修士論文

# 再利用表および投機機構の改良による区間再利用の高速化

指導教員　　富田　真治 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

李　森

平成18年2月3日

# 再利用表および投機機構の改良による区間再利用の高速化

李　森

**内容梗概**

近年，主要な商用マイクロプロセッサはプログラム資産を継承しつつ高速化を図るためにスーパースカラとスーパーパイプラインを採用し，動作周波数と命令レベル並列性を極限まで追求する高速化競争を繰り広げている。しかしプロセッサと主記憶などの記憶階層間の速度差が拡大しているため，キャッシュミスによる性能低下が著しくなり，IPC 向上率は鈍化している。キャッシュミスペナルティによる性能低下を軽減する手法として，投機マルチスレッド (SpMT)，区間再利用などの研究が多く行われている。

本論文では，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再利用する統合モデルを提案した。メインスレッド（MSP）の実行履歴を再利用表に格納するのみならず，実行履歴に基づいて，各命令区間を評価して投機実行命令区間を選択し，MSP の実行と平行して事前実行を行い，投機実行結果も再利用表に登録することより，命令区間の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。しかし，再利用表のサイズを大きくすると，動作速度が低下する，このため，高い再利用率を維持しつつ，再利用表を高速動作させるために，再利用表の構成と投機実行機構の改良を行った。

まず，従来モデルでは，再利用表を検索する際，再利用表全体を対象としていたため，多くのアクセス時間を要したのに対して，再利用表を分割し，命令区間の先頭アドレスによる分割 (row)，レコード ID による分割 (col)，および両方併用する分割 (cr) の 3 種類の再利用表分割手法を提案した。Stanford ベンチマークを用いた評価の結果，各分割数 (8，16，32，64) に対して，cr により，各 7.4%,9.3%,10.8%,9.1%の実行時間削減が可能となった。

また，投機スレッドを 2 種類に分類する，投機実行命令区間の選択方法の改良を考案した。SPEC ベンチマークを用いて評価した結果，cr 分割法かつ 16 分割の場合に，実行時間を平均 24.8%，最大で 61.6%削減できた。

# Improvement of Reuse-Tables and Speculative-Execution for Highspeed Region-Reuse

Sen Li

**Abstract**

In recent years, modern commercial microprocessors have been adopting superscalar technigue and superpipeline to make the best of the clock frequency of CPU and instruction-level parallelism. But the gap of speed between memory and CPU has been widenning and the cache miss panlty becomes sevire problem. To reduce the influence of cache miss, many researches have been performed, such as Speculative MultiThreading(SpMT), region-reuse and so on.

This paper proposes an intergrated SpMT model exploiting hardware based multilevel region speculation. It dynamically detects instruction regions from conventional load modules generated by widely used compliers, and memorizes the input and output of the regions to reuse buffer. At the same time, based on the evaluation of each region a qualified instruction region is selected to execute speculatively with predicted input value or address. The output of speculative execution are also written into the reuse buffer so that the main thread can reuse not only the result of previous execution but also the result of speculative execution. In this technique, the processor have to employ the large reuse buffer to achieve high reuse ratio. But it is difficult to achieve fast access speed in large reuse buffer. So, I propose dividing reuse buffer into several small buffers and improving the SpMT model.

The reuse buffer division methods that I propose are: division by region head address(row), division by record ID(col) and division by both of them. These mothod were evaluated with Stanford benchmark for 8,16,32,64 blocks(128,64,32,16 entries, respectively) division and it show that cr division can get the best performance(7.4%,9.3%,10.8%,9.1% execution time is reduced,respectively).

Forthermore, the following improvement is also proposed: speculative threads are classified into two groups and the selection method of region for speculative execution. The evaluation shows that the best division number is 16 blocks with cr division method. In the SPEC benchmark, the execution time is reduced by 24.8% in average and 61.6% for the best.

## 再利用表および投機機構の改良による区間再利用の高速化

# 目次

# 第1章　はじめに

主要な商用マイクロプロセッサは，プログラム資産を継承しつつ高速化を図るためにスーパースカラとスーパーパイプラインを採用し，動作周波数と命令レベル並列性を極限まで追求する高速化競争を繰り広げている。しかし，プロセッサと主記憶などの記憶階層間の速度差が拡大しているため，IPC 向上率は鈍化している。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 MultiThreading）が注目されている。ただし一般的な SpMT では，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するため，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も必要である。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。近年，増大するキャッシュミスによる性能低下の軽減手法として，投機マルチスレッディング（SpMT:Speculative MultiThreading）や区間再利用の研究が数多く行われている。従来，SpMT では，1 つのアプリケーションを複数のスレッドに分割し，並行して処理を行う。投機スレッドにデータをプリフェッチさせ，通常スレッドに必要なデータを遅れなく供給することにより高速化を図る。一般的に，投機スレッドと通常スレッドの間では，一対一でデータの引き継ぎが行われる。

一方，区間再利用とは，関数やループなどの命令区間ごとに，入出力セットを記憶しておき，再び同じ入力セットにより命令区間を実行しようとする場合に，記憶しておいた出力を利用して実行を省略する高速化手法である。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，区間に含まれる命令数が増えても，ハードウェアの複雑さが増大しない点にある。

さて，本論文では，区間再利用機構を備える効率のよい SpMT を提案し，通常スレッドの実行履歴を再利用機構に格納しておき，各命令区間の再利用有効性を評価して投機実行区間を選択し，通常スレッドの実行と平行して事前実行

