特別研究報告書

区間再利用を用いる
投機的マルチスレッディングの提案と評価

指導教官　　富田 眞治 教授

京都大学工学部情報学科

笠原 寛壽

平成17年2月4日

区間再利用を用いる
投機的マルチスレッディングの提案と評価

笠原 寛壽

内容梗概

スーパスカラ方式による性能向上が頭打ちになってきており，さらにIPCを向上させる手法である投機的マルチスレッディング（以下SpMT）の研究が多く行われている．ただし，従来のSpMTでは，投機実行中の先行スレッド１つのみを主スレッドが引継ぐ．これに対し本研究では，命令区間の再利用性に着目し，様々な順序により得られた実行結果を主スレッドが利用可能とすると，高い性能を得られると考えた．また，専用コンパイラを使わず，手続きやループ構造の動的な検出により，既存ロードモジュールへの適用を目指した．

本研究では，区間再利用機構を用いるSpMTを提案し，重要な４つの要素技術について各々どの程度貢献するか，詳細な調査を行った．第１は，再利用を試みるか否かを判断する機構である．命令区間の入出力を保持する外部連想バッファの検索オーバーヘッドにより，かえって性能が低下する場合，再利用を強制的に禁止する．第２は，命令区間の入出力を複数のグループに分割する機構である．互いに入出力依存が生じないようグループ分割可能なら，各グループごとに再利用の機会を増やせるだけでなく，複数グループの再利用テストを並列実行することで検索オーバーヘッドの削減が期待できる．第３は，アドレスの分類により，投機スレッドの主記憶値予測をより正確に行う機構である．参照すべき入力値のアドレスが変化する場合にも追随を狙う．第４は，投機スレッドが命令区間を実行する場合に，最外命令区間だけでなく内側命令区間の実行結果も保存する機構である．再利用の機会を広げることによる性能向上を狙う．

以上の機構を備える実行モデルを考案し，サイクルシミュレータを用いて評価を行った．全ての機構を搭載した場合，Stanford，Spec95の実行時間を平均32%，23%削減することができた．第１のオーバーヘッド評価機構を外した場合，平均28%，21%に性能が低下した．第２のグループ分割機構の効果はほとんど見られなかった．分析の結果，グループ分割可能な命令区間が多くないことが明らかになった．第３の主記憶値予測機構は，幅広い効果は得られなかったものの，compressでは最大5%のサイクル数を削減できることを確認した．第４の多重再利用機構もcompressで最大5%削減できることがわかった．

Proposal and Evaluation of Speculative Multi-Threading with Region Reuse

Hirohisa KASAHARA

Abstract

Many researches on speculative multithreading (SpMT) are reported for boosting the performance of microprocessors. In conventional SpMT, main thread takes over data from only a proceeding speculative thread. However, this paper focuses on the potential reusability of regions. Each set of input are accumulated in the associative buffer, and reused by main thread out-of-order. The nested regions are identified dynamically while executing legacy load modules generated by widely used conventional compilers, so that no recompilation or binary annotation is required.

This paper proposes a new SpMT model exploiting region reuse and focuses on four major mechanisms as follows. First, a mechanism to reduce the reuse overhead related to the associative search of a large buffer is proposed. The performance may fall if main thread tries to reuse regions with huge overhead, so that some mechanism is required to estimate the promisingness of each region. Second, a mechanism to divide a set of input and output into several groups is proposed. If several search for independent input can be started, the total latency of reuse test are significantly reduced and the reusability may increase. Third, a mechanism which correctly predicts the input for speculation based on memory addresses classification is revealed. Fourth, a multilevel region reuse is shown that exploits the reusability of internal regions in addition to outermost regions initially assigned to each speculative threads.

I developed a cycle simulator and evaluated these mechanism using Stanford and SPEC95. These mechanism with 64KB external buffer offers average speedup of 32% in Stanford-integer and 23% in SPEC95 compared to the base model. Considering first technique is not equipped, the performance degrades to average speedup of 28% in Stanford-integer and 21% in SPEC95. Second technique shows poor effectiveness. I found that the opportunity to divide regions is extremely little. Without third mechanism, the performance degrades 5% in compress. Finally, I found multilevel speculation boosts 5% in compress.

区間再利用を用いる
投機的マルチスレッディングの提案と評価

目次

第1章　はじめに

1.1　背景

現在，商用マイクロプロセッサが多く採用しているスーパースカラ方式では，一般的なプログラムにおいて並列実行できる命令の数はせいぜい２程度と言われており，ウインドウサイズを拡大し，命令発行数を増やすだけでは，今後の性能向上が見込めなくなってきている．

一方，次世代ハイエンドプロセッサにおいて消費電力を抑えつつ並列度を向上する中核的技術として，ハードウェア実装の視点からはチップマルチプロセッサ構成，また，プログラム実行モデルの視点からは投機的マルチスレッディング（Speculative Multi-Threading）が注目されている．

ただし普及するには，コンパイラによる緻密な並列化や専用命令の埋め込みを前提とせず，一般的なコンパイラが生成する平凡なロードモジュールを高速実行できるマルチスレッドモデルが必要である．本論文では，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実行モデルを提案する．既存研究との違いは，区間再利用と投機的マルチスレッディングを１つの実行モデルに統合した点にある．

1.2　関連研究

命令間に依存関係が存在する場合でも，先行命令列の実行結果を予測し，後続命令列の投機的実行を開始することにより，命令レベルの並列度を確保する研究が数多く行われており，代表的なものとして，SMT（Simultaneous Multithreading）および投機的マルチスレッディングがある．precomputation は，SMT におけるヘルパースレッドの先行実行により主スレッドのメモリレイテンシを縮める方法である．Luk は 3 本の投機スレッド，L1 キャッシュ32K(4way) ミス時 12cycle，L2 キャッシュ1MB(8way) ミス時 72cycle，コンパイラによる precomp 命令の配置およびソフトウェアによるレジスタ初期化を仮定して，平均 24%の性能向上を報告している [2]．同様に Roth らは 3 本の投機スレッド，L1 キャッシュ16K(2way) ミス時 6cycle，L2 キャッシュ256KB(4way) ミス時 100cycle，簡略化命令列（slice）の生成および map table のコピーによるレジスタ初期化を

仮定して，最大 25%(vpr) の性能向上を報告している [3]．Sohi らによる投機的マルチスレッディングの先駆的研究 [4] はコンパイラが生成したレジスタマスクに基づき，リング構造に配置した投機スレッドが先行スレッドを待ち合わせる．投機スレッドのロードアドレスに先行スレッドがストアした場合，値に関わらず投機スレッドを無効化する点が異なる．Akkary らによる DMT[6] は主スレッドが投機スレッドの開始地点に到着すると，主記憶値を含む出力値と次スレッドの予測値を比較し，異なる部分に関連する命令のみをトレースキャッシュから取り出して再実行（selective recovery）する．計 8 スレッドにより平均 35%の性能向上を報告している．Hammond らによる Hydra をベースとする投機実行 [7] はストア時に無効化しており，300〜3000 命令程度を対象とするのが最適としている．Marcuello らはループを対象とする 4 スレッドのリング構造により m88ksim を 45%高速化した [8]．SPECint95 の最内ループ回転数は 2〜6 程度と少なく最内ループのみの投機実行は効果が低いと報告している．Krishnan らはバイナリアノテーション [9] によりループを識別し投機スレッドを開始する方法を提案している．他と同様，先行スレッドのストアにより投機スレッドを無効化する．Steffan らはコンパイラによりループを識別し投機スレッドを開始する方法 [10] を提案している．競合アドレスにおけるスレッド待ち合わせ機構を備えている．さらに Marcuello らはリング構造ではなく完全結合の投機スレッドが高性能を発揮できることを報告している [11]．ただし競合アドレスについては待ち合わせるか perfect 予測を仮定している．

また，プログラムには値の局所性が存在することが指摘されている．値の局所性を利用しデータ依存を解消する手法として，値予測に基づく投機的マルチスレッディングと並んで区間再利用が提案されている．区間再利用 [12, 13, 14, 15] は，プログラムの一部分に関する入出力値を表に登録しておく．同じ箇所を再度実行するとき，入力値が既知の場合には，正しい出力値を瞬時に求めることができる．区間再利用の特長は，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である．副次的な効果として，冗長なロード／ストア命令や消費電力を削減できることも報告されている [16, 17]．Connors ら [18] は，コンパイラが切り出した再利用区間も用い，記憶可能な入出力レジスタ数を各々 8 とする表を用意し，SPEC ベンチマークプログラムを 10%から 60%短縮している．ただし，主記憶上の値は再利用の対象外としているため，適用範囲が限られる．Huang ら [13, 19] は，再利用区間内に閉じたレジスタ (dead register) をハード

ウェアに伝達し，出力値としての登録を抑制するようGCCを改良し，コンパイラの支援を受けた基本ブロックの再利用により，SPECベンチマークプログラムの実行時間を1%から14%短縮している．記憶可能なレジスタ値は，入力５，出力６，主記憶値は，入力４，出力３を仮定している．Wuら[20]は，再利用と投機を組み合わせた方法として，同様にコンパイラが再利用の区間の切り出しを行い，実行時に再利用可能である場合には再利用を行い，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研究を報告している．ただし，出力値の予測がはずれた場合，後続区間の投機的実行をキャンセルしなければならず，このためのオーバーヘッドが問題になる．

1.3 本研究の概要

コンパイラやバイナリアノテーションツール[9]の助けを借りずに区間再利用を行う際の課題は，どのように命令区間や入出力を識別するかである．２次キャッシュミスの隠蔽を狙う場合には，数十命令程度の最内ループや末端手続きを対象としたのでは効果が期待できない．複数のループや手続きを含む，より外側の入れ子構造を対象として，数百から数千命令規模の多重命令区間を対象とする機構を導入している．投機的マルチスレッディングにおいては，注目すべき命令区間や投機実行の効果が刻々と変化する中，投機スレッドにどの命令区間を実行させるかが課題となる．主スレッドの挙動に追随しながら，最適と思われる入れ子構造を選択する仕組みを取っている．

また，複数のスレッドが生成した各命令区間の入力値（投機スレッドの場合は入力予測値）および出力値をどのように結びつけるかを考慮しなければならない．従来の投機的マルチスレッディングでは先行スレッドの出力値と後続スレッドの入力予測値が異なる場合は，コンパイラの支援により先行スレッドの実行結果を後続スレッドが待ち合わせる[10, 11]か，後続スレッドを無効化する[4, 6, 7, 8, 9]のが一般的である．区間再利用により，前者に対しては，主スレッドが待ち合わせに巻き込まれて性能低下しない機構，後者に対しては，単に無効化せずしばらく保存しておき，命令区間の再利用性を活用して高速化できる機構を導入している．更に，命令区間の入出力関係を全て検証するのは非効率であるため，必要最小限の検証に絞る機構も導入している．検証によるオーバーヘッドを最小限に抑えたとしても，再利用可能性が低い場合や命令区間が

小さい場合に逆効果になることが在りうる．そこで本研究では，再利用によるオーバーヘッドを評価し，主スレッドの性能低下を抑制する仕組みを提案する．命令区間の細分化により，入出力関係の検証を並列化し高速化を図る手法も提案する．

また，本研究では，入出力記憶の構造により，投機スレッドの入力予測が不可能な場合の存在を指摘し，改善機構を提案する．

以下では，まず，区間再利用を用いた投機的マルチスレッディングについて説明する．次に，入出力検証に生じるオーバーヘッドを考慮し，再利用を実行するか否かを判断する評価機構について述べる．それから，更なる検索高速化および再利用率向上を図るために区間内の独立部を切り分ける手法を示す．また，投機スレッドで用いる入力予測を向上のための機構を説明する．さらに，従来の投機的マルチスレッディングの実装について述べ，本機構の投機的マルチスレッディングの性能について考察する．Stanford ベンチマークおよび Spec ベンチマークを用い各提案を評価する．

第２章　区間再利用と投機的マルチスレッディングの統合モデル

本研究の背景となっている再利用と並列事前実行による投機的マルチスレッディングについて説明する．本節では，第１の課題に関し，提案モデルの基本部分について述べる．

2.1　動的に検出可能な命令区間を利用する高速化

投機実行は，命令キャッシュや分岐予測ヒット率の向上を狙い，手続き呼び出しに関しては復帰直後の命令列（subroutine continuations）を割り当てるのが一般的である．ループに関しては，後方分岐不成立直後（loop continuations）を割り当てる方法 [6] や，ループ本体（loop iterations）を割り当てる方法 [7] がある．本提案は命令区間の飛び越しを狙うため，始点と終点を容易に特定できる手続き本体およびループ本体を対象とする．なお，コンパイラやバイナリアノテーションツールに頼らないため，図 1(a) に示すように，ループ１回目の始点は検出できない．仮に PC をインデックスとする表を設けても，異なるループが同一先頭アドレスを共有する場合があるため，始点を検出できない点に注意されたい．ループ２回目に入る後方分岐成立時に，分岐先（addr.A）を始点とし後方分岐命令自身（addr.B）を終点とする命令区間を認識できる．同様に call 命令検出時に，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手続きと認識する．なお，コンパイラの最適化によっては，手続きが return 命令により終結せず他の手続き先頭に無条件分岐することがある．この場合，最初に到達する return 命令までの区間を手続きとして認識する．図 1(b) および (c) は，ループや手続き呼び出しをいかに飛び越すかを示す．(b) は主スレッドのループ１回目終了を契機として，主スレッドが実行する２回目と，投機スレッドが実行する３および４回目が並列動作し，主スレッドが投機スレッドの実行結果を再利用することにより３および４回目を飛び越す．(c) は過去の手続き呼び出しを再利用することにより飛び越す．

