修士論文

大規模再利用による
投機的マルチスレッディングの効果

指導教官　　富田 眞治 教授

京都大学大学院情報学研究科
修士課程通信情報システム専攻

清水 雄歩

平成17年2月4日

大規模再利用による
投機的マルチスレッディングの効果

清水 雄歩

内容梗概

近年，プロセッサの処理速度の向上に伴い，主記憶へのアクセス速度が相対的に低下している．主記憶レイテンシを隠蔽する手法として，単一プロセッサに複数のスレッドを同時に実行させる同時マルチスレッディング（SMT）を利用した予備実行（Precomputation）や，値予測に基づく投機実行により命令レベルの並列度を確保する投機的マルチスレッディング（SpMT）がある．

SpMT において，主スレッド および複数の投機スレッドによる実行結果を out-of-order に利用可能だとすれば，高性能が得られる．ただし実現のためには，従来研究のような，投機スレッドと主スレッドの 1 対 1 のデータ引継ぎ構造では不十分であり，多くの実行結果の中から 1 つを選ぶ，多対 1 のデータ引継ぎ 構造が必要となる．本論文では，SpMT に区間再利用を統合したモデルを提案した．

区間再利用とは，実行時に命令区間における入出力セットを再利用表に記憶しておき，再び同じ入力により命令区間を実行する場合に，記憶しておいた出力を利用して命令区間の実行を省略する高速化手法である．本機構では，再利用表の規模（エントリ数）によって記憶できる入出力セットの数が決まる．大規模なほど多くのセットを記憶でき，再利用表のヒット率が上がるものの，再利用表のレイテンシも増大し，再利用効果の低下が予想される．そこで本論文では，再利用表と同規模のキャッシュのレイテンシを参考にし，再利用表のレイテンシを考慮した評価を行った．

また，評価に要する膨大なシミュレーション時間を短縮するため，汎用 CAM 搭載 PCI ボード の設計を行った．ソフトウェアシミュレータに対し最大約 70 倍の高速化を達成し，多くのプログラムの挙動を調査することが可能となった．

再利用表を小規模な 4K エントリかつ短レイテンシと仮定したモデルを Gs, 大規模な 256K エントリかつ長レイテンシを仮定したモデルを Gl，さらに，理想的な 256K エントリかつ短レイテンシを仮定したモデルを I として，主スレッド の性能評価を行った．測定の結果，理想的なモデル（I）が実行サイクル数を平均 31%削減でき，最も高性能であること，また，現実的なレイテンシでは平均 27%を削減できるモデル（Gs）が最も高性能であることがわかった．

Effectiveness of Speculative Multithreading exploiting Large-Scale Region Reuse

Yuho SHIMIZU

Abstract

Recently, modern commercial microprocessors have been adopting aggressive wide-issue superscaler technique and pursuing ultra high frequency. Nevertheless, the widening disparity in speed between L1-cache and main memory puts the breaks on the pace of boosting instruction level parallelism. Besides, two major techniques are proposed to overcome the large latency of main memory. One is precomputation exploiting Simultaneous Multithreading (SMT) that executes several threads in parallel on a single processor. The other is speculative multithreading (SpMT) equipped with multiple program counters exploiting thread level parallelism.

This paper proposes an integrated SpMT model exploiting hardware-based multilevel region speculation and tree-structured associative buffer. The integrated mechanism of SpMT and region reuse is different from related works. The key point of this work is the asymmetric structure of SpMT so as to reorder the multiple results produced by speculative threads.

Region reuse is one of techniques to boost ILP. The set of input and output for an instruction region is memorized and reused if the region is encountered with the same set of input. In this microarchitecture, the tradeoff between the size and speed of associative buffer is extremely important. This paper focuses on the tradeoff and evaluates the performance on several models that include realistic and optimistic implementations.

For the purpose to speed the simulation of large-scale associative buffer, I developed a PCI board exploiting general purpose CAMs that accelerates the associative search. The maximum speedup reaches to 70 times faster than a software based simulator.

Finally, this paper concludes that an ideal model exploiting large-scale associative buffer is potentially promising. However, considering realistic parameters, a model exploiting small-scale buffer is most effective among several models.

大規模再利用による
投機的マルチスレッディングの効果

目次

第1章　はじめに

近年，プロセッサの処理速度の向上に伴い，主記憶へのアクセス速度が相対的に低下している．この主記憶のレイテンシを隠蔽する手法として，**同時マルチスレッディング**（Simultaneous Multithreading: 以下，**SMT**）や，**投機的マルチスレッディング**（Speculative Multithreading: 以下，**SpMT**）がある．SMTは，単一プロセッサに複数のスレッドを同時に実行させる手法である．SMTにおけるヘルパースレッドによりload命令を先行実行することにより，効率的なプリフェッチ機構を実現する研究（Precomputation: 以下，予備実行）が多く行われている．一方SpMTは，値予測に基づく投機実行により，命令レベルの並列度を確保する手法である．

SpMTにおいて，主スレッドおよび複数の投機スレッドによる実行結果をout-of-orderに利用可能だとすれば，高性能が得られる．ただし実現のためには，従来研究のような，投機スレッドと主スレッドの1対1のデータ引継ぎ構造では不十分であり，多くの実行結果の中から1つを選ぶ，多対1のデータ引継ぎ構造が必要となる．本論文では，1モデルとして，再利用機構を備えた非対称SpMTを提案する．

再利用（Reuse）とは一般に，関数呼び出しやループなどの命令区間に対し，実行時に入出力セットを再利用表と呼ぶバッファ（Reuse Buffer）に記憶しておき，再び同じ入力により命令区間を実行する場合に，記憶しておいた出力を利用して命令区間の実行を省略する高速化手法である．本論文では，プログラムがSPARC ABI（Application Binary Interface）に従っているものと仮定し，コンパイラやバイナリアノテーションツールなどの助けを借りることなく，命令区間や入出力を動的に把握することにより，既存ロードモジュールを高速化する手法を提案する．

本提案では，再利用表に登録されている命令区間について今後の入力値を予測し，主スレッドの実行と並行して，投機スレッドが事前に実行し，結果を再利用表に登録する．これにより，入力が単調変化する場合など，過去の実行結果の単純な再利用では効果がない命令区間に対しても，高速化を図ることができる．

本機構では，再利用表の規模（エントリ数）によって記憶できる入出力セットの数が決まる．規模が大きいほど多くのセットを記憶でき，再利用表のヒッ

ト率が上がるものの，再利用表のレイテンシも増大し，再利用効果の低下が予想される．そこで本論文では，CAM と同規模のキャッシュのレイテンシを参考にし，CAM のレイテンシを考慮した評価を行った．

ところで，本機構の評価は，ソフトウェアにより実現したシミュレータを用いて行っている．そのため，再利用表に対する連想検索機構を，シミュレータでは RAM 上の 1 次元配列の逐次検索により実現しており，大規模な再利用表を装備した再利用機構のシミュレーションに膨大な時間を要する．そこで，シミュレーション時間を短縮するために，汎用 CAM 搭載 PCI ボードを開発し，シミュレータに組み込み，シミュレーションの高速化を図った．

以下，第 2 章で関連研究について述べ，本研究の位置付けを明らかにする．次に，第 3 章では，本論文が提案する SpMT の詳細について述べ，第 4 章で SpMT 機構を有するプロセッサシミュレータのハードウェアアクセラレータである GP600M について述べる．その後，GP600M を用いたシミュレータによる SpMT の評価を第 5 章で行う．

第2章　従来の投機的マルチスレッディング

本章では，これまでに行われている，SMT，SpMT，および再利用による高速化技術に関連する研究について述べ，本研究の位置付けを明らかにする．

2.1　投機実行

投機実行（Speculative Execution）は，命令を対象とするものとオペランドを対象とするものとに大きく分けることができる．

2.1.1　命令レベルの投機実行

命令レベルの投機実行には分岐予測があり，静的分岐予測と動的分岐予測に分けることができる．静的分岐予測は，コンパイラがプログラムを解析して分岐予測を行い，命令中に予測した情報を埋め込む手法である．プログラム実行時に動的な分岐予測は行わず，コンパイラの指示に従う．動的分岐予測は，コンパイラは分岐予測を行わず，プログラムの実行時に過去の分岐履歴を用いて分岐先を予測する手法である．命令分岐の方向には偏りがあり，過去 2 回程度の分岐履歴情報を用いて十分に予測可能と言われている．単純なハードウェア機構により実現でき，大きな効果が得られるため，多くの商用プロセッサが採用している．

2.1.2　オペランドレベルの投機実行

オペランドレベルの投機実行は，さらに，アドレスに関する投機実行と値に関する投機実行に分けることができる．オペランドアドレス予測の代表的なものとして，Next Line Prefetchがある．あるキャッシュラインが連続して参照されると，近い将来次のキャッシュラインを参照すると予測し，あらかじめ主記憶からプリフェッチしておく手法である．一方，近年，多くの研究が行われているのは，アドレスではなく値に基づく投機実行に関するものである．具体的には，過去の履歴に基づいて先行命令の実行結果を予測し，予測値を入力として後続命令列を投機的に実行する．予測が正しければ後続命令が先行命令を待つ時間が短縮される一方，誤っている場合には，投機的に実行した命令を全て無効化し，正しい入力値を用いて再実行する必要がある．このため，投機実行を適用しなかった場合よりも実行時間が長くなることがある．さらに，予測値を常に検証する必要があるため，ハードウェアが複雑になるという欠点もある．値予測の手法には，次のものがある．

Last-value 予測　同じ命令アドレスにおける前回の演算結果をそのまま使用する手法である．さらに，動的に予測可能かどうかを判断する CT（Classification Table）を用いて，予測ミスを減らす機構が提案されている [1].

Stride-based 予測　最近の 2 回の演算結果の差分を S，最近の演算結果を B とした場合に，次の予測値を $S + B$ とする手法である [2].

two-level 予測　命令ごとに最近の 4 種類の演算結果を表に記録しておく．また，別の表にそれらの演算結果が過去に出現した回数を格納する．後者の値があらかじめ決めておいた値に達した時に，対応する演算結果を予測値とする手法である．

Context-based 予測　過去の連続した有限個の値の履歴（Context）と，過去の履歴とを比較し，高い確率で実行されるパターンに基づいて予測を行う手法である．

ハイブリッド 予測　Last-value 予測と Stride-based 予測，Stride-based 予測と two-level 予測を組み合わせた手法である．

投機実行においては，予測値の検証の必要性から，先行命令の実行時間そのものを削減することができない．このため，厳密な検証が必要となる値そのものを投機対象とするのではなく，SMT を利用して load 命令を事前に実行し，効果的なプリフェッチ機構として利用する予備実行（Precomputation）の研究が数多く行われている [3, 4, 5].

2.2 SpMT

SpMT は値予測に基づく投機実行により，スレッドレベルの並列度を確保する手法である．複数の予測値に基づいて，複数のプロセッサを投入して高速化を図る研究が数多く行われている．SpMT では主記憶一貫性の保証が重要となる．主記憶一貫性を保証するために一般的に用いられる手法は，投機実行の無効化である．投機スレッドの開始後に，主スレッドが投機スレッドのソースオペランドを上書きした時点で投機実行の結果は利用できないとみなし，投機実行を無効化する．

Gopal ら [6] は，階層的マルチプロセッサシステムにおいて，各主記憶値の予測値を複数保持する Speculative Versioning Cache を提案している．各キャッシュラインには順序づけされたキャッシュセットが保持される．キャッシュはキャッ

シュセットから無効化応答を受け取ると，無効化信号をプロセッサに送信する．Marcuello ら [7, 8] も，投機的マルチスレッドプロセッサのための，トレースに基づく増分予測を提案している．増分は，当該トレースの前回実行時における，トレース終了時の値とトレース開始時の値の差として計算される．Oplinger[9] らは，call された関数本体およびその関数の復帰後に続くコードを，並列実行する手法を提案している．文献 [6, 7, 8, 9] とは対照的に，Codrescu ら [10] は，ループイタレーションや関数呼び出しではなく，load 命令を契機にスレッドを起動することにより，細粒度の投機スレッドを生成可能な自動分割ポリシーを提案している．投機的データキャッシュは主スレッドを実行するプロセッサが発行した store により生じる擾乱を検出するために保持される．また Marcuello ら [11] は，コントロールグラフおよび到達可能性に基づく，プロファイルベースの投機スレッド起動ポリシーを提案している．

2.3 再利用

再利用とは，関数呼び出しやループなどの命令区間に対して，実行時に入出力セットを再利用表に記憶しておき，再び同じ入力により命令区間が実行されようとした場合に，過去に記憶しておいた出力を利用することにより命令区間の実行自体を省略する高速化手法である．投機的手法ではないため，予測失敗による無効化を必要としない特長がある．

再利用の実現方法は，ハードウェアによるものや，ソフトウェア支援によるものが提案されている．一般に，プロセッサが動的かつ効率よく基本ブロックを切り出すことは難しく，簡単化するには，コンパイラが基本ブロックの範囲をハードウェアに伝達しなければならない．このため，コンパイラが再利用を行うための専用命令を生成する．ただし，専用命令を使用する場合は，専用のコンパイラが生成したロードモジュールのみが高速化の対象となり，既存ロードモジュールは高速化できない欠点がある．

ハードウェアのみによるものとして，Sodani ら [12] は，最大 1024 エントリからなるフルアソシアティブの再利用表を用い，単命令を対象とした汎用的な再利用を提案している．再利用表の各エントリは，命令オペランドの値および命令結果を保持する．また，load 命令が再利用可能であることを保証するために，主記憶有効ビットおよび主記憶アドレスが保持される．store 時には，主記憶アドレスが連想検索され，無効化される．他の方法として，再利用表にオペ

ランドレジスタの識別子を保存する方法も提案されている．González ら [13] は，最大 256K エントリからなる Reuse Trace Memory（RTM）と呼ばれる表を用いて評価を行っている．RTM は PC の一部によるインデクシングを仮定しており，256K エントリの場合，インデクスを 11bit，8way とし，PC ごとに最大 16 エントリを記録する．比較的小規模の CAM により実現可能となっているものの，総容量は 32MB 以上にも及ぶ．Costa ら [14] は，load/store 命令を対象から除外したうえで，フルアソシアティブの表による再利用手法を提案している．PC およびオペランドの値は，表により連想検索される．

ソフトウェア支援によるものには，Huang ら [15] が，コンパイラに機能を追加することにより，再利用区間内に閉じたレジスタによる出力の登録を省く，基本ブロック再利用方式がある．キャッシュの内容も比較するため，ポインタにも対応している．比較に 1 サイクル，再利用に 1 サイクルを仮定している．Connors ら [16] は，コンパイラによる区間切り出しとハードウェアサポートを組み合わせた手法を提案している．最後の load から次の load までの間に store 命令が存在しなければ主記憶比較を省略する．再利用表の最大サイズは 128KB 以上である．Connors ら [17] はまた，コンパイル時の情報だけではなく，実行時情報も用いることにより，再利用表の割り当てを最適化する手法を提案している．Huang ら [18] は，一般に基本ブロックの入出力数は様々であるため，有限幅の再利用表を効率的に使うことができないとし，基本ブロックをサブブロックに分割することにより，再利用表に登録可能な入出力数に揃える方法を提案している．

