出し規約では，6 つ目までの引数はレジスタに，7 つ目以降の引数は関数呼び出し時の%sp 以降に格納される．

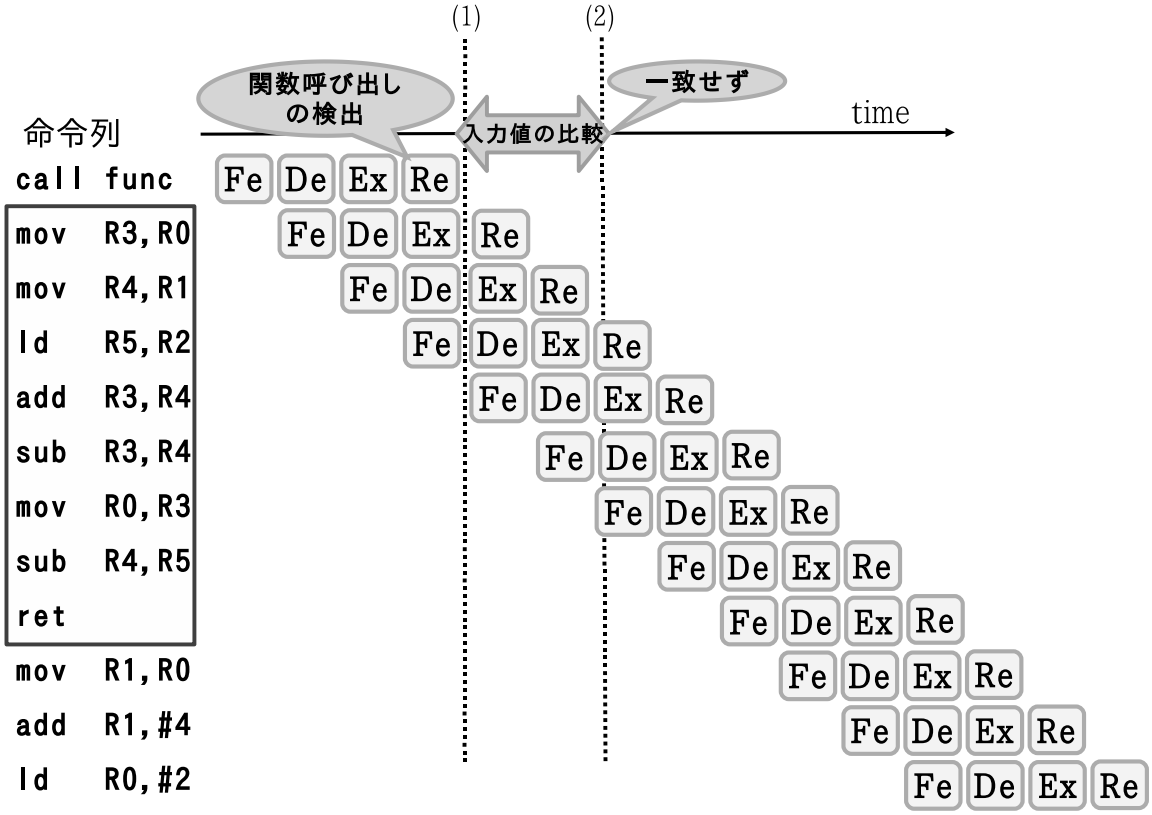


図 9: 動作モデル:オーバヘッド削減モデル

関数の入力とは，引数と大域変数，及び当該関数の呼び出し元関数の局所変数である．ある関数実行中に，%sp+92 以降のアドレスからの読み出しがあった場合，それが 7 つ目以降の引数読み出しであるのか，それとも当該関数を呼び出した関数の局所変数読み出しであるのかを知ることはできない．つまり，%sp+92 以降のアドレスから読み出しがあった場合，それが当該関数の入力であるのか否かを判断することができない．

では，上記の問題を解決せずに，関数の自動メモ化を行った場合に起こる現象を**図 11** のプログラム例を用いて説明する．

引数を 7 つ受け取る関数 FuncB をメモ化する時を考える．main 関数から呼び出された FuncA において，FuncB を呼び出す時のスタックの状態は**図 12** 中 (1) のようになる．そして，MemoTbl には図 12 中 (2) のような形でメモ化される．次に，main 関数から直接呼び出される FuncB のスタックフレームの状態を考える (図 12 中 (3))．この時，FuncB の再利用のために入力値の比較が行われるが，この比較は間違ったものであり，誤った再利用の可能性がある．FuncA においての FuncB 呼び出し時と main 関数においての FuncB 呼び出し時では，スタックフレームの状態が異なるので，第 7 引数

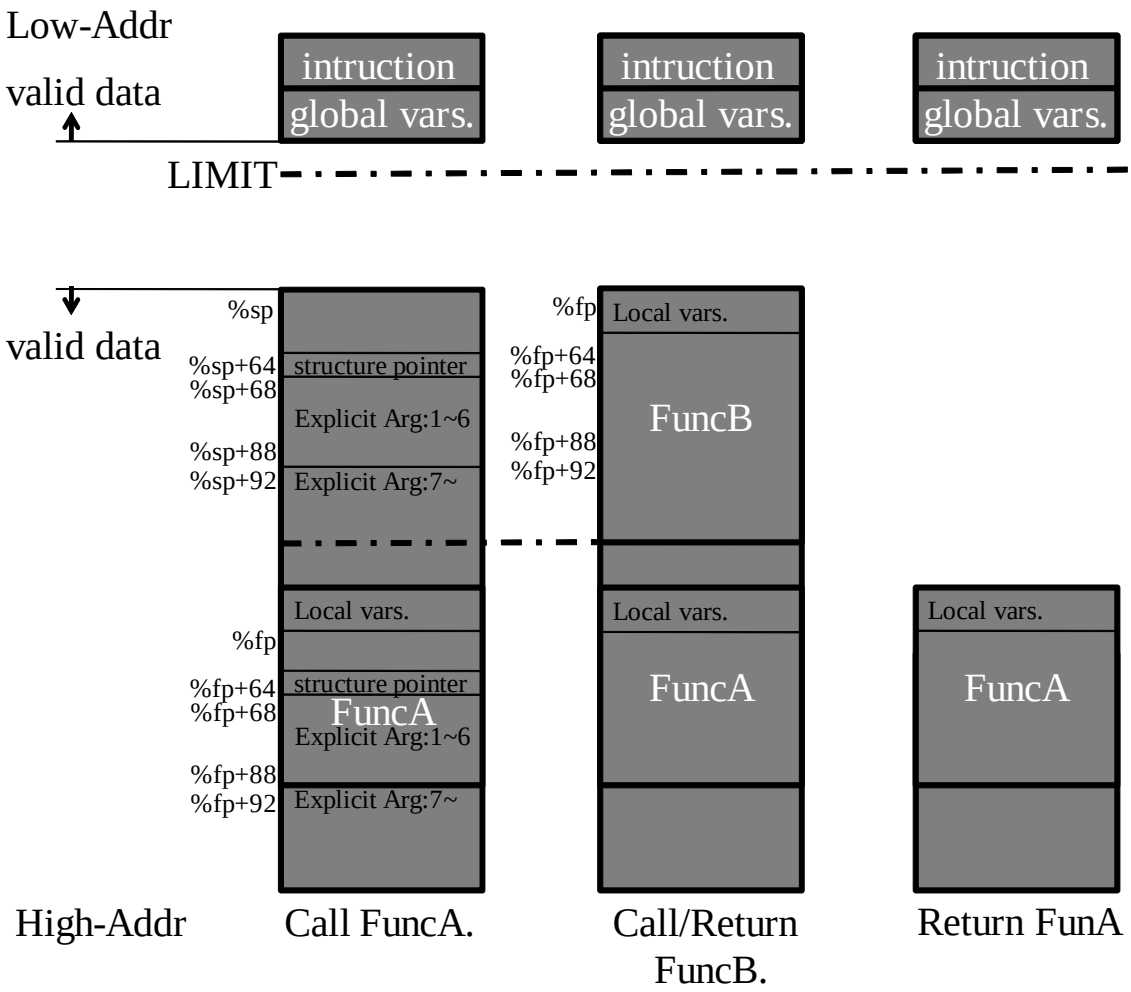


図 10: SPARC の関数スタックフレーム

の比較時に本来比較すべき対象と異なるアドレスで比較してしまう．これは，引数のアドレスを絶対番地形式で登録していることが原因である．スタックフレーム上の引数のアドレスは関数コンテクストによって異なるので，%sp からの相対アドレスという形でメモ化を行うべきである．

では，引数が格納されている可能性のあるアドレス%sp+92 以上の読み出しを%sp からの相対アドレス形式でメモ化することを考える．この場合，`FuncA` における `FuncB` のメモ化は図 12 中 (2) のようになる．main 関数における `FuncB` の入力値の比較はまたもや間違ったものとなる．`FuncB` の第 7 引数で受け取ったポインタの実体は絶対番地形式で登録されるべきであり，これを%sp からの相対オフセット形式でメモ化すると，本来比較すべきアドレスと異なるアドレスで比較を行ってしまう．よってこれも

```
 1:int FuncA(int *A)
 2:{
 3: reuturn(FuncB(1, 2, 3, 4, 5, 6, &A));
 4:}
 5:
 6:int FuncB(int a,int b,int c,int d,int e,int f,int *g)
 7:{
 8: return(a + b + c + d + e + f + g);
 9:}
10:
11:main()
12:{
13: int m=7;
14: FuncA(&m);
15: int m=8;
16: FuncB(1, 2, 3, 4, 5, 6, &m)
17: return();
18:}
```

図 11: プログラム例 1

また誤った再利用が行われる可能性がある．

結局，入力値の比較を正しく行うには，引数のみ%sp からの相対オフセット形式とするべきである．しかし，前述の通りスタックフレーム上のどこまでが引数領域であるかということを実行時にバイナリから知ることは困難である．

さて，このような誤った再利用を避けるために，既存モデルでは主記憶読み出しは全て絶対番地形式で登録することとし，7 つ以上の引数を持つ関数のメモ化をしないこととしている．引数が 6 つ以下の関数しかメモ化できないが，%sp+92 以上のアドレスからの読み出しは全て呼び出し元関数の局所変数，つまり当該関数の入力と判断できるので，上述のようなある主記憶読み出しが関数の入力かどうか判断できないという状況を回避できる．

では，ある関数が 7 つ以上の引数を受け取るのか否かをどのように判断しているのかを説明する．先ほどの例で用いた FuncA が FuncB を呼び出す時を再び考える．まず，6 つの引数は%o0-%o5 に格納され，最後の 1 つはアドレス%sp+92 にストアされる．こ

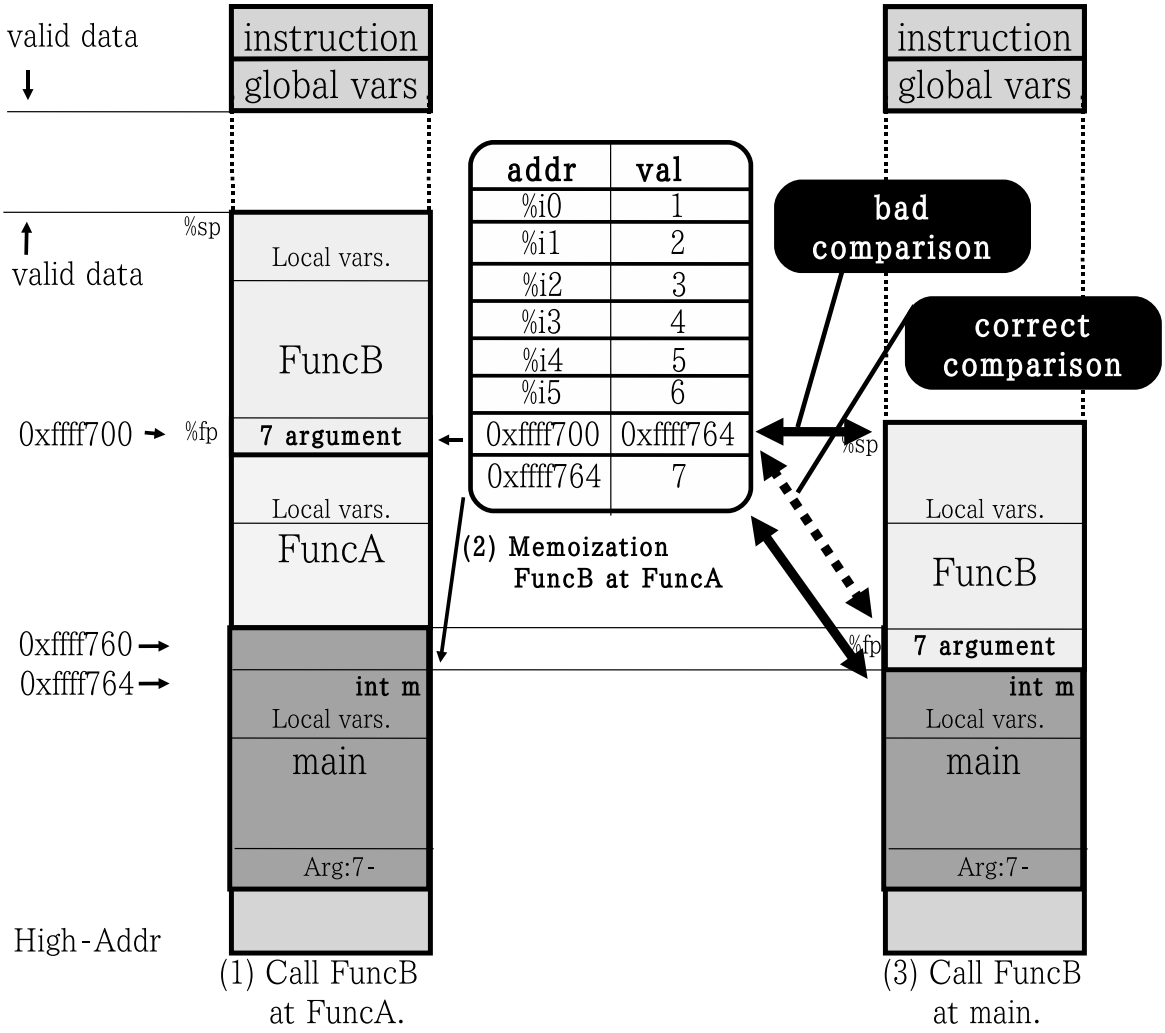


図 12: FuncB のメモ化

のベースレジスタが%sp，オフセットが 92 のストア命令実行を監視することで，7 つ目の引数の受け渡しがあったかどうかを判断することができる．

一方，ARM ABI が定める関数呼び出し規約による関数スタックフレームでは，引数をどの位置に格納すべきかが明確に規定されていない．**図 14** に ARM ABI が定める関数スタックフレームの構造を示す．ARM ABI に含まれる AAPCS(Procedure Call Standard for the ARM Architecture) [8] では，引数の受け渡しに関しては，4 つ目までの引数はレジスタを用い，それ以上の引数は順次スタックに格納するべきという記述がされている．