2.2　命令区間を飛び越すための十分条件

投機スレッドがループや手続きをあらかじめ実行できたとして，主スレッドが命令区間を飛び越す十分条件を把握する必要がある．プログラムの基本機能

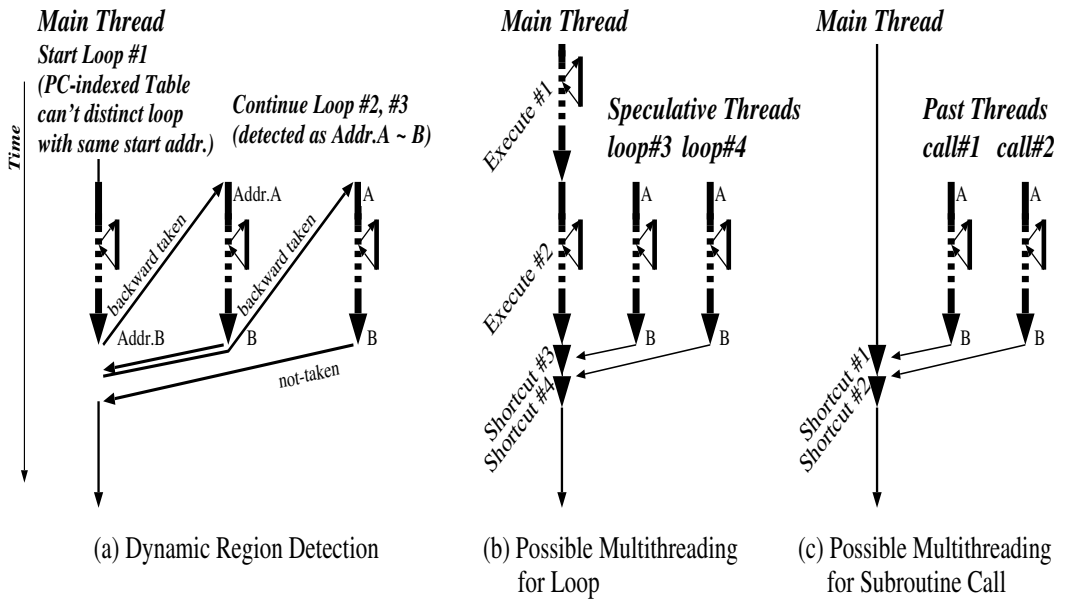

図 1: 動的に検出可能な命令区間を利用したマルチスレッディング．主スレッドが少ない手間でいかに命令区間を飛び越すかが鍵となる．

は，レジスタや主記憶アドレスを参照（入力）し，各内容に応じて処理を行い，結果をレジスタや主記憶アドレスに格納（出力）することである．命令自身が変更されない限り，入力が同じである間，実行結果も同一であり，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た場合，命令区間の入力パターンは，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，可変長の入力パターンを取り扱えるのみならず，枝に相当する記憶領域を節約できる．冒頭に述べた「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」ことは，まず命令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し，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ことに対応付けられる．もちろん，入力セット中の全入力値を比較する必要はなく，入出力の組を保存してから比較までの間に変更しなかったレジスタやキャッシュについては比較を省略できる．省略の具体的方法については後述する．

2.3 入出力の記録

ループの場合，レジスタ参照やロード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使用する．命令区間毎に，入力は初期値，出力は最終値のみを記録する点が，ARB[1] や SVC[5] と異なる．

手続き呼び出しについても同様の入出力を記録する．ただし，スタックフレーム上に配置する内部変数は，通常，初期化後に読み出し手続き終了時に捨てる．内部変数を初期化せずに読み出すプログラムは論理的に正しくないことを根拠に，内部変数は入出力記録から除外できる．一般に，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手続き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，どの手続きの局所変数であるかを識別できる．さらに，SPARC アーキテクチャのようにローカルレジスタの規定がある場合，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，既に出力側に登録しているかどうか検査（disambiguation）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（以後 RW と呼ぶ）と，複数組の入力パターンを格納する大規模な構造（以後 RB と呼ぶ）は分けるべきと判断できる．

RW に要求されるアドレス解決（disambiguation）には，参照を時系列に並べアドレスを連想検索する直接的な方法と，アドレスごとに時系列に並べる方法 [1, 5] がある．本論文では RW から RB への登録を容易にするために，参照順に並べる前者を採用する．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 2 に RW の概要を示す．RW は，現在実行中の最内命

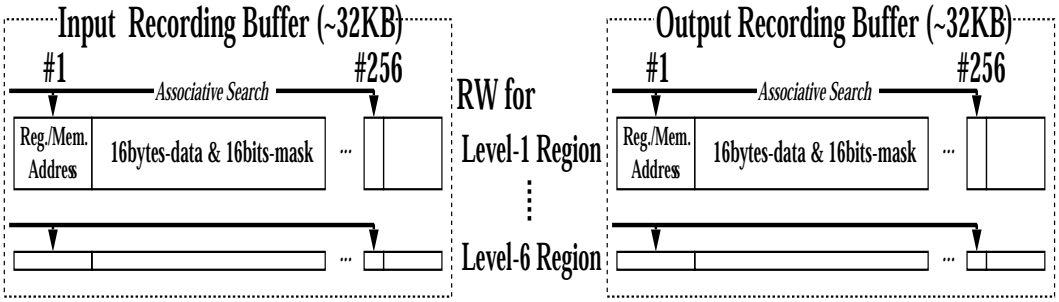

図 2: 連想検索機構を備えた多重入出力記録バッファ(RW)．In/Out の各総量は一次キャッシュ程度のハードウェア規模を想定している．

令区間（レベル 1）から最外命令区間（レベル 6）の各々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）を破棄し，登録中のレベル 1〜5 をレベル 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．各 RW に最高到達レベルを記録しておき，RB 登録時に RB_{out} に保存しておくことにより，再利用した命令区間のレベルを測定する．レベル 1 を多く再利用する場合は多重 RW が不要であり，レベル 2 以上が多い場合は必要と言える．

高速に連想検索するためには，連想度を 256 程度に抑える必要がある．また，入力側 (in)，出力側 (out) の総容量を各々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記録を採取するためには，1 つの命令区間あたり約 5KB が利用可能となる．1 バイト毎に 1 ビット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．したがって，16 バイト（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レジスタ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．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，手続き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

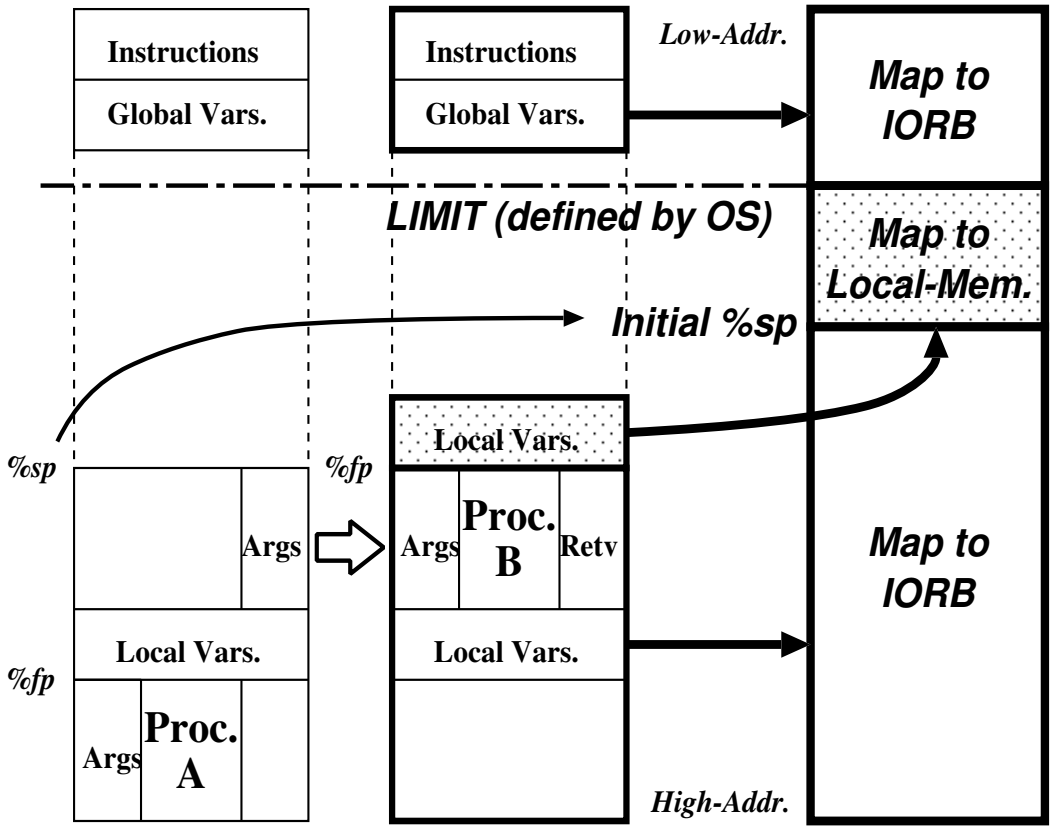

図 3: 投機スレッドにおけるスタックフレームと記録先の関連付け．投機スレッドが最初に手続きを実行する際には，スタックポインタを大域変数領域に続く低位アドレスに初期値しなければならない．

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照は入力として RW_{in} へ，各々振り分ける必要がある．このためには，図 3 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手続きの場合，手続き自身がローカルメモリ上にフレームを作成し，以後，フレーム内アドレスの参照にはローカルメモリを使用する．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ロードは，自身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，図 1 に示した終点（addr.B）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 が溢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行命令数が極端に多い場合は異常と見なす．命令区間の実行に要した最新の命令数を記録しておき，4Kstep 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 16Kstep と仮定する．

2.4　RB への蓄積

図 4 に RW および RB の構成例を示す．Strlen(str) は，NULL 文字により終端した文字列引数のバイト数を数える手続きとする．まず，第 1 引数に対応するレジスタ R0 から文字列”ABCDEF” の先頭アドレス (00010000 と仮定) を読み出してローカルレジスタ Rs に複写する．Rs が指すアドレスに NULL 文字を検出するまで Rs を１ずつ増加させ，文字列長 Rs-R0 を改めて R0 に格納

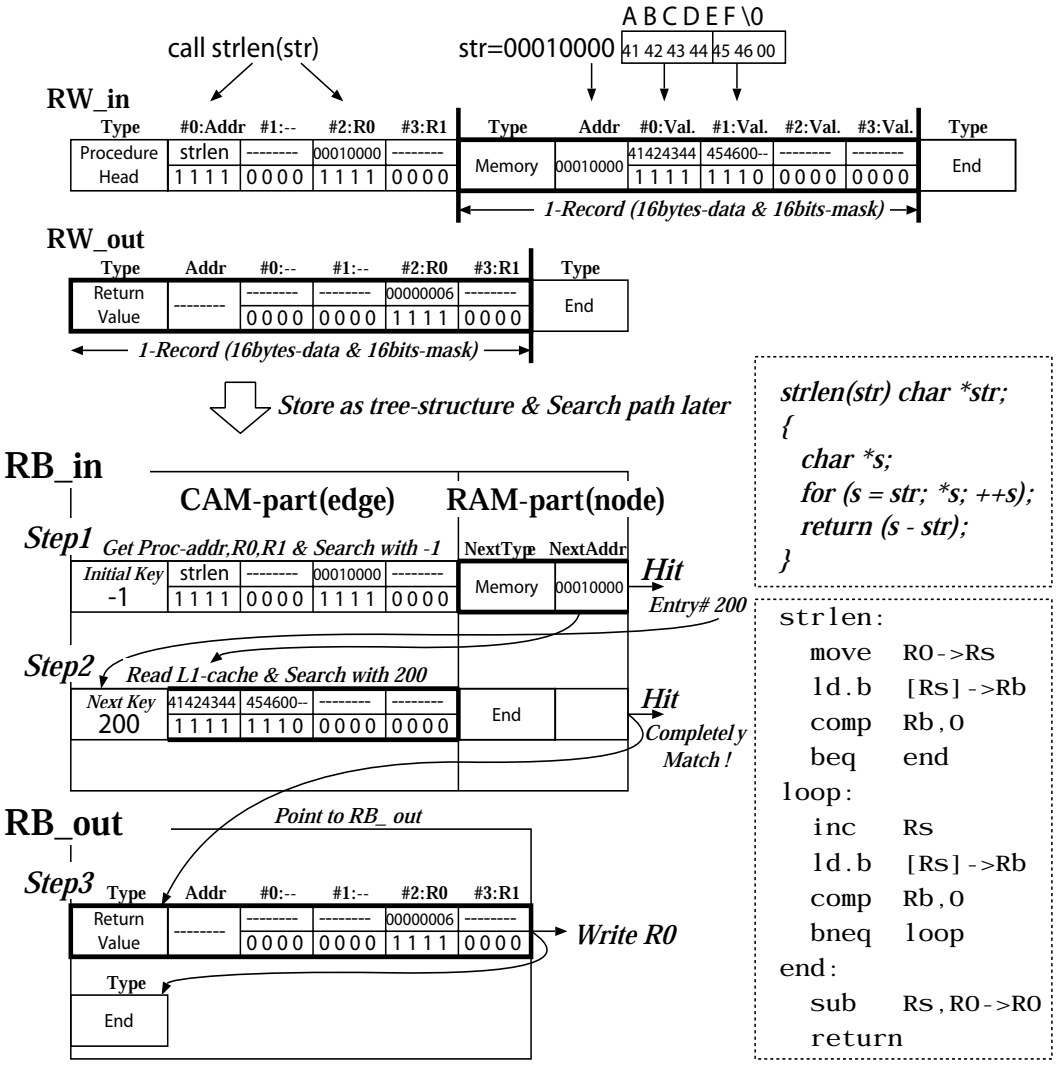

図4: RB の構成．RW_{in} の１行分の入力を木構造の１パスとして格納する．CAM-Part が枝，RAM-Part がノードに対応する．

し手続きを終了する．RW_{in} には，第 1 レコードに手続きの先頭アドレスおよびレジスタ R0 の内容 00010000，第 2 レコードにアドレス 00010000 と 7 バイト値”41424344454600”，第 3 レコードに終端を記録する．RW_{out} には，第 1 レコードにレジスタ R0 の内容 6，第 2 レコードに終端を記録する．一方 RB_{in} は，RW_{in} における 16 バイトデータ部分に検索キーを追加した CAM 部分と，RW_{in} の次レコードにおけるタイプおよびアドレスを保持する RAM 部分からなる．手続き終了時には，RW_{in} の先頭レコードから順に RB_{in} へ登録する．RW におけるアドレス 00010000 の内容が既に RB_{in} に登録した内容と異なる場合は，nextkey を 200 とする新たなエントリを追加する．

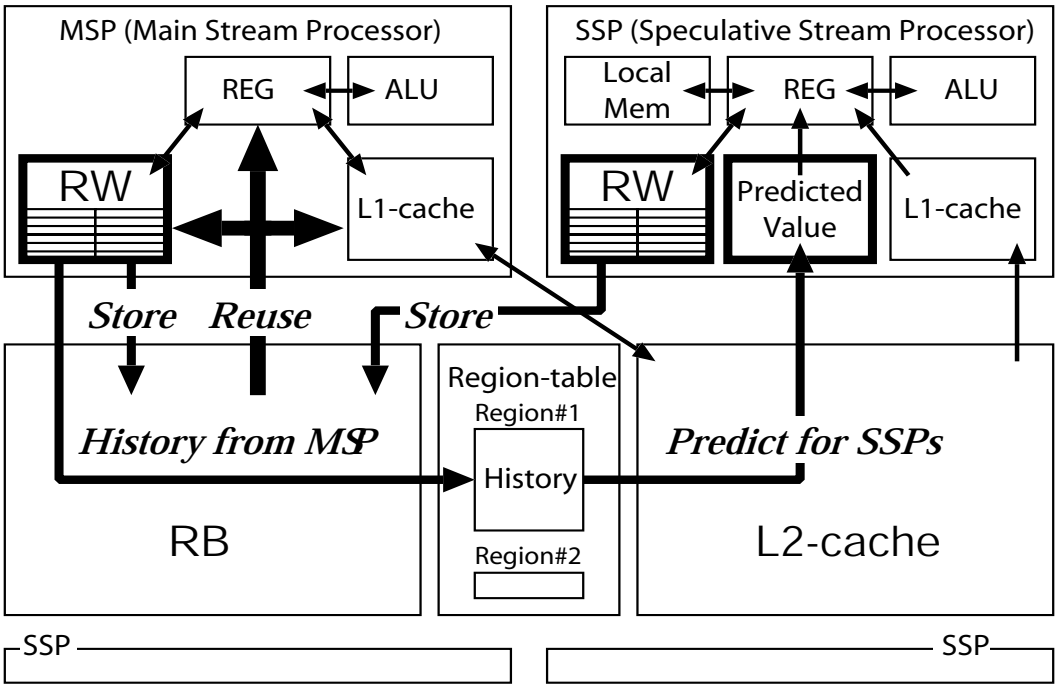

図 5: ハードウェアモデルへの写像．全てのプロセッサが RB と L2 キャッシュを共有する

以後 strlen(str) を呼び出す前に初期キー (-1)，strlen の先頭アドレスおよびレジスタ R0 の値により RB の CAM 部分を連想検索し，エントリ番号 (200) にヒットした場合はさらに RAM 部分の nextkey(200) および nextaddr から読み出した主記憶値を用いて RB の連想検索を繰り返すことにより，木構造から１つのパスを選択する．各バイトごとに検索マスクを設けているため，文字列長が異なっていても正しく検索できる．検索を繰り返し，RAM 部分に終端 (end) を検出した時，対応する RB_{out} が正しい出力値である．さらに各エントリにタイムスタンプを設け，参照したパスに属するエントリについてはタイムスタンプを更新し，古いタイムスタンプのエントリを定期的に消去することにより，長期間使用しない枝を刈り取ることができる．一般的な汎用 CAM には検索条件を満たす全エントリを一斉に無効化する機能があり，本機構でも同様の機能を仮定する．