する。投機実行結果も再利用表に登録することにより，命令区間の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少ない命令区間についても，高速化を図る。また，本機構では，再利用表の動作速度およびヒット率が全体の性能に大きく影響する。再利用表を分割することにより，高速動作および消費電力の両方を達成できることを示す。また，投機スレッドの分類および投機実行命令区間の改良により，投機スレッドの効率をあげることができることを示す。

以下，第 2 章では関連研究について述べる。第 3 章では提案する SpMT について述べる。第 4 章では，提案する再利用表の分割手法について詳述し，評価を行う。第 5 章では，投機スレッドの分類および投機実行命令区間選択方法の改良について詳述し，SPEC ベンチマークによる性能評価を行う。

# 第2章　先行研究

本章では，先行研究について述べる。

## 2.1　値予測

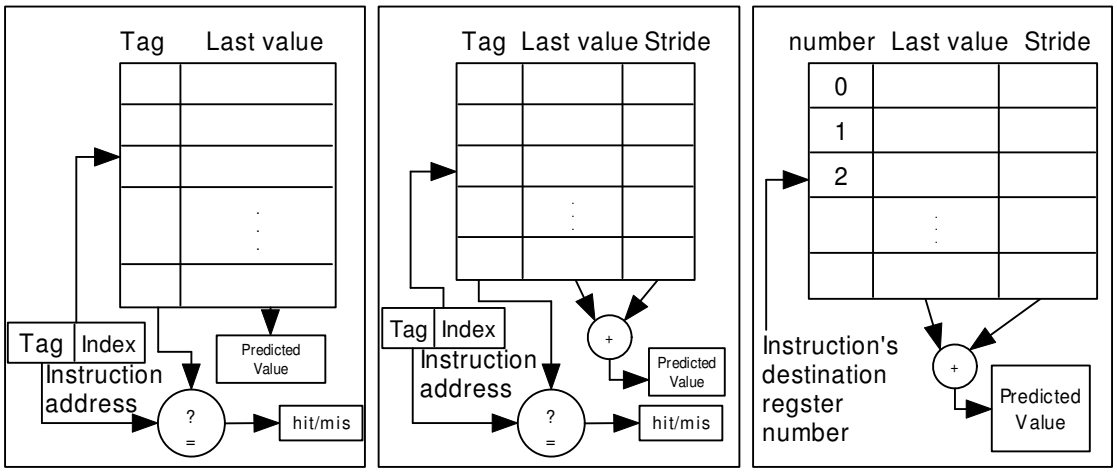
近年，値予測に基づく投機実行に関する研究が多くなってきた。具体的には，履歴に基づいて先行命令の実行結果を予測し，後続命令を投機的に実行する。値予測に基づく投機実行は，さらに，アドレスに関する投機実行と値に関する投機実行に分けることができる。アドレス予測の代表的なものとして，Next Line Prefetch がある。あるキャッシュラインが連続して参照されると，近い将来次のキャッシュラインを参照すると予測し，あらかじめ主記憶からプリフェッチしておく手法である。一方，近年，多くの研究が行われているのは，値に基づく投機実行に関するものである。具体的には，履歴に基づいて先行命令の実行結果を予測し，予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。

(a) ラストバリュー予測器: 前回の出力値に基づき，各命令の出力値を予測する。図 2.1(a) に示す表からなる。各行は各命令アドレスの下位でインデックスされる。Tag は命令アドレスの上位ビット，Last value は対応する命令の前回の出力値を保持する。命令アドレスの下位ビットにより，対応する値を予測値とする [1]。

(b) ストライド予測器: 前回の出力値および差分に基づき予測する。図 2.1(b) に示す構成である。Tag や last value はラストバリュー予測器と同様であり，Stride に保持する最近 2 回の差から，予測値を求める [2]。

(c) レジスタファイル予測器: 図 2.1(c) に示すように，各行はレジスタ番号によりインデックスされ，各行の Last value と stride を加算した結果を予測値とする。

(d) 2 レベル予測法：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




(a) Last-value predictor (b) Stride predictor (c) Register File predictor

図 2.1: 値予測器

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。

(e) コンテキスト予測法：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。

ラストバリュー予測器は前回と同じ値しか予測できないが，ストライド予測器は一定のストライドで変化する値も予測できる。レジスタファイル予測器はストライド予測器に対し，予測表の容量が小さいことが魅力だが，同じレジスタの値を同一命令が生成するとは限らないため，正確に予測できる確率は低くなる。

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はできない。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 投機的マルチスレッディング（SpMT）

近年，投機的マルチスレッディング (SpMT: Speculative MultiThreading，以下 SpMT と略す) による単一プログラムの高速化に関心がもたれている。SpMT