誤った再利用を避けるために，引数受け渡しにスタックが使われる場合，すなわち ARM においては 5 つ以上の引数を受け渡す場合では，SPARC でのメモ化同様に誤っ

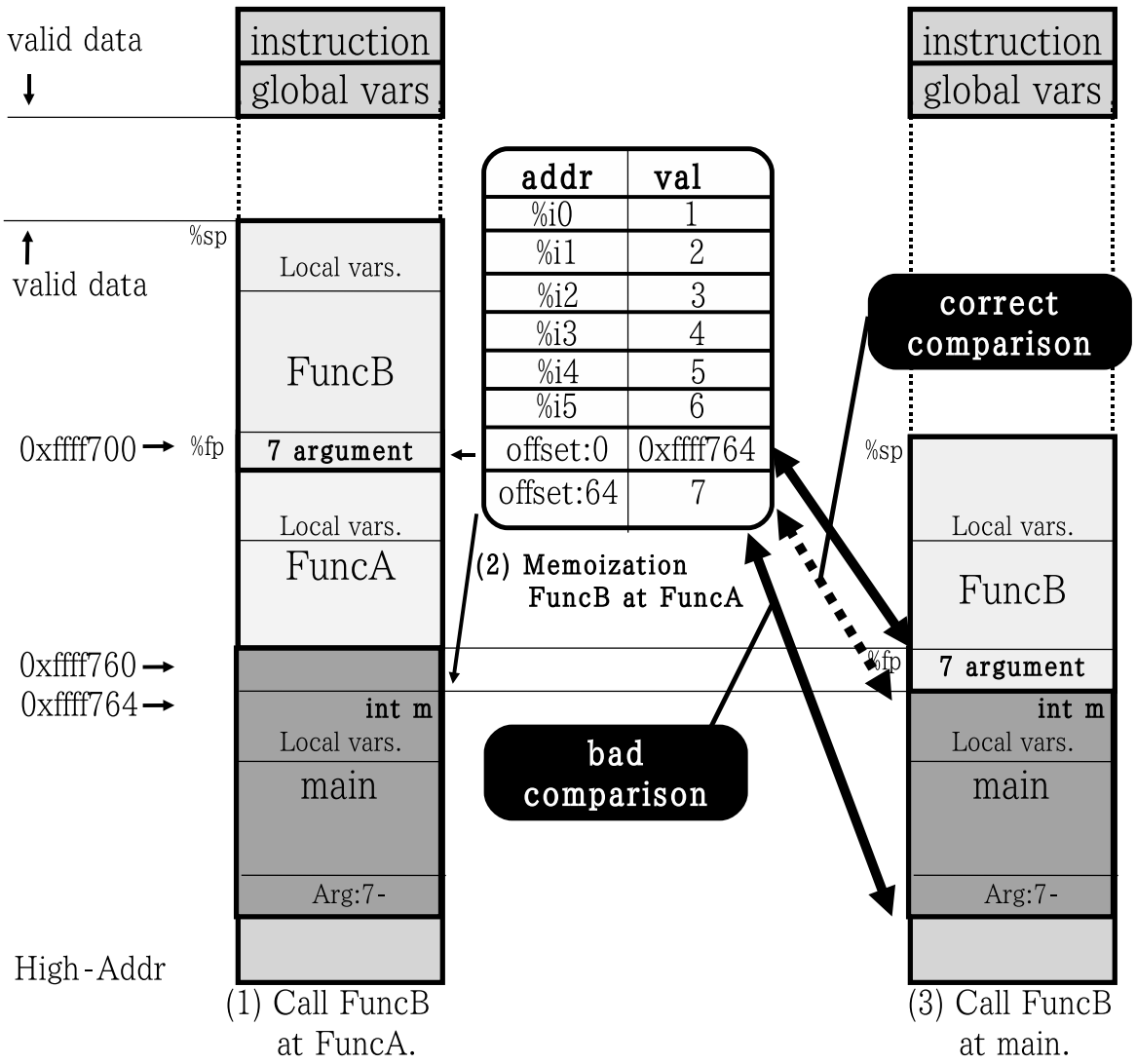


図 13: FuncB のメモ化 (%sp 相対形式)

た再利用を避けるために，メモ化を行わない方法を考える．

FuncA が 5 つの引数を受け取る FuncB を呼び出す時を考える (図 14 中左の状態)．まず，4 つの引数は R0-R3(ARM で引数格納用に予約されているレジスタ) に格納され，最後の 1 つはスタックの一番上にストアされる．しかし，このスタックへのストアは，FuncA が自身の局所変数をただスタックへ待避させただけである可能性がある．SPARC ではベースアドレス%sp，オフセット 92 のストアは引数受け渡しであると暗黙に定まっていたので，局所変数のストアと判別できたが，ARM ではこのどちらかを判断することができない．

このように，SPARC をベースアーキテクチャとしている既存モデル同様に，ARM

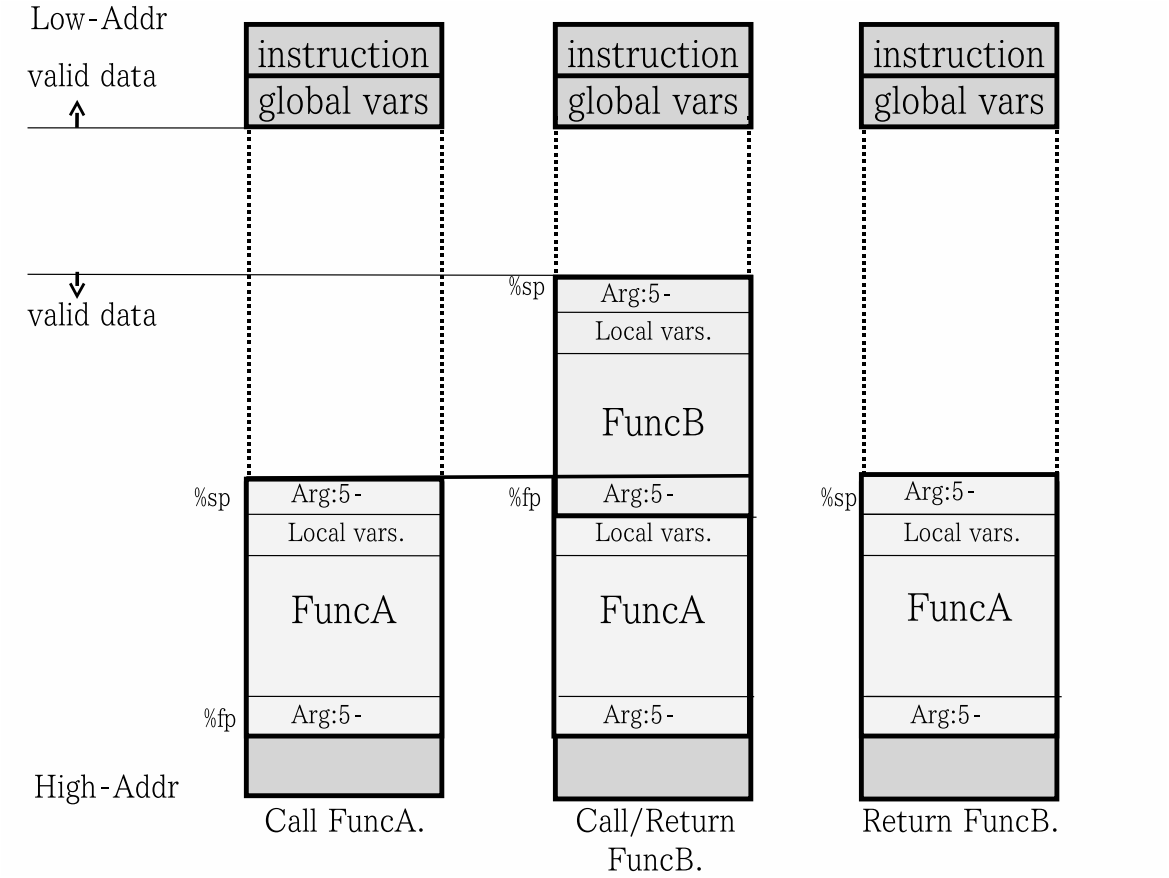


図 14: ARM の関数スタックフレーム

をベースアーキテクチャとして，既存バイナリを対象としたハードウェアによる関数自動メモ化は困難である．

3.2.3　2 つのメモ化形式モデル

以上に述べたように，SPARC との関数呼び出し規約の違いから，ARM では既存モデル同様の手法では関数の自動メモ化を行うことは難しい．そこで，これを解決するために，2 つのメモ化形式モデルを考案した．一つは，**コンテクスト固定メモ化形式**と呼ぶもので，関数が過去に呼び出されたときと同じコンテクストである状態でしか再利用が行えないモデルである．もう一方は，**コンテクストフリーメモ化形式**と呼び，既存モデル同様，関数コンテクストに依存しない再利用が可能なモデルであるが，既存バイナリをそのまま実行することはできない．

ではまず，コンテクスト固定メモ化モデルを用いた場合の動作例を説明する．**図 15** に説明に用いるプログラム例 2 を示す．図 15 中右 (2) の木は図 15 中左 (1) のプログラムの関数 f を実引数 1，2 で呼び出した時の関数コンテクストの移り変わりを表してい

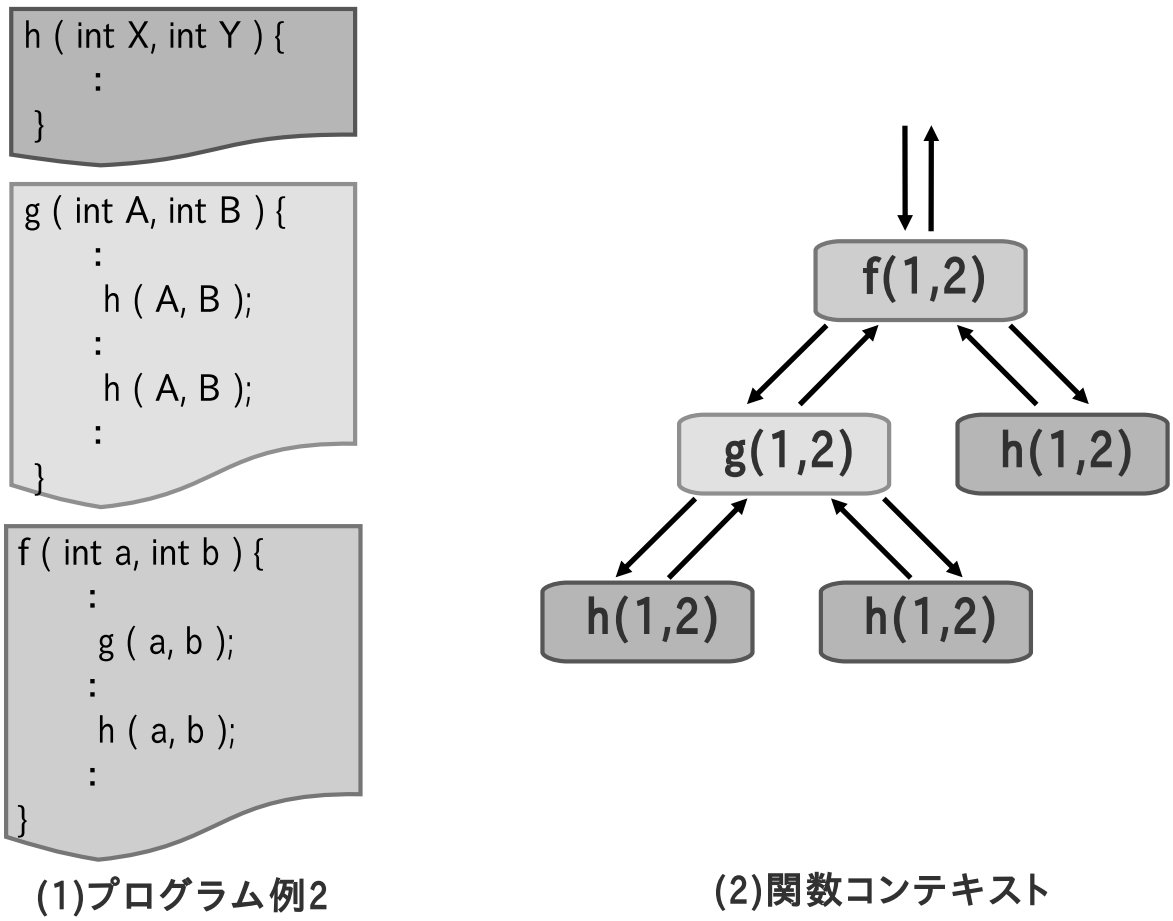


図 15: プログラム例 2 とその関数コンテクスト

る．説明の簡略化のため関数 h のみをメモ化対象とする．**図 16** に，各実行手法においてプログラムが実行される様子を示す．横軸は時間軸を示し，色分けされたグラフは各関数の実行に費やされた時間を意味する．また，h に関しては何番目の呼び出しであるのかを記載してある．図 16(b) はコンテキスト固定メモ化モデルによる再利用の要すを示している．まず，f から呼び出された g からの 1 度目の h 呼び出し時に，h はメモ化される (図 16(1))．その後，g 内で再び h を呼び出そうとするが，先ほどの呼び出し時にメモ化されているので，再利用により h の実行は省略される (図 16 中 (2))．その後，f において h を呼び出そうとする．2 度目の呼び出し同様に再利用により h は実行省略できそうだが，本モデルではこの再利用を行わない (図 16(3))．h のメモ化は f から呼び出された g という関数コンテクストで行われており，3 度目のこの呼び出しとは関数コンテクストが異なる (図 15(2))．本モデルでは関数コンテクストが固定された再利用しか許していないので，この再利用は失敗し，h は通常実行される．