2.5 ハードウェアモデルへの写像

図 5 に，本機構を装備するチップマルチプロセッサモデルを示す．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，RB および

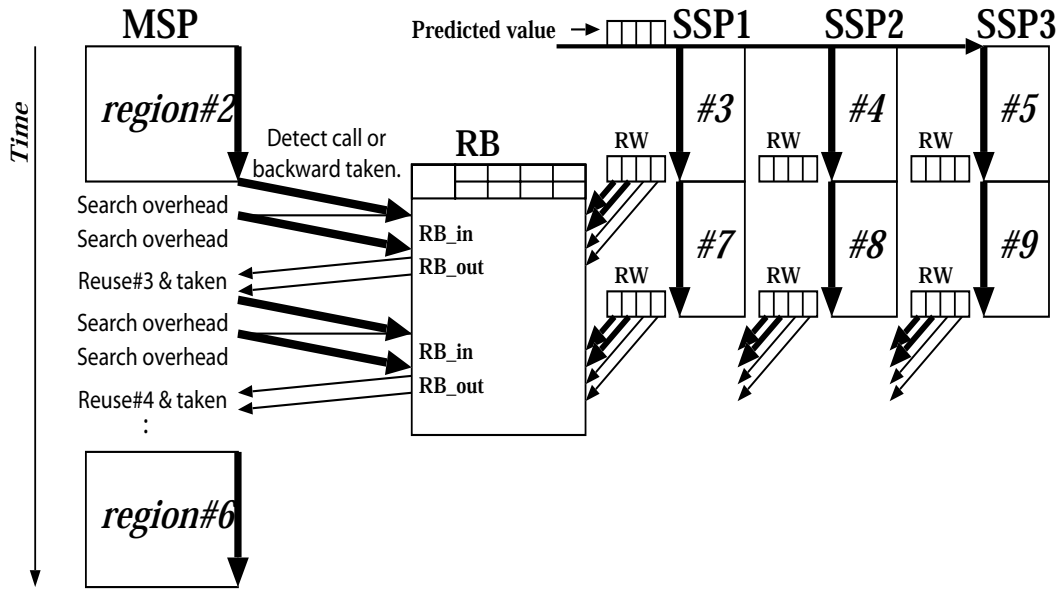

図 6: 評価モデル．RB は長レンテンシであるが検索と書き込みはオーバラップできるとする．

L2-cache を共有する．Region table 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．SPARC の場合，定数アドレスは sethi;ld，フレーム内定数アドレスは ld(%fp-const.) の各命令パターンから判別でき，RW_{in} の該当レコードにフラグを立てておく．RW_{in} を RB へ蓄積する際，同時に入力履歴として Region table 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 のロード命令は 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 RB へ送る．MSP は，後方分岐命令および手続き呼び出し命令の検出と同時に RB_{in} 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，RB_{out} 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 L1-cache は演算器およびレジスタと同じ速度で動作するとし，*RB* や L2-cache 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 6 に *RB* に関する評価モデルを示す．一般的な SpMT では，命令区間の検出と同時に投機スレッドを起動するのに対し，本モデルでは，Region table が自律的に有用と判断した命令区間および予測値を SSP に割り当てて起動する．また，MSP が命令区間を飛び越そうとする際には，*RB* の連想検索オーバーヘッドと，ヒット時の書き込みオーバーヘッドが生じる．*RB* 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 がどの命令区間の投機実行を開始するかについては後述するとして，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 *RB* 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 *RB* に記録した各レコードは MSP が直ちに検索できるとする．*RB* 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．

2.6 連想検索バッファ

MSP の *RB* 検索オーバーヘッドを下げる方法について考察する．MSP が命令区間の先頭を検知し，RB_{in} の連想検索を行う際，レジスタは頻繁に更新することを考え，レジスタに関するレコードについては常に比較を行うとして，主記憶アドレスの内容については，RB_{in} に記録後，更新しない限り比較を省略できる．Strlen(str) を用いた例を図 7 に示す．まず，引数が 16 バイト境界をまたぐ文字列”ABCDEF” であるとする．RW_{in} には手続き先頭アドレスとレジスタ引数 R0 を記録する第 1 レコードと，アドレス 10000 および 10010 に関する第 2 および第 3 レコードを格納する．RB_{in} に記録する際には，不要な比較を行わないよう cont フラグを 0 にリセットする．当該アドレスを更新するまでの間，第 1 レコードの一致のみにより cont=0 に到達し，直ちに命令区間を再利用できる．その後 10010 の内容を更新すると，以後，アドレス 10010 を含む第 3 レコードについては内容を比較しなければならないものの，アドレス 10000 については比較の必要がない．このように変更するには，RB_{in} のアドレス部を連想検索し，更新した 10010 と一致するエントリ (210) から当該エントリの key 部 (200)，type 部 (mem)，addr 部 (10010) を取り出し，key 部により指定さ

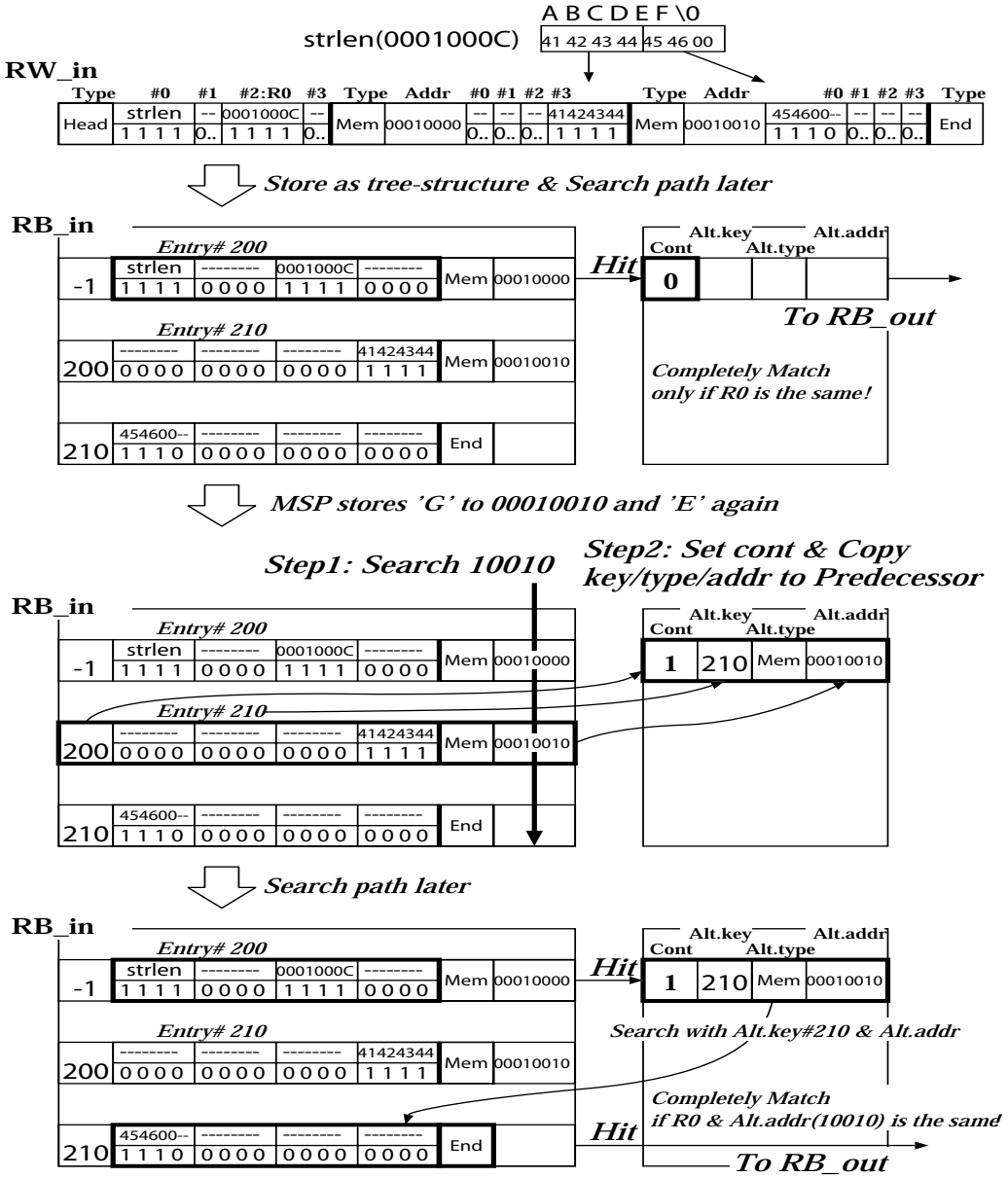

図 7: 検索オーバーヘッドの削減．更新した主記憶アドレスが順次比較対象として組み込まれる．

れるエントリ (200) の cont に 1 をセットし，alt.key，alt.type，alt.addr にそれぞれ 210，mem，10010 を格納する．以後，strlen(str) を再利用する際には，第 1 レコードの一致により cont=1 に到達し，本来の key/type/addr の代わりに，alt.key，alt.type，alt.addr を用いることにより，第 2 レコードを飛び越えて第 3 レコードを検査でき，連想検索回数を抑制できる．

既存研究の多くがストア値に関係なく投機スレッドを無効化するのに対し，本

機構では RW_{in} を用いることにより値が異なる場合のみ無効化できる点を強調しておく．値によらず無効化した場合，定数アドレスへのストアでも無効化することになり，定数アドレスの値予測を行う意味がなくなる．

以上に示した機構により検証オーバーヘッドを最小限に抑えたとしても，再利用により性能が低下する命令区間がある．そこで本研究では，再利用によるオーバーヘッドを評価し性能低下を抑制する「オーバーヘッド評価機構」を提案する．

第3章　オーバーヘッド評価機構

再利用では，過去に実行された命令区間が再び呼出される場合，RB の中から同一入力のエントリを検索する．その際，CAM による RB の実現により，CAM 自体を何度も繰り返し検索するため，特に比較する項目 (入力値) が多い場合，大きなオーバーヘッドが生じる．これにより，再利用対象にすると遅くなる命令区間が出てくる．再利用によって高速化されない場合を認識し，事前に再利用対象とするか否かを判定する機構を提案する．投機スレッドの対象区間を選択する際に，性能低下区間を対象候補から外すことにより，投機スレッドでの実行の有効性を向上させる．また，主スレッドにおいて命令区間呼び出しがあり，再利用を試みるとき，性能低下区間であれば，入出力検証をそもそも行わない．

本章では，まず，従来の投機実行対象区間の選択方法を述べ，本研究が提案するオーバーヘッド評価機構を説明する．

3.1　投機対象区間選択の従来手法

前述の Region table において，履歴から各命令区間の評価値を求めておき，評価値が最大の命令区間を SSP に割り当てるとして，MSP の高速化に繋がる評価値について考える．命令区間 i に関し，実行に要するサイクル数を C_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．MSP 自身が RB 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，従来の手法では評価値 E_i を次のように求めている．

$$E_i = C_i * (X_i + R_i) * S_i; \tag{1}$$

効率良く求めるために，C_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，命令区間毎に 3 本の 64 ビット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，ループの立ち上り時に優先的に投機実行する．なおループについては，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外すことにより精度を高めるプロセッサ数が少ない場合は，図 6 に示したように，敢えて MSP に担当させるイタレーションをプロセッサ数間隔にて予約し，その他を SSP に担当させることにより MSP と SSP での重複計算を回避している．

3.2 オーバーヘッド評価法

実行に要するサイクル数 C_i から RB の連想検索オーバーヘッドと，ヒット時の書き込みオーバーヘッドを差し引いた再利用によるゲインを G_i とすると，SSP の貢献により MSP が全てを再利用した場合は $G_i * (X_i + R_i)$ の高速化が可能となる．よって，評価値 E_i を従来の手法から次のように変更する．

$$E_i = G_i * (X_i + R_i) * S_i; \tag{2}$$

また，RB を検索する再利用オーバーヘッドが高い場合には，性能低下を抑えるために，MSP が RB を参照すべきかどうかを判断する機構が必要である．

前述の G_i，R_i を用い，さらに入力値の検査オーバーヘッドと出力値の書き込みオーバーヘッドからなる再利用オーバーヘッドのうち前者を T_i とすると，RB ヒットによる最近のゲイン Ga_i，および，ミスによる最近のロス Lo_i は次のように表現できる．

$$Ga_i = G_i * R_i/64; \tag{3}$$

$$Lo_i = T_i * (64 - R_i)/64; \tag{4}$$

$Ga_i < Lo_i$ の場合は RB を参照すべきでないと判断する．この手法では，再利用しないケースが続いた場合，R_i の値が小さくなるため，Ga_i の値も小さくなり，再利用効果が現れるにも関わらず，RB の検索を中止することがある．このため，長く再利用しない状況が続いたときには，過去に再利用した回数 R_i の値を変えることにより，再利用効果の出現に対応する．

再利用を実行する以前にオーバーヘッドの大きさを知る必要があるため，命令区間のが終了し入出力値を RB に登録するとき，RB の連想検索オーバーヘッドと，ヒット時の書き込みオーバーヘッドを算出し記録しておく．

過去に実行された区間が再び呼び出されると，まず，無駄になった検索オー

バーヘッドと削減サイクル数を計算する．これらを比較し，検索オーバーヘッドのほうが大きい場合，再利用を中止する．一方，検索オーバーヘッドが小さい場合は同一入力のエントリを検索し，見つかると再利用される．エントリが見つからなかった場合は，その命令区間を実行する．命令区間の実行を RB に登録するときは，実行した命令ステップ数，同一入力値を検索する際にかかるサイクル数および，出力値を書き込む際にかかるサイクル数を算出しておく．ここで，命令区間における実行ステップ数が，検索および書き込みに要するサイクル数の合計よりも小さい場合は，そもそも再利用による効果が得られないため，RB へ登録しないこととする．