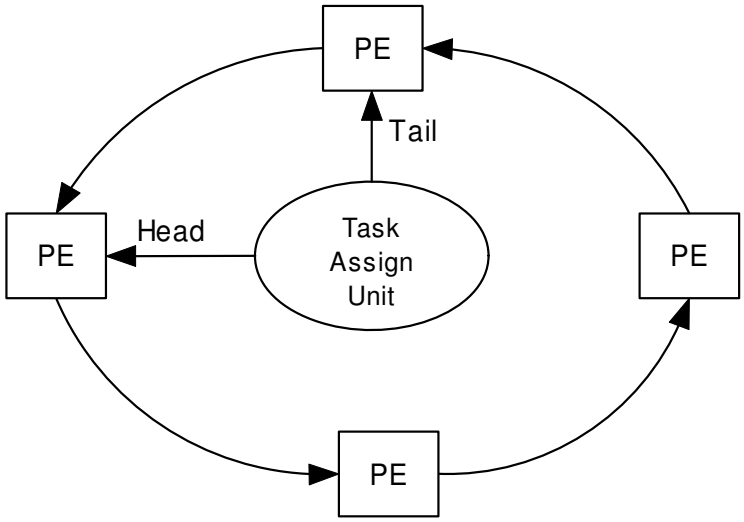



図 2.2: PE のリング構造

とは，スレッド単位の高速化手法である。ただし，高い性能を発揮するために，性能向上に寄与するスレッドを正確に選択し,各プロセッサに割り当てることが極めて重要である。また，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）を割り当てる方法や，ループ本体（loop iterations) を割り当てる方法がある。これまで数多く提案されている SpMT では図 2.2(a) に示すようなリング構造によりスレッドが各 PE(Processing Elements) に割り当てられる [14]。

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，主スレッドの実行時に，投機スレッドによる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。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。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結果は利用できないとみなして無効化する。

## 2.3 区間再利用

プログラムでは同一アドレスから同一値をロードしたり,同一値をストアするなど同じ計算を繰り返し実行することがある。区間再利用は，入出力セットを再利用表に記憶しておき，同一入力による実行の際には，記憶しておいた出力

を利用することにより命令区間自体の実行を省略する，投機実行とは異なる高速化手法である。クリティカルパス上の命令実行を省略することにより，性能向上を図ることができる。

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 [6] 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。ストア時には，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。Gonzalez ら [7] 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。Costa ら [8] 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。

# 第3章　提案するプログラム実行モデル

## 3.1　命令区間の検出

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。

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する 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 $\%o_n$，$\%l_n$，$\%i_n$　（$1 \leq n \leq 6$）から構成され，$\%i_1$ は反対側に隣接するウィンドウの $\%o_6$ と同一のレジスタとなっている。$\%l_n$ 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される。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，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，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。

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。

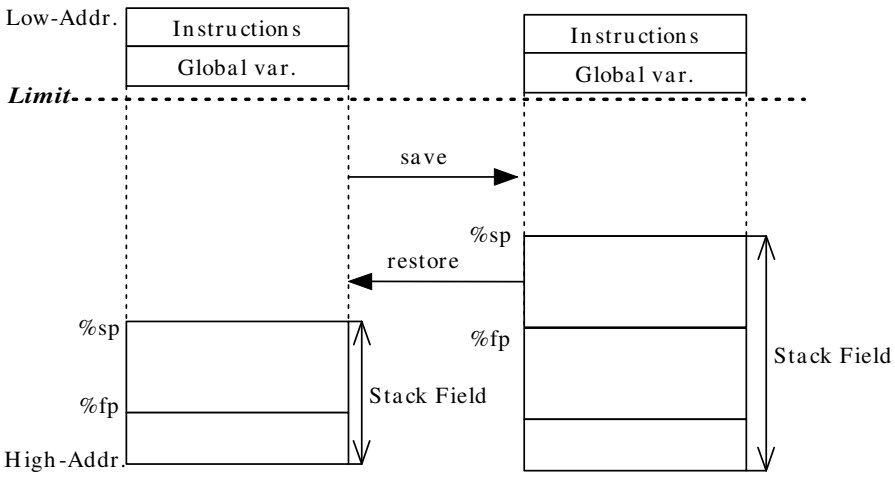



図 3.1: スタック

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 3.1 に示すように，save 命令が実行されると，レジ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

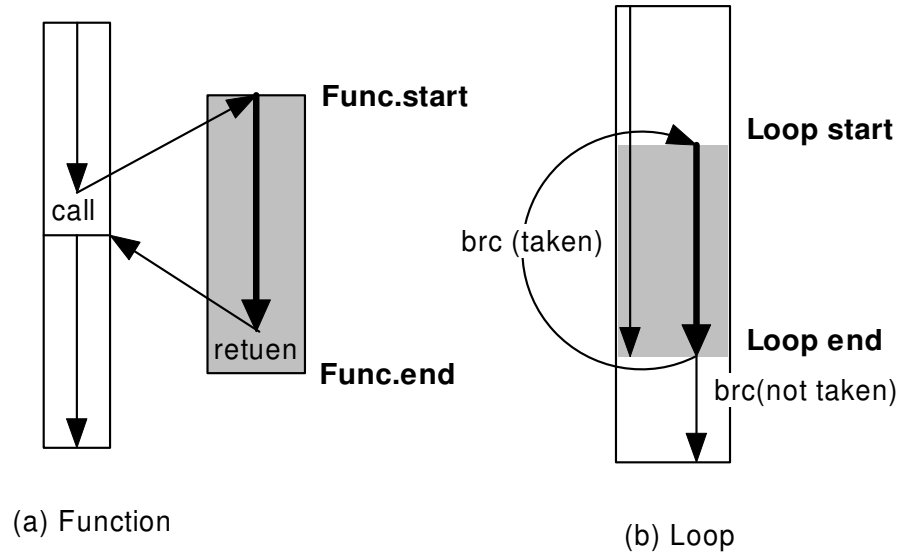



図 3.2: 関数とループの類似性

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（α ＞ 0）である jmpl 命令が用いられる。引数は%o0 ～%o5 が用いられ，返り値は%o0 および%o1 に格納される。

図 3.2 に，関数とループの類似性を示す。call 命令の分岐先を始点とし，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ことができる。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。