この他にも再利用に関しては，入力の一致判定に寛容性を持たせる曖昧再利用 [19, 20] や，load 命令に限ってアドレスおよび load 値を再利用する手法 [21, 22] など，様々な手法が提案されている．

投機実行の結果を一部再利用する研究も行われている．Roth ら [23] は，過去のレジスタマッピングを書き戻すことにより，以前の失敗した投機実行において書き込まれた物理レジスタを再利用する方法を提案し，SPEC ベンチマークプログラムの実行時間を最大 11%削減している．また，再利用とデータ駆動スレッドを統合した，投機的データ駆動マルチスレッディングを提案しており [24]，SPEC ベンチマークプログラムの実行時間を最大 25%削減している．事前実行の結果は，レジスタリネーミングを利用して物理レジスタに格納される．

再利用と投機実行を組み合わせた手法も研究されている．Wu ら [25] は，コンパイラが再利用区間の切り出しを行い，再利用不可能である場合には再利用

区間の出力値を予測して，後続区間の実行を投機的に開始する手法を提案している．これにより，4 命令および 8 命令同時発行の構成において，SPEC ベンチマークプログラムの実行時間を 1.25 倍から 1.4 倍高速化している．この手法では，出力値の予測が外れた場合，後続区間の投機実行をキャンセルする必要があり，コストおよびオーバヘッドが問題となる．Molina らは，投機スレッドと主スレッドを組み合わせる手法を提案している．投機スレッドによる実行済みの命令は FIFO に格納され，主スレッドはそこから命令を取り出してソースオペランドを比較し，一致した場合結果を用いる．4 命令同時発行の構成において，SPEC ベンチマークプログラムの実行時間を，最大 1.33 倍，平均で 1.16 倍高速化している．

第3章　再利用機構を備えたSpMT

本章では，本論文が提案する，再利用機構を備えた非対称のSpMTについて述べる．

3.1　本論文が提案するSpMT

前章において述べたSpMTに対し，本論文では，主スレッドや投機スレッドが並列実行した結果を再利用する，多対1のデータ引継ぎ構造を有するSpMTを提案する．本機構では，プログラムを構成する命令区間を多入力多出力の複合命令と捉え，動的に検出した複数の複合命令を並列実行し，主スレッドの実行結果も含む複数の実行結果をout-of-orderに再利用することにより，主スレッドが実行する命令を大幅に削減する．投機スレッドが事前に実行することにより，同一入力が出現する間隔が長い場合や，入力が単調に変化する場合など，過去の実行結果の単純な再利用では効果がない場合においても再利用の効率を上げ，高速化を図る．この点において，事前実行は従来の予備実行とは異なる．また，命令区間の複雑な入れ子構造についても多重的に再利用する機構を提案する．

本SpMTの特長は，文献[25]とは異なり，コンパイラ支援を必要としない点，および再利用区間の入力値を予測の対象としているため，失敗した投機実行をキャンセルする必要がない点である．また，考え方は文献[26]に比較的近いものの，入力の比較の際にキャッシュの内容まで比較することにより，主記憶参照を必要とする命令区間に対しても再利用が適用できるという利点がある．

また本論文では，主記憶一貫性保証の手法として，一般的なSpMTで採用されている無効化ではなく，一部比較を用いている．書き換えられたアドレスを記憶しておき，再利用時に当該アドレスのみを比較することにより，再利用率を低下させることなく主記憶入力値比較コストを軽減している．

3.2　区間再利用

プログラムは，入力に対して処理を行い，結果を出力する．入力とは，レジスタや主記憶アドレスの参照であり，出力とは，処理結果のレジスタや主記憶アドレスへの格納である．命令自身が変更されない限り，入力が同じであれば実行結果も同じであり，入力が異なる命令以降について実行結果が枝分かれしていく．すなわち，参照順に入力を並べると，命令区間の入力セットは，**図1**

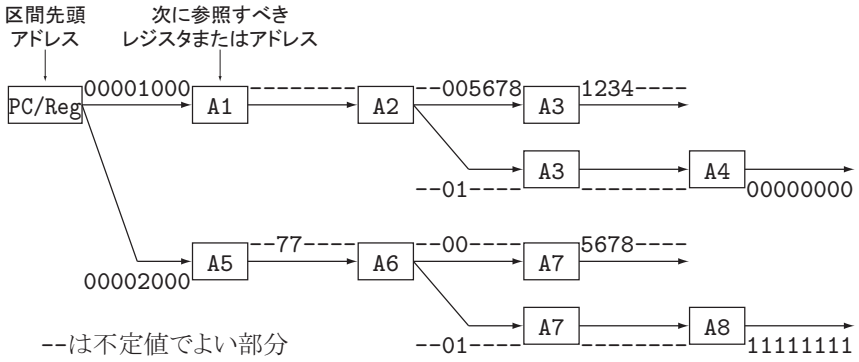


図 1: 入力セットの多分木構造

に示すように，レジスタ番号や主記憶アドレスをノードとし，各内容を枝とする多分木中の 1 つのパスとして表現できる．過去に出現した入力および予測した入力をこのような木構造に格納することにより，可変長の入力セットを取り扱えるだけでなく，枝に相当する記憶領域を節約できる．

命令区間を認識すると，区間の識別子から木構造の根を選択し，各ノードに記録されたレジスタ番号または主記憶アドレスから現在の値を読み出し，複数の枝の中から値が同一である枝を順に選択することを繰り返す．最終的に末端に到達した場合，対応する出力を再利用する．

すなわち，再利用とは，プログラム中の命令区間に対して，実行時に区間の入出力の組を再利用表に記憶しておき，再び同じ入力値によりその命令区間を実行しようとした場合に，記憶しておいた出力値を利用して命令区間の実行自体を省略する高速化手法である．再利用の特徴は，入力値さえ一致すれば実行結果を検証する必要がない点，および再利用の対象区間に属する命令の数が増えても再利用機構の複雑さがそれほど増大しない点にある．

3.3 再利用と SpMT の統合

図 2 を用いて，区間再利用の仕組みを具体的に説明する．和を求める関数 `Add` は，整数 `a` および `b` を引数とし，和 `x` を返す．すなわち，`a` および `b` が入力，`x` が出力である．関数 `Add(1, 2)` が呼び出されると，再利用表から入力セットが `(1, 2)` であるエントリを検索する．登録されていなければ関数を実行する．実行が終わると，入力セット `(1, 2)` と，出力値 `3` をそれぞれ再利用表の空きエントリに登録する．次に，`Add(3, 4)` が呼び出されると，入力セット `(3, 4)` も登録されていないので同様に実行し，結果を登録する．さらに，再び `Add(1,`

```
int Add(int a, int b)
{
  int x;
  x = a + b;
  return x;
}
```

	in		out
a	b	x	
1	2	3	
3	4	7	

図 2: 関数と再利用表

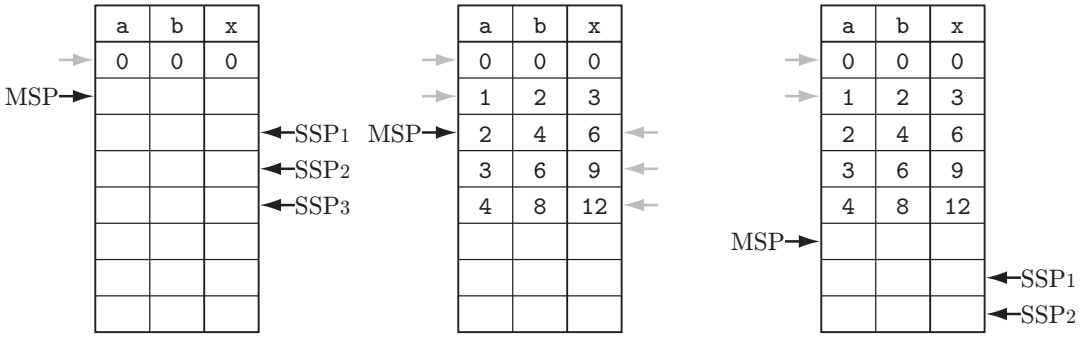


図 3: 入力が単調変化する場合

2) が呼び出され，再利用表を検索すると，以前に登録したエントリにヒットする．この場合，関数 Add は実際に実行することなく，再利用表に登録されている出力値 3 をレジスタに書き戻して高速に終了する．

ところで，一般に同一入力値が出現する間隔の長い命令区間や，入力値が単調変化し続ける命令区間に対しては，投機実行による効果が高いと予想できる．しかし，各命令区間の性質や投機実行による削減ステップ数は事前には分からない．このため，初めて実行する命令区間については，直ちに投機実行用プロセッサにより数回分の事前実行を試みる．その結果，対象の命令区間が主実行用プロセッサによる高い登録頻度を持ち，かつ投機実行用プロセッサが登録したエントリの再利用頻度も高い場合，事前実行による効果が高いと見て継続して投機実行用プロセッサによる投機対象とする．投機実行対象区間の選択方法については 3.9 節で詳述する．以下では，主実行用プロセッサを**MSP**（Main Stream Processor），投機実行用プロセッサを**SSP**（Speculative Stream Processor）と呼ぶことにする．

入力値が単調に変化する場合，今後の入力値を予測し，MSP と並行して SSP

が投機実行を行う．**図 3** に，SSP を 3 台用いた場合の実行例を示す．図 2 の関数 Addが，

```
for (i = 0; i < 10; i++)
  total += Add(i, i * 2);
```

と単調変化する引数により呼び出されるとする．MSP のみの場合，入力セットは (0，0)，(1，2)，(2，4)，...と変化し，どの入力セットも 1 度しか出現しないため，全て実行する必要があり，再利用では全く高速化できない．そこで，予測した入力値に基づき SSP に投機実行させる．具体的には，(0，0) の実行を終了し，(1，2) という入力セットが現れた（a）の時点において，後述するストライド予測により，(2，4)，(3，6)，(4，8)，...と入力セットが変化すると予測し，MSP と並行して SSP が投機実行を行い，結果を再利用表に登録しておく．MSP が (1，3) の実行を終え，（b）のように MSP が (2，4) を実行する前に，再利用表から入力セット (2，4) を検索する．MSP は過去に実行していないにも関わらず，入力セット (2，4) は既に登録されており，出力値 6 を再利用できる．同様に，(3，6)，(4，8) についても SSP により既に登録されており，MSP は（C）のように(5，3) の実行を開始する．以上のように，SSP による投機実行の結果を再利用することにより，MSP の実行を高速化できる．一方，予測が外れた場合においても投機実行をキャンセルする必要はなく，キャンセルに伴うオーバヘッドも削減できる．

3.4 SPARC ABI に基づく命令区間の特定

前節で説明したような再利用を行うためには，プログラムから適切な命令区間を選択しなければならない．本機構は SPARC アーキテクチャへの応用を仮定しており，プログラムは SPARC ABI（Application Binary Interface）に従っているものとする．これにより，一般に行われている再コンパイルや静的解析に基づく付加情報の埋め込み（バイナリアノテーション）を必要とせずに，既存ロードモジュールにおける命令区間の構造を動的に把握できる．

SPARC アーキテクチャにおいて，プログラムは常に計 32 個の汎用レジスタを使用することができる．汎用レジスタには，大域変数を格納する global レジスタ（%g0〜%g7），引数を渡したり作業用として使用される out レジスタ（%o0〜%o7），局所変数を格納する local レジスタ（%l0〜%l7），引数を受け取ったり返り値を格納する in レジスタ（%i0〜%i7）がある．

さらに，SPARC アーキテクチャには，関数呼び出しの時の引数の受け渡しの際に，主記憶を介する必要をなくすために規定されたレジスタウィンドウがある．各ウィンドウは上記の$\%o_n$，$\%l_n$，$\%i_n$（$0 \leq n \leq 7$）から構成され，$\%o_n$ は隣接するウィンドウの$\%i_n$ と同一，また，$\%i_n$ は反対側に隣接するウィンドウの$\%o_n$ と同一のレジスタとなっている．それに対して，$\%l_n$ は各ウィンドウに固有である．

現在のウィンドウは，CWP（Current Window Pointer）レジスタの内容により指定できる．CWP の値は save 命令によってインクリメントされ，restore 命令によってデクリメントされる．save 命令と restore 命令は，一般に関数呼び出し時と関数終了時に実行される．CWP の値がインクリメントされるとウィンドウが 1 つ進められ，以前に%o として参照していたレジスタが%i となり，以前に%l および%i として参照していたレジスタは使用できなくなる．その代わりに，新しく%l および%o が割り当てられる．逆に CWP の値がデクリメントされるとウィンドウが 1 つ戻り，以前に%i として参照していたレジスタが%o となり，以前に%l および%o として参照していたレジスタは使用できなくなる．その代わりに，新しく%l および%i が割り当てられる．

save 命令により割り当てられたレジスタの内容は，restore 命令を実行するまでは保存される．しかし，save 命令が続くと，レジスタウィンドウの容量を超過し，新たなレジスタを割り当てることができなくなる．この場合にはウィンドウオーバフロー割り込みが発生し，レジスタの内容がスタックに一時退避される．逆に，restore 命令が続くと，ウィンドウアンダフロー割り込みが発生し，スタックに退避していた値がレジスタに戻される．このようなオーバフローおよびアンダフローの際には主記憶参照が生じるため，プログラムの実行が遅れる．

SPARC におけるスタックは，以下の用途に用いられる．

- ウィンドウオーバフローが起こった際にレジスタの値を退避する．
- 関数の返り値が構造体の場合に，構造体へのポインタを格納する．
- 関数呼び出しの際に，第 7word 以降の引数を格納する．
- 引数の一時退避．
- 局所変数の退避．

スタックは，主記憶の高位アドレスから低位アドレスに向けて伸びていく．現在有効なスタックの下限アドレスが，スタックポインタと呼ばれるレジスタ%o6（%sp）に格納されており，**図 4** に示すように，save 命令が実行されると，レジ