以上のような再利用オーバーヘッドを考慮する機構をサイクルシミュレータとして実装した．この機構を用いない場合と用いた場合に分けて評価し，再利用オーバーヘッドを考慮する必要性を調査する．評価結果は 7 章に記す．

第4章　木構造のグループ化機構

入力パターンを木構造として持つ従来の手法は，木の上流にあたる検索が完全に終了するまで下流の検索が行われないため，CAM のスループットを生かすことができないという問題がある．さらに従来の手法では，命令区間の入力が全て一致しなければ，再利用することができない．区間内の一部の出力が入力セットの一部にのみ依存する場合，原理的にはその部分入力セットさえ一致すればその一部は再利用可能のはずである．そこで，入力パターンを互いに依存のないグループに分割し，各グループ毎に過去の入力パターンを保持するよう木構造を構成する．そして，CAM のパイプライン機能を用いてそれぞれを並列に検索することで，命令区間の部分的際利用を実現しつつ再利用テストのオーバーヘッドを削減することが可能となる．

本章では，まず入出力の依存関係を保持する機構を説明し，依存関係の調査方法を示す．それから，依存関係からグループを抽出する手法を述べる．

4.1　依存関係保持機構

入出力の依存関係は，依存関係表（以下，RD）を用いて保持する．再利用の対象とする各命令区間に対し，それぞれ RD を作成する．RD の概要を図 8 に示す．図 8 において，横エントリは入力，縦エントリは出力に対応する．入出力は，レジスタ１本毎に１エントリ，主記憶は１バイト毎に１エントリを割り当てる．それぞれの入力エントリに対し，依存する出力のビットを１とし，依存しない出力のビットを０にする．依存関係が生じる命令が実行されるごとに，RD を更新していく．

依存関係（入力 A →出力 B）が登録される場合，まず出力 B のエントリにおいて，入力 A に対応するビットを１とする．さらに出力 A のエントリを検索し，その出力 A が依存する入力を，出力 B が依存する入力として登録する必要がある．それには，出力 A の行と出力 B の行の OR を取ったものを，出力 B の行とすればよい．ここで，手続き区間では区間の出力とならない局所変数も，RD の出力エントリとして加えることにする．（入力 C →局所変数 D →出力 E）のような依存関係を認識するためである．また，条件分岐命令が存在する場合，分岐以降の命令を実行することそのものが依存関係にあるため，CC（Condition Code）が依存する入力を記憶しておき，この入力を以後の命令の入力に加える．

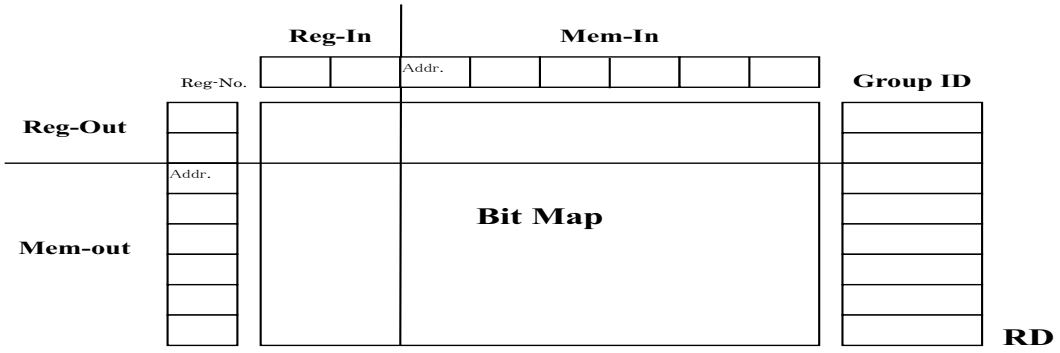

図 8: RD 概要

命令区間が多重構造になっている場合，レジスタを参照し依存関係をたどる．手続きが return される際，外部区間の o レジスタに return される手続きの i レジスタの情報を元に依存関係を書き加える．また，ループが終了する際，ループ内部のローカルレジスタや出力レジスタへの出力における依存関係が，外部区間の依存関係に作用する場合があるため，外部区間の入力として RD に登録する必要がある．

4.2 グループ化機構

以上のように作成された RD を用いて，１つの命令区間の入出力セットを複数の独立なセットにグループ化する．それぞれのグループには自然数の ID を持たせることとする．各出力エントリはそれぞれが所属するグループの ID を保持する．

出力エントリが更新あるいは新規に作成された場合，その出力エントリが依存する入力の少なくとも１つに依存する他の出力を検索する．更新（新規）出力の行と他出力の行で OR をとることにより検索できる．検出されたすべての出力からそれぞれが持つグループ ID を提出させ，その ID の中で最小の ID を，更新（新規）出力および検索された出力エントリの新しいグループ ID とする．複数のグループがまとめられる場合，破棄された最小以外の ID は回収し，以後の空 ID 一覧に加える．同じ入力に依存する出力が検出されない場合，空 ID 一覧から最小の ID を選択し，更新（新規）出力の ID とする．

下に示す命令列を持つ手続き区間を例とし，グループ化の動作を説明する．

```
Load[R1+100] => R2
```

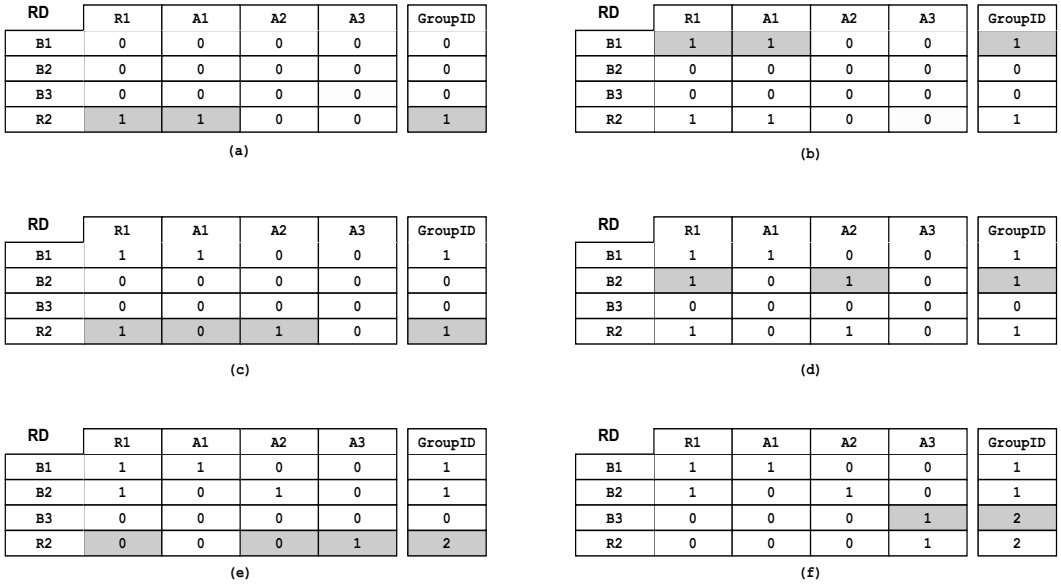

図 9: グループ化の動作

```
Store R2      => B1
Load[R1+200] => R2
Store R2      => B2
Load A3       => R2
Store R2      => B3
```

命令列１行目では，レジスタ R1 に 100 を加えたアドレス（A1 とする）に格納されているデータの値が，レジスタである内部変数 R2 に格納される．内部変数 R2 はレジスタ R1 と主記憶のアドレス A1 に依存するため，図 9 で示すところの R2 の行は，1100 となる．この命令は，本区間において最初の命令のため，グループ ID はまだ１つも使われておらず，空 ID のうち最小の ID １が R2 のグループ ID となる．(図 9a)

命令列の２行目では，１行目でストアされた内部変数 R2 の値を区間の出力 B1 に格納している．ここで，B1 が依存する R2 は内部変数であり，R2 自体が他の入力に依存している可能性があるため，R2 の行を参照する．すると，R2 の行は 1100 となっており，R2 が他の入力に依存していることがわかるため，R2 の行を B1 の行に OR で足しこむ．これにより，B1 が依存する入力部（R1 および A1 の列との交点）に１を立てられる．共通して依存する入力を持つ行を検索すると，R2 の行のみヒットし，ID １が提出されるため，B1 の行の ID も１

となる.（図 9b）

命令列３行目では，レジスタ R1 に 200 を加えたアドレス（A2 とする）に格納されているデータの値が，レジスタ R2 に格納される．ここで，R2 の行は 1010 となる．共通して依存する入力を持つ行を検索すると，B1 の行のみヒットし，ID １が提出されるため，R2 の行の ID は１のままである.（図 9c）

命令列４行目では，出力 B2 の行が 1010 となり，共通して依存する入力を持つ行を検索した場合，複数行から ID が提出されるものの，その ID はすべて１であるため，B2 行の ID は１となる.（図 9d）

命令列５行目では，R2 の行が 0001 となる．共通して依存する入力を持つ行を検索するものの，見つからないため，空 ID のうち最小の ID ２が R2 のグループ ID となる.（図 9e）

命令列６行目では，B3 の行が 0001 となり，ID は２となる.（図 9f）

例えば，ここで A2 および A3 の値を用いる命令が呼ばれる場合，その出力の行は 0011 となり，共通して依存する入力を持つ行を検索すると，複数行がヒットし，ID １および２が提出される．この場合，最小の ID １がこの出力の ID に選択され，ID １あるいは２を提出した行の ID を全て１とし，ID ２を再び空 ID とすることにより，グループ関係が維持される.

７章において，グループ化機構の評価，考察を行う.

第5章　主記憶値予測機構

従来，投機スレッドが利用する入力の予測は，命令区間内の全ての入力を対象としてきた．本章では，従来の入力予測機構の問題点を明らかにし，入力値の分類による入力値予測の効率化手法について述べる．

5.1　入力予測機構の問題点

SSP は命令区間の入力値をストライド予測し，その予測した値を入力として MSP に先がけて命令区間の実行を行う．またこの際，各入力にフラグ（C-FLAG）を設け，RB 登録時以降，当該アドレスに対して store が発生していないことが保証される場合など，そのアドレスを実際に読み出して比較する必要のない場合は，このフラグを用いて入力値テストを省略する機構を導入している．

図 10 に示した命令列を実行した場合の，入力予測の概要を図 11a に示す．

図 10 の命令列２行目は，アドレス A1 からロードした４バイトデータ 00110000 をレジスタ R1 に格納する．この場合，アドレス A1 とその値 00110000 が入力となる．命令列４行目では，A2 および値 02 が入力，レジスタ番号 R2 および値 02 が出力となるものの，A2 の読み込まれない下位３バイトに関しては don't care として登録する（「-」は don't care を示す）．命令列５行目は，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，命令列２行目の入力と併せて，02–22–となる．A2+01，A2+03 に相当する部分は，don't care のままである．同様に登録を行っていくが，命令列８行目において 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．

今，A1 については値が変更されることがないとする．これにより再利用テスト自体を省略でき，A1 の履歴を保持する意味がない．また，この命令区間を実行する度に，アドレス A2 の値は，02，03，04，05 と変化していくと仮定する．このような場合，A2 の格納値が変化するため，1 度目と 2 度目の実行では A2+4 の位置が変化し，A2 に対するマスク位置も変化してしまうため，予測が不可能である．また，3 度目の実行では参照されるアドレス数そのものが変化してしまうため，表のカラムがずれ，A2 の後続の入力 A3 および A4 についても予測が成功しなくなってしまう．

そもそも入力値予測が当たると期待できるアドレスは，ループカウンタなど

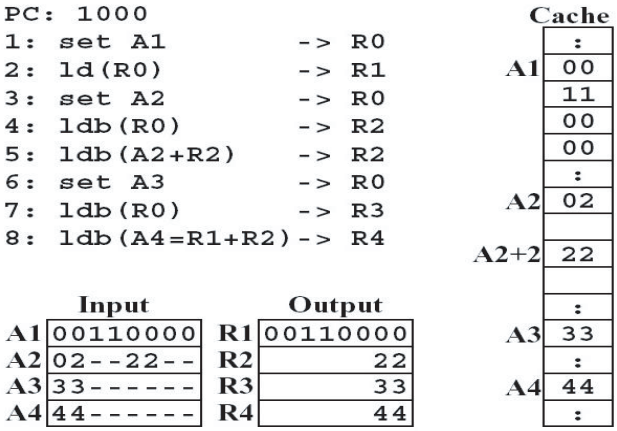

図 10: 命令列と cache の状況

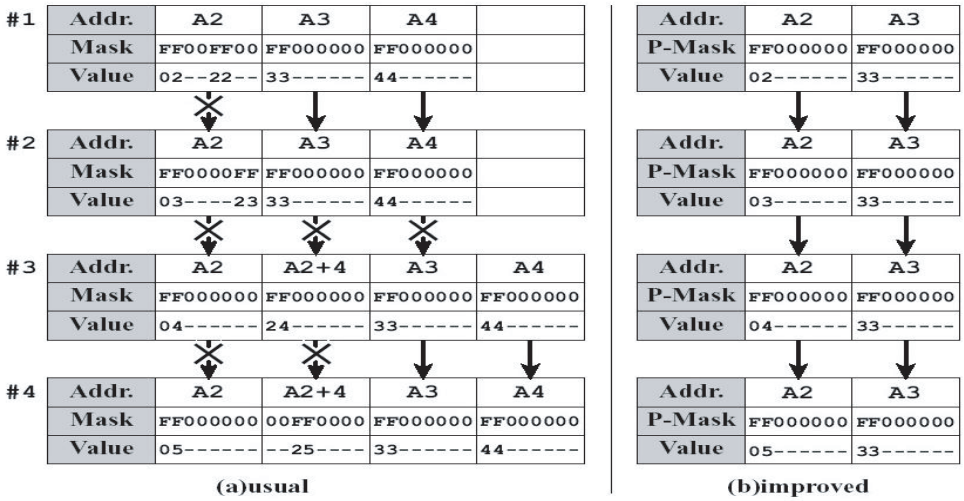

図 11: 入力予測

のフレーム内局所変数に代表されるような，まずそのアドレス自体が変化せず，かつその格納値が単調変化するものに限られる．同様の変数には，ラベルによる参照される大域変数や，スタックポインタやフレームポインタをベースレジスタとして参照される局所変数である．逆に，これら以外のアドレスは予測の対象とすべきではない．従来の手法の問題をまとめると以下のようになる．

- 例えば配列などの場合，配列添字は単調変化するが，配列の格納値は一般に単調変化しない．また，添字をアドレスとして用いる主記憶参照は，アドレス自体が変化し，参照されるアドレス数そのものも変化する可能性がある．このような場合，RB の同一列の変化には規則性がなくなり，予測の