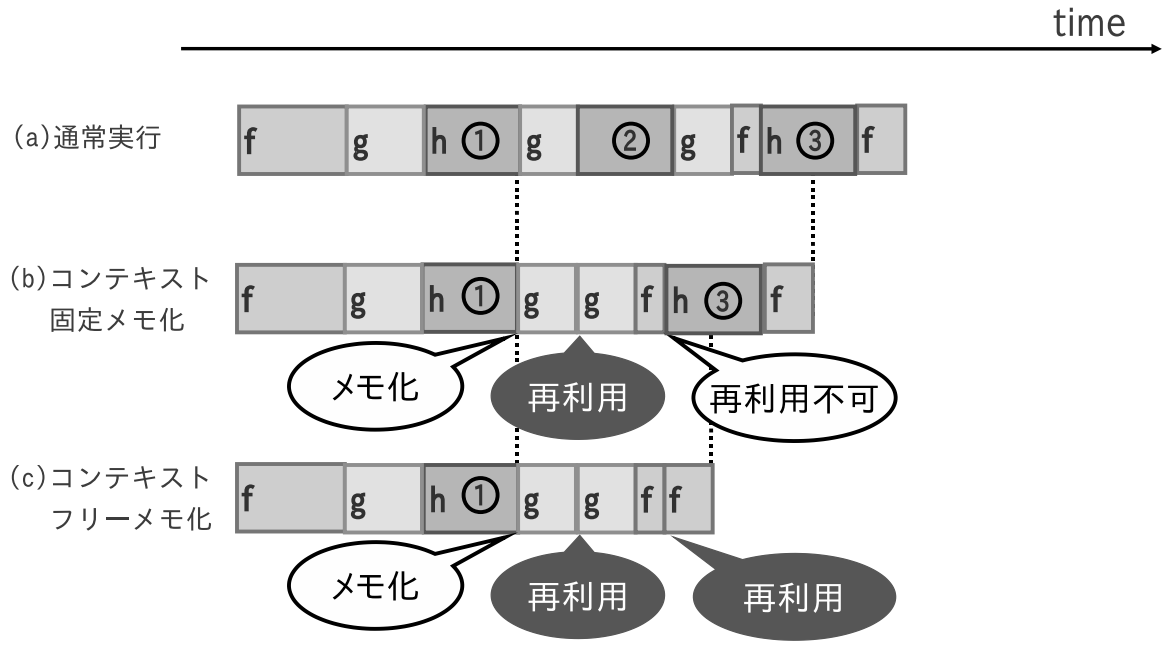


図 16: メモ化モデル毎のプログラム例 2 実行時間

このように，コンテクスト固定メモ化モデルでは再利用適用場面が非常に限定されるので，既存モデルに比べ，再利用率の低下が懸念される．

では次に，コンテクストフリーメモ化形式モデルでの動作例を述べる．図 16 中 (c) にコンテクストフリーメモ化による再利用の様子を示す．図から分かる通り，コンテクスト固定メモ化では再利用できなかった 3 度目の関数 h の再利用に成功している (図 16(3)．本モデルでは，このようにコンテクストに依存しない再利用が可能となっており，先述のコンテクスト固定メモ化に比べて高い再利用率が期待できる．しかし，コンテクストに依存しないメモ化を行うためには，実行前にプログラム中に存在する各関数の引数に関する情報をソースプログラムから抽出しておく必要があり，既存バイナリをそのまま実行することはできない．

以上述べてきた両モデルの利点と欠点についてまとめると，以下のようになる．

コンテクスト固定メモ化

利点　既存バイナリがそのまま実行可能

欠点　コンテクストに依存した再利用しか行えない

コンテクストフリーメモ化

利点　コンテクストに依存しない再利用が行える

欠点　既存バイナリはそのまま実行不可能

4 実装

本章では，提案手法の実装について説明する．まず，本提案手法で扱うベースアーキテクチャの詳細を述べ，その後，そのアーキテクチャ上にどのようにしてメモ化機構を実装するのかについて説明する．

4.1 ベースアーキテクチャについて

まず，命令セットアーキテクチャとして採用した ARM について述べた後に，ベースアーキテクチャの具体的なハードウェア構成について述べる．

4.1.1 ARM 命令セットの特徴

ARM 命令セットは RISC に分類されるが，純粋な RISC である SPARC に比べると複雑な命令を多数持つ．以下，いくつか特徴的な ARM 命令セットの主な特徴について概説する．

条件実行

ARM では全ての命令を条件付きとすることができる．条件は，各命令の先頭にある 4bit の条件コードと呼ばれる領域で指定する．これにより，即値オペランドの指定領域等が大幅に削られてしまうが，分岐命令を削減することができるので，コード密度の向上が期待できる．

多重ロード，ストア

1 つのロード/ストア命令で最大 16×4bytes のデータをロード/ストア可能な命令が存在する．命令の下位 16bits がロード先/ストア元のレジスタを意味する．アドレスは指定したベースレジスタをインクリメントしていくことで求められる．連続したメモリ領域へのアクセス (例えば，作業レジスタをスタックへ待避させる場合等) が必要な場合，大幅にコード密度が向上する．

インライン・バレル・シフタ

バレル・シフタはある特定の bit 数分だけデータをシフト可能なデジタル回路である．ARM ではこれをインライン (直列に) 演算器に接続しており，オペランドが演算器で処理される前にシフト処理を加えることが可能である．シフト処理の後に加算を実行するといった，複雑な命令が 1 命令で実行可能となる．

Thumb16bit 命令セット

ARM はプロセッサコアを拡張し Thumb とよばれる第 2 の 16bit 命令セットを追加している．このため ARM プロセッサは 16bit 命令も 32bit 命令も実行でき，16bit

表 1: ARM の汎用レジスタ

レジスタ番号	使用用途
R0-R3	引数渡しレジスタ 1/作業レジスタ/返り値格納レジスタ
R4-R8	一時レジスタ
R9	一時レジスタ/静的ベースレジスタ
R10	一時レジスタ/スタック制限/スタックチャンク処理
R11	一時レジスタ/フレームポインタ
R12	ip(instruction pointer)/作業レジスタ
R13	sp(stack pointer) スタックフレームの最下部
R14	lr(link register)/作業レジスタ
R15	pc(program counter)
cpsr	条件コードレジスタ
spsr	条件コードレジスタの内容を待避するレジスタ

命令を用いることでコード密度を高める事ができる．

拡張命令

拡張命令拡張ディジタル信号処理（DSP）命令が標準的な命令セットに追加されている．そのため高速に DSP 処理ができる．

以上あげたように，ARM 命令セットはコード密度を向上させるために随所に工夫がされている．これは，ARM がメモリ制限の厳しい組み込み用途に用いられるためである．

4.1.2 ARM の論理レジスタ

ユーザモードでアクセス可能な論理レジスタを表**表 1** に示す．これは，ユーザモードすなわちアプリケーション実行時に通常使用される保護モードにおいて，アクティブなレジスタを示している．ユーザーモードでアクセス可能な論理レジスタは 18 個あり，データレジスタが 16 個，プログラムステータスレジスタが 2 個ある．プログラマは R0-R15 のレジスタを使用することができる．その中でも R0-R3 の 4 つのレジスタは関数の引数渡し及び返り値受取りに用いられる．それとは別に ARM プロセッサでは特別な役割を担うデータレジスタが 3 つある．それは R13，R14，R15 のレジスタである．順に R13 はスタックポインタ (sp) の格納場所であり，R14 はリンクレジスタ (lr) と呼ばれ，サブルーチンを呼び出す時にに現在のプログラムカウンタ (pc) を格納

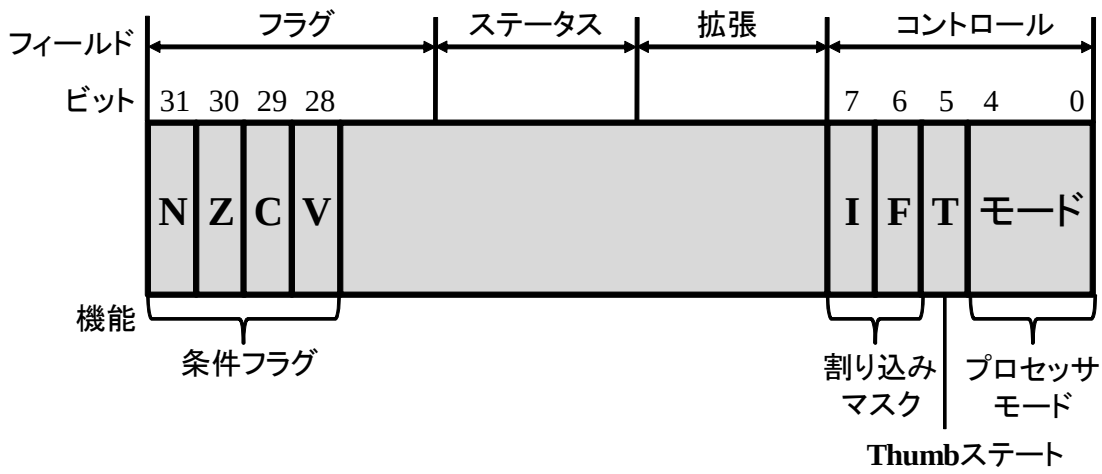


図 17: プログラムステータスレジスタ

し，サブルーチンからの復帰時に使用する.R15 はプログラムカウンタ格納レジスタであり，プロセッサが次にフェッチするアドレスを保持する．

また，ARM にはプログラム・ステータス・レジスタというものがある．プログラム・ステータス・レジスタはには cpsr(current program status register) と spsr(セーブド・プログラム・ステータス・レジスタ) の 2 つがある．spsr は cpsr を保存するために用いられる．一般的なプログラム・ステータス・レジスタの基本レイアウトを**図 17** に示す．cpsr はフラグ用，ステータス用，拡張用，コントロール用の 4 つのフィールドに分かれ，それぞれが 8bits 幅である．拡張用フィールドとステータス・フィールドは将来の使用に備えて予約されている．コントロール・フィールドには，プロセッサ・モード・ビット，Thumb ステート・ビット，割り込みマスク・ビットがあり．フラグ・フィールドには条件フラグがある．プロセッサ・モードはどのレジスタがアクティブで，cpsr レジスタ自体へのアクセス権を持つかを決定する．プロセッサ・モードは特権もしくは非特権モードのいずれかであり，特権モードでは cpsr すべてのフィールドの読み書きが可能である．非特権モードの場合，cpsr のコントロールフィールドは読み出しのみが可能である．ただし，条件フラグは読み出し書き込みが可能である．

プロセッサモードは合計 7 つあり，6 つの特権モード (アボート，高速割り込み要求，割り込み要求，スーパバイザ，システム，未定義) と 1 つの非特権モード（ユーザ）である．プロセッサはメモリアクセスに失敗するとアボートモードになる．また，高速割り込み要求モードと割り込み要求モードは ARM プロセッサの 2 つの割り込みレベルに対応する．スーパバイザモードは，リセット後のプロセッサモードであり，一般に

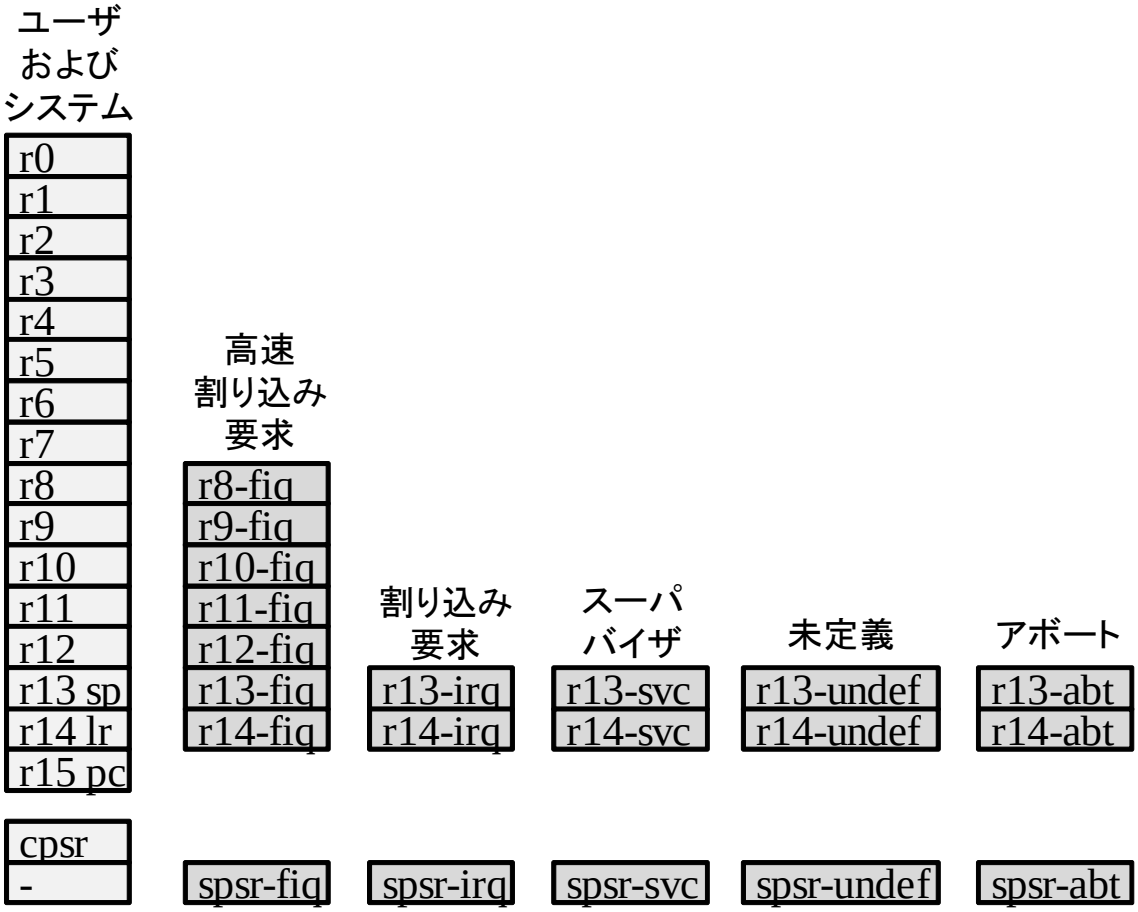


図 18: バンクレジスタ

システムのカーネルが動作するモードでもある．システムモードはユーザモードの特殊版で，cpsr への読み出し，書き込みを許容する．未定義モードは，未定義命令の実行時のモードであり，ユーザモードはプログラムとアプリケーションの両方で使用されるユーザアプリケーションが一般に実行されるモードである．また，**図 18** にレジスタファイルを構成する 37 個のレジスタを示した．このうち 20 個のレジスタはプロセッサモードによってプログラムから隠される (図 18 中の濃い灰色部分)．このようなレジスタをバンクレジスタと呼ぶ．図 18 はユーザおよびシステムモード時を示しており，プログラムは左端のレジスタ群以外にはアクセスすることはできない．ユーザモードを除いたすべてのプロセッサモードでは，cpsr のモードビットを直接書き変えることでプロセッサモードを変更することができる．またバンクレジスタはユーザモードのレジスタと 1 対 1 に対応しており，プロセッサモードを変更すると，既存モードのバンクレジスタと新しいモードのバンクレジスタが入れ替る．図 18 には，割り込みモー