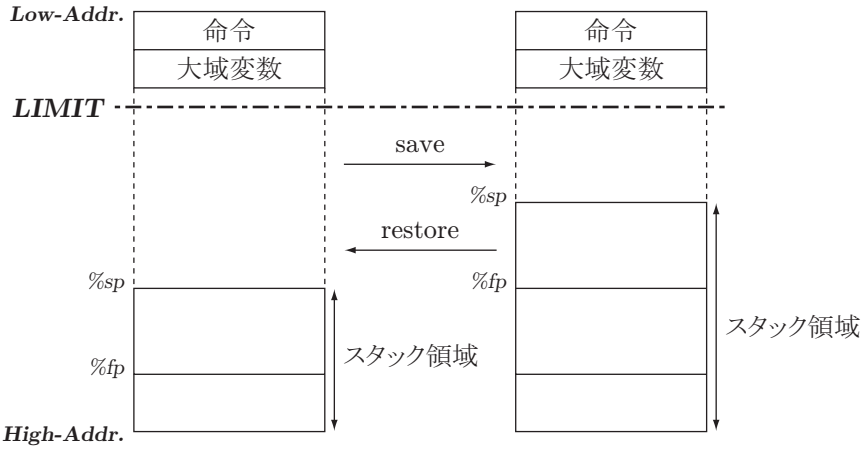


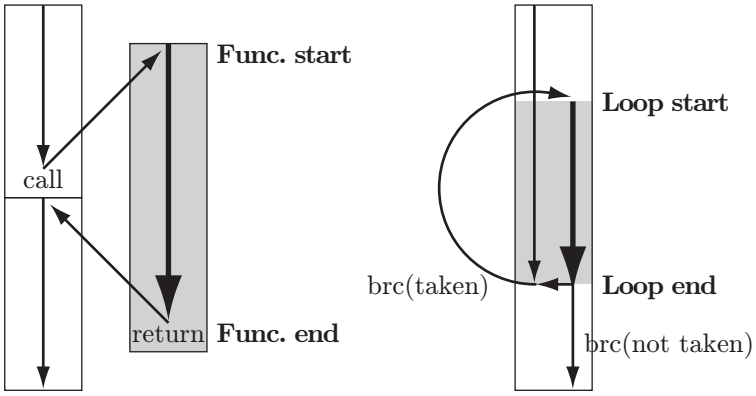
図 4: スタック

スタ退避に必要な領域を確保するために，積まれる関数フレーム分だけ%sp を減じ，元の値はフレームポインタと呼ばれるレジスタ%i6（%fp）に格納される．restore 命令の場合は逆の操作が行われる．また一般的に，主記憶上では，大域変数を格納するためのデータ域とスタックのためのデータ域との境界が OS により設けられる．この境界を LIMIT と呼ぶことにし，大域変数と局所変数の区別に用いる．なお，%sp は LIMIT を超えて減じられることはないとする．LIMIT 以上%sp 未満の領域は無効データ領域である．

関数は，call 命令，または，現在のプログラムカウンタ（PC）を%o7 に書き込む jmpl 命令により呼び出される．現在の PC の値が%o7 に格納され，関数の先頭アドレスに書き換えられる．なお，SPARC アーキテクチャでは，分岐先の命令を実行する前に分岐命令の次アドレスの命令が実行される．関数の引数は%o0〜%o5 に入る．引数が 7word 以上ある場合には，前述のようにスタックに格納する．この場合，第 7word は%sp+92 に，第 8word は%sp+96 に格納される．それ以降の引数も同様にスタックに積まれる．

さらに関数呼び出しを行う関数（非 leaf 関数）は save 命令を含む．save 命令の実行により，%i0〜%i5 に引数，%i6（%fp）に以前のスタックポインタ，%i7 には関数呼び出し時の PC の値が格納されている．返り値は 1word の場合%i0 に，2word の場合は%i0 および%i1 に格納される．さらに restore 命令によって返り値は，%o0 および%o1 に見えるようになる．

なお，leaf 関数の場合は，save 命令および restore 命令はなく，復帰には，第 1 オペランドが%o7+α（$\alpha > 0$）である jmpl 命令が用いられる．引数は%o0〜



(a) Function (b) Loop

図 5: 関数とループの類似性

%o5 が用いられ，返り値は%o0 および%o1 に格納される．

命令区間や入出力の特定を動的に行うために，再利用の対象を，始点と終点を容易に特定できる関数およびループとする．**図 5** に，関数とループの類似性を示す．call 命令から分岐する分岐先を始点とし，最初に到達した return 命令を終点とする命令区間を関数と認識する．関数の入力は，引数および大域変数である．同様にループは，後方分岐命令の分岐先から，同じ後方分岐命令までの命令区間として認識する．ただし，後方分岐命令を検出して初めてループの始点がわかるため，ループ 1 回目の始点は検出できない．また，ループ内の局所変数は動的に識別不可能であるため，参照されたレジスタおよび主記憶アドレスの全てを入力とみなして記録しておく必要がある．

ループにおいては，関数とは異なり，多重ループなど，複数の異なるループが同じ先頭アドレスを共有する場合がある．このため，ループの再利用では分岐先アドレスも再利用表に格納しておく必要がある．また，ループ内の局所変数は動的に識別不可能であることも関数の場合とは異なり，参照されたレジスタおよび主記憶アドレスの全てを記録しておく必要がある．ループが完了するより前に関数から復帰したり，前節で示したような入出力の容量オーバーなどによりループの入出力登録が中止されなかった場合，登録中のループに対応する後方分岐命令を検出した時点で登録中の入出力表エントリを有効にし，ループの登録を完了する．後方分岐命令が成立する場合には，次ループが再利用可能であるかどうかを関数と同様の手順で判断する．再利用した場合は再利用表に登録されている分岐方向に基づいて，さらに次のループに関して同様の処理

を繰り返す．一方，次のループが再利用不可能である場合は，さらに次のループの実行と再利用表への登録を開始する．

3.5 入力と出力の分類

3.5.1 入出力の記録

ループの場合，レジスタ参照や load のうち，自身が上書きする前の読み出しについては，レジスタ番号や主記憶アドレス，および各読み出し値の全てを入力として記録する．また，書き込みについては最終値が残るように逐次記録し，入力に対する上書きの検査にも用いる．

関数呼び出しについても同様に入出力を記録する．ただし，スタックフレーム上に配置する内部変数は，初期化後に読み出し，関数終了時に捨てる．SPARC アーキテクチャでは，大域変数は命令領域に続く低位アドレスに配置し，スタックフレームは高位アドレスから低位アドレスに向かって伸びる．大域変数とスタックフレーム下限の境界は OS が静的に決定すること，また，スタックフレーム間の境界は関数呼び出し直前のスタックポインタの値により決まることを利用して，あるアドレスが大域変数であるか，またはどの関数の局所変数であるかを識別できる．さらに SPARC アーキテクチャにはローカルレジスタの規定があり，同様に記録を除外できる．

さて，命令区間実行中に入力を記録する際には，既に出力側に登録されているかどうか検査する必要がある．重複登録を避けるためには入力側の検査も必要である．出力についても，既に出力側に登録している場合には上書きしなければならない．前述した多分木構造は，プログラムの入力パターンを素直に表現できるものの，一次キャッシュと同程度の高速性が要求されるハードウェア機構としては効率が悪い．このため，各命令区間実行中に一組の入力パターンを記録する小規模かつ高速な機構（**RW**）と，複数組の入力パターンを格納する大規模な構造**RB** は分けるべきと判断できる．

RB に要求されるアドレス解決は，RW から RB への登録を容易にするために，参照を時系列に並べアドレスを連想検索する方法を採用する．すなわち，既登録か否かを検査するための連想検索機構が必要である．**図 6** に RW の概要を示す．RW は，現在実行中の最内命令区間（レベル 1）から最外命令区間（レベル 6）のそれぞれについて一組の入出力を時系列に記録する構造である．より内側に新たな命令区間を検出した場合は，最外命令区間の記録（レベル 6）を破棄

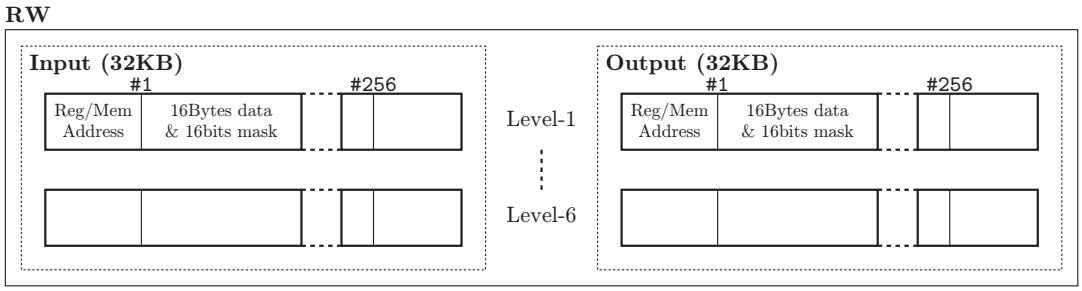


図 6: RW の構成

し，登録中のレベル 2〜6 に読み替え，空いた RW をレベル 1 として使用するリング構造としている．

高速に連想検索するために，連想度を 256 程度としている．また，入力側（in），出力側（out）の総容量をそれぞれ一次キャッシュ程度の 32KB に抑えつつ，入れ子関係にある 6 重の命令区間の入出力を記録するためには，1 つの命令区間あたり約 5KB が利用可能となる．1Byte ごとに 1bit のマスクビットを用意するとして，データに利用できるのは 4KB である．従って，16Byte（4KB/256）を 1 レコードとして，連続するレジスタまたは主記憶アドレスの内容を記録する．各レコードには先頭アドレス番号または先頭アドレスを付加する．

主スレッドについては以上の機構により，命令区間を実行しながら入出力を RW に記録できる．

3.5.2 投機スレッドにおける入出力

一方，投機スレッドは実行結果が保証できないため，キャッシュを経由した主記憶への書き込みはできない．命令区間の入出力に矛盾が生じないためには，ストア後のロードはキャッシュを参照してはならない．また，自身がストアしない限り同一アドレスからロードする値は同じでなければならない．前述のように，ループについては代わりに RW_{out} を出力先として利用できるものの，関数は内部変数の格納場所として RW 領域を利用できないため，一次キャッシュと同程度のローカルメモリを用意する必要がある．以上をまとめると，投機スレッドは，ローカルメモリ，RW，一次キャッシュを以下の優先順に参照しなければならない．

（1） ローカルメモリ

内部変数を再利用対象としないためには，手続きが参照する内部変数はローカルメモリへ，大域変数および上位の命令区間が使用するスタックフレームの参

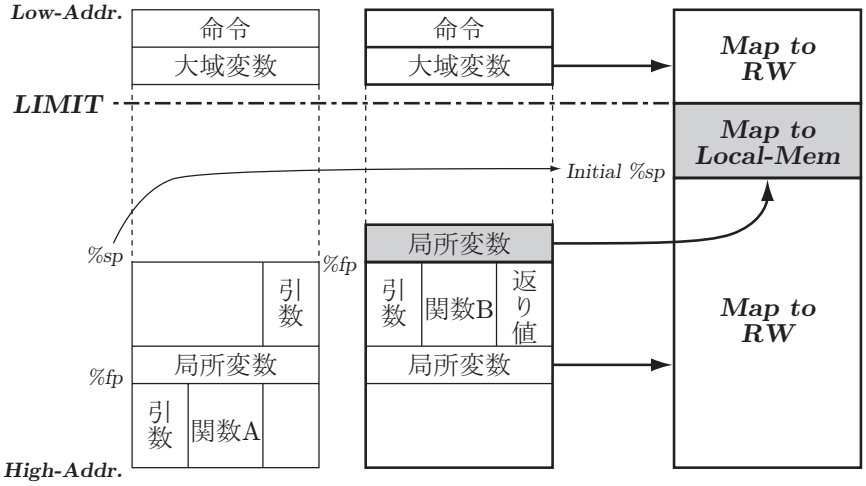


図 7: 投機スレッドにおけるスタックフレームと記録先の関連付け

照は入力として RW へ，それぞれ振り分ける必要がある．このためには，**図 7** に示すように，上位の命令区間が使用するスタックフレームを避けて，OS が静的に決定する大域変数とスタックフレーム下限の境界（LIMIT）にローカルメモリを割り付ける．さらにローカルメモリの最上位アドレスを投機スレッド開始時のスタックポインタ初期値とすることにより，この問題を解決できる．投機対象の最外区間がループの場合，フレームは作成せず，正常なプログラムである限り，ローカルメモリに該当する領域（LIMIT からスタックポインタ値まで）のアドレスを使用することもない．

（2） **レベル 1 の RW$_{out}$**

ローカルメモリ範囲外からの load は，自信が書き込んだ値を最優先するために，レベル 1 の RW$_{out}$ を優先する．

（3） **レベル 1 の RW$_{in}$**

RW$_{out}$ にない場合は，過去に一次キャッシュから読み出して RW$_{in}$ に登録したものを優先する．

（4） **上位 RW**

以上を最高レベルまで繰り返す．

（5） **一次キャッシュ**

いずれの RW にもない場合は，命令区間にとって初めての参照であるため，一次キャッシュを参照して RW$_{in}$ に登録する．

一方ストアについては，再利用しないアドレス範囲はローカルメモリへ，再

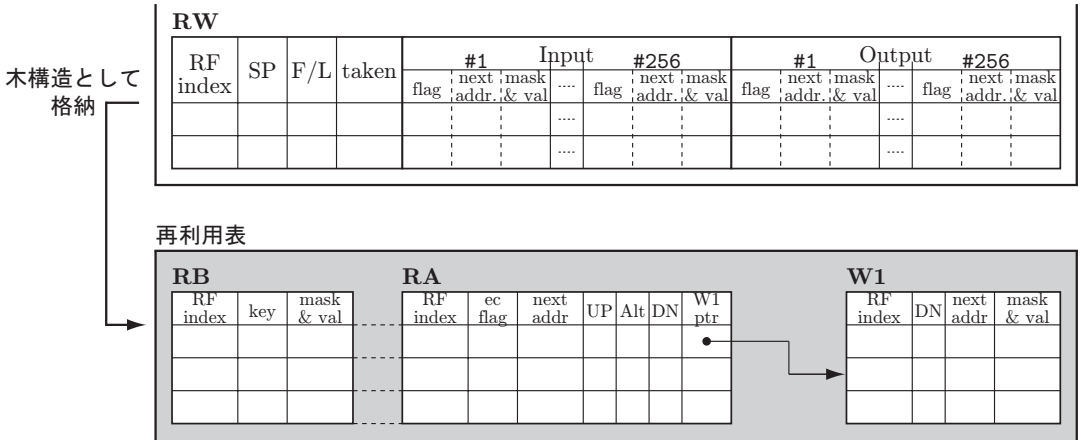


図 8: 再利用表の構成

利用するアドレス範囲は RW_{out} にそれぞれ格納する．また，予測値に基づく投機スレッドは，結果に誤りがあるだけでなく，区間の終点に到達しない可能性もある．以下の状況では投機スレッドを打ち切る必要がある．

ローカルメモリの容量超過　ローカルメモリの容量を超えてスタックフレームが伸びる場合，継続できない．

RW の容量超過　投機スレッドでは RW が主記憶の投機状態を保持するため，レベルの深さが許容範囲を超えた場合，主スレッドのように最外命令区間の登録を破棄することはできず，継続できない．最外レベルの RW があふれた場合も同様である．