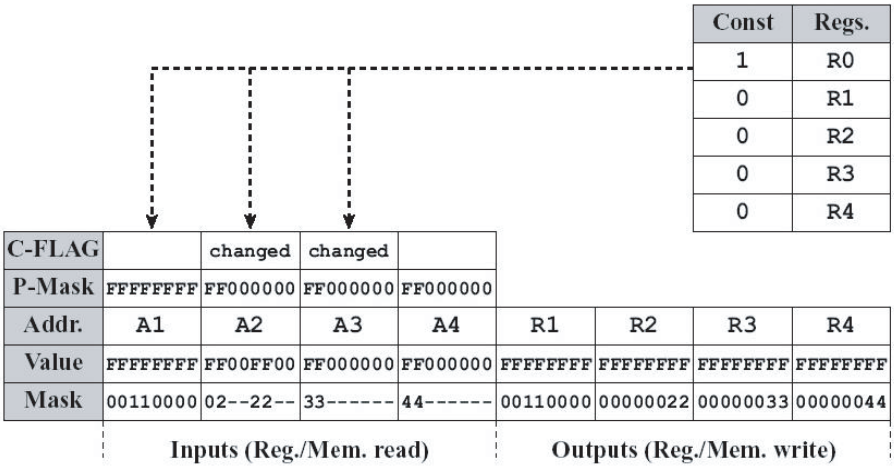

図 12: 改良した入力予測機構

成功率が激減する．

- 格納値が変化しないアドレスを予測したり，格納値の変化において過去の履歴から全く規則性が見いだせないアドレスに関しては，予測そのものを省略するべきである．また，マスク位置が変化するアドレスに関しては，マスク位置の変化まで予測するのは困難であるため，やはり予測を省略すべきである．

これらは，従来手法では登録された全てのアドレスを同等に扱っている点に原因がある．この問題を解決するため，予測が当たることが期待できるアドレスと，そうでないアドレスを区別する必要がある．

5.2 主記憶値予測

予測が当たる可能性のアドレスを区別するため，以下のような主記憶値予測機構を導入する．この予測機構の概要を図 12 に示す．

まず，アドレス自体が変化するか否かの判断のため，load 命令実行時のアドレス計算が参照するレジスタに，フラグ（Const-FLAG）を設ける．スタックポインタやフレームポインタなどの命令区間実行中変化しない値に用いるレジスタについては，Const-FLAG をセットする．またその他のレジスタについても，定数セット命令の実行時に Const-FLAG をセットする．

また，格納値の変化に規則性が見られるアドレスであるか否かの判断のため

に，当該アドレスを履歴保存対象とするかどうかを示す履歴マスク（P-Mask）を設ける．このマスクは，当該アドレスを RB に新たに登録する時点でリセットする．また，load 命令実行時に，当該アドレスを生成したレジスタの Const-FLAG を参照し，これがセットされている場合に，P-Mask のうち load 対象であるバイト位置をセットする．

図 10 で示した命令列を用いて具体的に説明する．まず，命令１が実行される際，レジスタ R0 に格納されるのはアドレス A1 であるため，R0 に対応する Const-FLAG をセットする．次に命令２が実行される際，従来手法と同様に入力として A1 および 00110000，出力をして R1 および 00110000 が記録される．このとき，アドレスとして用いた R0 に対応する Const-FLAG はセットされているため，A1 の P-Mask として４バイトデータ 00110000 に対応する FFFFFFFF がセットされる．また，R1 に対応する Const-FLAG はリセットされる．

この手法を用いた場合に従来と異なるのは，命令５の実行である．アドレスとして用いた R2 に対応する Const-FLAG がリセットされているため，A2+4 に対応する P-Mask はセットされない．このように命令の実行を進めていくと，C-FLAG がセットされ，かつ P-Mask がセットされている入力は，A2 の第１バイトおよび A3 の第１バイトのみとなる．このようにして，A2 および A3 のみの入力履歴を保持し，これらの値の予測を行うことによって，図 11 にしめすように，従来手法では予測が正確に行えなかった A2 および A3 について正しく予測が行えるようになる．なお，アドレス A2+4 については，予測値を求めずに主記憶を直接参照することで，正確な値を得ることができる．

７章において，評価，考察を行う．

第6章　投機スレッド実行結果の保存機構

本機構では，多重命令区間の投機実行を行う．多重命令区間の実行結果を RB に登録すると，再利用効率を向上させると予想している．

多重命令区間の投機実行の効果を検証するため，再利用を用いる投機的マルチスレッディングの投機的実行動作に関し，従来とは異なる２つの手法を実装する．

１つは，SSP の実行区間のうち，最外の区間のみを RB に登録し，多重実行を保存しないモデルを考える．もう１つは，SSP が実行している区間の主記憶入力値に store があった場合，その SSP の実行を中止するモデルを考える．本章では，この２つのモデルについて説明する．

6.1　多重実行を保存しないモデル

この手法では，SSP の実行区間のうち，最外の区間の実行結果のみを RB に登録し，内部区間の実行結果は破棄する．MSP による実行については，すべての区間を登録候補とする．

SSP により実行される区間は，将来，MSP によって呼び出される可能性の大きい区間で，その入力値はストライド予測によって将来の入力値になりやすいものが選択される．その予測された区間を事前に実行し RB にあらかじめ登録することは，再利用の効果を大きく向上させるものの，従来の手法のようにその内部の区間の実行結果まで RB に登録することは，再利用率向上の効果があるか否か疑問である．なぜなら，将来の予測に成功していた場合，多重再利用により，最外の区間の入出力セットさえ登録されていれば，再利用が可能となるためである．また，SSP が実行した内部区間の結果を RB に登録し，その内部区間の結果が再利用の役に立たなかった場合，RB のエントリを無駄にすることとなる．資源は有限のため，RB に効果の小さいエントリが登録されることにより，性能低下を起こす可能性がある．また，登録されるエントリが少ない場合，木構造における枝分かれが少なくなり，検索オーバーヘッドや登録オーバーヘッドが小さくなる．以上の理由から，多重命令区間の投機実行により再利用効率が落ちる可能性が考えられる．

最外の区間の実行結果のみを RB に登録するモデルを実装し評価を比較する必要がある．

6.2 投機実行を中止するモデル

この手法では，ある区間を実行中の SSP において，実行中の区間で入力となる主記憶値が変化した場合，つまりその主記憶値に store があった場合，命令区間の実行を中断し，将来予測される他のケースに実行対象を変える．

入力となっていた主記憶値が書き変わる場合，その SSP の実行結果は，その主記憶値が SSP 開始時の値に再び戻ることがなければ，再利用に効果的は見込めない．将来の実行になりえる期待が小さくなるため，SSP の実行を中止し，他の期待の大きい区間を実行するほうが，より効果が大きい可能性があると考えられるため，このモデルを実装し評価結果を比較する．

これらの評価，考察は 7 章に記す．

第7章　評価と考察

以上に述べた手法を用いて，区間再利用を用いた投機的マルチスレッディングのシミュレータを開発し評価を行った．本章では，まず評価環境を説明する．次にオーバーヘッド評価機構を，CAM レイテンシの小さい場合と大きい場合それぞれを想定し評価する．それからグループ化の評価を示し，主記憶予測機構を評価する．最後に従来とは異なる動作を行う SSP の評価を述べる．

7.1　評価環境

評価には，再利用機構を実装した単命令発行の SPARC-V8 シミュレータを用い，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．評価に用いた各パラメータを表 1 に示す．命令レイテンシは，SPARC64-III[22] を参考にしている．

ハードウェア構成に関しては，共有 2 次キャッシュは 2MB : 4way とした．また，1 次キャッシュ，共有 2 次キャッシュ，主記憶のレイテンシはそれぞれ，2 サイクル，10 サイクル，100 サイクルと仮定した．1 次キャッシュを 32KB : 4way とし，RW は 32Byte 幅× 256 エントリ が 4 セット から成り，1 次キャッシュと同サイズの 32KB とし，レイテンシも 1 次キャッシュと同じと仮定した．一方，*RB*（CAM）のサイズは，主に 128KB とした．しかしながら，より大容量の CAM を使うことも想定し，一部に関して，*RB* を 2MB，あるいは 8MB としても評価している．

MSP が D1/D2 に対し store を行うと，SSP 内の D1 で invalidate が発生し，後続命令をそれぞれ 10 サイクル/100 サイクルだけ stall させる．以後 SSP には，D1 ミスとして観測される．また，MSP/SSP で D2 に対し load が発生した場合，それより 100 サイクル後以降は MSP/SSP 内の D1 上で hit すると仮定した．

評価には，Stanford ベンチマークおよび Spec95 ベンチマークを用いた．いずれも 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．

7.2　オーバーヘッド評価機構の効果

まず，オーバーヘッド評価機構を評価し検証する．CAM レイテンシの小さい場合と大きい場合それぞれを想定し評価した．検索オーバーヘッドの大きさは，

表 1: シミュレーション時のパラメータ

ラインサイズ	32 Byte
1 次 cache 容量	32 KByte
1 次 cache ウェイ数	4
1 次 cache ミスペナルティ	10 cycles
2 次 cache 容量	2 MByte
2 次 cache ウェイ数	4
2 次 cache ミスペナルティ	100 cycles
Register-Window	4 set
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
RW 深さ	6
RF エントリ数	256
SSP	3

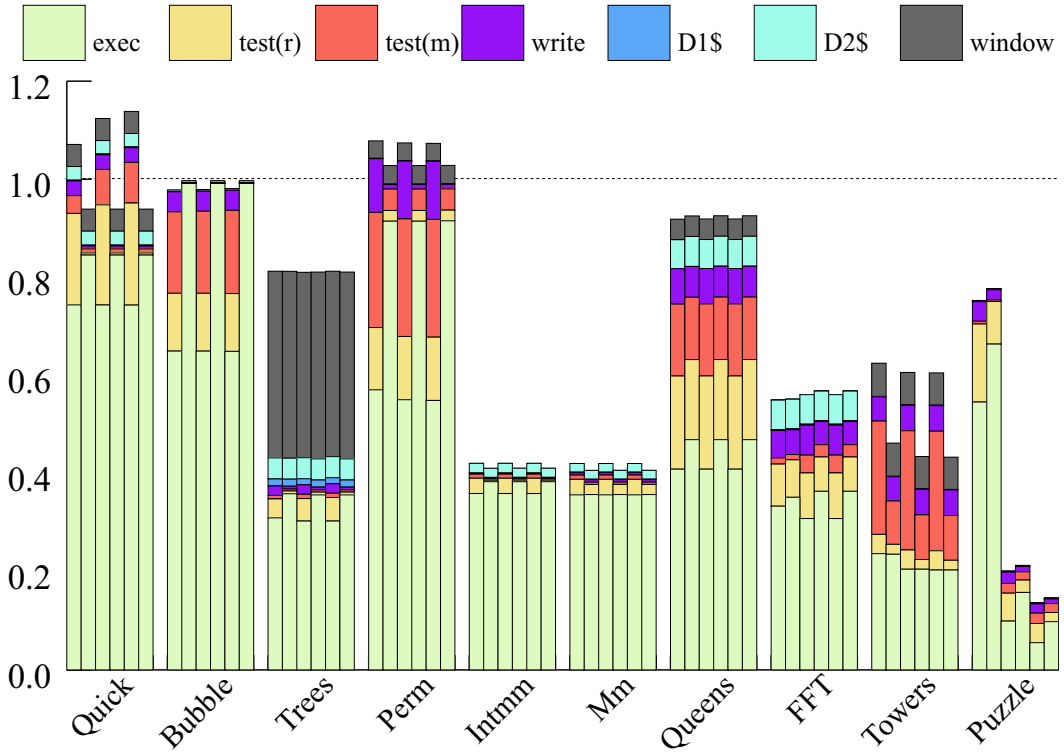

図 13: MSP の実行サイクル数 (Stanford，オーバーヘッド評価機構の評価，短レイテンシ)

RB の容量にも依存すると予想されるため，規模の異なる 3 種類の CAM を想定して評価している．

以下では，Stanford ベンチマークおよび Spec95 ベンチマークでの評価結果を示し，考察を述べる．

7.2.1 Stanford による評価

まず，*RB*CAM のレイテンシを，レジスタとの比較に 32Byte/1 サイクル，キャッシュとの比較に 32Byte/2 サイクルと仮定し評価した結果を図 13 に示す．

各ベンチマークのグラフは，左から順に，

1. (Gv-s) オーバーヘッド評価機構なし (CAM…幅 256bit ×深さ 4K = 128KB)
2. (G-s) オーバーヘッド評価機構あり (CAM…幅 256bit ×深さ 4K = 128KB)
3. (Gv-m) オーバーヘッド評価機構なし (CAM…幅 256bit ×深さ 64K = 2MB)
4. (G-m) オーバーヘッド評価機構あり (CAM…幅 256bit ×深さ 64K = 2MB)
5. (Gv-l) オーバーヘッド評価機構なし (CAM…幅 256bit ×深さ 256K = 8MB)
6. (G-l) オーバーヘッド評価機構あり (CAM…幅 256bit ×深さ 256K = 8MB)

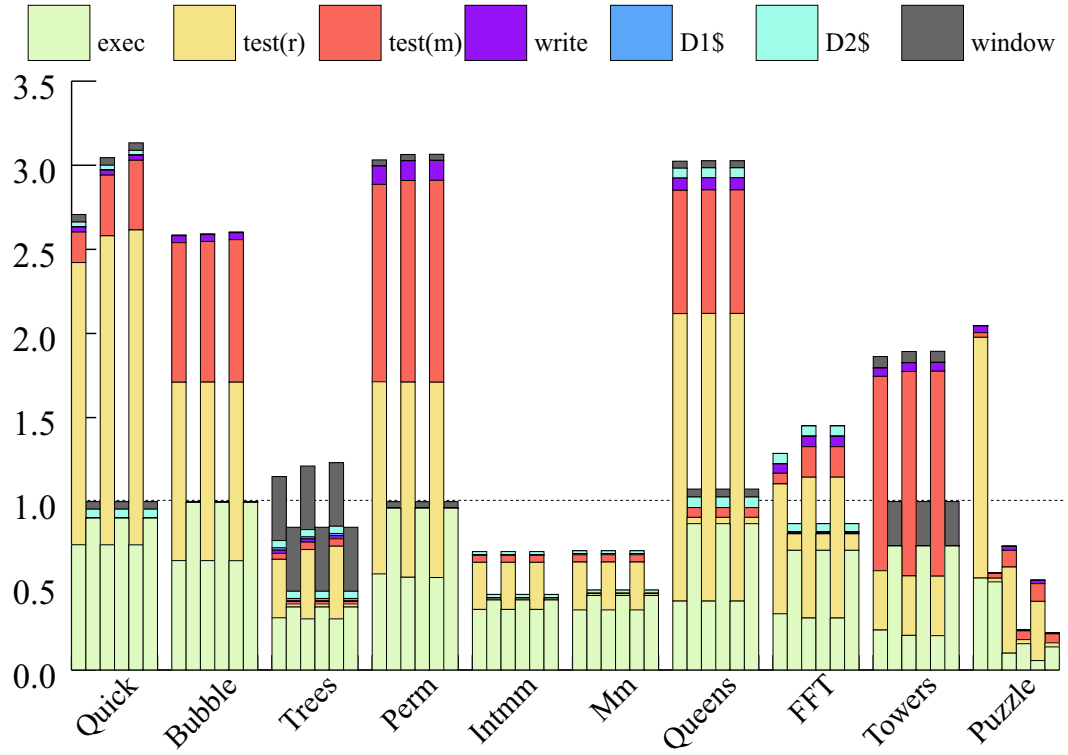

図 14: MSP の実行サイクル数 (Stanford，オーバーヘッド評価機構の評価，長レイテンシ)