ドで現れる新しいレジスタも示している．前のモードに戻るときに使用するレジスタである spsr(saved program status register) は，特権モード時しか変更できず，ユーザモードには存在しない．

4.1.3 ベースアーキテクチャのハードウェア構成

前節では，一般的な ARM プロセッサの仕様について述べた．本説では，実際に提案手法を実装するベースアーキテクチャ「OROCHI[9]」について詳しく説明する．OROCHI は VLIW 命令と ARM 命令を同時実行可能なスーパースカラ型プロセッサである．一般に，プログラムが持つ ILP には限界があり，高性能なコンパイラを用いたとしても VLIW 命令内には多数の NOP 命令が混入してしまい，プロセッサの使用率は低下してしまう．そこで OROCHI は，NOP 命令により空いている演算器に元々ILP が期待できないようなスレッド (例えば Kernel スレッド) の命令を投入することにより，プロセッサ全体の使用率を向上させるということを意図して設計された．つまり，OROCHI プロセッサは VLIW 命令と ARM 命令それぞれ別に，命令フェッチ，デコード等を行うフロントエンドを持ち，実行ステージを含むバックエンドは 1 つというハードウェア構成になっている．今回，提案手法を実装する上で，この VLIW 命令用のフロントエンドは必要ないので，無効化している．

図 19 に VLIW 命令用のフロントエンドを除いたベースアーキテクチャのハードウェア構成を示す．ベースアーキテクチャのパイプラインは全 19 段で構成され，2 次キャッシュは命令キャッシュと 1 次データキャッシュに共有される．次に，各パイプラインステージについて簡易的に説明する．

IA(Instruction Address)

プログラムカウンタの計算を行う．

IF(Instruction Fetch)

命令キャッシュから命令をフェッチする．また，g-share による分岐予測を行う．

ARM-D(Arm-Instruction Decode)

ARM 命令を高速に実行可能な内部命令に分解する．例えば，マルチ LD 命令は，複数の 4byte の LD 命令に，乗算命令などは，複数の加算命令とシフト命令に分解される．

HOST-D(Host-Instruction Decode)

条件実行命令を二命令に分解する．一方は，条件が真の場合に実行される命令であり，もう一方は条件が偽の場合の命令 (つまり何も実行せず，ディスティネーションレジスタが持つ値をパスするだけの命令) である．このように分解することで，

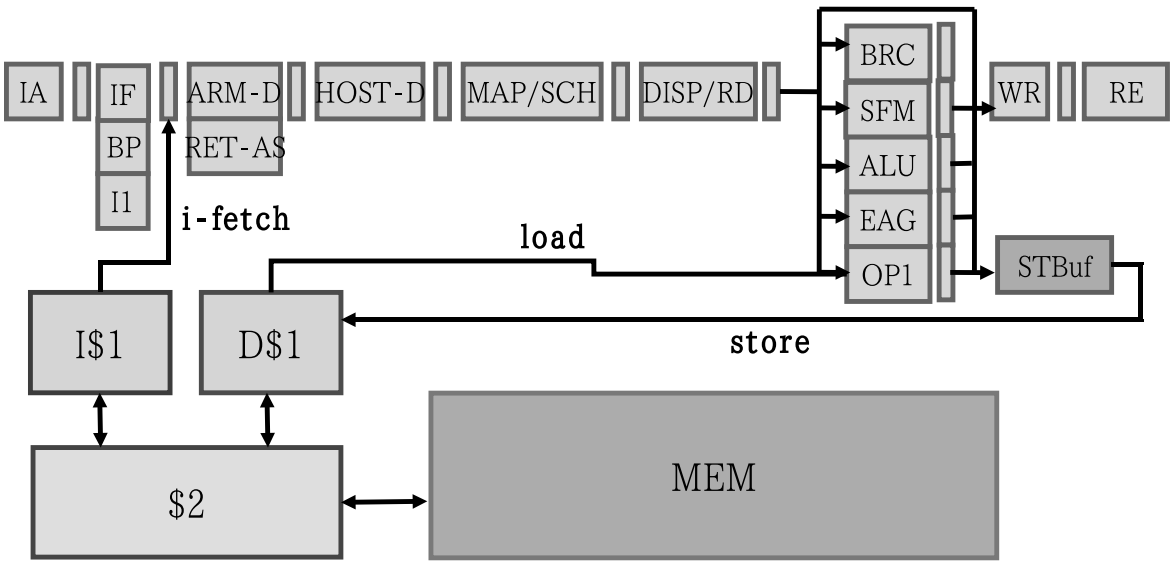


図 19: ベースアーキテクチャのハードウェア構成図

条件実行に必要な条件フラグの値が定まっていない時点でも，命令の実行が可能となる．最終的に，Retire ステージで条件フラグが参照され，条件に合致する方の命令はコミットされ，もう一方の命令は破棄される．

MAP/SCH(Register mapping/Schedule)

各命令のオペランドに対して，汎用レジスタ (EREG) と拡張レジスタ (IREG) で構成される論理レジスタから，命令ウィンドウと物理レジスタを兼ねる RoB(Reoder Buffer) へマッピングされる．また，同時に各命令のスケジューリングも行われる．

SEL/RD(Select and Read)

命令の発行 (Issue) を行うステージである．各演算ユニットからオペランドのバイパスが可能かどうか調べ，依存関係にあるオペランドを持つ命令の発行を制御する．

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．また，本ステージの各ユニットは前段に 5 段の入力 FIFO を，後段に 5 段の出力 FIFO を持っている．以下，各ユニットについて記述する．

BRC　分岐予測の正誤を判定する．予測が誤っていた場合，パイプラインをストールさせ，正しい命令を再フェッチするよう信号を発行する．

SFM　シフト演算を処理する．また，積和補助演算の処理も担当する．

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，積和補助演算の処理も担当する．

OP1　ロード，ストア命令を処理する．

WR(WriteBack)

RoB への書き込み処理を行う．

RE(Retire)

先行命令がすべて完了した命令を RoB から論理レジスタへ書き戻し処理を行う．なお，分岐予測ミス時等に実行再開をすみやかに行うために，命令がリタイアされる順番は，分解前命令列と同じであることが保証されている．

本ベースアーキテクチャでは ARM 命令を RISC 型内部命令へ分解する．これは，本ベースアーキテクチャのモデルが，異種命令混在実行を目指して設計されたプロセッサであるからである．また，命令分解を採用しているのは，ARM 命令セットが RISC 設計でありながら，1 命令で複雑な処理も可能な命令形態をとっていることも理由の一つである．パイプラインで命令を効率よく実行するには，命令が常に同じサイクルで実行されるほうが良いとされている．そのため，本モデルでは ARM 命令を内部命令に変換する手段をとっている．一方で既存の命令分解機構に，Intel 社の Netburst 機構 [10] があるが，本モデルの分解機構は演算機をより単純化している．

前項では，一般的な ARM プロセッサが持つ論理レジスタについて説明してきたが，ベースアーキテクチャには，これらのレジスタに加えて，6 個の拡張論理レジスタが存在する．この拡張論理レジスタは，ARM 命令分解後の内部命令が使用する論理レジスタであり，この内部命令用のレジスタを IREG（Inplicit Register）と呼び．対して ARM プロセッサで規定される R0-R15 までの汎用論理レジスタを EREG（Explicit Register）と呼ぶ．IREG は，内部命令がデータを一時保存するために用いられる．

4.2　入力値検索のオーバヘッド削減手法の実装

ベースアーキテクチャをスーパースカラ型プロセッサとしたことで可能となった入力値の検索によるオーバヘッド削減手法の実装について述べる．

図 20 は関数 f が呼び出す関数 g の入力値検索が失敗に終わる時の様子を示している．横軸は時間を意味し，3 本のグラフは，上から (1) メモ化無しの通常実行，(2) 通常のメモ化モデルでの実行，(3) 再利用オーバヘッド削減モデルでの実行における関数呼び出し前後の様子を示す．まず，通常のメモ化モデルでの実行の様子について述べる．関数 f 実行中に g 呼び出し命令が Retire されると，全てのパイプラインステージを停止させる (図 20 中 (1))．その後，入力の一致比較が始まる．この時，OP1 ユニッ

トからのロード/ストア要求をキャッシュコントローラが受け付けている場合，要求の処理が完了するまで，入力値の一致比較はまだ開始されない (図 20 中 (2))．要求の処理が完了すると，入力値の一致比較が開始される．g の入力値検索は失敗に終わるので，結局入力値検索のかかった時間と，キャッシュコントローラのレディ待ち時間分だけ，メモ化無しの通常実行より実行時間が増えてしまっている．

次に，再利用オーバヘッド削減モデルでの実行の様子について述べる．まず，g 呼び出し命令が Retire されると，Retire ステージに Retire 禁止シグナルが送られる．これは，入力値の一致比較対象のレジスタやキャッシュが上書きされるのを防ぐためである (図 20 中 (1))．通常のメモ化モデル同様に，OP1 ユニットからのリクエストをキャッシュコントローラが受け付けている場合，リクエストを処理するまで，入力値の一致比較は開始されない．一方，Retire ステージ以外のパイプラインステージは動作しており，g 内の命令は解釈実行されていく (図 20 中 (2))．キャッシュコントローラがレディとなると，以降キャッシュコントローラは入力値の一致比較のための読み出し要求のみを受け付け，OP1 ユニットからのロード/ストア要求を無視するようにする．このようにすることで，パイプラインに命令の解釈実行をさせながら，入力値検索を行い，入力検索によるオーバヘッドを隠蔽する．しかし，Retire を禁止した状態なので，入力値の一致比較対象が多かったり，キャッシュミスが多発した場合，Reorder Buffer に空きが無くなり，stall が発生する．(図 20 中 (3))．このため，どこまでオーバヘッドを隠蔽できるかは，Reorder Buffer のエントリ数に依存することになる．

4.3　2 つのメモ化形式モデルの実装

3.2.2，及び 3.2.3 で述べたように，実行中の ARM バイナリからは，スタックからの読み出しが引数参照なのか呼び出し元関数の局所変数参照であるのかを判断することができない．そのため，関数入力値を正しいアドレス形式で登録することができない．その結果，既存モデル同様のメモ化手法では誤った入力値の比較が発生し，間違った再利用が行われる可能性がある．これを回避するために提案した 2 つのメモ化形式モデルの実装手法について述べる．

4.3.1　コンテクスト固定メモ化

誤った入力値の比較は，過去に関数をメモ化した時とコンテクストが異なる状態で入力値の検索を行った場合に発生する．これを避けるために，コンテクスト固定メモ化モデルは，関数のコンテクスト情報も関数の入力値として扱うことで，メモ化時と異なるコンテクストにおける入力値検索を強制的に失敗させる．本モデルで扱う関数

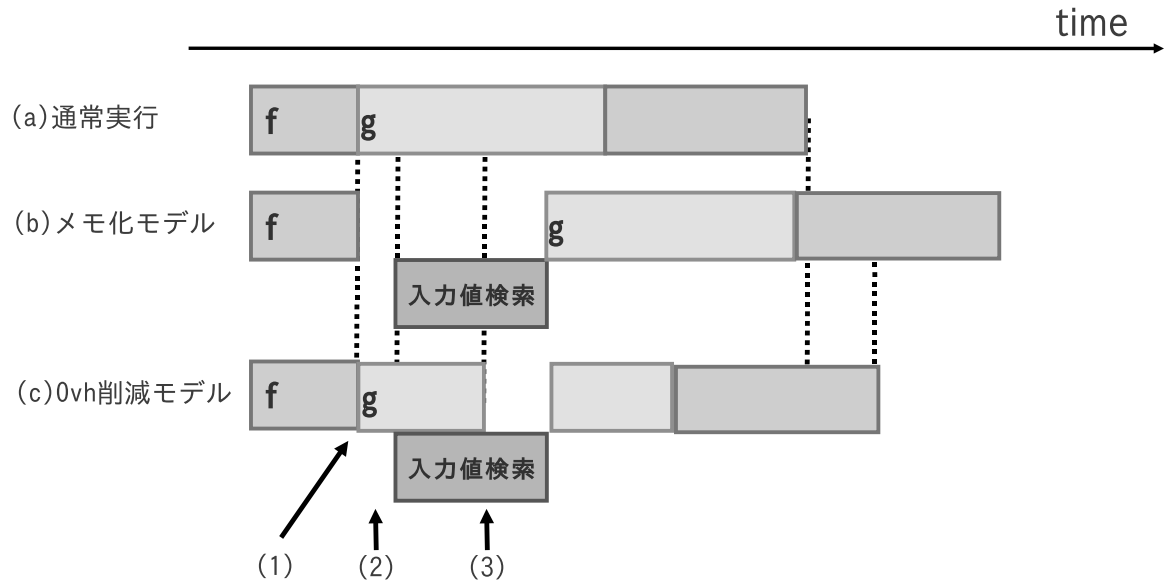


図 20: 入力値検索オーバヘッド削減時のプロセッサ状態

のコンテクスト情報は，具体的には関数開始時における sp(R13) の値である．ARM の関数プロローグにおいて，関数内で扱われる局所変数を考慮に入れたスタックフレームが確保される．そのため，sp の値が過去に関数を実行した時と一致すれば，関数コンテクストの一致が保証される．以下，**図 21** のプログラム例を用いて具体的な実装について述べる．