例外やシステムコールの検出　例外やシステムコール検出時は RW と主記憶値の一致が保証できないため，継続できない．

実行命令数の異常　主スレッドの実行履歴と比較して，投機スレッドの実効命令数が極端に多い場合は異常とみなす．命令区間の実行に要した最新の命令数を記録しておき，4K ステップ未満の区間を投機実行の対象とする．また，実行打ち切りの上限を 16K ステップと仮定する．

3.6　再利用表の構成

命令区間の実行が終了すると，RW に生成された入出力セットを，再利用表本体に格納する．**図 8** に，再利用表の構成を示す．RB は入力セットを格納する**入力表**，RA は次に参照すべき入力の主記憶アドレスを格納する**アドレス表**，W1 は出力セットを格納する**出力表**である．

また，入出力セットがどの命令区間に属するかを識別するために，**区間表**RFを用意する．RFは命令区間の先頭アドレスを格納し，256種類程度の区間を管理することを想定している．

RW_{in}の1行分の入力を，図1で示した木構造の1パスとして再利用表に格納する．幅広のエントリを木構造により表現するアルゴリズム [27] に基づき，RBに木の各枝を，RAには各ノードを格納する [28]．このうち連想検索の対象となるRBをCAMで構成し，RAの各エントリはRBの各エントリと1対1に対応させる．

RBの各エントリは，当該エントリの親エントリを指すインデクス（key）と，入力値およびそのマスクを格納する部分（Val., Mask）からなる．そしてRAの同じエントリが，その入力の次に参照すべき主記憶アドレス（next addr.），およびそのアドレスに対する書き換えが発生したか否かを記憶するフラグ（flag）を保持する．なお，RA内のUP，Alt.，DNは，再利用テストを軽減するための拡張であり，3.8節において詳述する．

MSPは命令区間の実行開始時に再利用表を参照し，入力セットが完全に一致するエントリが見つかった場合，当該エントリに対応する出力をW1から一次キャッシュおよびレジスタに書き戻す．これにより，命令区間の実行が省略される．SSPは，再利用表のこれまでの入力セットから予測される入力セットを受け取り，投機実行を行う．投機実行の結果得られた入出力セットは，MSPと同様に命令区間の実行終了時にRWから再利用表本体に登録される．

3.7 再利用表の操作

本節では，再利用表に対する登録，削除，および検索の操作方法を述べる．ただし，前述のようにRW_{in}の1レコードには16Byte（4word）をまとめて格納するが，本節では簡単のために1レコードに1wordを格納するとして説明を進める．

3.7.1 RWへの登録

図9に示す命令区間を場合を考える．命令区間の先頭アドレスが`1000`であるとする．命令2はアドレス`A1`からロードした4Byteデータ`00110000`をレジスタ`R1`に格納する．この場合，アドレス`A1`およびデータ`00110000`が入力，レジスタ番号`R1`およびデータ`00110000`が出力となる．これらを時系列順にRWに登録する．

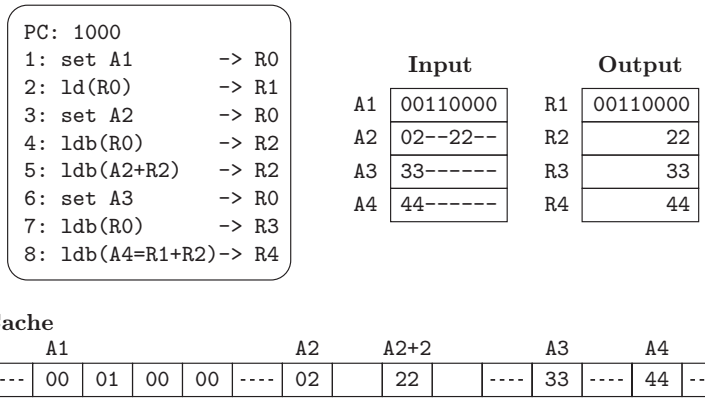


図 9: 命令区間の入出力セット

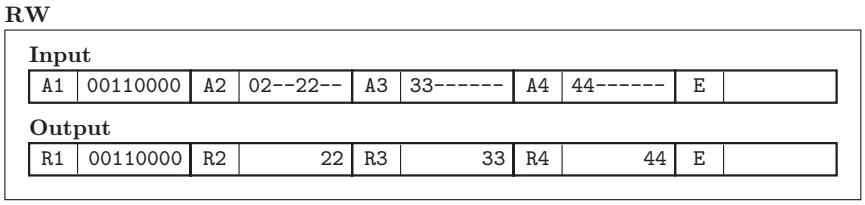


図 10: RW への登録

命令 4 では，A2 および 02 が入力，R2 および 02 が出力となる．この際，A2 の残り 3Byte については，*don't care* として登録する（図中「-」部）．命令 5 では，A2+02 および 22 を入力として登録する．この際，22 は A2 の項に追加登録され，命令 2 の入力と併せて 02--22--となる．A2+01，A2+03 相当部分は，*don't care* のままである．

同様に登録を行っていくが，命令 8 では R1 および R2 は当該命令区間内で上書きされたレジスタであり，命令区間の入力として登録する必要がない．このため A4 および 44 のみを入力として登録する．

以上のように入出力を時系列順に登録した結果，**図 10** のようになる．

3.7.2 再利用表への登録

命令区間の実行が終了すると，各 MSP/SSP の RW に一時的に登録しておいた入出力セットをまとめて再利用表に登録する．RB から空きエントリを検索し，空きエントリがなければ，次項に述べるように，タイムスタンプの最も古いエントリを一括削除する．空きエントリが見つかれば，RW の各レコードを 1 ノードとして順に登録する．

登録の結果，ある一つの命令区間に対する入力セットのパターンは，**図 11** 上

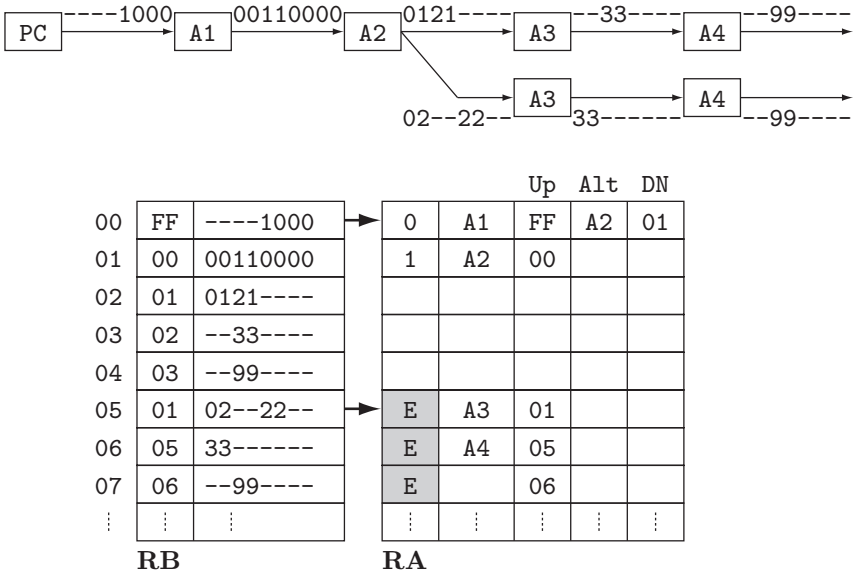


図 11: 再利用表への登録

のような木構造を形成する．ノードは参照すべき入力アドレス，枝はその格納値であり，根から葉までの経路数が，命令区間の入力セットのパターン数となる．再利用表上では各枝と次に参照すべきアドレスを 1 エントリとする．RB に当該枝の値と親エントリを指すインデクスを保存し，RA の同一インデクスに，次に参照すべきアドレスを登録する（図 11 下）．ただし，親が根である場合は -1 とする．さらに RA の各エントリに，

UP　親 RB エントリのインデクス

Alt.　次に検索すべきアドレス

DN　次に検索すべきインデクス

の 3 項目を追加する．これにより，検索時に値を比較する必要のないアドレスに対する比較を省略できる．

3.7.3　削除

再利用表の容量には限界があり，エントリの追加を続けると空きエントリがなくなってしまう．そのため，適宜エントリを削除する必要がある．エントリの削除は LRU に基づいて行う．RB のエントリに 4bit のタイムスタンプを用意し，これを用いてエントリのエイジングを行う．具体的には，システム全体の時刻を表すサイクリックな 4bit カウンタを用意し，命令区間の実行の際に，このカウンタをインクリメントする．インクリメントの後に，カウンタの値と同じタイムスタンプを保持している RB エントリを検索し，該当エントリを最も

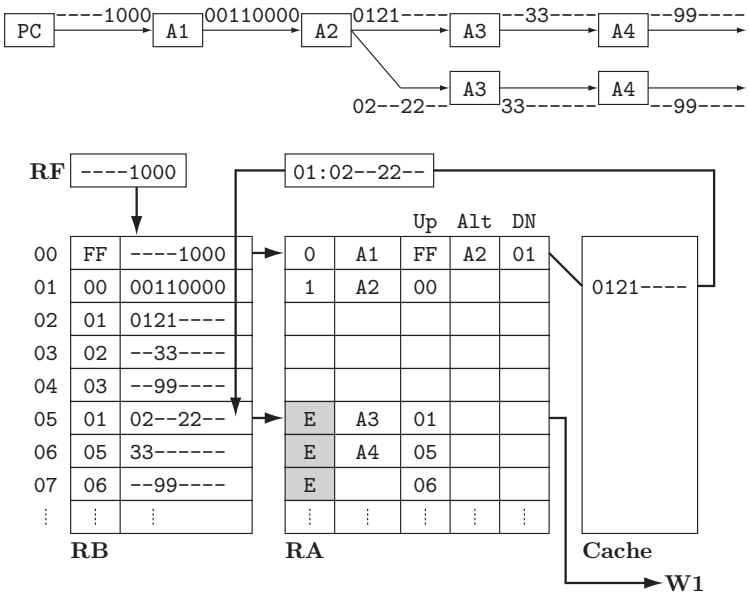


図 12: 検索

古いエントリとみなし，全て削除する．RA，W1 においても同様にタイムスタンプを保持し，LRU に基づいて削除する．

命令区間に対し再利用が行われた場合には，その入力セットを構成する RB エントリ，すなわち参照された全ての RB エントリのタイムスタンプを，現カウンタの値に更新する．これにより，木構造を構成する各ノードにおいて，親ノードのタイムスタンプが必ず子ノードのタイムスタンプと同じ，あるいはより現時刻に近くなることが保証できるため，上述した削除の際には必ず部分木のみが削除され，全体として木構造に矛盾が生じることはない．

また，RF に登録されている命令区間自体も LRU によりエントリの入れ替えを行う．その際には，当該 RF エントリのインデクスに基づいて，RB，RA，および W1 も削除する．

3.7.4 検索

図 11 のように入力セットが登録されているとする．検索は**図 12** のように行う．まず，命令区間の先頭アドレスであることを表すインデクスおよび先頭アドレスにより RB を検索する．例では先頭アドレスは 00001000 である．RB のインデクスのエントリがマッチし，RA の同インデクスのエントリが参照される．RA の該当エントリには次に参照すべき主記憶アドレス A1 が記憶されているものの，比較必要フラグがオフであり，A1 を参照する必要はない．DN によ

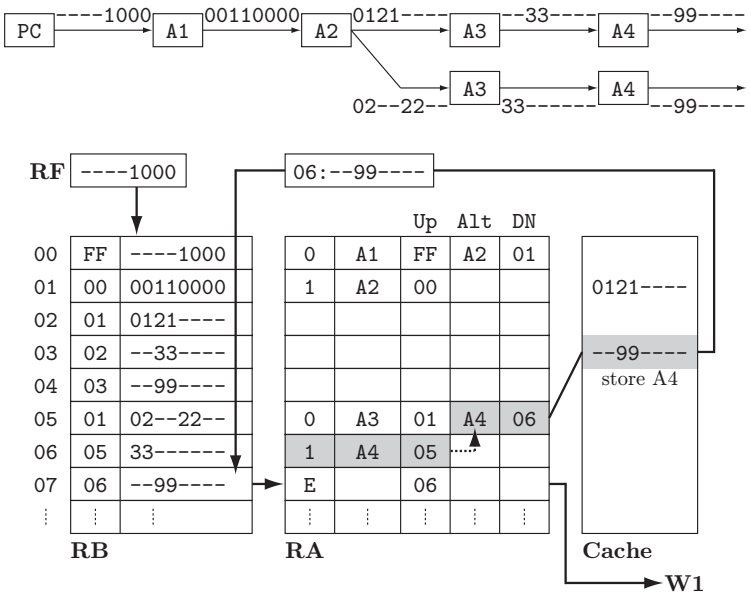


図 13: ストア後

り次ノードが 01 であることがわかる．01 はフラグがオンであり，A2 を参照し，A2 の値と当該 RB インデクスを用いて検索を続ける．

最終的に RA に終端フラグ Eが現れると，当該 RA エントリに格納されている W1 へのポインタにより，W1 から出力値を読み出す．

このように，フラグがオフの場合はそのアドレスにはストアが行われておらず，キャッシュから値を読み出す必要はないため，Alt. および DN をキーとして次の検索を行う．これによって，主記憶テストの必要がない途中の入力セットを迂回して検索を続けることができる．比較必要フラグがオンであるエントリに関しては，インデクスおよびキャッシュから読み出した値の双方をキーとして RB の検索を行う．

比較必要フラグがオフであるエントリに関しては，アドレス A4 に対しストアが発生したときの Alt.，DN に対する操作を**図 13** に示す．まず RA から A4 を連想検索し，マッチした行の比較必要フラグを立てる．次にその行の UP 項が示す RF の行（図 13 では 05）に関し，Alt. 項を A4 で，DN 項を A4 の格納されている RA のインデクス（図 13 では 06）で埋める．これにより，次回検索時には，ストア前には行われなかった A4 の内容に対する比較が行われるようになる．

3.8 主記憶テストの高速化

SpMTにおいては，主記憶一貫性の保証が重要となる．主記憶一貫性保証のための一般的手法として，全比較または無効化がある．**全比較**は，再利用時に，RBに記録されている主記憶値を全て比較検査する手法である．MSPによるstoreが発生した場合，主記憶の内容が書き換えられるものの，RBの各エントリに対しては特に何も操作は行わない．また，SSPによるstoreは主記憶自体を書き換えないため，これもやはり特に操作は行わない．一方，再利用テスト時においては，入力となる全ての主記憶値について参照，比較を行う．これにより，MSPによるstoreにより登録時と再利用時の主記憶値に不整合が生じている場合においても，不正な再利用を抑止できる．しかし，全ての入力に関して主記憶値を比較するため，再利用テストのオーバヘッドが大きくなり，再利用による性能向上の効果が減少してしまうという問題がある．