を用いた場合に要したサイクル数であり，それぞれ高速化手法を一切使用しない場合のサイクル数を 1 とした正規化を行っている．

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．test(r)，test(m) はそれぞれ，RB とレジスタ，RB 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 RB 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 1 次キャッシュミスペナルティ，2 次キャッシュ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．

オーバーヘッド評価機構により，Bubble では僅かに悪化しているものの，Quick，Perm，Towers において高速化が見られる．

平均サイクル削減率を調べたところ，(Gv-s)，(Gv-m)，(Gv-l) ではそれぞれ 28%，36%，38% という値であったのに対し，(G-s)，(G-m)，(G-l) ではそれぞれ 32%，40%，42% という高い値となった．

次に，より現実的な環境として汎用 CAM のレイテンシを大きく仮定した場

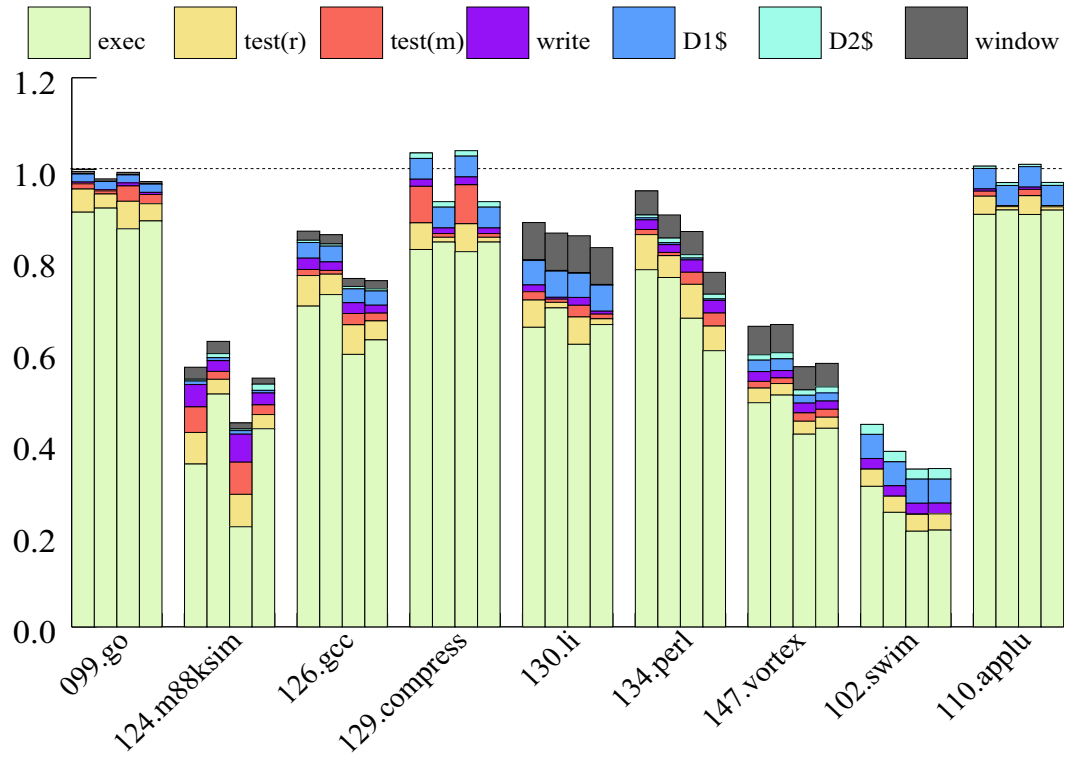

図 15: MSP の実行サイクル数 (SPEC95，オーバーヘッド評価機構の評価，短レイテンシ)

合の測定を行った．具体的には，レジスタとの比較に 32Byte/9 サイクル (短レイテンシ時の 9 倍)，キャッシュとの比較に 32Byte/10 サイクル (短レイテンシ時の 5 倍) 仮定した．この結果を図 14 に示す．レイテンシが非常に大きくなっているにもかかわらず，FFT，Towers 以外では大きな性能低下は見られない．特に，*RB* のサイズが大きくなるに従い発生するはずの性能低下を回避できており，オーバヘッド評価機構の有効性が見てとれる．平均サイクル数削減率においても，(G-s)，(G-m)，(G-l) はそれぞれ 21%，27%，28% となり，良好な結果となった．

7.2.2 SPEC95 による評価

次に，SPEC95 による結果を示す．まず前節と同じく，*RB* のレイテンシを，レジスタとの比較に 32Byte/1 サイクル，キャッシュとの比較に 32Byte/2 サイクルと仮定し，評価を行った．評価結果を図 15 に示す．それぞれのベンチマークのグラフは，左から (Gv-s)，(G-s)，(Gv-m)，(G-m)，による結果である．

129.compress などから分かるように，やはり Stanford と同様に，再利用の適

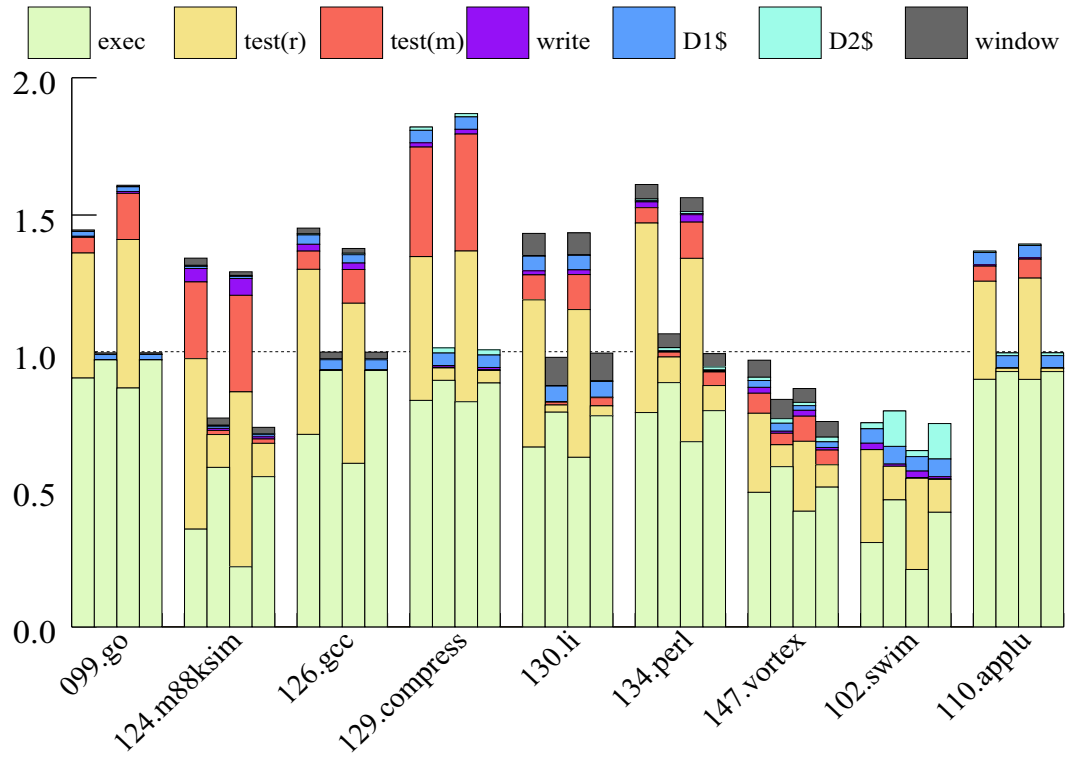

図 16: MSP の実行サイクル数 (SPEC95，オーバーヘッド評価機構の評価，長レイテンシ)

用による性能低下を回避することができている．平均サイクル数削減率は，(Gv-s)，(Gv-m) でそれぞれ 21%，27% であったのに対し，(G-s)，(G-m) ではそれぞれ 23%，29% という値となり，Stanford 同様良好な結果となった．

次に，*RB* のレイテンシを，*RB* とレジスタとの比較に 32Byte/9 サイクル，キャッシュとの比較に 32Byte/10 サイクル と仮定し評価した結果を図 16 に示す．全体的に性能は低下するものの，オーバヘッド増大による速度低下を抑えることができている．平均サイクル数削減率は，(G-s)，(G-m) それぞれ 7%，10%となった．

7.2.3 考察

Stanford ベンチマークの評価では低レイテンシを仮定している場合でも，オーバーヘッド評価機構を実装したことにより，Quick，Perm，Towers において高速化が見られた．高速化した原因について各プログラムの実行を分析した．ここでは，プログラム Towers について特に詳しく説明する．

Towers は，ハノイの塔問題を解くプログラムである．Towers が高速化した

のは，手続き Move が呼び出される場合の再利用テストが，オーバーヘッド評価機構によって省略されたことによる．ハノイの塔のディスクを移動させる手続き Move は，手続き Pop および手続き Push を呼び出してディスクの移動を行う．手続き Pop はハノイの塔のディスクを持ち上げる手続きで，手続き Push はディスクを置く手続きである．そして，手続き Move は入力として各塔の状態を取る．ここで，ハノイの塔問題では同じ状態は 2 回と現れない．そのため同じ入力が 2 回と登場しない Move は再利用されることがない．なお，状態を表す入力値の個数は多い．従来の機構では，手続き Move が再利用されないことを解析できなかったため，手続き Move に対する RB とレジスタおよび主記憶との比較を省略しなかった．しかしながら，オーバーヘッド評価機構により，手続き Move の再利用有効性が低いことが判定でき，手続き Move が呼ばれた場合でも RB を検索しない．これにより，一度も再利用できない命令区間について検索を行っていた無駄が解消されていることがわかる．

Quick，Perm においては，従来，ステップ数の小さい命令区間に再利用を用いることにより，逆に性能低下を起こしていた．オーバーヘッド評価機構によって，このような場合を回避できることがわかる．

また，Bubble ではオーバーヘッド評価機構の適用により効果が落ちている．オーバーヘッド評価機構では，過去数回分の実行および検索オーバーヘッドと書込みオーバーヘッドから，再利用効果を予想しているにすぎず，その予想が外れることによって，逆効果をもたらす可能性を示している．しかしながら，全てのベンチマークの平均サイクル削減数を見ても，この機構の有効性は確かなものである．

次に，CAM のレイテンシを大きく仮定し評価したことから，オーバーヘッド評価機構により，大きな性能低下が防げることを示した．高レイテンシでは，オーバーヘッド評価機構がない場合，多くのプログラムで再利用しないケースよりも遅くなることがわかっている．しかしながら，FFT や Towers ではオーバーヘッド評価機構を用いた場合でさえ，高レイテンシによる大きな性能低下が起こっている．そもそも低レイテンシの際，FFT および Towers では，実行ステップ数が小さい区間が非常に頻繁に再利用されることによって高速化されていた．ここでは CAM のレイテンシが大きくなり検索オーバーヘッドも大きくなったため，実行ステップ数が小さい区間の再利用有効性がなくなり，FFT や Towers で性能が低下した．例えば，Towers で非常に頻繁に再利用される手

続き Push は区間の命令ステップが 50 ステップ程度であり，同じく手続き Pop は 30 ステップ程度である．評価に使用した塔の高さを 14 とする Towers では，手続き Pop および手続き Push がそれぞれ 16400 回呼び出されるのに対し，入力パターンはそれぞれ 1400 程度であるため，両手続きの実行は低レイテンシ時には大きく削減される．

しかし，オーバーヘッド評価機構により，高レイテンシでの再利用で大きな性能低下を引き起こす Push や Pop の再利用を見送ったことにより，それ以上の性能低下は抑えているといえる．

本機構では，縦長 CAM で *RB* を構成するため，*RB* の検索オーバーヘッドが大きい場合がある．しかしながら，オーバーヘッド評価機構により，再利用の適用によって却って性能が低下するような場合を回避可能のため，検索オーバーヘッド増大時の再利用での逆効果を防ぐことができる．

7.3 木構造グループ化機構の効果

木構造のグループ化機構を評価し検証する．

以下では，Stanford ベンチマークおよび Spec95 ベンチマークでの評価結果を示し，考察を述べる．

7.3.1 Stanford による評価

Stanford による評価結果を図 17 に示す．*RB*CAM のレイテンシは，レジスタとの比較に 32Byte/1 サイクル，キャッシュとの比較に 32Byte/2 サイクルを仮定している．

各ベンチマークのグラフは，左から順に，

1. (Gg) グループ化機構なし (CAM…幅 256bit ×深さ 4K = 128KB)
2. (G) グループ化機構あり (CAM…幅 256bit ×深さ 4K = 128KB)

を用いた場合に要したサイクル数であり，それぞれ高速化手法を一切使用しない場合のサイクル数を 1 とした正規化を行っている．なお，(Gg)，(G) はオーバーヘッド評価機構および主記憶値予測機構を実装済みの結果となる．

凡例はサイクル数の内訳を示す．exec は命令サイクル数である．test(r)，test(m) はそれぞれ，*RB* とレジスタ，*RB* 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 *RB* 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１次キャッシュミスペナルティ，２次キャッシュペナルティとレジスタウィンド

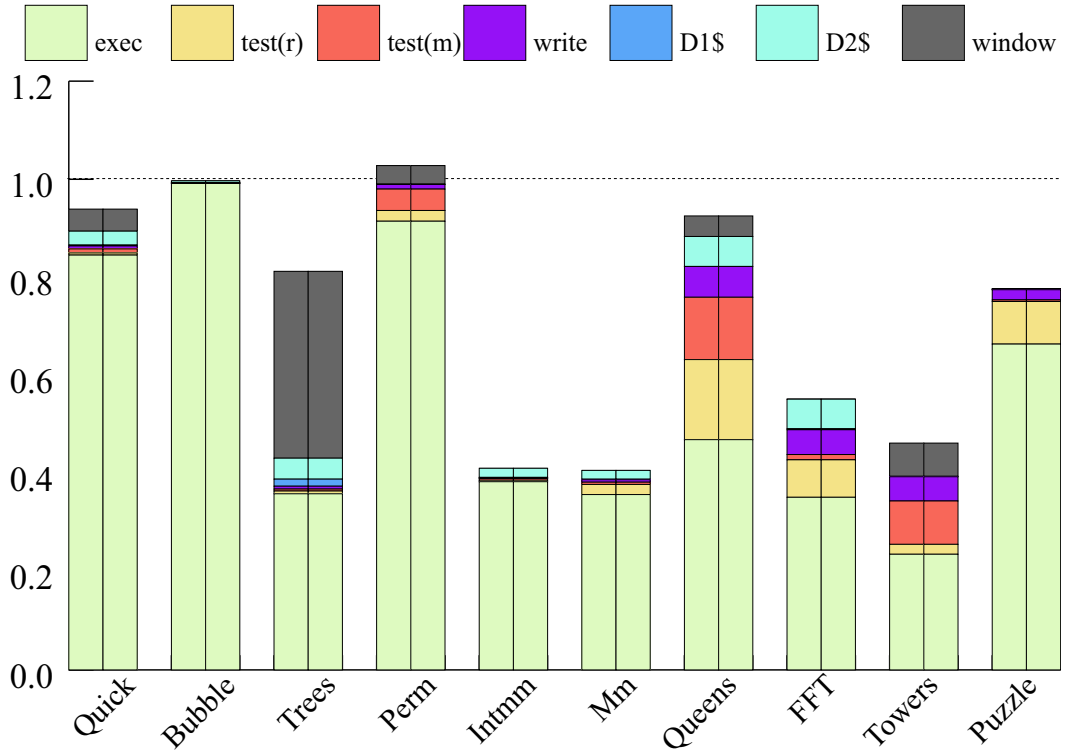

図 17: MSP の実行サイクル数 (Stanford，木構造グループ化機構の評価)