ループについては，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が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することなく,，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

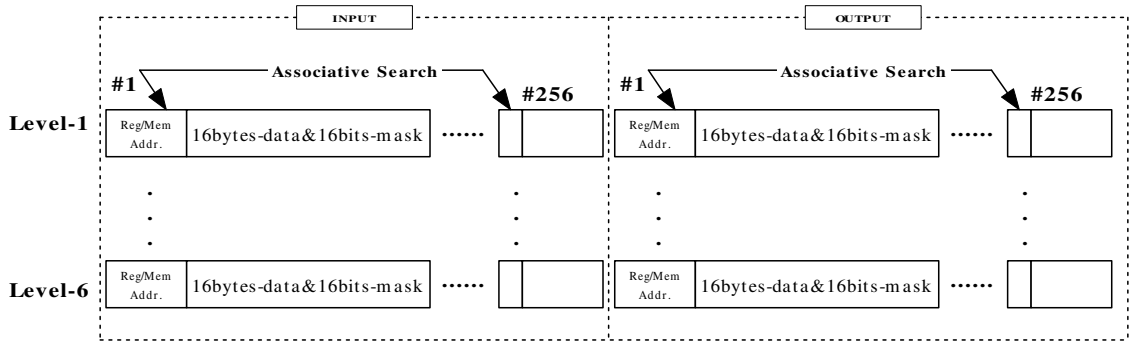



図 3.3: 再利用ウインドウ (RW)

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。

## 3.2 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，すでに出力側に登録されているかどうかを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，すでに出力側に登録している場合には上書き

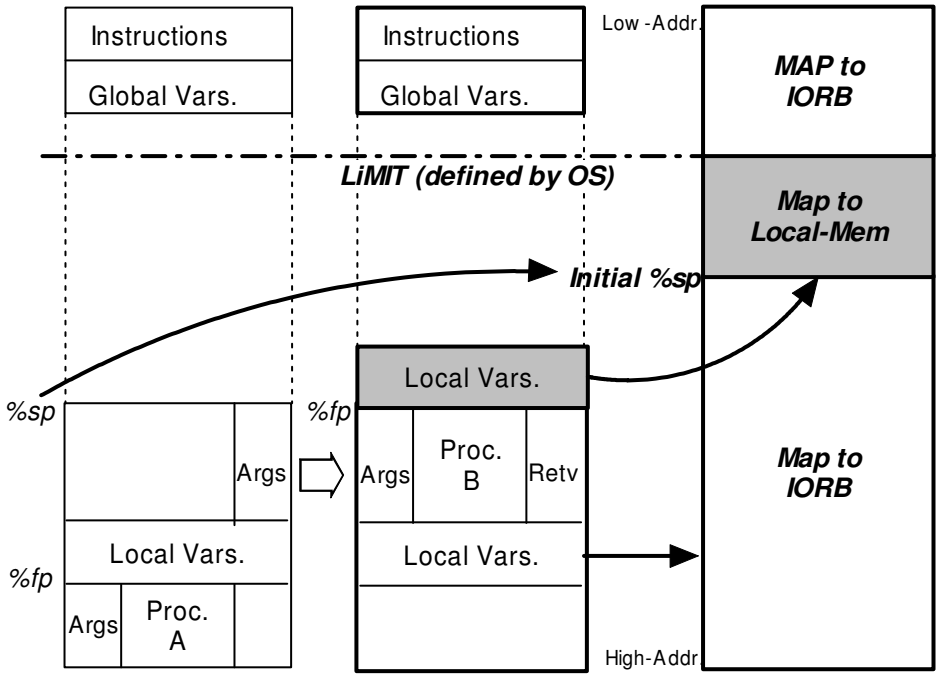



図 3.4: スタックフレームと登録先の関連つけ

しなければならない

図 3.3 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，連想度は 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。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 RWout 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。

- 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 RWin へ，各々振り分ける必要がある。このためには，図 3.4 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。
- レベル 1 の RWout: 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 RWout を優先する。
- レベル 1 の RWin: RWout にない場合は，過去に一次キャッシュから読み出して RWin に登録したものを優先する。
- 上位 RW: 以上を最高レベルまで繰り返す。
- 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 RWin 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 RWout 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 3.5 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。

## 3.3　再利用の考え方

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，前述したRW 記録する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。再利用が行われる際に，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す，そして，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。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。

## 3.4　SpMT との統合

本提案では，始点と終点を容易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 3.5(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