これに対し，SpMTで最も一般的に用いられている手法は**無効化**（invalidation）である．RFに登録されている主記憶アドレスに対し，MSPによるstoreが発生した場合，当該アドレスを入力として参照するRBエントリを全て検索し，これを無効化する．SSPによるstoreは主記憶自体を書き換えないため，特にRBに対する操作は必要ない．一方，再利用テスト時においては，内容が変更された主記憶アドレスを入力として参照していたRBエントリは無効化されるため，RB内で有効であるエントリにおいては，主記憶値が変更されていないことが保証される．これにより，再利用時の主記憶値の比較が不要となり，再利用テストのオーバヘッドは大幅に削減される．しかし，有効なRBエントリが減少するため，再利用率が低下してしまい，全比較を用いた場合のような高速化は望めない．

そこで本論文では，主記憶において書き換えられた可能性のある箇所のみを比較する手法（以下，**一部比較**）を用いる[29]．各アドレスに対し比較の必要があるか否かを記憶しておき，再利用テスト時に，そのアドレスのみについて主記憶値の読み出しおよび比較を行う．

本機構を実現するために，RFに対し，各入力アドレスに1bitのフラグを付加する．そして，RFに登録したアドレスから読み出した主記憶値と，RB内に保持されている主記憶値を比較する必要があるか否かを，本フラグを用いて表す．新しい命令区間の実行に伴い，再利用表への登録を開始する時点で，この

フラグはリセットされる．

再利用表への登録後にMSPやI/Oが同一アドレスにstoreを行った場合，storeは実際に主記憶を更新するため，当該アドレスのフラグを立てる必要がある．RBへの登録時における主記憶書き込みの際には，まず登録されている主記憶入力アドレスが，これから登録しようとする主記憶書き込みアドレスと一致するか否かを検査する．一致した場合，前述のフラグをセットする．なお主記憶読み出しの際には，過去に当該主記憶に書き込んだことがある場合はその値を，書き込みがなく読み出したことがある場合はその値を使用する．いずれもない場合は実際に主記憶から読み出して値を得る．

SSPで発生したstoreは実際には主記憶値は更新せず，再利用表内のデータを更新するのみである．このとき一般には，同一命令区間におけるstore後loadはその区間の入力とはならないため，比較の必要は生じない．ただし再利用区間が多重構造をなしている場合，外側区間Aにおいてstoreした値を内側区間Bがloadし得るため，Bにおける再利用時にはその主記憶値の比較が必要となる．よってBの入力が当該アドレスを参照している場合，フラグを立てておく必要がある．また，さらに内側区間Cが存在し，入力が当該アドレスを参照している場合，もし将来Bが再利用表から追い出されても比較の必要の有無を記憶しておくために，Cの入力にもフラグを伝播しておく必要がある．以上のように，SSPによるstoreにおいては，区間をまたぐstore後loadが存在する場合，比較の必要が発生し，内側区間の全てにおいて当該アドレスのフラグを立てておく必要がある．

再利用テスト時は，RF内の主記憶入力アドレスにおいて，比較が必要であることを示すフラグが有効となっているRBエントリに関してのみ主記憶の比較を行う．比較が必要な全ての入力に関して値が一致すれば，記憶してある主記憶値を書き戻し，再利用を終了する．これにより，再利用率を損なうことなく，主記憶比較のためのオーバヘッドを抑えることができる．

3.9 投機実行対象区間の選択

プログラム中に出現する全ての命令区間を事前実行および再利用の対象とした場合，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おいては，再利用により削減されるサイクル数よりも再利用表操作に要するオーバヘッドの方が大きい場合も存在する．そこで，命令区間に対し，再利用の効果がある

か否かを事前に判断し，効果が得られないと判断された命令区間は，再利用の対象としないことにする．これにより，無駄な再利用を削減することができ，プログラム全体としての高速化が図れる．具体的には，命令区間の再利用により削減できるステップ数と，その再利用に必要となるオーバヘッドについて概算を行う小さなハードウェアを再利用表に付加する．これにより，再利用効果の評価が行え，実際に効果が得られると考えられる命令区間についてのみ再利用を行うことができる．

並列事前実行では，SSP による投機実行の対象とする命令区間をいかに選択するかが重要である．命令区間のうちでも，MSP による登録頻度が高く，SSP が登録したエントリの再利用頻度も高いものが，最も並列事前実行による効果が得られる．再利用機構では，この動的に変化する登録頻度や再利用頻度を把握するために，一定期間における登録および再利用の状況を，シフトレジスタを用いて記憶している．そこで，この記録をオーバヘッドの算出に利用することにする．

命令区間 i に関し，実行に要するサイクル数から再利用のコストを差し引いたゲインを G_i，MSP が実行後 RB に登録した回数を X_i，再利用回数を R_i，再利用回数のうち SSP が生成した RB パスが貢献した割合を S_i とする．$G_i < 0$ の場合は対象にならない．また，前述したように MSP 自身が RM に登録できない場合は $X_i + R_i = 0$ となる．再利用の可能性がある命令区間 i の出現回数は再利用回数に関わらず $X_i + R_i$ であるため，SSP の貢献により MSP が全て再利用した場合は $G_i \times (X_i + R_i)$ の高速化が可能となる．すなわち評価式 E_i は，

$$E_i = G_i \times (X_i + R_i) \times S_i \tag{1}$$

のように表現できる．これにより，常に再利用頻度が高いか削減ステップ数の多い命令区間について投機実行を行うことができる．

効率よく求めるために，G_i は直前の実行結果をそのまま用いる．また，X_i，R_i，S_i は，命令区間ごとに 3 本の 64bit シフトレジスタを設け，最近実行した 64 命令区間に対応する各ビット位置に 1 を立てて得られる 1 の合計（0 から 64 の範囲）により表現する．なお，S_i の初期値が 0 の場合は投機実行を開始しないため，ループを命令区間として最初に認識した時の初期値は 64（最大値）とし，ループの立ち上がり時に優先的に投機実行する．ループについては，MSP において後方分岐成立時に候補に含め，後方分岐不成立時には候補からはずす

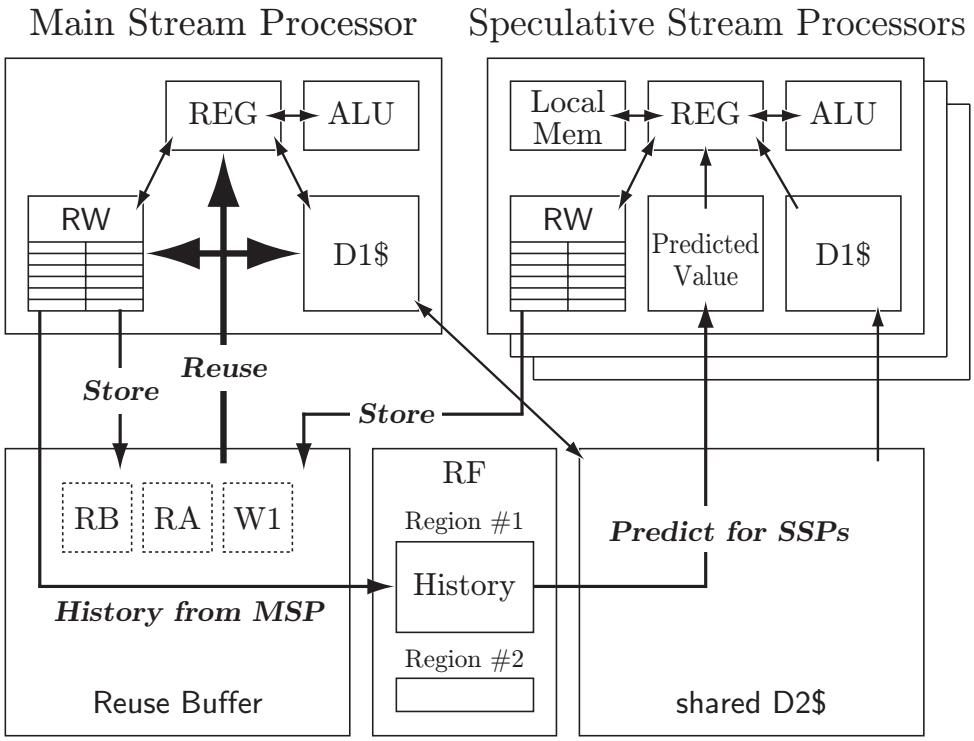


図 14: SpMT のハードウェアモデル

ことにより精度を高める．

SSP に投機実行をさせるためには，過去の履歴に基づいて将来の入力を予測する必要がある．このために，RF の各エントリごとに小さなハードウェアを用意し，MSP や SSP とは独立に入力予測値を求める．具体的には，最後に出現した入力 a および最近出現した 2 組の入力の差分 d に基づいてストライド予測 [2] を行う．$a+d$ に基づく命令区間の実行は MSP が既に開始していると考えると，n 台の SSP が存在する場合，SSP_i に対し入力予測値 $a+d\times(i+1)$ を用意する．$a+d\times(n+2)$ は MSP に割り当てる．

3.10 ハードウェアモデル

これまでに述べてきた SpMT 機構のモデルを**図 14** に示す．主スレッドを担当する MSP および投機スレッドを担当する複数の SSP が，再利用表（Reuse Buffer）および二次キャッシュを共有する．RF では，ストライド予測により，MSP が実行あるいは再利用した命令区間の入力履歴から予測値を生成し，SSP 起動に間に合うように各 SSP の Predicted Value 領域へ送る．予測対象はレジスタ，定数アドレス，フレーム内定数アドレスである．RW_{in} を再利用表へ蓄積

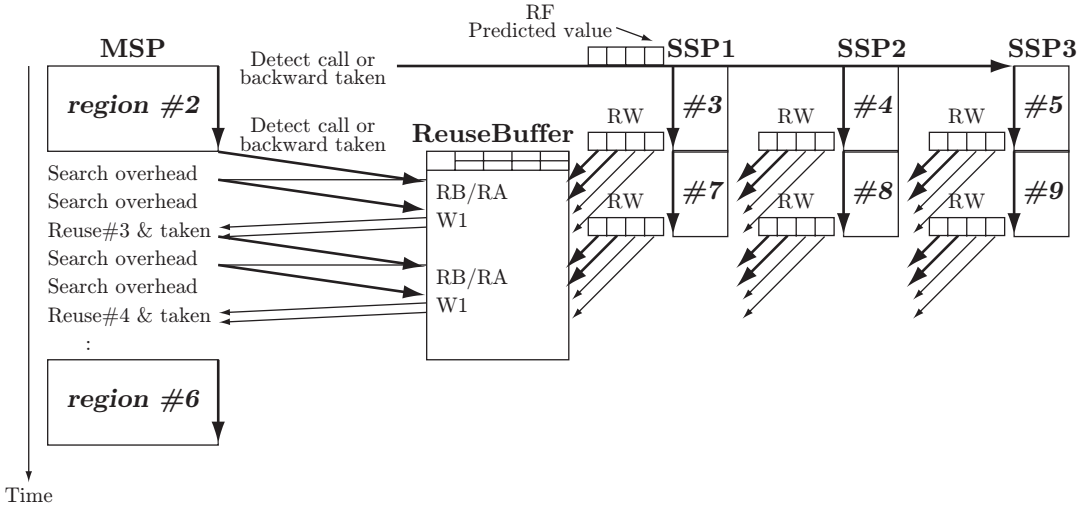


図 15: 評価モデル

する際，同時に入力履歴として RF に格納する．入力履歴は RW_{in} の 1 行分が時系列に 2 セット並んだ FIFO であり，フラグを立てたレコード単位にストライド予測を適用して予測値を求める．予測値のレコードも RW_{in} と同様に参照順に並ぶため，SSP は全予測値の転送を待たずに投機実行を開始できる．

SSP の load 命令は Predicted Value 領域の予測値を優先的に使用し，RW_{in} に登録する．以降は前述のように，(1) RW_{out}，(2) RW_{in} の優先順に参照するので，SSP から見た主記憶空間は他プロセッサの干渉を受けない．MSP および SSP は，各命令区間の入出力を各 RW へ記録し，命令区間実行完了時に再利用表へ送る．MSP は，後方分岐命令および関数呼び出し命令の検出と同時に RB の連想検索を行い，再利用可能なパスが存在する場合には，W1 の出力値をレジスタおよび主記憶アドレスに書き込む．

次に評価モデルについて述べる．RW や一時キャッシュは演算器およびレジスタと同じ速度で動作するとし，再利用表や二次キャッシュは内部のパイプライン動作によりスループットは確保するものの，演算器やレジスタに対しては長レイテンシとする．**図 15** に再利用表に関する評価モデルを示す．MSP が命令区間を検出すると SSP を起動する．また，MSP が命令区間を飛び越そうとする際には，再利用表の連想検索コストと，ヒット時の書き込みコストが生じる．再利用表の検索に必要な主記憶値を取得する際にキャッシュミスが発生した場合はサイクル数を加算する．SSP は前述のように投機実行対象区間を選択し，SSP が担当する命令区間の投機実行を完了した際には，RW から再利用表

に対して書き込みを開始すると同時に，空き RW エントリを用いて次の担当命令区間の実行を開始できると仮定する．また SSP が RW から再利用表に記録した各レコードは MSP が直ちに検索できるとする．再利用表へ記録するレコードは参照順であるため，SSP が全レコードの記録を完了する前に MSP が該当パスの検索を開始できる．

第4章　汎用CAMによるSpMT機構シミュレータの高速化

本章では，汎用CAM搭載PCIボード GP600Mによる，大規模再利用表を備えたSpMT機構シミュレータの高速化手法について述べる．

4.1　ソフトウェアシミュレータの問題点

前章で述べたSpMTの評価は，ソフトウェアによるシミュレータを用いて行っている．しかし，命令区間の実行のたびに行う再利用表からの連想検索を，ソフトウェアシミュレータではRAM上の1次元配列の逐次検索により実現しており，大規模な再利用表を想定したシミュレーションに非常に時間がかかる．例えば，再利用表の規模を4Kエントリとした場合，SPECfp95（train）を1本走行するのに3〜4日を要する．64Kエントリの場合は2カ月以上を要し，途中で計算サーバがダウンすることもあるため，極めて効率が悪い．さらに256Kエントリの場合，現実的時間ではシミュレーションが完了しない．

ところで，プログラムの高速化手法としては，ソフトウェアによるものと，専用ハードウェアを用いるものが考えられる．

ソフトウェアによる検索の高速化手法としては，ハッシュ法を用いることが考えられる．しかし本機構では，あるアドレス情報を保持するノードに接続された複数の枝について，互いにマスク情報が異なることが許されている．任意の位置に *don't care* を含んだ値からハッシュ値を求めるのは困難であり，ハッシュ法は使用できない．