ウミスによるペナルティを表している．

Stanford では，どのベンチマークにおいても, (Gg) と (G) との結果で違いが見られない．Stanford ベンチマークにはグループ化が可能なプログラムが存在しなかったためである．

7.3.2 SPEC95 による評価

SPEC95 による評価結果を図 18 に示す．*RB*CAM のレイテンシは，

どのベンチマークプログラムにおいても，グループ化機構を用いることによる効果は，ほとんど目に見えないほどである．しかし，Stanford ベンチマークにおける評価とは異なり，半分程度のプログラムで若干効果があった．グラフを表示しているプログラムの中では，099.go，126.gcc，132.ijpeg，110.applu に対して効果が見られた．若干ではあるが，グループ化機構の効果が一番現れていたのは 110.applu であり，(G) は (Gg) と比較して，命令の実行ステップ数を 0.13%削減している．また，*RB* 比較にかかるオーバーヘッドを 3%削減している．

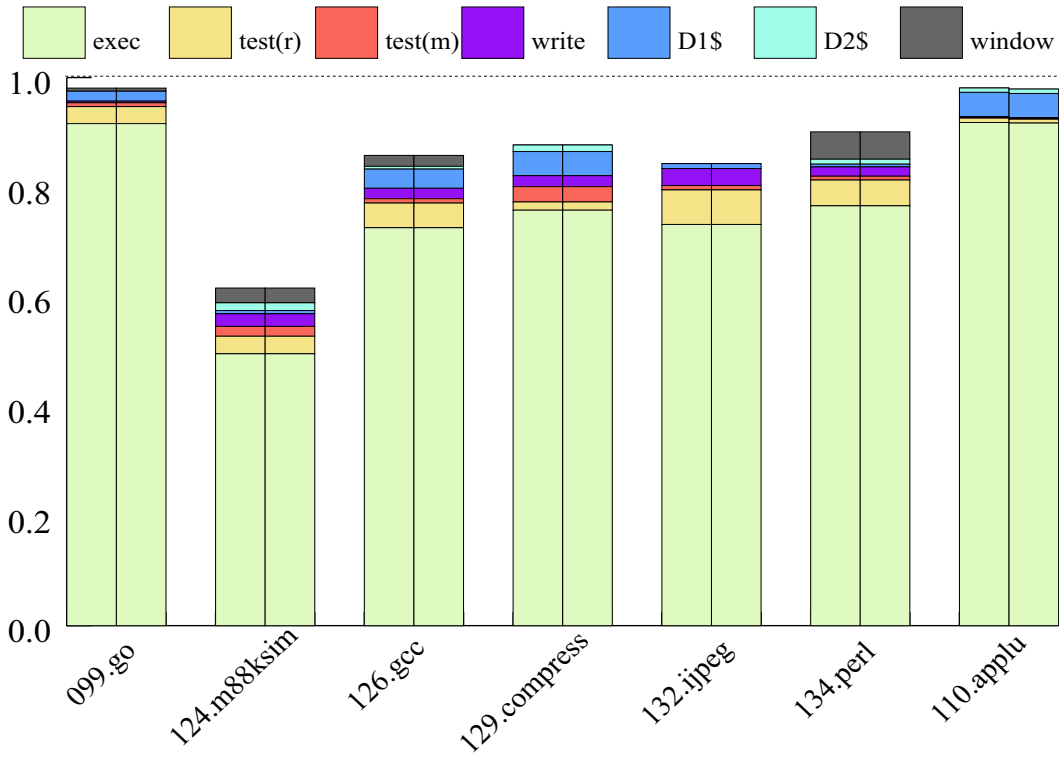

図 18: MSP の実行サイクル数 (SPEC95，木構造グループ化機構の評価)

7.3.3 考察

木構造のグループ化機構では，再利用効率がほとんど向上しなかった．その原因として，そもそもグループ分割可能な命令区間が少ないことが挙げられる．Spec ベンチマークにおいても，まったくグループ分割できる命令区間のないプログラムが多くあり，グループ分割可能な命令区間を持つプログラムでもそのような命令区間の数は極めて少ない．グループ化機構により微々たる効果のあった 099.go，126.gcc，132.ijpeg については，グループ化可能な命令区間は 10 区間にも満たない．それらのプログラムよりも多少効果が出た 110.applu では，20 個のグループ分割可能な命令区間が存在している．グループ分割可能な命令区間が少ないのは，次のような理由による．そもそも命令区間内に条件分岐命令があると，分岐命令以降の命令が実行されるか否かは，その条件分岐の方向次第となる．つまり，条件分岐方向の決定に用いられる Condition Code に，分岐以降の命令全てが依存することになるため，分岐以降の命令は同じグループにならざるを得ない．このため，グループ分割できる区間は少ない．

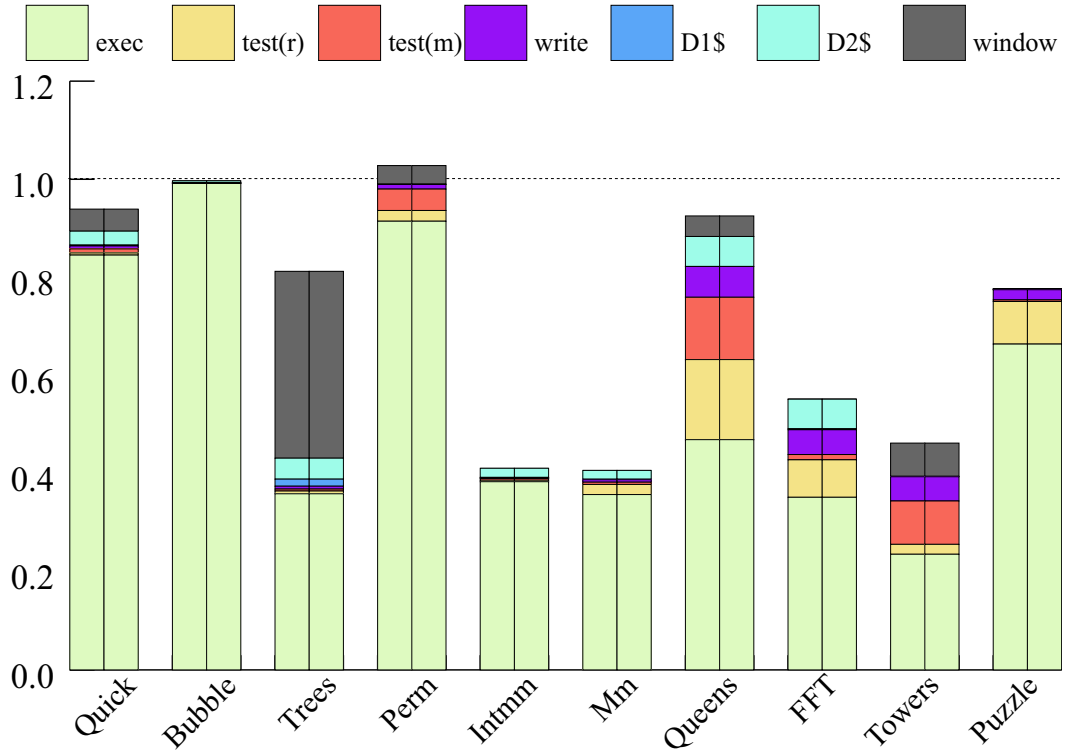

図 19: MSP の実行サイクル数 (Stanford，主記憶値予測機構の評価)

7.4 主記憶値予測機構の効果

主記憶値予測機構を評価し検証する．

以下では，Stanford ベンチマークおよび Spec95 ベンチマークでの評価結果を示し，考察を述べる．

7.4.1 Stanford による評価

Stanford による評価結果を図 19 に示す．*RB*CAM のレイテンシは，レジスタとの比較に 32Byte/1 サイクル，キャッシュとの比較に 32Byte/2 サイクルを仮定している．

各ベンチマークのグラフは，左から順に，

1. (Gq) 主記憶予測機構なし (CAM…幅 256bit ×深さ 4K = 128KB)
2. (G) 主記憶予測機構あり (CAM…幅 256bit ×深さ 4K = 128KB)

を用いた場合に要したサイクル数であり，それぞれ高速化手法を一切使用しない場合のサイクル数を 1 とした正規化を行っている．なお，(Gq)，(G) にはどちらもオーバーヘッド評価機構および木構造のグループ化機構を実装済みである．

凡例はサイクル数の内訳を示す．exec は命令サイクル数である．test(r)，test(m)

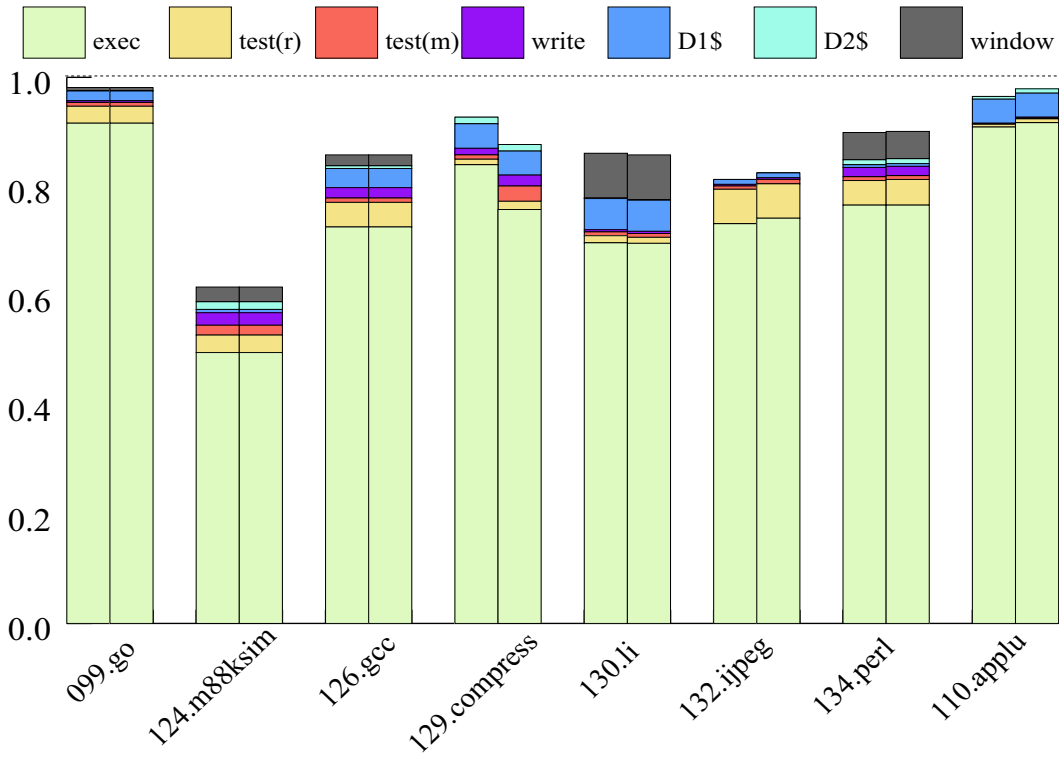

図 20: MSP の実行サイクル数 (SPEC95，主記憶値予測機構の評価)

はそれぞれ，*RB* とレジスタ，*RB* 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 *RB* 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１次キャッシュミスペナルティ，２次キャッシュ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．

Stanford では，Perm と Towers 以外のベンチマークにおいて, (Gq) と (G) との結果に違いは見られない．Perm，Tower に関しても，結果の違いは効果として感じられない程度であった．

7.4.2　SPEC95 による評価

SPEC95 による評価結果を図 20 に示す．

129.compress では，実行ステップ数に着目すると，(Gq) では高速化しない場合の 10%を削減したものの，(G) では 19%削減した．他のプログラムに関しては，主記憶値予測機構を用いることにより，サイクル数が大きく変化することはなかった．しかしながら，プログラムによっては，逆効果となるものも存在した．

7.4.3 考察

まず，主記憶値予測機構の効果が大きく表れた 129.compress について分析する．129.compress はデータ圧縮のプログラムであり，まずサイズの大きなデータを作成し，それから，そのデータを圧縮して別の領域に書込み，最後に圧縮結果を確認して終了する．129.compress では，主記憶値予測により大きく効果が上がった区間が複数存在する．手続き getbyte は，129.compress 内で約 250,000 回呼び出される命令区間であり，従来の手法では，約 67.000 回ほど再利用されたものの，主記憶予測値機構により約 190,000 回再利用できた．手続き getbyte の内容は，

```
if( InCnt > 0 ) {
 InCnt--;
 return( (unsigned int)*InBuff++ );
 }
else return( -1 );
```

である．参照アドレスが固定で変化しないため以前は再利用できない部分であったものの，アドレスの格納値が単調に変化するため，主記憶値予測機構により再利用が可能になっている．他に効率が向上した区間として，手続き getcode があり，

```
code |= *bp++ << r_off;
```

という部分について上記の InBuff と同様に再利用が効果をあげている．

129.compress のように特殊な区間が存在しない場合，主記憶値予測機構の適用により，再利用効果に大きな変化がないことがわかった．

しかしながら，小さな変化は多く存在する．本機構のストライド予測は，例えば

```
for( i = 0 ; i < 100 ; i++ ) {
 for( j = 0 ; j < 100 ; j++ ) {
      ...
 }
}
```

のような多重ループ構造において，j のループの実行が終わり，i がインクリメントされたのち，再び j が 0 からインクリメントされていくときに，j が 0 に戻った後，すぐに次の j の値は 1 であることが予想できるように，ストライド変化

が見つかったら，例外的な数値の変化が存在しても，現在の値からストライド分を加える仕組みをとっている．以上のような動作のため，主記憶値予測の適用により予測する数値の個数が変化したら，将来値での再利用成功率に若干の違いが出てくることは明白である．特に，持続的に入力値がストライド変化せず，ストライド予測により実行しておいた結果が，予想よりも後に使われ，再利用効率を上げている場合では，主記憶値予測機構により，結果に違いが生じることが多い．

例えば，Stanford ベンチマークの Towers において，手続き Pop は塔（スタック）から，ディスクを持ち上げる手続きであり，引数に持ち上げるディスクの番号を取り，他の入力としては，そのディスクの下にあるディスクの番号などを取る．ハノイの塔問題では，番号が１ずつ違うディスクを連続して持ち上げることが多く，頻繁にストライド予測される．その予測の実行結果は，すぐ使われず，しばらく後に使用されることがある．この手続き Pop も主記憶値予測によって，再利用率が変化した．この場合は，主記憶値予測を用いない場合よりも，効率が落ちている．

再利用できる命令区間においては，イタレーション間にデータ依存の関係が存在しない．ストライド予測不可能なデータ依存関係がイタレーション間にある場合，並列処理を行っても，有効な結果が得られない．