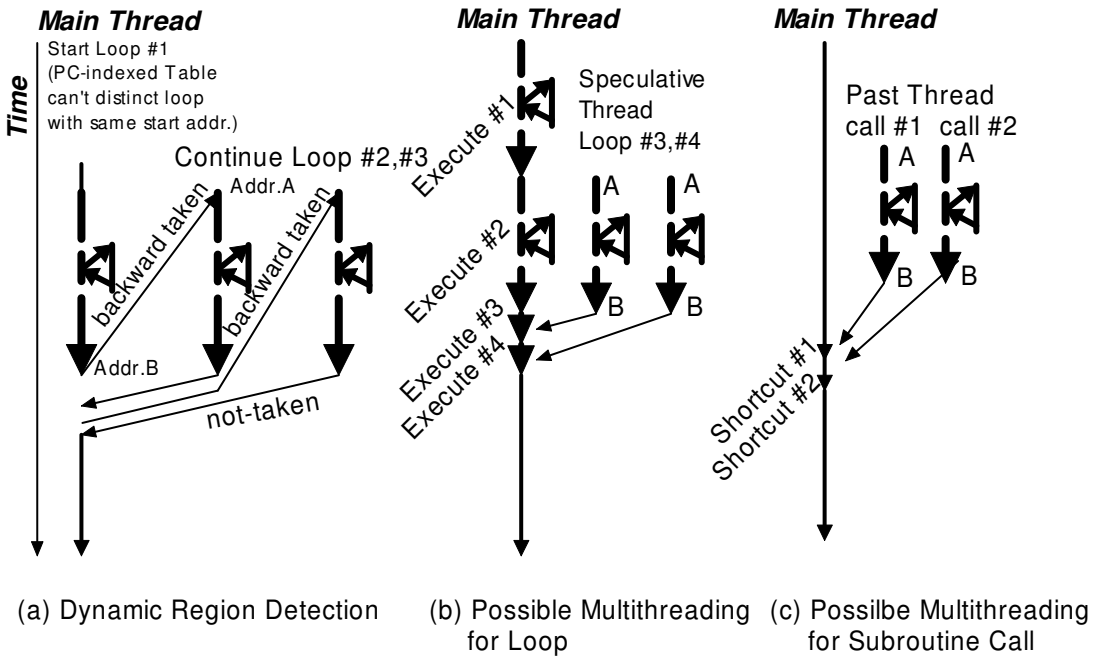



図 3.5: 動的に検出可能な命令区間を利用したマルチスレッディング

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 3.5(b) および 3.5(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。3.5(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。

加算を行う関数 x=Add(a,b) を例とし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

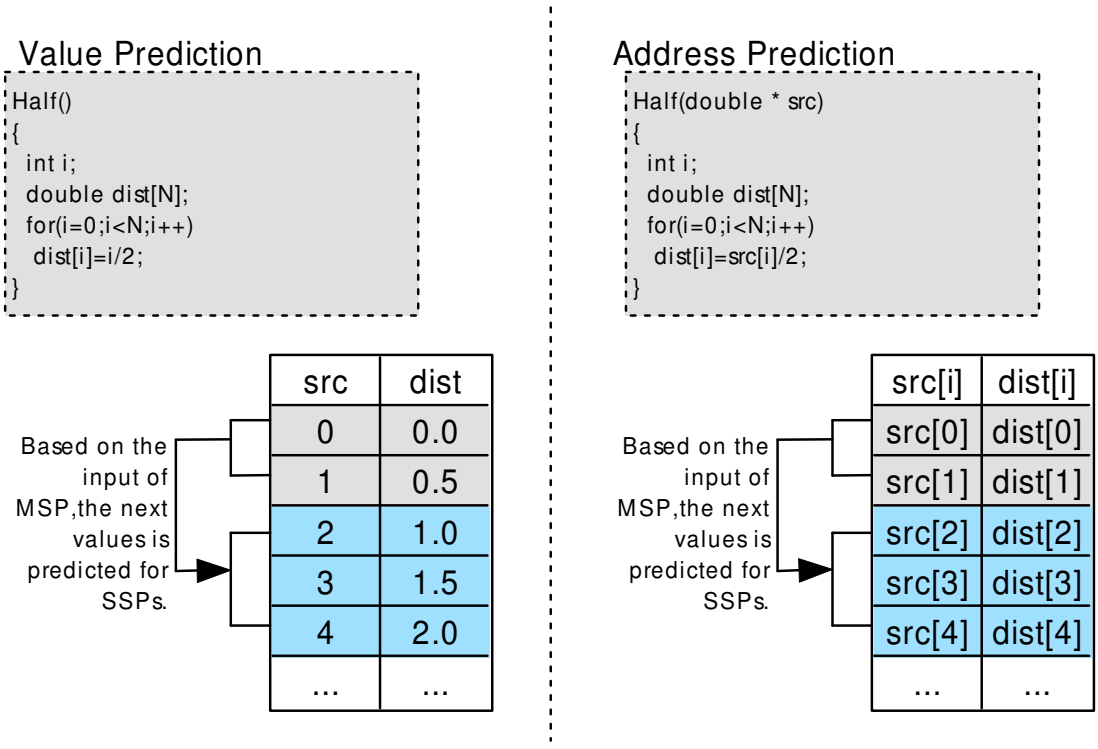



図 3.6: 区間再利用と SpMT の統合

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(1,2) 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考え，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 3.7 節で詳述する。以下では，通常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3.6(a) に，SSP を3 台用いた場合の実行例を示す。図3.6(a) に示すforループの命令区間が単調変化する引数により実行する。MSPのみの場合，入力値は0，1，2. . . と変化し，どの入力値も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の実行を終了し，1という入力セットが現れた時点において，後述するストライド予測により，2，3，4. . . と入力値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が入力値1 の実行を終え，図3.6（a）のようにMSPが入力値2として を実行する前に，再利用表から入力セットを検索する。MSPは過去に実行していないにも関わらず，入力値2 に関する実行履歴は既に登録されており，出力値1.0 を再利用できる。同様に，3，4についてもSSP により既に登録されており，MSPは3.6 のように再利用表から出力値を得る。図3.6(b) に示すように，命令区間の入力値が単調変化ではなく，入力データのアドレスが単調変化する場合にも，入力データのアドレスに対してストライド予測し投機実行を行う。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。

## 3.5 再利用機構の構成

図3.7 に示すように，再利用機構は再利用ウィンドウ（RW:Reuse Window），命令区間表 (RF) および再利用表（RB:Reuse Buffer）などからなる。