2.6 節で述べたように，関数の入力アドレスは先行する入力値に依存する．そのため，様々な入力パターンでメモ化された関数の入力値系譜は木構造で表す事ができる．コンテクスト固定メモ化モデルでプログラム例 3(図 21) の関数 FuncB をメモ化した際に行われる関数入力登録の様子を，木構造を用いて**図 22** 中 (2) に示す．プログラム例 3 では 5 つの引数をとる FuncB は FuncA において 2 回，main において 2 回の計 4 回呼び出される．図 22 中 (2) の各リーフ下の番号が何回目の呼び出しに対応しているかを示している．また，FuncB が各関数コンテクストで実行される時の関数スタックフレームを図 22 中 (1) に示す．

1 回目の FuncB 呼び出しで，レジスタを介して渡される第 1 引数-第 4 引数までが順に入力として登録される．次に，第 5 引数はスタックを介して渡されるが，第 5 引数を入力として登録する前に関数コンテクスト情報である sp の値を入力として登録する．その後，第 5 引数が入力として登録される．最後に，FuncB 内で参照される e の実体 (main 関数内の m) がアドレス 0xfff764 で登録される．2 回目呼び出しは，第 3，第 4

```
 1:int FuncA(int *A)
 2:{
 3: FuncB(1, 2, 3, 4, &A);    // … 1 回目
 4: FuncB(1, 2, 30, 40, &A);  // … 2 回目
 5:   return(0);
 6:}
 7:
 8:int FuncB(int a,int b,int c,int d,int *e)
 9:{
10: return(a + b + c + d + *e);
11:}
12:
13:main()
14:{
15: int m=5;
16: FuncA(&m);
17: FuncB(1, 2, 3, 4, &m)     // … 3 回目
18: FuncB(1, 2, 30, 40, &m)   // … 4 回目
19: return();
20:}
```

図 21: プログラム例 3

引数が異なるので入力値検索に失敗し，メモ化される．第 3，第 4 引数が異なるだけで，メモ化の過程は一回目の呼び出しと同じである．3 回目の呼び出しは，1 回目の呼び出しで登録された入力系譜と第 1-4 引数まで一致するが，次の sp の値が一致しないので入力値検索は失敗に終わる．また 4 回目の呼び出しも，2 回目の呼び出し登録された入力系譜と第 1-4 引数まで一致するが，次の sp の値が一致しないので入力値検索は失敗に終わる．

以上に述べたように，引数受取りなのか呼び出し元関数の局所変数なのか判断できない領域 (関数開始時点における sp 以上の領域) からの読み出しがあった場合，メモ化機構は関数コンテクスト情報として sp の値を関数入力に含める．これによって，以後，同関数の入力値検索で関数コンテクストの一致比較が行われることになり，間違った入力値検索を避けることができる．なお，sp 以上の領域からの読み出しがない関数については，間違った入力値検索の可能性はない．本モデルで sp 以上の領域からの読み

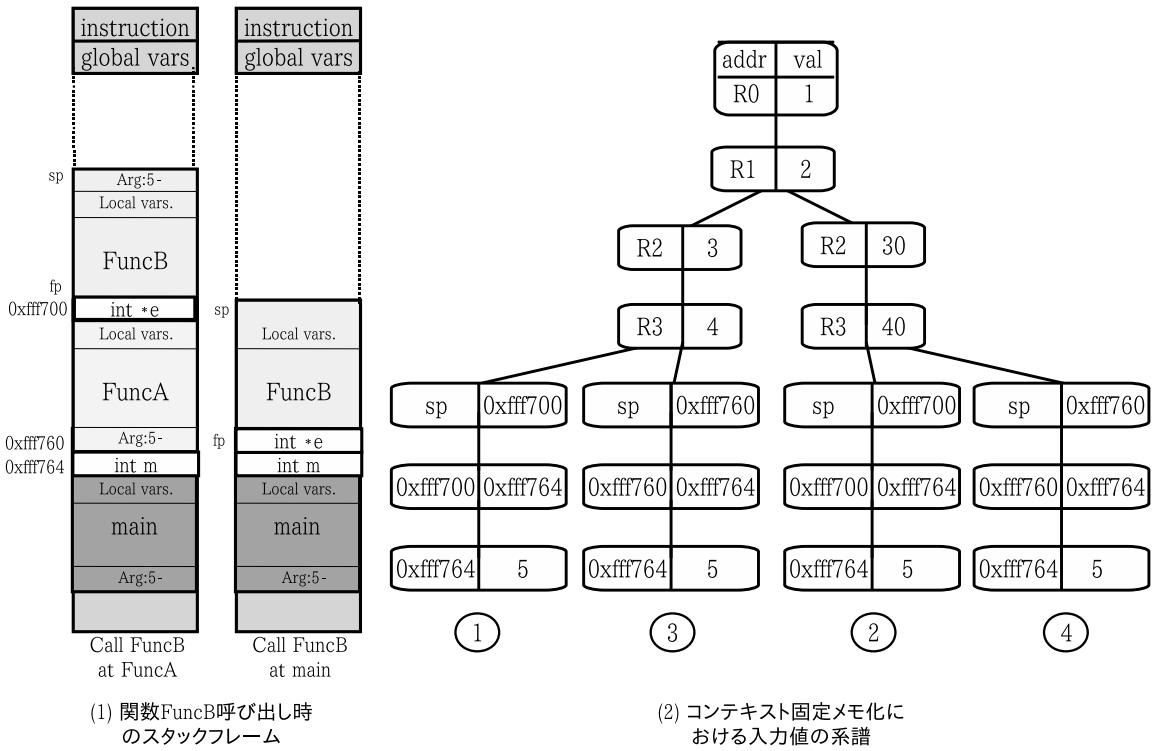


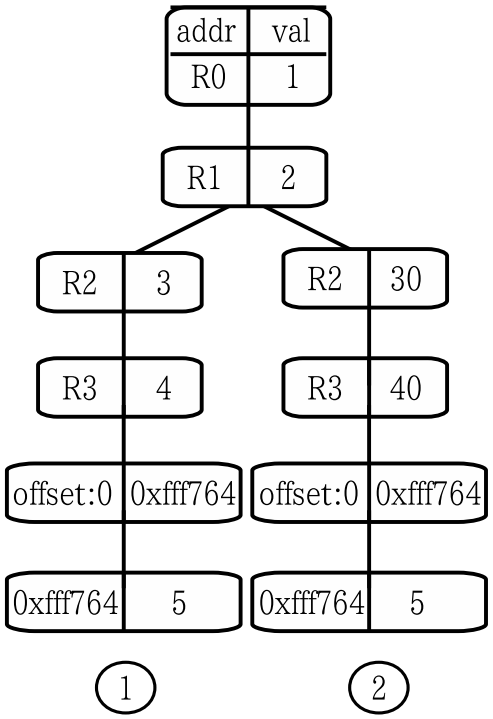
図 22: コンテクスト固定メモ化モデルでの関数入力登録

出しがあった時にのみ，コンテクスト情報を入力系譜の木に挟む方法をとっているのは，そのような関数についてはコンテクストに依存しないメモ化を行うためである．

4.3.2 コンテクストフリーメモ化

前節で扱ったコンテクスト固定メモ化モデルは関数コンテクストが異なる場合，強制的に入力値検索を失敗させることで，誤った再利用を避ける．当然，関数コンテクストが過去と同じでないと再利用ができないため，コンテクストに依存しない再利用が可能な既存モデルに比べ，再利用率の低下が懸念される．本節で扱うコンテクストフリーメモ化モデルは，誤った入力値検索を避けるために，静的なソフトウェア支援を用いる．実行対象プログラム中の各関数が，スタックフレーム上のどの領域を引数受け渡しに用いるのかという情報を，事前に実行対象プログラムから抽出しておく．この抽出されたヒント情報を実行時に参照することにより，正しいアドレス形式で関数入力値を登録し，間違った入力値検索を避ける．以下から，その具体的な実装手法について説明する．

まず，ヒント情報を抽出する方法を述べる．GNU gcc により実行対象プログラムのソースプログラムをコンパイルして生成されたアセンブリソースプログラムには，アセ



(1) コンテキスト固定メモ化における入力値の系譜

```
                    :
        .ascii   "modification by `asm'¥000"
        .text
        .align  2
        .global c_expand_asm_operands
        .type    c_expand_asm_operands, %function
c_expand_asm_operands:
        @ args = 12, pretend = 0, frame = 0
        @ frame_needed = 1, uses_anonymous_args = 0
        mov      ip, sp
        stmfd    sp!, {r4, r5, r6, r7, r8, sl, fp, ip, lr, pc}
        sub      fp, ip, #4
        sub      sp, sp, #12
        mov      r6, r0
        mov      r0, r1
        mov      sl, r3
        mov      r5, r1
        mov      r8, r2
        bl       list_length
        ldrb     r3, [r6, #8]    @ zero_extendqisi2
                    :
                    :
```

(2) GAS例

図 23: コンテクストフリーメモ化モデルでの関数入力登録

ンブラやプログラマのための情報がコメントとして記述されている．このコメントをヒント情報抽出に利用する．**図 23** 中 (2) は，実際に gcc でコンパイルされたアセンブリソースプログラムの一部である．GAS(Gnu ASsembler) 記法 [11] ではターゲットが ARM の場合，@がコメントアウト行を意味する．6 行目から始まる関数 c_expand_asm_operands は，7 行目の args = 12 からスタックフレームの 12bytes の領域を介して引数を受け取る関数であることが分かる．また，8 行目の uses_anonymous_args = 0 は本関数が可変長引数をとる関数でない事を示している．もし，uses_anonymous_args = 1 であれば，その関数は可変長引数をとる関数である．このように，アセンブリソースプログラムのコメントからヒント情報が抽出できる．ヒント情報の抽出と整形は自作の perl スクリプトにより行う．

次に，実際にヒント情報を参照し，関数の入力値が登録される様子を前節でも用いた図 21 のプログラム例を用いて述べる．先程と同様に，FuncB がメモ化される際の入力値登録を考える．まず，一度目の関数 FuncB が呼び出された時に，ヒント情報が参照さ

れRF及びMemoBufにキャッシュされる．以後，メモ化機構はこのキャッシュを参照することによりヒント情報を得る．int型へのポインタである第5引数のために，スタックフレームが4bytes使われることが分かる．第1-4引数は前節のモデル同様に順に登録される (図23中(1))．次に，第5引数であるアドレス0xfff700参照時に，参照アドレスが`FuncB`開始時のsp (図22中(1)左) +4(0xfff700 + 4 = 0xfff704)未満の領域であることから，メモ化機構はこれを引数であると正しく認識する．節3.2.2で述べたように，スタックを介して受け渡しされる引数は関数開始時点でのspからの相対アドレス形式で登録されるべきであり，呼び出し元関数の局所変数は絶対番地アドレス形式で登録されるべきである．そのため，第5引数はアドレス0という形で登録される．最後に，eの実体であるmain関数内のaが登録される．この参照はsp+4(0xfff700+4 = 0xfff704)以上の領域であるので，メモ化機構はこれを呼び出し元関数の局所変数参照であると正しく認識し，絶対番地形式で登録する．2度目の呼び出しも1度目の呼び出し時と同様に登録される．

3度目の呼び出し時の入力値検索では，レジスタを介して受け渡しされる第1-4引数の比較はこれまでどおり行われ，その後第5引数の比較が行われる．現在の関数コンテクストにおける関数開始時点のsp(図22中(1)右) +offset(0xfff760 + 0 = 0xfff760)で第5引数の格納アドレスは求められ，一致が確認される．このように，スタックを介して受け渡しされる引数のアドレスは現在の関数コンテクストでのspに対してメモ化時に計算したoffsetを加えることで求められる．最後にeの実体参照の一致が確認され，再利用が適用される．また同様に，4度目の呼び出しは2度目の呼び出しと入力が全て一致するので再利用が行われる．

以上述べてきたように，コンテクストフリーメモ化モデルではコンパイル時のコメント情報から，各関数の引数が占有するスタック領域を抽出する．メモ化機構はこの情報を実行時に参照することで，引数は正しい形式で入力値として登録され，間違った入力値検索は避けられる．

4.4 メモ化に必要な主な追加ハードウェアについて

本節では，メモ化に必要な追加ハードウェアについて述べる．**図24**に提案するモデルのハードウェア構成を示す．MemoTbl及びMemoBufの基本的な構造は既存モデルに準ずるものとする．