本論文で提案した主記憶予測機構をより有効に機能させるには，更に入力の依存性の解析を動的に行い，依存のあるアドレスに関して，各プロセッサ間で値を待ち合わせる機構を実現すればよいと考えられる．

7.5　投機スレッド実行結果の保存機構の効果

６章で述べた，異なる動作をする SSP について評価する．以下では，Stanford ベンチマークおよび Spec95 ベンチマークでの評価結果を示し考察を述べる．

7.5.1　Stanford による評価

Stanford による評価結果を図 21 に示す．*RB*CAM のレイテンシは，レジスタとの比較に 32Byte/1 サイクル，キャッシュとの比較に 32Byte/2 サイクルを仮定している．

各ベンチマークのグラフは，左から順に，

1. (Gt) 多重実行を保存しないモデル (CAM…幅 256bit ×深さ 4K = 128KB)
2. (Gw) 投機実行を中止するモデル (CAM…幅 256bit ×深さ 4K = 128KB)

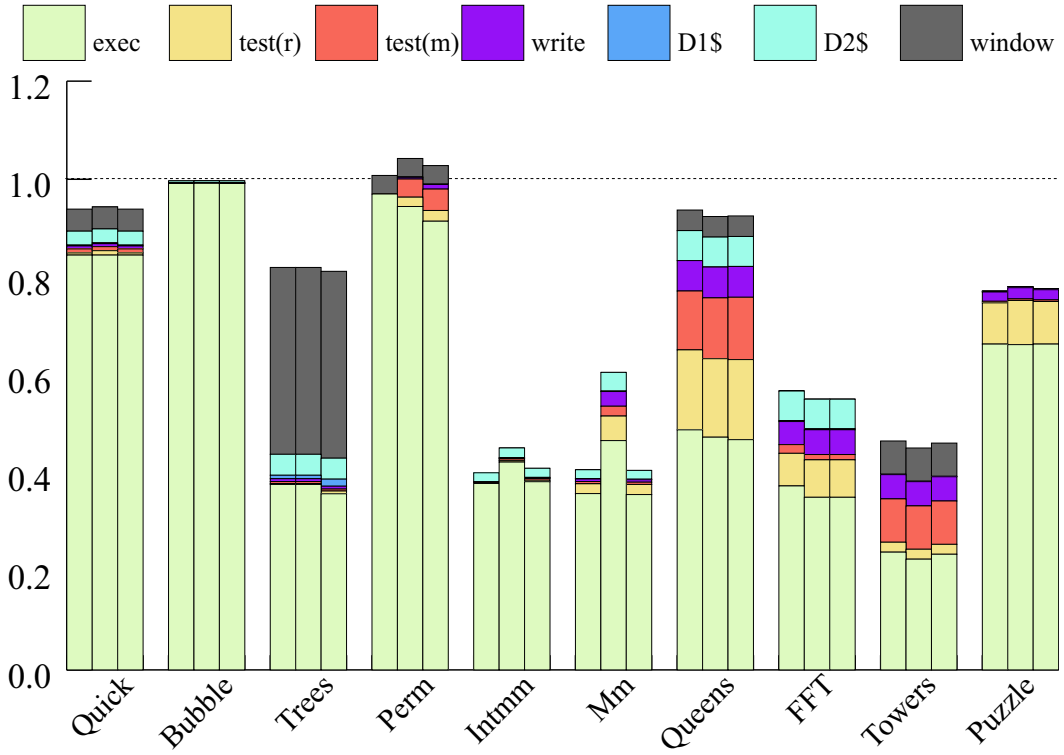

図 21: MSP の実行サイクル数 (Stanford，投機スレッド動作の評価)

3. (G) 多重実行を保存するモデル (CAM…幅 256bit ×深さ 4K = 128KB)

の場合に要したサイクル数であり，それぞれ高速化手法を一切使用しない場合のサイクル数を 1 とした正規化を行っている．なお，(Gt)，(Gw)，(G) はオーバーヘッド評価機構，グループ化機構および主記憶値予測機構を実装済みである．

凡例はサイクル数の内訳を示す．exec は命令サイクル数である．test(r)，test(m) はそれぞれ，*RB* とレジスタ，*RB* 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 *RB* 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１次キャッシュミスペナルティ，２次キャッシュ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．

(G) と (Gt) とを比較すると，おおかた (G) の命令実行ステップ数のほうが小さく，再利用の効果が大きいことがわかる．一方，どのプログラムに関しても，(G) のほうがより再利用時の検索オーバーヘッドが大きくなっている．Perm において (G) の命令実行ステップ数が (Gt) と比較して 6%小さいものの，検索オーバーヘッドのため，トータルのサイクル数では (G) のほうが大きくなった．(G)

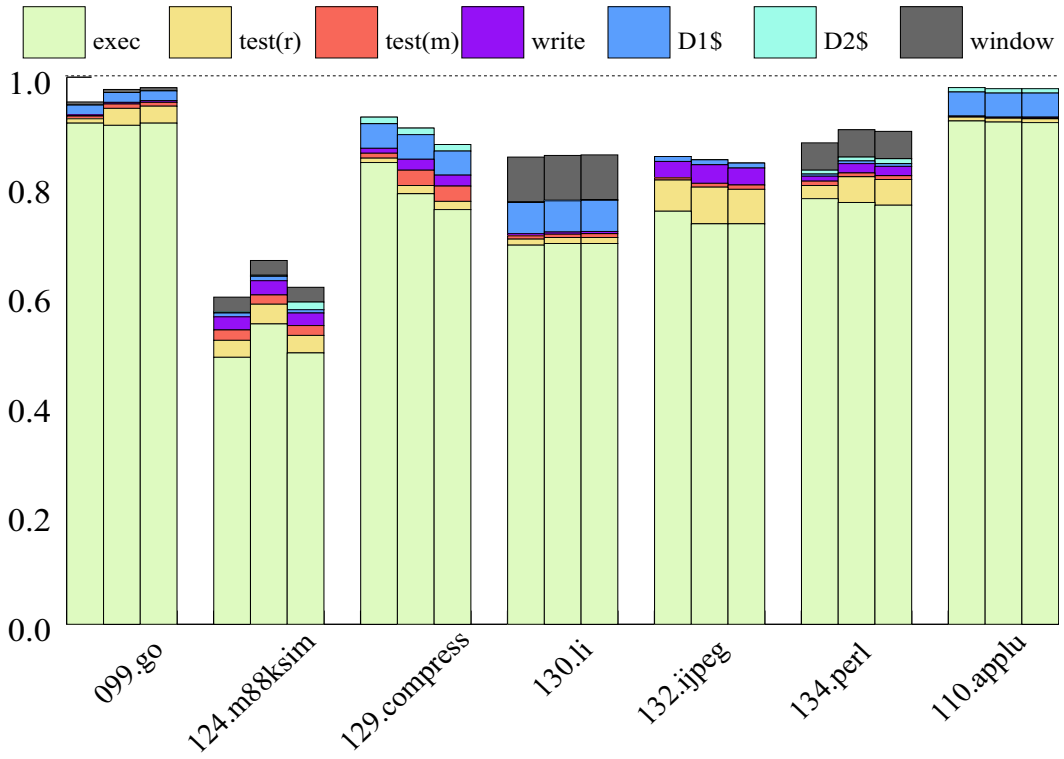

図 22: MSP の実行サイクル数 (SPEC95，投機スレッド動作の評価)

をと (Gw) とを比較すると，(Gw) よりも (G) において再利用の効果がおおかた大きい．ほとんどの Stanford ベンチマークにおいて，(Gw) と (G) とで検索オーバーヘッドに大きな違いは生じていないものの，Mm では，(Gw) のほうが断然大きくなった．

7.5.2 SPEC95 による評価

SPEC95 による評価結果を図 22 に示す．

Stanford ベンチマークの場合と同様に，(G) と (Gt) とを比較すると，(G) の命令実行ステップ数のほうが小さく再利用の効果が大きい．特に 129.compress では命令実行ステップ数が (Gt) と比較して 10%小さい．しかしながら，124.m88ksim においては，(Gt) での実行ステップ数が (G) でのステップ数より小さくなった．またどのプログラムに関しても，(G) のほうがより再利用時の検索オーバーヘッドが大きくなっている．特に 099.go において，(G) の検索オーバーヘッドは (Gt) の検索オーバーヘッドのおよそ 3 倍となっている．また 134.perl においても，(G) では (Gt) の 1．5 倍の検索オーバーヘッドを要する．

(G) をと (Gw) とを比較すると，(Gw) よりも (G) において再利用の効果がお

おかた大きい．124.m88ksim において，(G) の命令実行ステップ数は (Gw) の実行ステップ数の 10%を削減している．そして，129.compress においても，(G) では (Gw) のステップ数の 4%を削減している．しかしながら，(Gw) と (G) では検索オーバーヘッドに大きな違いは生じていない．

また余談ではあるが，本機構において 128.m88ksim などで，2 次キャッシュミスが大幅に削減されている．Stanford の結果にはほとんど表れていなかったが，本機構が共有 2 次キャッシュに対して，効果的なプリフェッチ機構としても働いていることが分かる．

7.5.3 考察

プログラムにより (Gt)，(Gw)，(G) の実行ステップ数の関係は多少異なるものの，(G) のステップ数が他の 2 つの手法のステップ数を下回る場合が多い．これは，将来実行するであろうと予測された区間だけではなく，その内部区間の実行結果も登録しておいた場合において，より再利用効率が向上することを示している．区間再利用を用いる投機的マルチスレッディングでは，投機スレッド間のデータの引継ぎとして，多対 1 のデータ構造を持つ．この結果から，多対 1 データ引継ぎの有効性が示され，本機構では，投機スレッドを有効に使っていることがわかった．

一方，(Gt) で (G) よりも再利用時の検索オーバーヘッドが小さくなる結果より，内部区間の実行結果も SSP が RB に登録することにより，検索オーバーヘッドの増大を招いていることもわかった．Perm では，その検索オーバーヘッドの増大のため，再利用効率が上がったにもかかわらず，実行時間は遅くなったまた，124.m88ksim において，(Gt) での実行ステップ数が (G) でのステップ数より小さくなるのは，内部区間の実行結果も登録することにより，RB の容量を圧迫しているためである．これは，容量を大きく取れば解決すると考えられる．(Gt) と (G) との全サイクル数を比較すると，ほとんど大差はないものの，若干 (G) のほうが少なく済むことがわかった．今後，更に RB から過去の入力を検索する際にかかるオーバーヘッドを小さくすることができれば，再利用効率のいい (G) のほうがより高速化可能となる．

第8章　まとめ

区間再利用を用いる投機的マルチスレッディングを提案し, サイクルシミュレータを用いて評価した．全ての機構を搭載した場合，Stanford，Spec95 の実行時間を平均 32%，23%削減することができた．オーバーヘッド評価機構を外した場合，平均 28%，21%に性能が低下した．グループ分割機構の効果はほとんど見られなかった．分析の結果，グループ分割可能な命令区間が多くないことが明らかになった．主記憶値予測機構については，幅広い効果は得られなかったものの，compress では最大 5%のサイクル数を削減できることを確認した．多重再利用機構についても compress で最大 5%削減でき，多重区間を投機実行する有効性を示した．今後，ハードウェアの性能向上に伴い，多重命令区間の投機実行効果は，より向上するとわかった．

謝辞

本研究の機会を与えてくださった，富田眞治教授に深く感謝の意を表します．

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，五島正裕助手に深く感謝いたします．

さらに，日頃暖かく御鞭撻下さった京都大学工学部情報学科富田研究室の諸兄に心より感謝いたします．

参考文献

[1] M.Franklin, et.al. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. IEEE Trans. on Comp. Vol.45, No.5, 552-571, 1996.

[2] C.K.Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. ISCA'01, pages 40-51, 2001.

[3] A.Roth, et.al. A quantitative framework for automated pre-execution thread selection. MICRO'02, pages 430-441, 2002.

[4] G.S.Sohi, et.al. Multiscalar Processors. ISCA'95, pages 414-425, 1995.

[5] S.Gopal, et.al. Speculative Versioning Cache. HPCA'98, pages 195-206, 1998.

[6] H.Akkary, et.al. A Dynamic Multithreading Processor. MICRO'98, pages 226-236, 1998.

[7] L.Hammond, et.al. Data speculation support for a chip multiprocessor. ASPLOS'98, pages 58-69, 1998.

[8] P.Marcuello, et.al. Clustered speculative multithreaded processors. ICS'99, pages 365-372, 1999.

[9] V.Krishnan, et.al. A Chip-Multiprocessor Architecture with Speculative Multithreading. IEEE Trans. on Comp., Vol.48, No.9, pages 866-880, 1999.

[10] J.G.Steffan, et.al. Improving Value Communication for Thread-Level Speculation. HPCA'02, pages 65-75, 2002.

[11] P.Marcuello, et.al. Thread Partitioning and Value Prediction for Exploiting Speculative Thread-Level Parallelism. IEEE Trans. on Comp. Vol.53, No.2, pages 114-125, 2004.

[12] A.Sodani and G.S.Sohi. Dynamic Instruction Reuse. 24th ISCA,pp.194-205,1997.

[13] J.Huang and D.J.Lilja. Exploiting Basic Block Value Locality with Block Reuse. 5th HPCA 1999.

[14] A.Gonzalez,J.Tubella and C.Molina. Trace-Level Reuse. ICPP,1999.

[15] 重田大介, 小川洋平, 山田克樹, 中島康彦, 富田眞治: 命令畳み込み, データ投機および再利用技術を用いた Java 仮想マシンの高速化, 情報処理学会論文

誌: ハイパフォーマンスコンピューティングシステム,HPS1,pp.13-18,2000.

[16] J.Yang and R.Gupta. Load Redundancy Removal through Instruction Reuse. ICPP,2000.

[17] J.Yang and R.Gupta. Energy-efficient load and store reuse. ISLPED,pp.72-75,2001.

[18] D.A.Connors,H.C.Hunter,B.C.Cheng and W.W.Hwu. Hardware Support for Dynamic Activation of Compiler-Directed Computation Reuse. 9th ASPLOS,pp.222-223,2000.

[19] J.Huang and D.J.Lilja. Extending Value Reuse to Basic Blocks with Compiler Support. IEEE Trans.on Comp.,Vol.49,No.4,pp.331-347,2000.

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

[21] 中島康彦, 緒方勝也, 正西申悟, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: 関数値再利用および並列事前実行による高速化技術, 情報処理学会論文誌: ハイパフォーマンスコンピューティングシステム,HPS5,pp.1-12,Sep.2002.

[22] HAL Computer Systems/Fujitsu: SPARC64-III User's Guide,1998

[23] MUSIC SEMICONDUCTORS: Using The MU9C1965A LANCAM MP For Data Wider 128 Bits,1998.

[24] 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３値CAMを用いた並列事前実行機構の効率的実現, 情報処理学会論文集: 先進的計算基盤システムシンポジウムSACSIS2004,pp.251-259,2004.

[25] 笠原寛壽，清水雄歩，津邑公暁, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: ２次キャッシュを用いた再利用および並列事前実行における高速化手法, 情報処理学会研究報告: HOKKE-2004(ARC157,HPC97)pp.133-138,2004.