一方，専用ハードウェアによる検索の高速化として，FPGAにより実現したCAMを用いることが考えられる．しかし，現在入手可能な最大規模のFPGA（Xilinx XC2V6000）を用いても，配線が極めて多いために288bit×256エントリ程度の非常に小規模な機構しか実現できていない．

以上のことから，シミュレータの高速化には，連想検索を高速に行うことができる大規模専用CAM-LSIの導入が不可欠である．そこで，汎用CAM搭載PCIボード GP600Mを用いたシミュレータを開発した[30, 31]．以下では，従来のソフトウェアシミュレータにおける再利用表の機能をGP600Mにより実現し，高速化を図る方法について述べる．

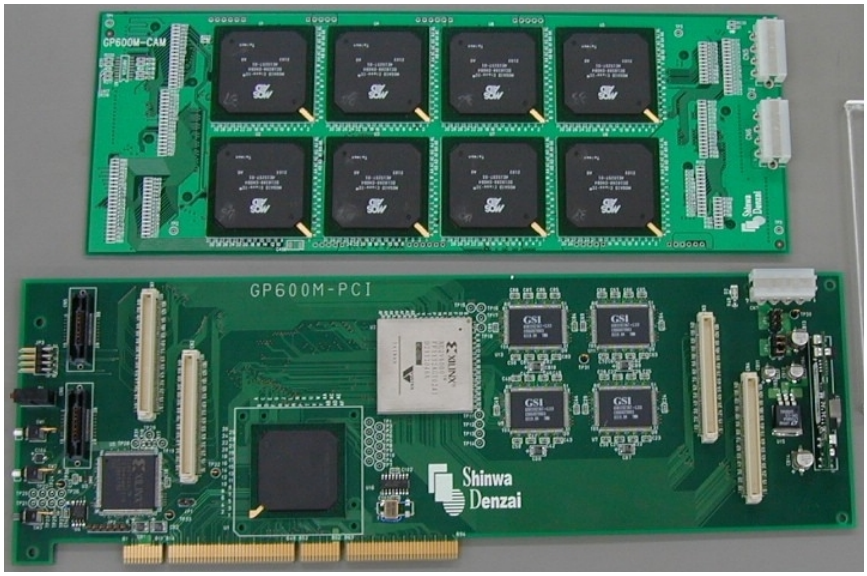


図 16: GP600M

4.2 GP600M の概要

GP600M は，新和電材株式会社製の汎用 CAM 搭載 PCI ボードである（**図 16**）．本ボードは CAM（MOSAID Class-IC DC18288[32]）を 8 個，および FPGA（Xilinx XC2V6000）を搭載している．

DC18288 は，IP ルーティングやスイッチング，QoS，データの暗号化や圧縮などに使われている．DC18288 はデータの幅とエントリ数を 3 種類のモードで利用でき，さらに，最大 8 個までカスケード接続することによりエントリ数を増やすことができる．各エントリにはそれぞれ，Empty，User，Age，Permanent のステータスビットが用意されている．Empty ビットはエントリにデータが書き込まれると自動的に 0 にセットされ，エントリが削除されると 1 にセットされる．**表 1** に DC18288 の仕様を示す．

DC18288 は，容量が 18Mbit の 3 値（Ternary）CAM である．すなわち，各ビットごとに 0，1，および *don't care* の値を持つことができる．**表 2** に，データとマスクの 3 値関係，および検索結果を示す．マスクが 1 であるビットが *don't care* を意味する．CAM 内のデータに対してある値の検索を行う場合，あるビットに注目すると，CAM データと検索データのどちらかが *don't care* であれば検索結果は「一致」となる．どちらも *don't care* ではない場合，CAM データと検索データが同じであれば「一致」，異なっていれば「不一致」となる．CAM のあるエントリの全ビットに対して一致すれば，そのエントリがマッチしたこと

表 1: DC18288 の仕様

周波数	83MHz
検索時間	96ns（パイプライン検索時 12ns）
容量	18Mbit
データエントリ数	72bit×256K エントリ 144bit×128K エントリ 288bit×64K エントリ
マスクレジスタ数	32 個
最大カスケード数	8 個
電圧	V_{DD}: 1.8V V_{DDQ}: 2.3 or 3.3V
パッケージ	432pin PBGA

表 2: 3 値データと検索結果

CAM データ		検索データ		結果
データ	マスク	データ	マスク	
0/1	0	0/1	1	一致
0/1	1	0/1	0	一致
0	0	0	0	一致
0	0	1	0	不一致
1	0	0	0	不一致
1	0	1	0	一致

になる．また，*don't care* となっているデータを読み込むと，データビットが 1，マスクビットが 1 とする値が得られる．

本論文では，DC18288 を 2 種類のモードで使用する．288bit×64K エントリのモードの DC18288 を 4 個カスケード接続し，256K エントリの CAM0 とする．さらに，144bit×128K エントリのモードで同様に 4 個カスケード接続し，512K エントリの CAM1 とする．CAM0 および CAM1 は互いに独立に動作する．すなわち，他の CAM が動作中であるか否かに関わらず，命令実行待ち状態にあ

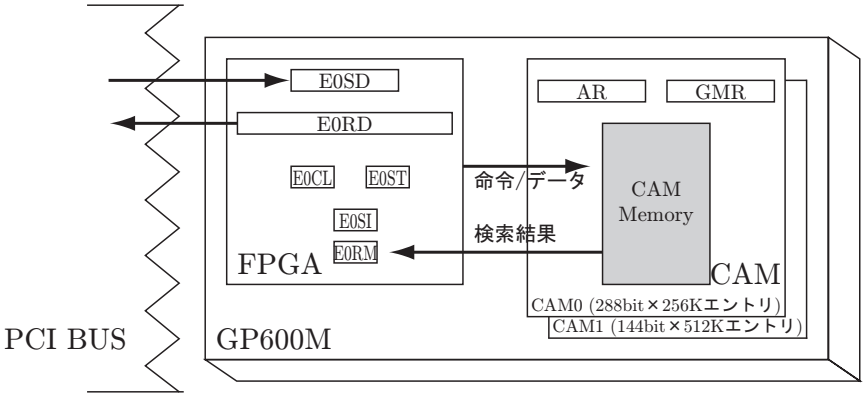


図 17: GP600M の構成

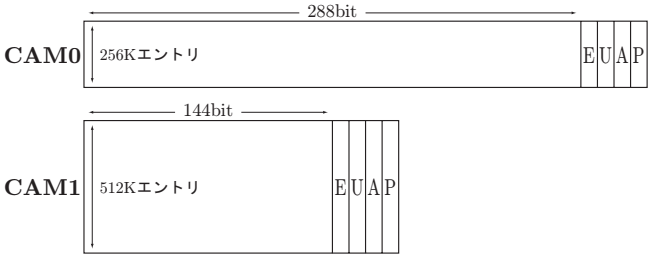


図 18: CAM の初期状態

る CAM に対して動作開始を指示し，正しい実行結果を得ることができる．**図 17** に GP600M の構成を示す．CAM0 および CAM1 は，後述する RESET 命令により，それぞれ**図 18** に示す構成に初期化される．

4.3 FPGA と CAM 間のインタフェース

市販の CAM は厳密なタイミング制御が必要であり，一般的なプロセッサ上で動作するソフトウェアシミュレータから利用するためには，FPGA を仲介したタイミング保証機構が必要である．すなわち，ソフトウェアシミュレータから観測できるのは，CAM そのものではなく，FPGA 上に実現されるインタフェースとなる．

GP600M 上の FPGA には，CAM にアクセスするための制御レジスタが用意されている．CAM0 制御レジスタを**図 19** に示す．CAM0 制御レジスタの詳細は以下の通りである．CAM1 についても，幅が 144bit である以外は同様である．

E0CL　STE0（STart External cam0）ビットに 1 を書き込むと，CAM0 が E0SI および E0SD により指定した動作を開始する．

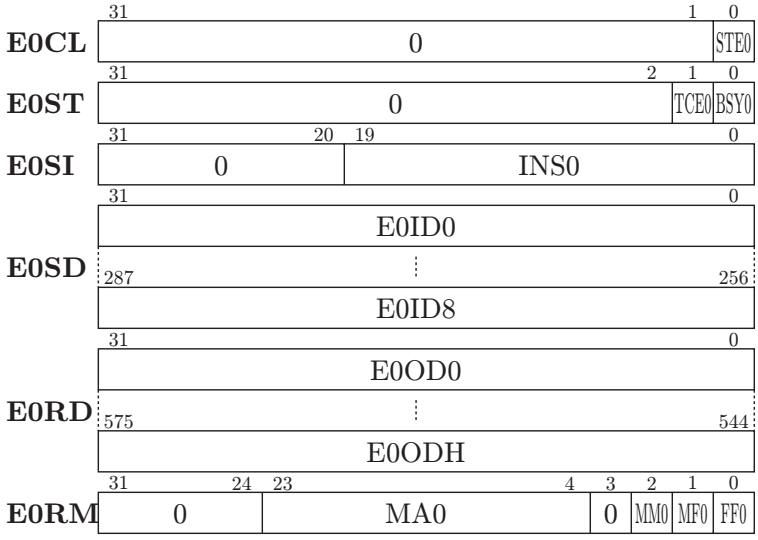


図 19: CAM0 制御レジスタ

E0ST　TCE0（Termination Code of External cam0）ビットが 1 の場合，STE0 ビットへの 1 書き込み時に指定した E0SIが未定義命令であったことを示す．BSY0（cam0 BuSY）ビットが 1 の場合，CAM0 が動作中であることを示す．

E0SI　INS0（INStruction for external cam0）に CAM0 の動作を指定する．主な命令の詳細については後述する．

E0SD　E0ID0-8（External cam0 Input Data 0-8）に CAM0 に書き込むデータを指定する．

E0RD　E0OD0-H（External cam0 Output Data 0-H）に CAM0 から読み出した値が表示される．

E0RM　後述する SSGL および SFRE 命令の実行結果が表示される．MA0（Match Address of external cam0）に CAM0 が出力する一致アドレスが表示される．MM0（Multi Match flag of external cam0）ビットが 0 の場合は複数一致を検出したことを示す．MF0（Match Flag of external cam0）ビットが 0 の場合は一致を検出したことを示す．FF0（Full Flag of external cam0）ビットが 0 の場合は Full を検出したことを示す．

制御レジスタへは，mmap システムコールによりメモリにマッピングされたアドレスに直接アクセスする．すなわち，

```
ureg = (unsigned char*) mmap((caddr_t)0, 0x200,
       PROT_READ|PROT_WRITE, MAP_SHARED, fd, (off_t)0));
```

としてureg にマッピングすることにより，例えばアドレス 0x148 の制御レジスタ E0CL に対して ureg[0x148] = 1; のようにアクセスできる．

GP600M 上の CAM は FPGA から命令を送信することにより動作する．GP600M における CAM マクロ命令には，主に以下の命令がある．

RESET（RESET） DC18288に定義されている powerup sequence および general reset を実行し，CAM の Configuration を行う．

CLEAR（CLEAR） CAM の全てのエントリにおいて Empty ビットを 1 にセットすることにより，全てのエントリを削除する．

WGMR（Write Global Mask Register） E0SD に格納されたマスクデータをグローバルマスクレジスタ（GMR）に書き込む．

RGMR（Write Global Mask Register） 指定した GMR からマスクデータを E0RD に転送し，E0RD からデータを読み込む．

WADR（Write ADdress Register） E0SD に格納されたアドレス値をアドレスレジスタ（ADR）に書き込む．

RADR（Read ADdress Register） ADR からアドレス値を E0RD に転送し，R0RD から値を読み込む．

WMEM（Write MEMory） WADR により指定したアドレスのエントリに，E0SD に格納されたデータを書き込む．

RMEM（Read MEMory） WADR により指定したアドレスのエントリからデータを E0RD に転送し，E0RD からデータを読み込む．

WSTB（Write STatus Bits） WADR により指定したアドレスのエントリにおけるステータスビットを変更する．

RSTB（Read STatus Bits） WADR により指定したアドレスのエントリからステータスビットを E0RD に転送し，E0RD から値を読み込む．

SSGL（Search SinGLe） WGMR により指定したマスクデータと E0SD に格納されたデータを用いてメモリ内を検索する．結果は E0RM に格納され，ヒットしたアドレスを返す．どのエントリにもヒットしない場合，−1 を返す．

SDEL（Search and DELete） SSGL と同様に検索を行い，該当した全エントリのステータスビットにおける Empty ビットを 1 にセットする．

SFRE（Search for next FREe address） 空きエントリ（Empty ビットが 1でないエントリ）のうちアドレスが最も小さいものを検索する．結果は E0RM

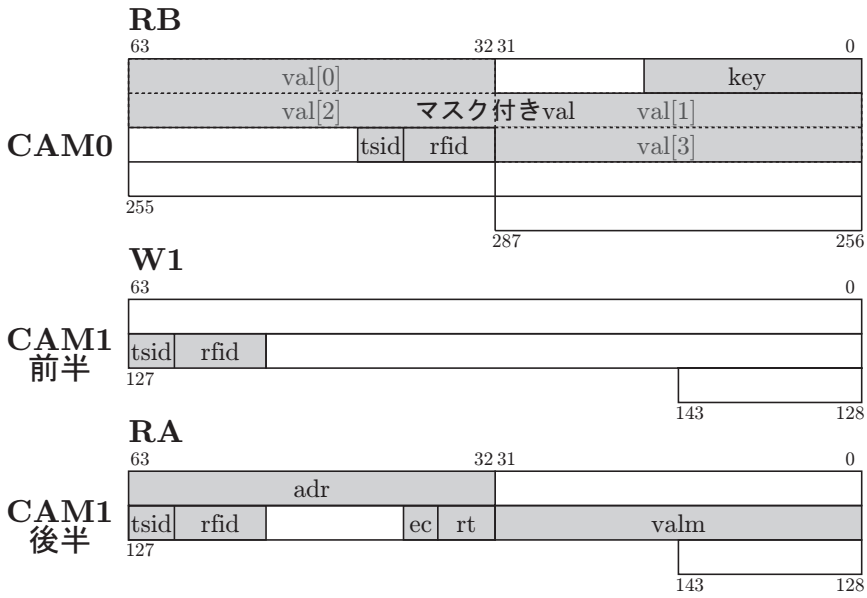


図 20: CAM へのデータの割り当て

に格納され，ヒットしたアドレスを返す．空きエントリが存在しない場合，−1 を返す．

4.4 CAM へのデータの割り当て

図 20 の RB，RA および W1 の CAM への割り当てについて詳述する．図の網掛け部が使用している部分である．RB は，最も古いエントリを消去する時に使用するタイムスタンプ tsid（4bit），RF から命令区間を削除する時に使用するインデクス rfid（8bit），親ノードの RB インデクス key（19bit），およびマスク付き入力データ val（4 ワード）の計 159bit からなり，幅 288bit の CAM0 に格納することができる．RA は，RB と同様の tsid（4bit）と rfid（8bit），入力アドレスに対し書き換えがあったか否かを示す値 ec（3bit），レジスタかメモリのどちらからの入力であるかを示す値 rt（5bit），次に参照すべきアドレス adr（32bit），次の入力となる値のマスク値 valm（32bit）の計 92bit からなり，幅 144bit の CAM1 に格納することができる．W1 は RB と同様の tsid（4bit）と rfid（8bit）の計 12bit からなり，幅 144bit の CAM1 に格納することができる．前述のように RB と RA のインデクスは対応しており，RA に対して SFRE 命令が発行されることはない．