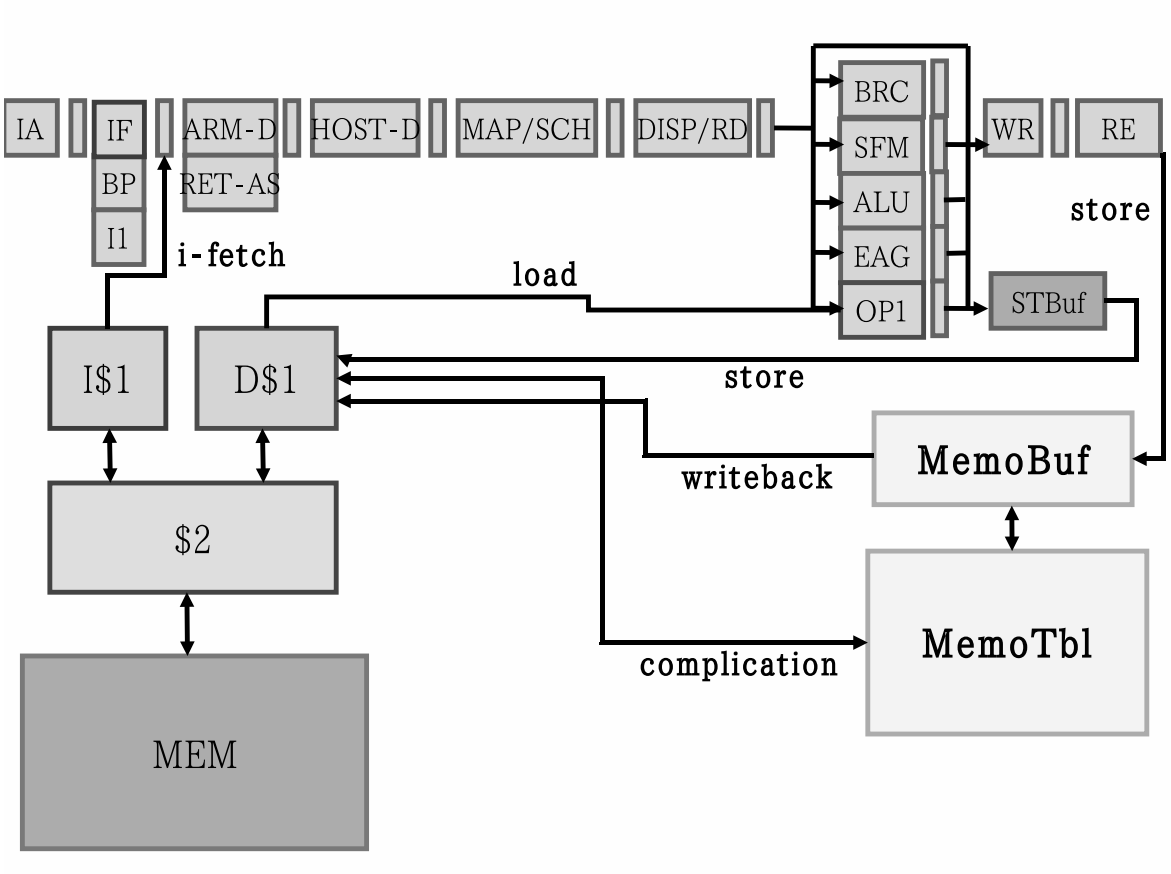


図 24: 提案モデルのハードウェア構成

4.4.1 関数の入出力登録，及び入力値検索開始ユニット

入力値の検索の開始タイミングでは，比較対象の値がストアされていることを保証する必要があるため，関数呼び出し命令が Retire された時点としていることは，以前述べた．また，関数呼び出し命令の Retire 時に後続する命令の Retire が可能であったとしても，入力値を上書きしてしまう可能性があるので，後続命令の Retire はキャンセルされる．

関数の入出力登録のタイミングもまた，入出力の原因となる命令が Retire された時点である．命令が Retire されるより前に入出力の登録を行うと，分岐予測ミスなどで実行された命令がキャンセルされた場合，それに応じて MemoBuf に登録した入出力をキャンセルする必要があり，ハードウェアが複雑となってしまうためである．

以上のように，関数の入出力登録，及び入力値検索開始ユニットは，Retire ステージを拡張することで実装される．

	必要な情報		リタイア時に得られる情報	
	アドレス	値	アドレス	値
入力	R0	0x1000	R0	N/A
	R1	0x2000	R1	N/A
出力	0x2004	0x1000	0x2004	0x1000

表 2: 関数の入出力登録情報

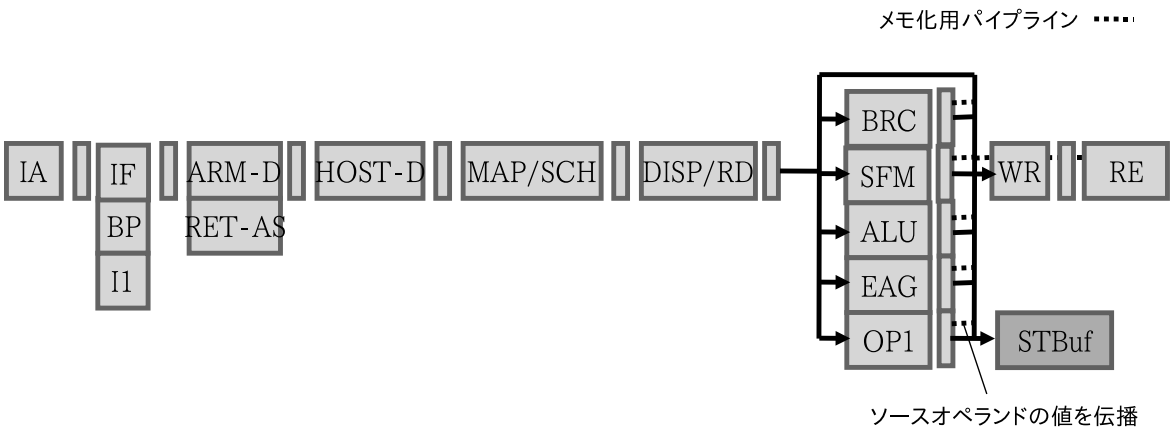


図 25: メモ化用パイプラインレジスタ

4.4.2 メモ化用パイプラインレジスタ

前項で述べた通り，関数の入出力登録は Retire ステージで行われる．しかし，入出力情報は Retire ステージに至るまでに失われてしまうものが多い．関数の入出力情報とは，関数内で読み書きされた値とアドレスの組である．

今、ある関数が 2 つの引数を受け渡されて (R0 に 0x1000，R1 に 0x2000) 呼び出されたとする．そして，関数呼び出し直後に str R0, [R1 #4] という命令が実行された場合を考える．この命令による関数入出力は**表 2** に示す通りである．この命令が Retire される時点でのパイプラインレジスタ及び Reorder Buffer からは 2 つの入力の値が失われている．これは，通常この命令を Retire するには出力先のアドレスと値のセットが必要であり，入力オペランドに関する情報は必要でないためである．

そこで，ソースオペランドの値を Retire ステージまで伝播させるために，**図 25** に示すように，メモ化用パイプラインレジスタを追加する．ソースオペランドの値は，DISP/RD(Dispatch & Read) ステージでは物理レジスタ読み出し，論理レジスタ読み出し，WR(Write) ステージからのバイパス，及び RE(Retire) ステージからのバイパ

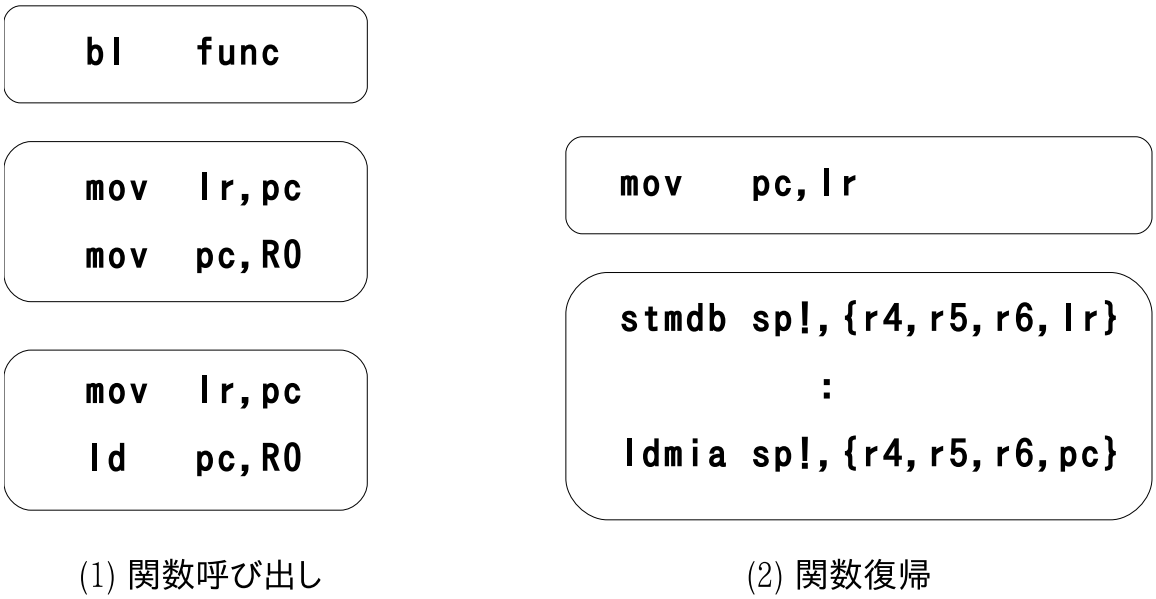


図 26: ARM の関数呼び出し，復帰コード

スによって供給され，実行ステージでは前 cycle の実行ステージからのバイパスにより供給される．そして，実行ステージ以降はソースオペランドの値は破棄され，後段に伝播されない．つまり，実行ステージ以降からリタイアステージまでソースオペランドの値を伝播させるパイプラインレジスタを追加すれば良い．

4.4.3 関数呼び出し，復帰検知デコーダ

関数をメモ化するためには，関数の呼び出し及び復帰を検知する必要がある．既存モデルが採用している SPARC の ABI では関数呼び出しは call 命令，復帰命令は ret 命令と規定されている．そのため，call 命令と ret 命令の実行を監視することにより，メモ化対象区間を特定している．

一方，ARM ABI では関数呼び出し及び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．関数呼び出し命令として bl(branch and link) 命令が存在するが，関数呼び出し時には必ずしもこの命令を用いなければならないわけではない．**図 26** に ARM の関数呼び出し，及び復帰コードを示す．このように，ARM の関数呼び出し，復帰コードは多岐に渡るので，既存モデル同様に単純に特定の命令を監視するだけでは，メモ化対象区間を特定することができない．そのため，メモ化対象区間特定のために，既存モデルに比べ複雑な順序回路を用いる必要がある．

関数呼び出し，復帰を検知するために，デコードステージ (ARM-D 及び HOST-D) とリタイアステージに，関数呼び出しと復帰検知回路，及びパイプラインレジスタを

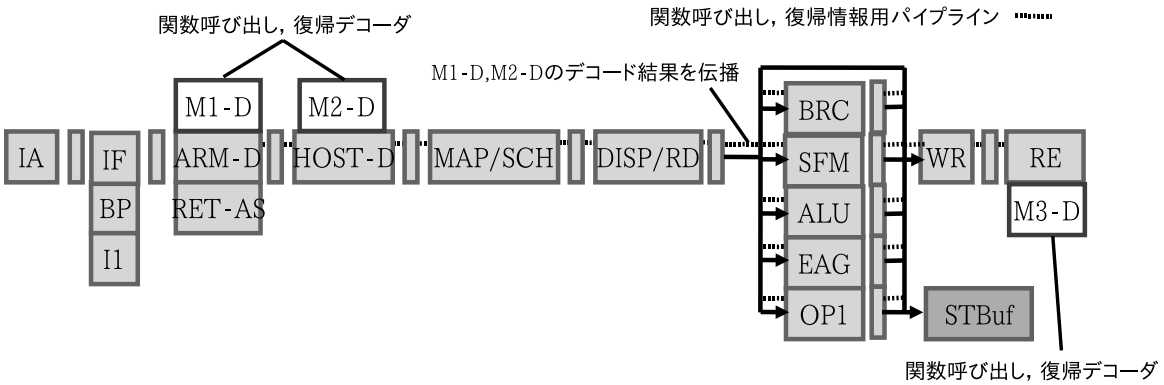


図 27: ARM の関数呼び出し，復帰デコーダ

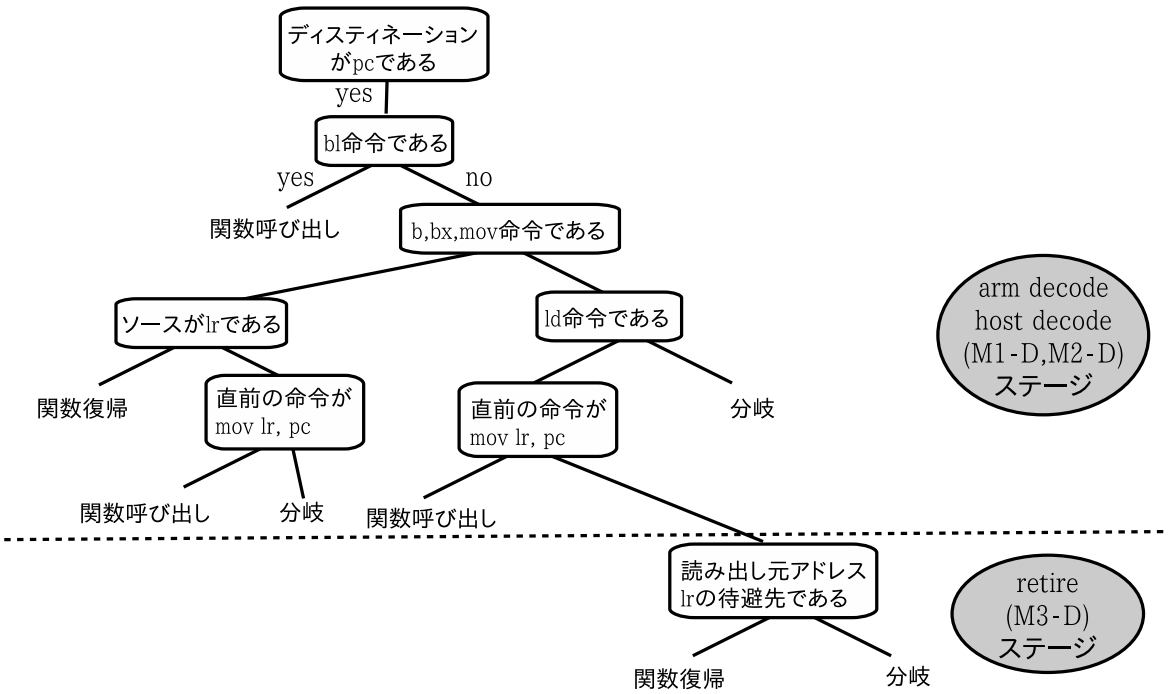


図 28: 関数呼び出し，復帰検知アルゴリズム

図 27 に示したように追加する．本機構は以下のように動作する．まず，デコードステージの復帰検知回路 (M1-D,M2-D) はプログラムカウンタを上書きする命令を監視する．検知された命令が bl 命令ならば，関数呼び出し命令であるとリタイアステージまでパイプラインレジスタを伝播させて伝えられる．その他のプログラムカウンタ上書き命令である場合，プログラムカウンタを上書きするソースオペランドレジスタを調べる．ソースオペランドレジスタが関数復帰アドレスを保持する lr(リンクレジスタ) であるならば，検知された命令は関数復帰命令であると認識する．そうでない場