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3.7 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を用いる。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。

RWinの1 行分の入力を，木構造の1 パスとして再利用表に格納する。RBに木の各枝を，RAには各ノードを格納する。このうち連想検索の対象となるRBをCAMにより構成し，RAの各エントリをRBの各エントリと1 対1 に対応させる。

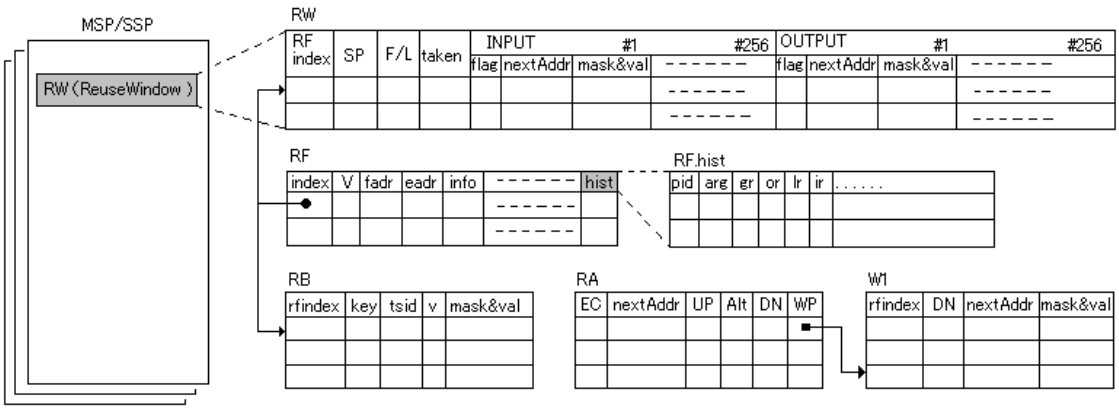



図 3.7: 再利用機構の構成

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，次に参照すべき主記憶アドレス（ next addr.），および，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。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，後節において詳述する。

## 3.6 再利用表の動作

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。実際には，RWin の 1 レコードには 16Byte（4word）をまとめて格納するものの，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。

### 3.6.1 再利用ウインドウ（RW）への登録

図 3.8(a) に示す，32bit の整数 4 つからなる，配列の総和を求める関数を命令区間として登録することを考える。この関数をコンパイルにしたものが図 3.8(b) に示す。命令区間の先頭アドレスは 0x00002000 であるとし，配列のデータはアドレス 0x00010000 から格納されているとする。図 3.8(b) で使われているレジスタは，それぞれ，%l0 がループカウンタ，%l1 が配列のインデクス，%l2 が配列の総和，%l3 がメモリより読み出した配列の値，%i0 が関数の引数，%o0 が関数の返り値である。

命令 i3 は，アドレス%l1(A1:0x00010000) からロードした 4byte データ 0x00000001 をレジスタ%l3 に直接格納し，0 に初期化された配列の総和と配列の最初のエントリの加算を省く。この場合，アドレス 0x00010000 およびデータ 0x00000001