以上のことから，汎用 CAM を用いた区間再利用プロセッサシミュレータで

(a) ソフトウェアによる空きエントリの逐次検索と登録

```
for (i = 0; i < RBSIZE; i++) {
  if (!rb[i].v) {
    add_entry(i);
    break;
  }
}
```

(b) CAMによる空きエントリの検索と登録

```
i = cam0_exec(SFRE);
cam0_exec(WADR, i)
cam0_exec(WMEM, data);
```

(c) 空きエントリ検索時はデータは使用しない

図 21: 空きエントリの検索と登録

は，CAM0 を RB，CAM1 を RA および W1 として使用する．CAM1 の前半を W1，後半を RA として使用し，W1 に対して SFRE を実行した結果，後半のアドレスがヒットすれば，W1 には空きエントリがないとみなす．

また，CAM へのアクセス速度は RAM より遅いため，RAM 上にも同じデータを格納しておき，値の読み込み時は RAM を用いるようにすることにより，高速化を図っている．

4.5 シミュレータの高速化

第 3 章で述べた再利用表の操作を CAM を用いて行う方法について述べる．

4.5.1 登録

図 21 に RB における空きエントリの検索と登録時のコードを示す．(a) はソフトウェアによる空きエントリの逐次検索と登録のコードである．要素数 RBSIZE の配列 rb を最初から順に調べ，空きエントリを検索する．ここで，RBSIZE は最大 256K の定数である．rb[i].v が 1 であるエントリが空きエントリである．空きエントリが見つかれば，add_entry() により RB にエントリを追加する．(b) は CAM による空きエントリの検索と登録のコードである．関数 cam0_exec() により SFRE 命令を発行すると，空きエントリのアドレス i が返される．その後，WADR 命令により i をアドレスレジスタに書き込み，WMEM 命令によりアドレス i のエントリにデータを書き込む．(c) は SFRE 命令において指定するデータであるが，本命令においては不要である．ただし，WMEM 命令によ

(a) ソフトウェアによる逐次削除

```
for (i = 0; i < RBSIZE; i++) {
  if (rb[i].v && rb[i].tsid == tsid)
    rb[i].v = 0;
}
```

(b) CAMによる一括削除

```
cam0_exec(SDEL, tsid);
```

(c) 削除時に使用するビット

図 22: 削除

るデータ登録時は全て使用する．

さらに，対応する RA にもエントリを登録する．W1 における空きエントリの検索と登録についても同様である．

4.5.2 削除

前述したように，エントリの削除はタイムスタンプに基づいた LRU により行う．**図 22** に削除時のコードを示す．(a) はソフトウェアによる逐次削除のコードである．前項と同様に，要素数 RBSIZE の配列 rb を最初から順に調べ，有効エントリかつ指定したタイムスタンプを持つエントリを検索し，該当エントリを全て無効化する．(b) は CAM による一括削除のコードである．関数 cam0_exec() により SFRE 命令を発行し，指定したタイムスタンプ tsid を持つエントリを全て削除する．(c) は SDEL 命令において使用する CAM のビットを示しており，網掛けの部分を使用する．ここではタイムスタンプのビットのみを用い，その他のビットは *don't care*，すなわちマスクビットを 1 としておく．

RA および W1 における削除についても同様である．また，RF におけるある区間の削除時には，rfid を有効にして削除を行う．

4.5.3 検索

図 23 に検索時のコードを示す．(a) はソフトウェアによる逐次検索のコードである．前項と同様に，要素数 RBSIZE の配列 rb を最初から順に調べ，有効エントリでかつ，親ノードアドレスを示す key が一致するエントリを検索する．さらに，そのエントリがの入力セット val が *don't care* のワードを除き全て一致すれば，当該エントリがヒットしたことになる．(b) は CAM による検索の

(a) ソフトウェアによる逐次検索

```
for (i = 0; i < RBSIZE; i++) {
  if (rb[i].v && rb[i].key == key) {
    for (j = 0; j < 4; j++) {
      if (rb[i].val_m[j]) {
        if (rb[i].val_m[j] & (rb[i].val_d[j] ^ val[j]))
          break; /* mismatch */
      }
    }
    if (j == 4) /* match */
      break;
  }
}
```

(b) CAMによる検索

```
i = cam0_exec(SSGL, data);
```

(c) 検索時に使用するビット

図 23: 検索

コードである．関数 cam0_exec() により SSGL 命令を発行すると，ヒットしたエントリのアドレス i が返される．(c) は SDEL 命令において使用する CAM のビットを示している．key とマスク付き入力セット val を使用する．

第5章　評価

本章では，GP600Mを用いたシミュレータの性能評価を行い，さらにそのシミュレータを用いて本SpMT機構の評価を行う．

5.1　GP600M自体の評価

まず，GP600Mにおける各CAM命令の実行時間を計測した．具体的には，各命令をそれぞれ100万回ずつ発行し，その実行に要した時間を測定し，1回の命令実行に要した平均時間を求めた．評価に用いたマシンは，Intel Xeon 2.8GHz，メモリ2GB，GP600Mを搭載したFreeBSD 4.10RのPCである．

結果を**表3**に示す．ほぼ全ての命令において，288bit幅のCAM0よりも144bit幅のCAM1のほうが実行速度が速い．CAMのRead/Write速度を比べると，グローバルマスクレジスタ，アドレスレジスタ，およびメモリのいずれのアクセスおいても，Readより Writeのほうが速い．そのため，前述したようにCAMに書き込んだデータと同じものをRAMにも記録しておき，Readの際はRAMから読み出すようにしている．

次に，実アプリケーションにおけるGP600Mの性能を調べるために，ソフトウェアシミュレータおよびGP600Mを用いたシミュレータによるStanfordベンチマークの実行時間を，timeコマンドにより計測した．

Stanfordベンチマークの各プログラムを，gcc-3.0.2（-msupersparc -O2）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: 各命令の実行時間 [μs]

命令	CAM0	CAM1
WGMR	4.26	3.21
RGMR	12.17	7.67
WADR	2.12	2.12
RADR	3.18	3.17
WMEM	6.29	5.20
RMEM	24.41	14.23

命令	CAM0	CAM1
SSGL	6.48	5.41
SDEL	6.47	5.40
SFRE	3.20	3.21

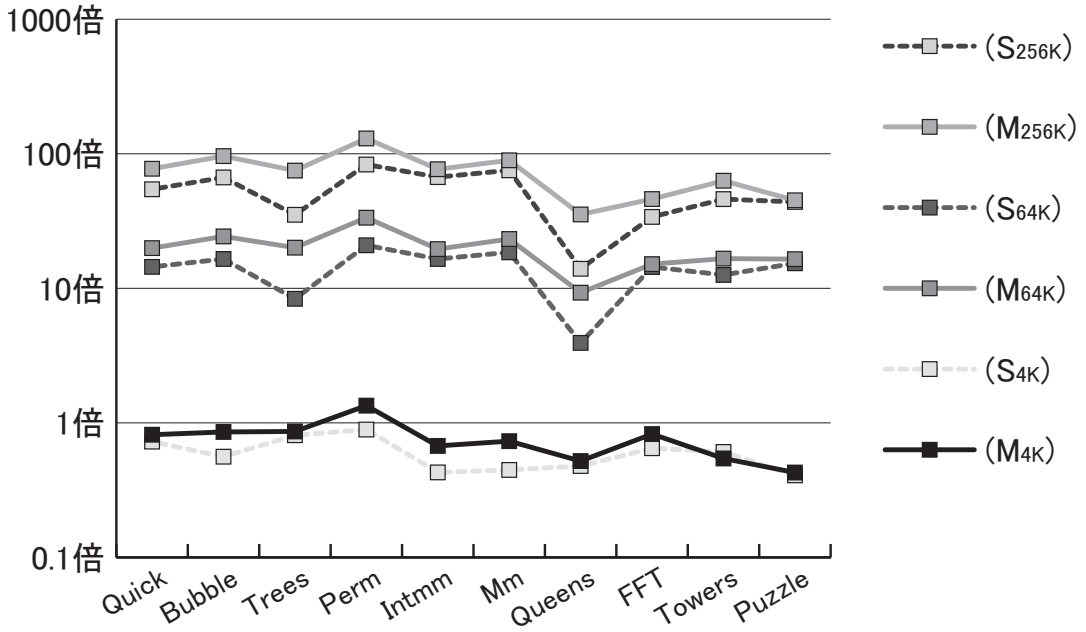


図 24: 高速化率

表 4: 高速化率の平均 [倍]

(M_{4K})	(S_{4K})	(M_{64K})	(S_{64K})	(M_{256K})	(S_{256K})
0.76	0.60	19.8	14.2	73.4	52.0

ソフトウェアシミュレータに対する高速化率のグラフを**図 24** に示す．各折れ線グラフは，使用したプロセッサと再利用表のエントリ数が，

(M_{4K}) MSP のみ，エントリ数 4K
(S_{4K}) MSP+SSP×3 台，エントリ数 4K
(M_{64K}) MSP のみ，エントリ数 64K
(S_{64K}) MSP+SSP×3 台，エントリ数 64K
(M_{256K}) MSP のみ，エントリ数 256K
(S_{256K}) MSP+SSP×3 台，エントリ数 256K

であることを示している．さらに，高速化率の平均を**表 4** に示す．

ソフトウェアシミュレータでは再利用表のエントリ数が増えるにつれて実行時間が長くなるのに対して，CAM を用いたシミュレータではエントリ数によらずほぼ一定である．そのため，再利用表のエントリ数が多いほど CAM による

高速化の効果が大きくなっている．

また，同じエントリ数の場合，MSP のみにおけるシミュレーションに対して SSP を追加したものは高速化率が低くなっている．これは，SSP による再利用表への登録回数が増えたことにより，比較的低速な CAM への書き込み回数が増加したためである．

一方，再利用表のエントリ数が 4K エントリの場合，逆にソフトウェアシミュレータよりも遅くなっている．これは，CAM による検索速度の向上よりも，CAM への書き込みがボトルネックとなったため，ソフトウェアによる逐次検索を行った方が高速なためである．

以上のことから，再利用表のエントリ数が 4K の場合はソフトウェアシミュレータ，それ以上の場合は GP600M を用いたシミュレータの方が効率的であるといえる．

5.2 SPEC95 による評価

本機構では，記憶できる入出力の組の数が再利用表の規模（エントリ数）によって決まる．規模が大きいほど多くの組を記憶でき，再利用表のヒット率も上がるものの，再利用表のレイテンシも増大し，再利用効果の低下が予想される．そこで，CAM と同規模のキャッシュのレイテンシを参考にし，CAM のレイテンシを考慮した SpMT 機構の評価を行った．

第 3 章で述べた SpMT 機構を実装した単命令発行の SPARC-V8 シミュレータに，第 4 章で述べた GP600M を追加したシミュレータを開発し，SPEC95 による MSP および SSP のサイクルベースシミュレーションを行った．前節と同様に，SPEC95 の各プログラムを gcc-3.0.2（-msupersparc -O2）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．SPEC95 の各プログラムには，test，train，ref の 3 種類が用意されており，実行時間は ref が最も長く，test が最も短い．本論文では train を用いた．

評価に用いた各パラメータを**表 5** に示す．なお，命令レイテンシは SPARC64-III[33] を参考にした．ハードウェア構成に関しては，一次キャッシュを 32KB，4way とし，共有二次キャッシュは 2MB，4way とした．また，共有二次キャッシュ，主記憶のレイテンシはそれぞれ，10cycle，100cycle と仮定した．なお，小規模な再利用表を用いた場合，再利用表のレイテンシとして，レジスタとの比較に 32Byte/cycle，キャッシュとの比較に 32Byte/2cycle（このパラメータセッ

表 5: シミュレーション時のパラメータ

D1 Cache 容量	32 KBytes
ラインサイズ	64 Bytes
ウェイ数	4
レイテンシ	2 cycles
Cache ミスペナルティ	10 cycles
共有 D2 Cache 容量	2 MBytes
ラインサイズ	64 Bytes
ウェイ数	4
レイテンシ	10 cycles
Cache ミスペナルティ	100 cycles
Register Window 数	4 sets
Window ミスペナルティ	20 cycles/set
ロードレイテンシ	2 cycles
整数乗算〃	8 cycles
整数除算〃	70 cycles
浮動小数点加減乗算〃	4 cycles
単精度浮動小数点除算〃	16 cycles
倍精度浮動小数点除算〃	19 cycles
Read アドレス	256 word/RW
Write アドレス	256 word/RW
RF エントリ数	256
SSP 台数	3

トを以下，$1\tau/2\tau$ と記す）を仮定している．一方，大規模な再利用表のレイテンシは $9\tau/10\tau$ を仮定している．このレイテンシは，同規模のキャッシュのレイテンシを参考にした．

MSP が一次キャッシュ/共有二次キャッシュに対し store を行うと，SSP 内の一次キャッシュにおいて invalidate が発生し，後続命令をそれぞれ 10cycle/100cycle だけ stall させる．以後 SSP には，一次キャッシュミスとして観測される．