合，直前の命令を調べ，それが関数復帰アドレスを待避する命令 (mov lr，pc 等) であれば，検知された命令は関数呼び出し命令であると認識する．これら全ての条件に当てはまらない命令はただの分岐命令であると認識する．なお，関数プロローグにおいてスタックに待避された関数復帰アドレスを，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を上書きして関数復帰が行われる場合がある．このような場合にも正しく関数復帰を検知するために，関数プロローグにおいて関数復帰アドレスの待避先アドレスを記憶しておく．プログラムカウンタを上書きするロード命令実行時に，ロードアドレスと記憶しておいたアドレスとを比較し，一致すれば関数復帰命令であると認識する．この比較は，前項で述べたパイプラインレジスタにより伝えられたソースオペランドを用いて，リタイアステージの復帰検知回路 (M3-D) にて行われる．以上で述べた関数呼び出し，復帰検知デコーダのアルゴリズムをまとめると**図 28** のようになる．

では次に，より具体的に，関数復帰命令 (図 26 中 (1)) が本機構により検知される様子を述べる．命令「ldmia sp!，{r4,r5,r6,pc}」のデコード時に検知デコーダは，その命令が関数呼び出し，分岐及び関数復帰のいずれを意味しているのかをパイプラインレジスタを伝播させてリタイアステージに通達する．リタイアステージでは，対象の命令は分解，実行済みであり，どのアドレスから読み込まれてきた値でプログラムカウンタが上書きされたのかがわかる (4.4.2 節参照)．関数プロローグ中にリンクレジスタの値がスタック上に待避された場合，本機構はそのアドレスを記憶している．このアドレスとプログラムカウンタを上書きしたアドレスを比較し，一致した場合この命令は関数復帰命令であると判断される．

5　評価

スーパスカラ型プロセッサ OROCHI をサイクルベースでシミュレーションする Osim 上に提案手法を実装し，その評価を行った．

5.1　評価環境

評価時の各パラメータを**表 3** に示す．ベンチマークプログラムとして，stanford と SPEC95 INT を gcc-4.1.1(-O2 -msoft-flaot -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 3: 評価パラメータ

A1 cache	miss penalty	8	cycles
	size	16	Kbytes
	line size	64	bytes
	way 数	4	ways
L1 cache	miss penalty	8	cycles
	size	32	Kbytes
	line size	64	bytes
	way 数	4	ways
L2 cache	miss penalty	40	cycles
	size	2	Mbytes
	line size	64	bytes
	way 数	4	ways
pipeline	命令フェッチ (IF)	1	命令/cycle
	デコード (ID)	2	命令/cycles
	命令分解能 (AD)	4	内部命令/cycle
	命令分解能 (HD)	4	内部命令/cycle
	レジスタマッピング (MAP)	4	内部命令/cycle
	命令発行 (SEL/READ)	4	内部命令/cycle
	命令実行 (EXE)	1	内部命令/cycle
	物理レジスタ書き込み (WR)	1	内部命令/cycle
	論理レジスタ書き込み (RE)	4	内部命令/cycle
	Reorder buffer	32	entry
CAM	size	4	Klines
	line size	64	bytes
入力値検索コスト	Register	1	cycle
	L1 cache	2	cycle

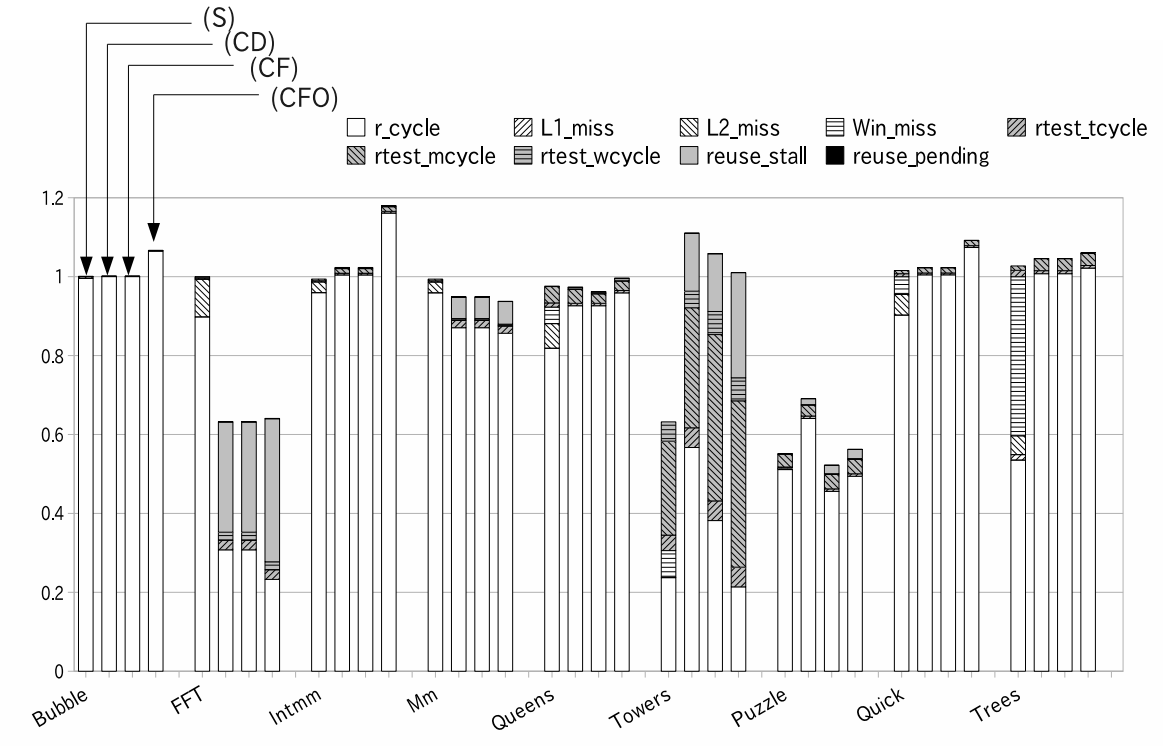


図 29: 評価結果:stanford

5.2 結果-stanford

図 29 に stanford ベンチマークソフトによる評価結果を示す．横軸がプログラム名を示し，縦軸はメモ化無しの通常実行を 1 として正規化した実行時間を示している．各実行プログラム毎の 5 本のグラフは左から SPARC をベースアーキテクチャとしている既存モデル (以下，S), コンテクスト固定メモ化モデル (以下，CD), コンテクストフリーメモ化 (以下，CF)，コンテクストフリーメモ化+入力値検索オーバーラップモデル (以下，CFO) を示す．凡例について説明する．まず，既存モデルの凡例から述べる．r_cycle がプロセッサ実行時間，L1_miss，L2_miss はそれぞれ L1，L2 キャッシュミスペナルティを，Win_miss はレジスタウィンドウミスペナルティを意味する．rtest_tcycle, rtest_mcycle は入力値の比較にかかった時間を示し，順にレジスタと CAM との比較，キャッシュと CAM との比較時間を意味する．次に，提案モデルの凡例を述べる．r_cycle はプロセッサ実行時間とキャッシュミスペナルティを意味する．rtest_tcycle,rtest_mcycle は既存モデルと同じく入力値の比較に必要な時間を示す．reuse_stall は再利用適用時に発生する stall によるペナルティを，reuse_pending はキャッシュコントローラが busy で入力値検索開始が遅延した時間を意味する．

(CD) では平均 5%，FFT において最大 37%の高速化を確認した．(CF) では平均 8%，Puzzle において最大 48%の高速化を確認した．(CFO) では平均 5%，Puzzle において最大 44%の高速化を確認した．FFT を除いて，再利用による実行時間の削減率に関して，既存モデルと同様の傾向を示していることがグラフから読み取れる．FFT では，既存モデルに比べ提案モデルが大幅な高速化を達成している．これは，既存モデルは浮動小数点演算ユニットを備えているが，提案モデルにはそれがなく，ソフトウェア浮動小数点ライブラリを用いているためである．FFT は浮動小数点演算を多用するプログラムであり，提案モデルでは，この浮動小数点演算全てがライブラリ関数呼び出しとなる．結果，既存モデルに比べて再利用対象区間が増加し，実行サイクル数の大幅な削減につながった．

Towers と FFT において，プロセッサ実行時間の削減率は高い．しかしながら，再利用適用時に発生する stall の影響から，プログラム全体の実行時間は大幅に上昇してしまっている．**図 30**，**図 31** に Towers と FFT における再利用状況を示す．横軸は再利用が行われた関数を意味し，縦軸は再利用により削減できた平均 cycle 数を示している．なお，この平均 cycle 数は再利用オーバヘッドによるペナルティを 0 として算出している．図からわかる通り，どちらのプログラムも実行ステップ数の非常に小さい関数が再利用されている．再利用適用時に発生する stall は，当然一回の再利用毎に発生するので，このような再利用特性を持つプログラムでは，再利用による高速化の恩恵を受けるのは難しい事が分かる．

入力値検索のオーバヘッド削減により (CF) に比べて，高速化が期待できる (CFO) であるが，実行時間の平均削減率は (CF) やに比べて悪くなっている．これは，(CFO) ではロード命令及びストア命令の発行に際して，実行ステージからのオペランド供給バイパスが使えないことに起因する．再利用機構は関数呼び出し命令のリタイアを検知すると，以降入力値検索が終了するまで，キャッシュコントローラを独占的に使用する．これは，ベースプロセッサ側から見ると，突然キャッシュアクセス不能になり，後段からのバイパスを期待して発行済みの命令がバイパスを受け取れなくなる場合がある．そのため，(CFO) では，ロード命令及びストア命令は発行ステージで論理レジスタ及び物理レジスタからオペランドが読み出せない限り発行されない．これは，パイプラインの使用効率を大きく低下させるものである．そのため，入力値検索のオーバヘッド削減による性能向上が打ち消されてしまっていたり，再利用がほとんど適用できないプログラムにおいては，メモ化無しの通常実行に比べても著しく性能が低下している．

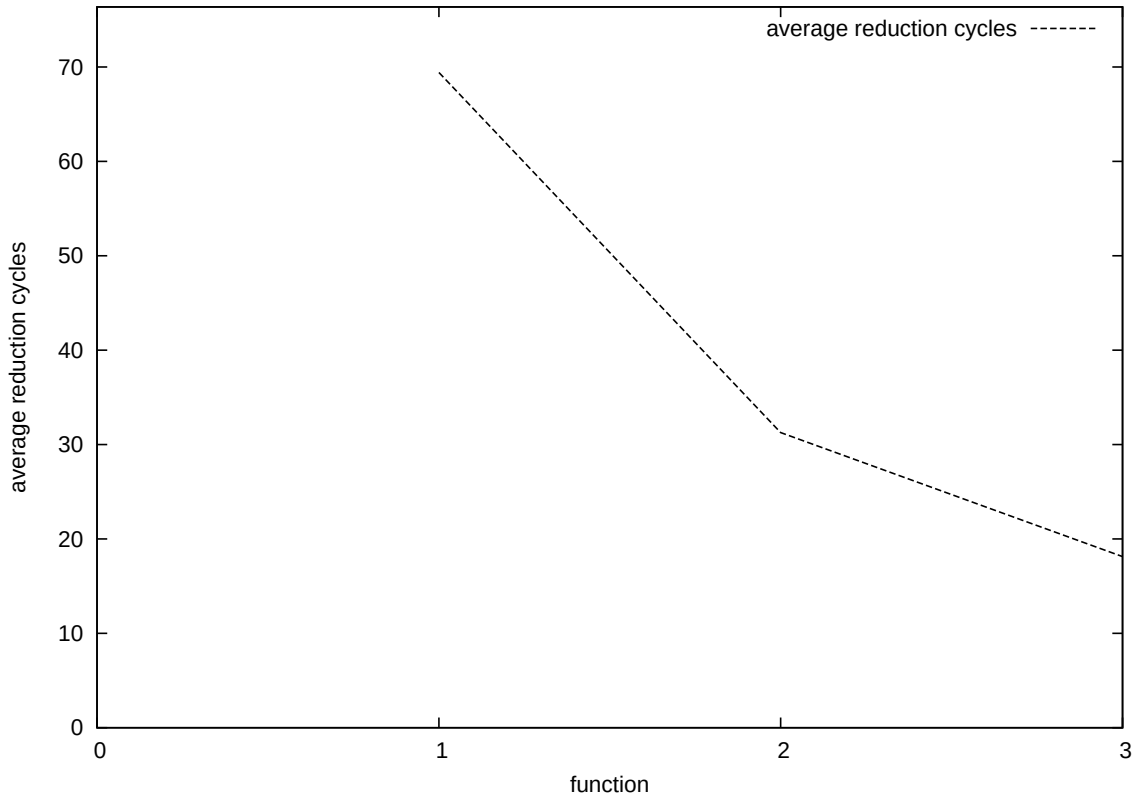


図 30: Towers における再利用状況

5.3 結果-SPEC CPU95 INT

図 32 に SPEC CPU95 INT ベンチマークソフトによる評価結果を示す．グラフの読み方は上述の stanford ベンチマークのものと同じである．(CD) では平均 1%，147.vortex において最大 12%の高速化を確認した．(CF) では平均 2%，147.vortex において最大 17%の高速化を確認した．stanford ベンチマーク同様 SPEC95 INT ベンチマークにおいても，再利用による実行時間の削減率は既存モデルと同様の傾向を示している．147.vortex において既存モデルに比べ提案モデルの実行 cycle 数に若干の上昇がみられるが，これは既存モデルと提案モデルのベースアーキテクチャの違いによるものと考えられる．例えば，既存モデルがベースアーキテクチャに採用している SPARC の整数乗算命令に比べ，本提案モデルが採用しているベースアーキテクチャの ARM の整数乗算命令は，非常に実行が遅い．また，上述したように，既存モデルは浮動小数点演算ユニットを備えているが，提案モデルにはそれがなく，ソフトウェア浮動小数点ライブラリを用いているため，浮動小数点演算実行に非常に時間がかかる．