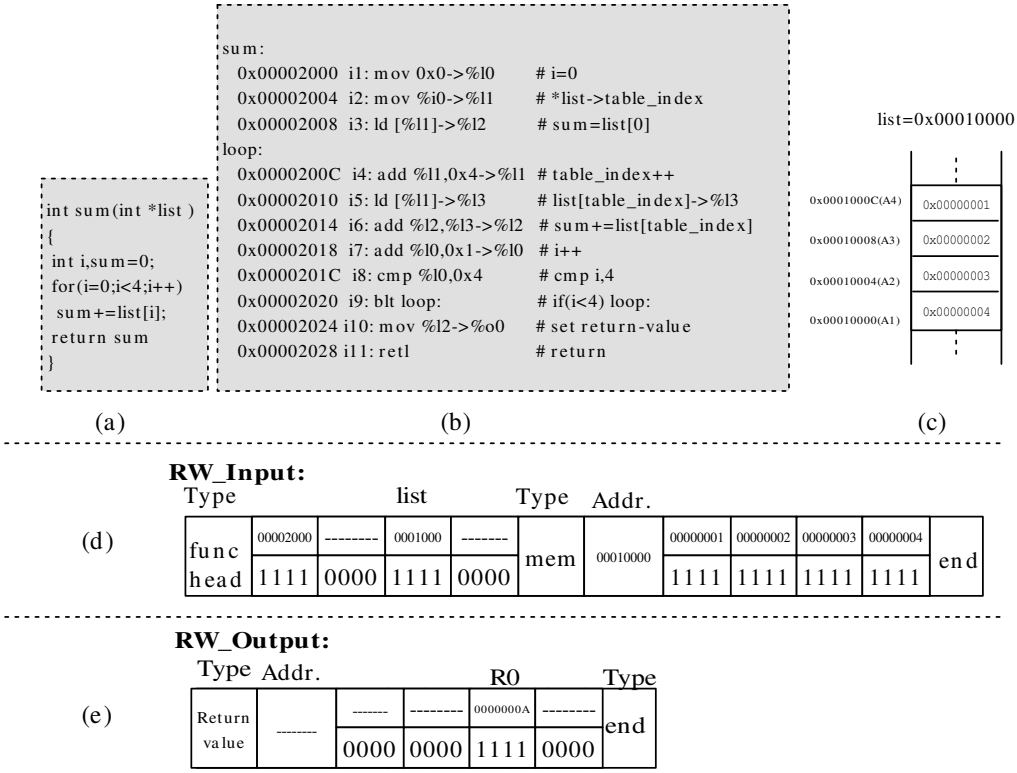



図 3.8: 入出力の記録

が入力，レジスタ番号%l2 およびデータ 0x00000001 が出力となる。これらを時系列順に RW に登録する。また，命令 i5,i6 ではアドレス%l1(A2:0x00010004) およびデータ 0x00000002 が入力，レジスタ%l1 および加算の結果 0x00000003 が出力となる。同様に命令 i5,i6 を実行するたびに登録を行っていくが，各ループのたびに命令 i4 では%l1，命令 i5 では%l3，命令 i7 では%l0 が本命令区間内で上書きされる。よって，本命令区間のローカル変数とみなし，入出力いずれもなく，登録する必要がない。最後の計算の結果を%o0 に書き込み，それを返り値として関数が終了するため，レジスタ%o0 と値 0x0000000A が出力として登録される。以上のように入出力を時系列順に登録した結果，図 3.8(d) と図 3.8(e) のようになる。命令区間を場合を考える。

### 3.6.2 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。

図 3.9（a）に示すように，RW では二つの入力パターンを再利用表へ登録しようとする。

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 3.9 (c) 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。ただし，親が根である場合は-1 を登録する。さらに RA の各エントリに，再利用テストの軽減するための次の 3 項目を追加する（図 3.10 を参照）:

- 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

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。これら 3 項目の詳細には 3.6.5 節で述べる。

### 3.6.3 削除操作

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削除を行う。具体的には，システム全体の時刻を表すサイクリック 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。よって，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。

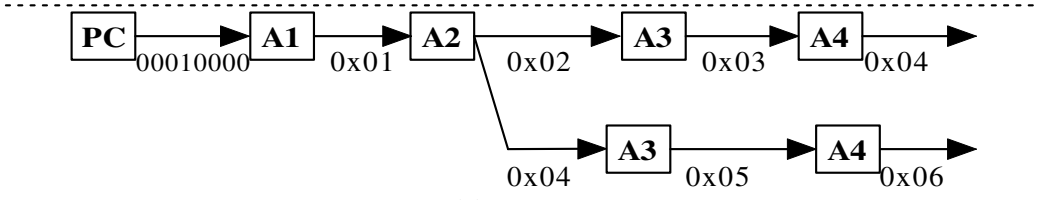
**RW_Input:**

| Type | | ID: 1 | | ID: 2 | | ID: 3 | | ID: 4 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| F/L | Head Addr. | A1 | Val1 | A2 | Val2 | A3 | Val3 | A4 | Val4 | End |
| | Mask | | mask | | mask | | mask | | mask | |
| F | 00010000 | A1 | 0x01 | A2 | 0x02 | A3 | 0x03 | A4 | 0x04 | End |
| | 1111 | | 1111 | | 1111 | | 1111 | | 1111 | |
| F | 00010000 | A1 | 0x01 | A2 | 0x04 | A3 | 0x05 | A4 | 0x06 | End |
| | 1111 | | 1111 | | 1111 | | 1111 | | 1111 | |

(a)

| RB | | | RA | Flag | Addr. | Up | Alt. | Dn |
|---|---|---|---|---|---|---|---|---|
| 00 | -1 | 00010000 | | 0 | A1 | -1 | A2 | 03 |
| | | | | ... | ... | ... | ... | ... |
| 03 | 00 | 0x01 | | 1 | A2 | 0 | | |
| | | | | ... | ... | ... | ... | ... |
| 06 | 03 | 0x02 | | E | A3 | Sb | | |
| 07 | 03 | 0x04 | | E | A3 | Sb | | |
| | | | | ... | ... | ... | ... | ... |
| 09 | 06 | 0x03 | | E | A4 | | | |
| 10 | 06 | 0x05 | | E | A4 | | | |
| | | | | ... | ... | ... | ... | ... |
| 12 | 09 | 0x04 | | E | | | | |
| 13 | 09 | 0x06 | | E | | | | |
| | | | | ... | ... | ... | ... | ... |

(b)




(c)