SPECint95 の結果を**図 25** に，SPECfp95 の結果を**図 26** に示す．各ベンチマー

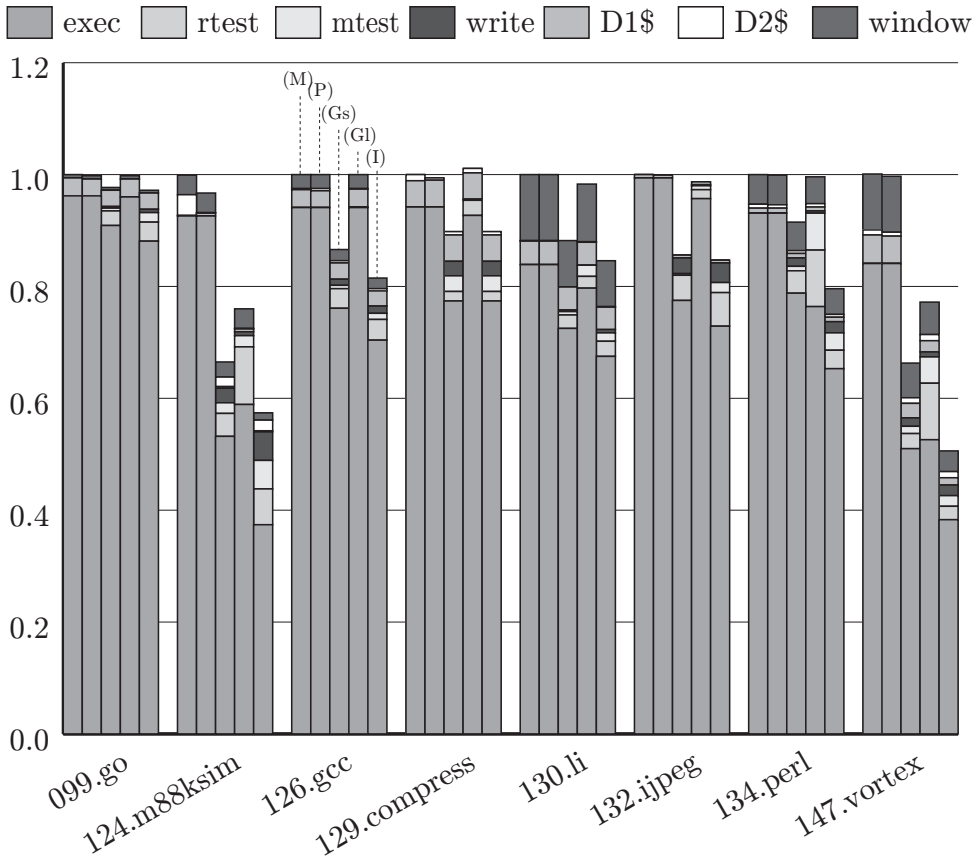


図 25: SPECint95

クプログラムのグラフは，左から順に，

（M） SSP と再利用のいずれもなし
（P） SSP×3 台，予備実行によるプリフェッチ
（Gs） SSP×3 台，エントリ数 4K，レイテンシ $1\tau/2\tau$
（Gl） SSP×3 台，エントリ数 256K，レイテンシ $9\tau/10\tau$
（I） SSP×3 台，エントリ数 256K，楽観的なレイテンシ $1\tau/2\tau$

を用いた場合に要したサイクル数であり，それぞれ（M）を 1 とした正規化を行っている．ここで,（P）の予備実行によるプリフェッチとは，予測された入力に基づいて SSP が事前実行を行うものの，結果は再利用表には登録せず，主記憶からロードした値を共有二次キャッシュに残しておくことにより，MSP による実行を高速化する手法である．SSP が実行する命令区間は，再利用機構と同様に選択している．ただし，式（1）において無意味となる S_i を除くことによ

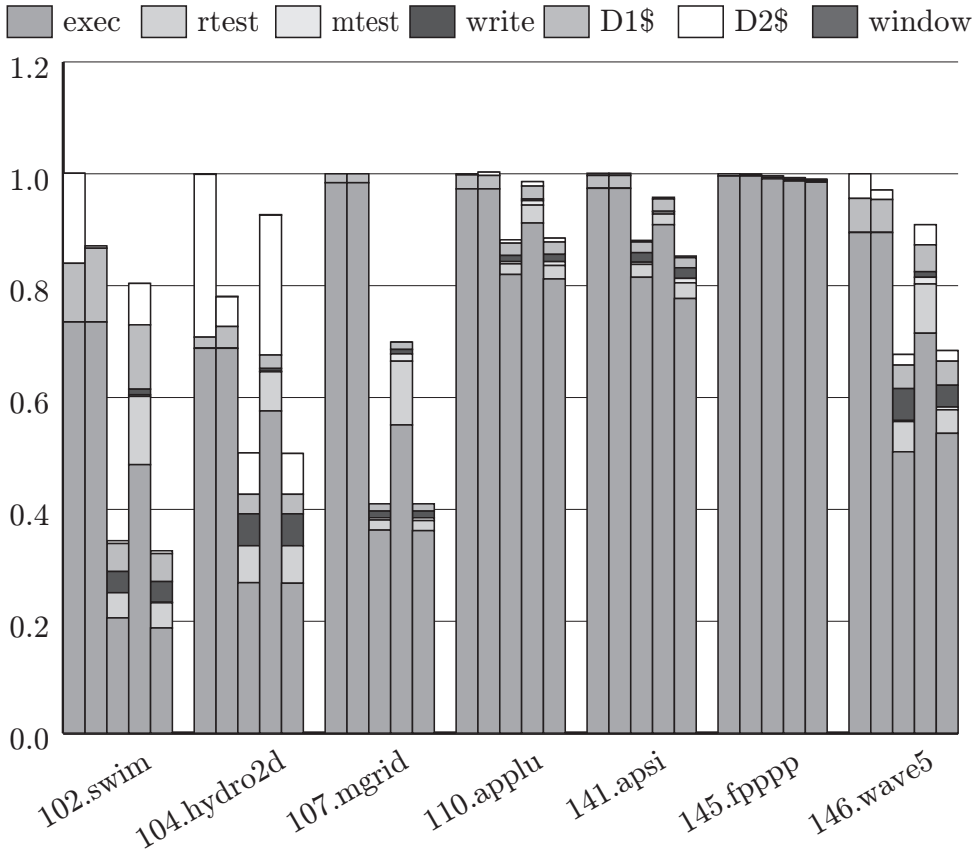


図 26: SPECfp95

り，実行に要するサイクル数が多く，かつ，最近の実行回数が多い区間を選択している．

グラフ中の凡例はサイクル数の内訳を示しており，exec は命令サイクル数である．rtest，mtest はそれぞれ，再利用表とレジスタ，再利用表とキャッシュの比較に要したサイクル数である．write は，命令区間の出力を再利用表からレジスタおよびキャッシュへ書き戻すのに要したサイクル数である．また，D1$, D2$，および window は，それぞれキャッシュミスペナルティとレジスタウィンドウミスによるペナルティを表している．

平均サイクル削減数は,（P）の 3%に対し,（Gs）は 27%,（Gm）は 8%,（Gl）は 31%となった．

（P）においては，プリフェッチにより二次キャッシュミスが減少しているものの，命令サイクル数自体は変わらない．それに対して，再利用においては命令サイクル数を大幅に削減できている．また，予備実行には及ばないものの，

102.swim や 104.hydro2d など，多くのプログラムにおいて二次キャッシュミスが大幅に削減されている．このように，再利用機構が共有二次キャッシュに対して，予備実行と同様に効果的なプリフェッチ機構としても働いていることがわかる．

さて,（Gs）と（Gl）を比較すると，再利用表のエントリ数が大規模な（Gl）の平均サイクル削減数が低くなっている．エントリ数の増加による性能向上よりも，再利用表を構成する CAM のレイテンシの増大の影響が大きくなり，性能が低下している．rtest や mtest だけでなく，命令サイクル数も増加しているのは，CAM のレイテンシが増加した結果，再利用によるオーバヘッドが増大し，オーバヘッド評価機構により，一部の短い命令区間が再利用されなくなったためである．

仮に将来，256K エントリの CAM においてもレイテンシが $1\tau/2\tau$ という楽観的な高速動作が実現できた場合,（I）のように,（Gs）よりもさらにサイクル削減数を向上できる．ところで,（Gs）と（I）において，大きく差があるものと，ほとんど差がないものがある．差があるものは，4K エントリでは足りないほどの多様な入力セットが存在することを示しており，差がないものは，4K エントリで十分であるか，または 256K エントリでも足りないことを示している．

以上のことから，予備実行によるプリフェッチに対して，4K エントリという小規模な再利用表を搭載した SpMT が有効であることがわかる．また，256K エントリという非常に大規模な再利用表を用いた場合，レイテンシの増加により，逆に性能が低下することがあるものの，レイテンシが減少すればさらなる性能向上が期待できることがわかる．

第6章　おわりに

本論文では，再利用機構を備えた多対1の非対称なデータ引継ぎ構造であるSpMTにおいて，再利用表の規模とレイテンシの観点から評価を行った．

大規模な再利用表を用いた場合のシミュレーションを高速化させるため，汎用CAMを搭載したハードウェアアクセラレータを開発し，ソフトウェアシミュレータに対して最大約70倍の高速化を達成した．

再利用表の規模が性能に与える効果を調べたところ，一般にSMTで採用されている予備実行によるMSPの実行サイクル数の平均削減率が3%程度であるのに対し，4Kエントリ（64KB相当）の小規模かつ高速（レイテンシ$1\tau/2\tau$）な再利用表を搭載したSpMTでは最小1%，最大66%，平均で27%のサイクル数を削減できた．一方，256Kエントリ（4MB相当）の大規模かつ低速（レイテンシ$9\tau/10\tau$）な再利用表では，平均削減率は8%に留まり，大規模化による性能向上よりも，CAMのレイテンシによる性能低下が目立つことがわかった．

今後の課題は，再利用表の規模がどの程度であれば最適であるかを調査することである．また，CAMの検索パイプライン機構を生かすための改良や，CAMの規模による性能低下を防ぐために再利用表を階層化することなどが考えられる．

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚なる謝意を表します．また，本研究に関して適切なご指導を賜った中島康彦助教授，森眞一郎助教授，五島正裕助手に心から感謝いたします．また，GP600M の開発においてお世話になった新和電材株式会社の蛭田由人氏，三精システム株式会社開発部の及部晴康氏に心から感謝いたします．さらに，日頃からご助力頂いた京都大学大学院情報学研究科通信情報システム専攻富田研究室の諸兄に心から感謝いたします．

参考文献

[1] Lipasti, M.H. and Shen, J.P.: Exceeding the Dataflow Limit via Value Prediction, *29th MICRO*, pp.226-237（1996）.

[2] Wang, K. and Franklin, M.: Highly Accurate Data Value Prediction Using Hybrid Predictors, *30th MICRO*, pp.281-290（1997）.

[3] Collins, J.D., Wang, H., Thllsen, D.M., Hughes, C., Lee, Y., Lavery, D. and Shen, J.P.: Speculative Precomputation: Long-range Prefetching of Delinquent Loads, *28th International Symposium on Computer Architecture*（*ISCA*）, pp.14-25（2001）.

[4] Luk, C.: Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors, *ISCA'01*, pp.40.51（2001）.

[5] Collins, J.D., Tullsen, D.M., Wang, H. and Shen, J.P.: Dynamic Speculative Precomputation, *34th MICRO*, pp.306-317（2001）.

[6] Gopal, S., Vijaykumar, T.N., Smith, J.E. and Sohi, G.S.: Speculative Versioning Cache, *4th International Symposium on High-Performance Computer Architecture*（*HPCA*）, pp.195-205（1998）.

[7] Marcuello, P., González, A. and Tubella, J.: Speculative Multithreaded Processors, *International Conference on Supercomputing*（*ICS*）, pp.77-84（1998）.

[8] Marcuello, P., Tubella, J. and González, A.: Value Prediction for Speculative Multithreaded Architectures, *32nd MICRO*, pp.230-237（1999）.

[9] Oplinger, J.T., Heine, D.L. and Lam, M.S.: In Search of Speculative Thread-Level Parallelism, *International Conference on Parallel Architectures and Compilation Techniques*（*PACT*）, pp.303-313（1999）.

[10] Codrescu, L., Wills, D.S. and Meindl, J.: Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications, *IEEE Trans. Comput.*, Vol.50, No.1, pp.67-82（2001）.

[11] Marcuello, P. and González, A.: Thread-Spawning Schemes for Speculative Multithreading, *8th HPCA*, pp.55-64（2002）.

[12] Sodani, A. and Sohi, G.S.: Dynamic Instruction Reuse, *Proc. 24th ISCA*,

pp.194-205（1997）.

[13] González, A., Tubella, J. and Molina, C.: Trace-Level Reuse, *Proc. International Conference on Parallel Processing*, pp.30-37（1999）.

[14] Costa, A.T., França, F.M.G. and Filho, E.M.C.: The Dynamic Trace Memorization Reuse Technique, *PACT*, pp.92-99（2000）.

[15] Huang, J. and Lilja, D.J.: Exploiting Basic Block Value Locality with Block Reuse, *Proc, 5th HPCA*, pp.106-114（1999）.

[16] Connors, D.A. and Hwu, W.W.: Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results, *32nd MICRO*（1999）.

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

[18] Huang, J. and Lilja, D.J.: Exploring Sub-Block Value Reuse for Superscalar Processors, *PACT*（2000）.

[19] Álvarez, C., Corbal, J., Salamí, E. and Valero, M.: On the Potential of Tolerant Region Reuse for Multimedia Applications, *ICS'01*（2001）.

[20] 津邑公暁, 清水雄歩, 中島康彦, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: ステレオ画像処理を用いた曖昧再利用の評価, *Symposium on Advanced Computing Systems and Infrastructures*（*SACSIS*）*2003*, pp.97-104（2003）.

[21] Yang, J. and Gupta, R.: Load Redundancy Removal through Instruction Reuse, *International Conference on Parallel Processing*（*ICPP*）, pp.61-68（2000）.

[22] Önder, S, and Gupta, R.: Load and Store Reuse Using Register File Contents, *ICS'01*, pp.289-302（2001）.

[23] Roth, A. and Sohi, G.S.: Register Integration: A Simple and Efficient Implementation of Squash Reuse, *33rd MICRO*（2000）.

[24] Roth, A. and Sohi, G.S.: Speculative Data-Driven Multithreading, *7th HPCA*, pp.37-50（2001）.

[25] Wu, Y., Chen, D. and Fang, J.: Better Exploration of Region-Level Value Locality with Integrated Computation Reuse and Value Prediction, *28th ISCA*, pp.98-108（2001）.

[26] Molina, C., González, A. and Tubella, J.: Trace-Level Speculative Multithreaded Architecture, *International Conference on Computer Design (ICCD) '02* (2002).

[27] MUSIC SEMICONDUCTORS Inc.: *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edition (2003).

[28] 津邑公暁, 笠原寛壽, 清水雄歩, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 大容量汎用3値CAMを用いた並列事前実行機構の効率的実現, *SACSIS2004*, pp.251-259 (2004).

[29] 津邑公暁, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 並列事前実行機構における主記憶値テストの高速化, *ACS4*, pp.31-42 (2004).

[30] 清水雄歩, 笠原寛壽, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 汎用CAMを用いた区間再利用プロセッサシミュレータの高速化, 電子情報通信学会技術研究報告 *CPSY2004-16 SWoPP* 論文集, pp.43-48 (2004).

[31] 清水雄歩, 笠原寛壽, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 汎用CAMを用いた区間再利用プロセッサシミュレータの高速化, 平成*16*年度情報処理学会関西支部大会講演論文集, pp.175-178 (2004).

[32] MOSAID Technologies Inc.: *MOSAID Class-IC DC18288 Feature Sheet* (2003).

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