全てのプログラムにおいて，再利用適用による stall の影響はあまり見られない．特

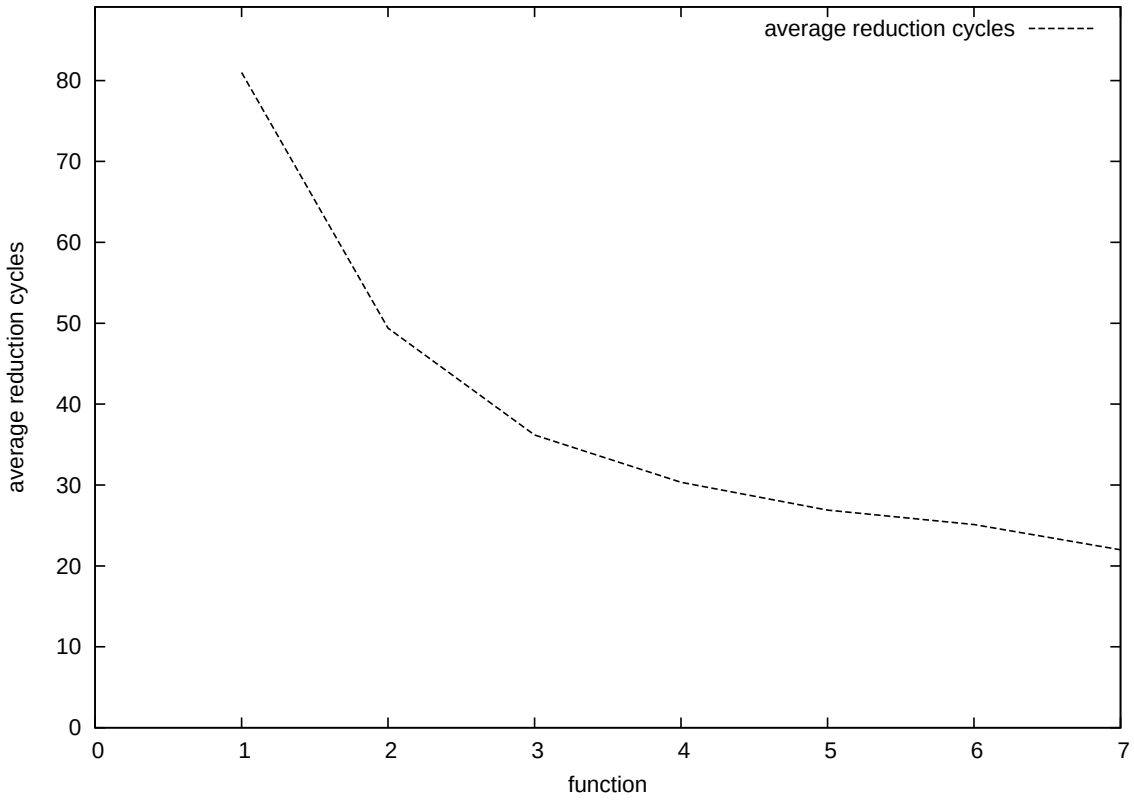


図 31: FFT における再利用状況

に再利用効果の高い 147.vortex の再利用状況を**図 33** に示す．グラフの読み方は図 29 と同じである．図から，147.vortex では再利用による平均削減 cycle 数が比較的大きいものが多いのが分かる．このため，再利用適用時に発生する stall の影響が相対的に小さくなっているのだと考えられる．

5.4 考察

stanford ベンチマークおよび SPEC95 INT ベンチマークにおいて，再利用率は既存モデルにほぼ等しい結果を得ることができた．しかし，再利用オーバヘッドを含めたプログラム全体の実行 cycle 数は一部のプログラムで大幅に上昇してしまっている．

入力値の検索後，再利用可能なことが分かると，すべてのパイプラインレジスタをフラッシュした後に後続区間の命令をフェッチしてくるので，再利用適用時には stall が発生する．この stall が一部のプログラムにおいて実行 cycle 数の大幅な上昇を引き起こしている．stall が実効性能に与える影響はプログラムの特性によって異なる．stanford ベンチマークの Towers や FFT のように，再利用率が高いプログラムであっても，再

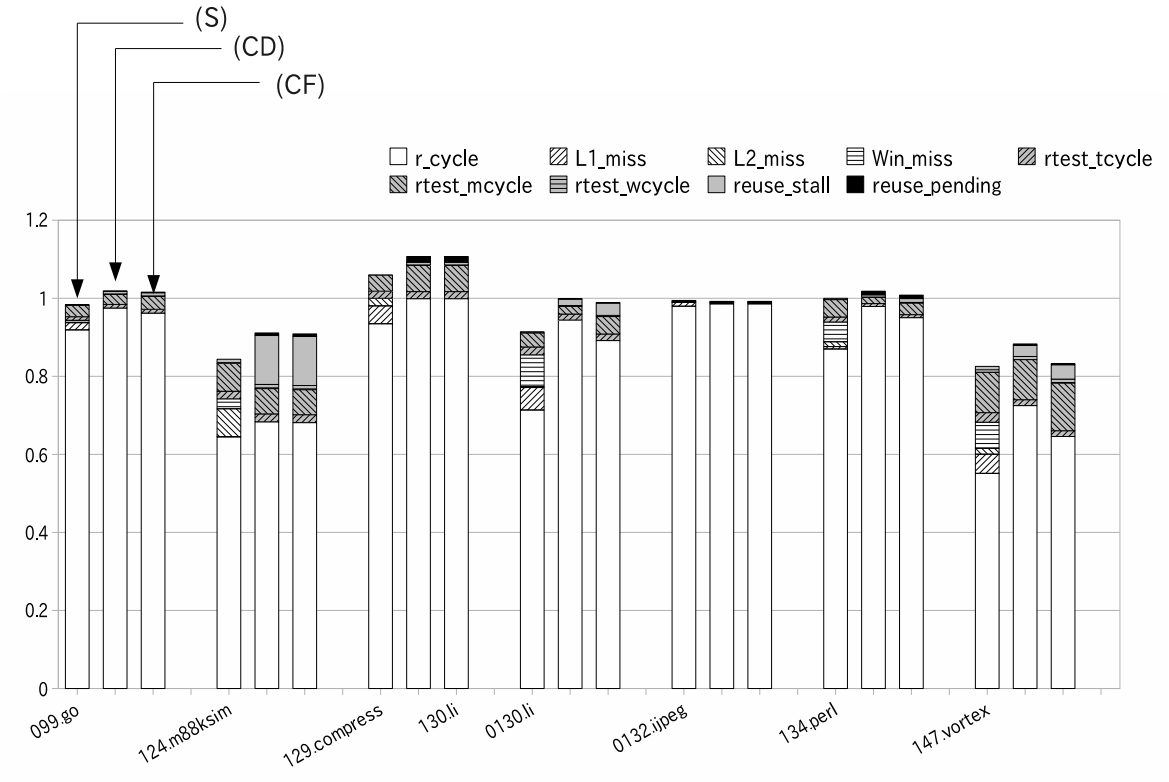


図 32: 評価結果:SPEC CPU95 INT

利用一回当りの削減 cycle 数が小さいものでは，一回の再利用毎に発生する stall によるペナルティが相対的に大きくなる．これにより，プログラム実行全体では再利用率に見合った高速化を達成する事ができていない．一方で SPEC95 INT ベンチマークの 147.vortex のような再利用一回当りの削減 cycle 数が大きいものでは，再利用による stall の影響は相対的に軽微となり，既存モデル同様の再利用効果が達成できていると考えられる．

ロード命令及びストア命令のオペランド供給バイパスが使えないため，性能低下を起こしている (CFO) であるが，今後，バイパス使用不能の期間を命令発行ステージで関数呼び出し命令を検知した時点から入力値検索終了までの間のみというように限定することで，性能の向上が期待できる．

6 おわりに

自動メモ化プロセッサは，メモ化をハードウェアにより自動的に行い，プログラム実行の高速化を達成する．本モデルには，単命令発行型プロセッサをベースアーキテクチャとしている点と，SPARC 命令セットアーキテクチャに依存した実装である点の

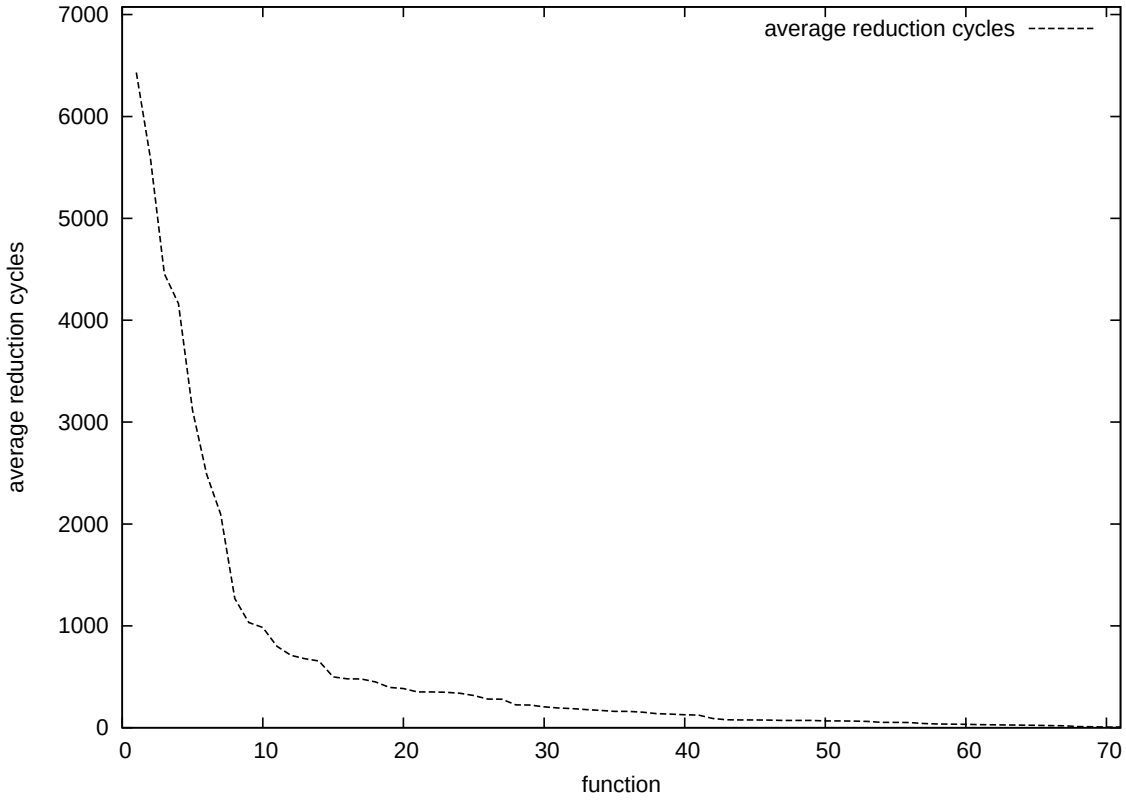


図 33: 147.vortex における再利用状況

2 つの問題点がある．本研究では，この 2 つの問題点を同時に解決することを目的に，スーパースカラ型 ARM プロセッサをベースアーキテクチャとした自動メモ化プロセッサモデルを提案した．SPARC ABI と ARM ABI が定める関数呼び出し規定の違いから，既存バイナリを実行可能であり，かつ既存モデルと同様の再利用率を維持することは困難であることが分かった．そこで，既存バイナリを実行可能であるが，ある程度の再利用率が低下するコンテクスト固定メモ化モデルと，既存バイナリの実行は不可能であるが，既存モデルと同等の再利用率が期待できるコンテクストフリーメモ化の 2 つのメモ化モデルを考案した．両モデルをサイクルベースのシミュレーションにより評価したところ，コンテクスト固定メモ化モデルでは，SPEC95 INT ベンチマークで平均 1%，最大 12%の高速化を，コンテクストフリーメモ化モデルでは，SPEC95 INT ベンチマークで平均 2%，最大 17%の高速化を達成し，その有効性が確認できた．また，パイプラインを持つプロセッサにハードウェアメモ化機構を搭載する上で必要な追加ハードウェアや，再利用適用時に発生する stall 等のハードウェアによるメモ化に関する新たな知見を得ることができた．

今後の課題としては，loop 区間を対象とした再利用がまずあげられる．次に，オーバヘッドフィルタやキャッシュの書き換え検知による入力値検索削減手法等の既存モデルに実装済みの拡張を本モデルで評価することである．特に，オーバーヘッドフィルタは既存モデルそのままではなく，既存モデルには存在しなかった再利用適用時の stall を考慮に入れたモデルを考える必要がある．

7　謝辞

本研究の為に多大な御尽力をいただき，日頃から熱心な御指導を賜った名古屋工業大学の津邑公暁准教授，奈良先端科学技術大学院大学の中島康彦教授に深く感謝いたします．また，本研究に対しての熱心な御協力をいただきました，名古屋工業大学の松尾啓志教授に深く感謝致します．

さらに，本研究のために多大な御協力をいただいた新美明仁氏に深く感謝致します．加えて，たびたびご検討・助言をしていただいた同大学の齋藤彰一准教授や松井俊浩助教授にも深く感謝致します．最後に，本研究の際に多くの助言・協力をしていただいた松尾・津邑研究室ならびに斎藤研究室の皆様，中でも本研究に関して様々な相談や検討を通じ，支援をして頂いた神谷優志氏，池谷友基氏，板橋世里華氏に対して深く感謝致します．

参考文献

[1] W.Wall, D.: Limits of Instruction-Level Parallelism, *WRL Research Report*, No. 93/6 (1993).

[2] Akio Nakajima, Ryotaro Kobayashi, H. A. T. S.: Limits of Thread-Level Parallelism in Non-numerical Programs, *IPSJ Degital Courier*, Vol. 2, pp. 280–288 (2006).

[3] 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).

[4] Huang, J. and Lilja, D. J.: Exploiting Basic Block Value Locality with Block Reuse, *Proc. 5th International Symposium on High-Performance Computer Architecture*, pp. 106–114 (1999).

[5] Sodani, A. and Sohi, G. S.: Dynamic Instruction Reuse, *Proc. 24th International Symposium on Computer Architecture*, pp. 194–205 (1997).

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

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

[8] ARM: *Procedure Call Standard for the ARM Architecture* (2008).

[9] KOJIMA, T. and NAKASHIMA, Y.: Design of a Centralized Instruction Window Superscalar for Evaluation of OROCHI, *IPSJ SIG Notes*, Vol. 2006, No. 127, pp. 61–66 (20061128).

[10] Koufaty, D. and Marr, D. T.: Hyperthreading technology in the netburst microarchitecture.

[11] Elsner, D. and Fenlason, J.: *using as*, No. 2.14 (2002).