図 3.9: 再利用表への登録

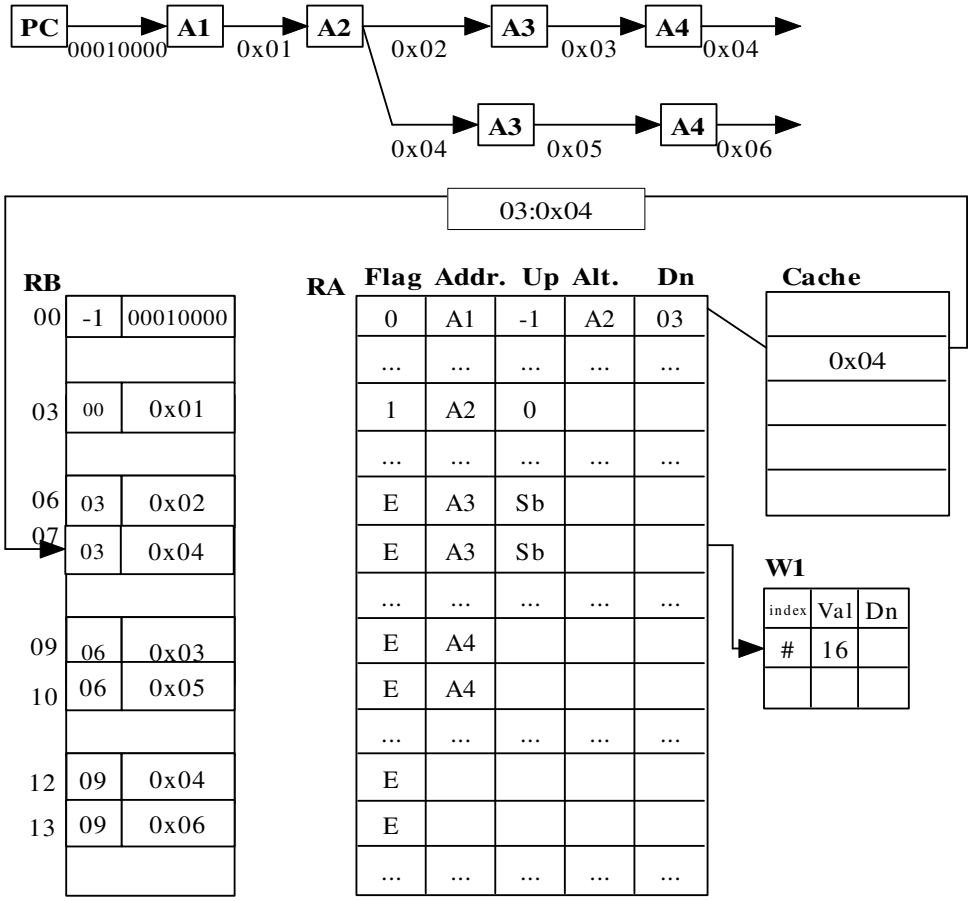



図 3.10: 検索操作（Sb：ブロックサイズ）

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，再利用表は全体的にデータを一致するため，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。

### 3.6.4 検索操作

図 3.9 のように入力セットが登録されているとする。検索は図 3.10 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 により次ノードが Sb（Sb：ブロックサイズ）であることがわかる。Sb 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.に格納しているデータおよび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両方をキーとして次の検索を行う。

### 3.6.5 主記憶一貫性

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。もし，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エントリを無効化した場合，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，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ー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。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，図3.11示すように，RAに対し，各入力アドレスに2bitのフラグ(flag)を付加する。そして，RA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。

再利用表への登録後にMSPやI/Oが同一アドレスにストアした場合，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。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主記憶から読み出して値を得る。

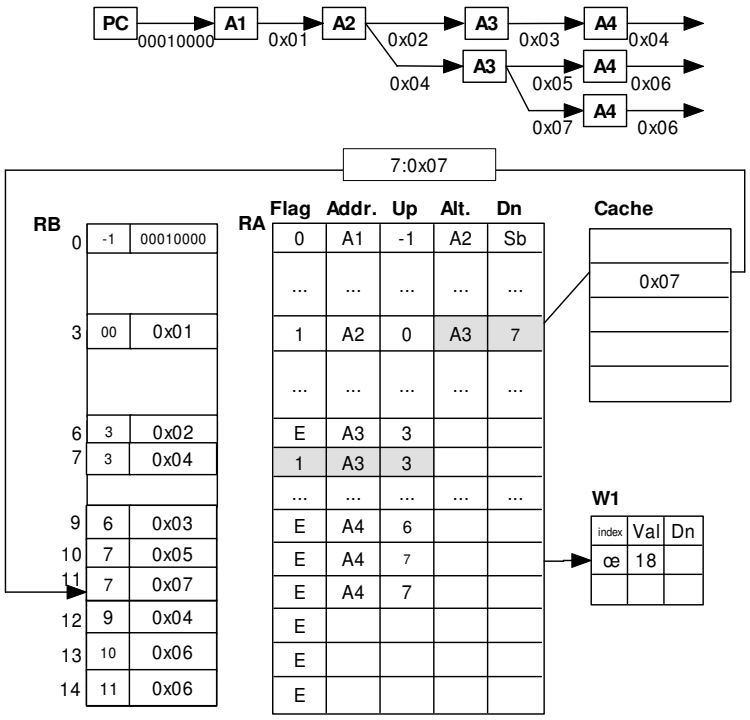



図 3.11: ストア

SSP で発生したストア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ストア 後ロードは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である場合，外側区間 A においてストアした値を内側区間 B がロードし得るため，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 によるストアにおいては，区間をまたぐストア 後ロードが存在する場合，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。

比較要フラグがオフであるエントリに関し，アドレス A3 に対してストアが発生したときの Alt.，DN に対する操作を図 3.11 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 RA の行に関し，Alt. 項を A3 で，DN 項を A3 の格納されている RA のイ

ンデクスで埋める。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 A3 の内容に対する比較が行われるようになる。

再利用テスト時は，RA 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ーヘッドを抑えることができる。

## 3.7 命令区間の実行履歴

本研究では，命令区間に関する情報記録は RF 表を用いて格納する。各命令区間が RF 表中の一行に対応し，新命令区間の実行が終ると，その命令区間に関する情報を RF 表に収めていく。RF 表の構成を図 3.12 に示す。

- V: 本行が有効であるかどうかを示す
- fadr: 命令区間の先頭アドレス
- eadr: 命令区間の最後尾アドレス
- info: 命令区間が関数かループかを示す
- rovh_mru: 前回再利用テストの read ステップ数
- wovh_mru: 前回再利用テストの write ステップ数
- step_mru: 前回の実行ステップ数
- msp: MSP により実行された回数
- ssp: SSP により実行された回数
- hist: 最近 4 回分の入力情報，各 RF エントリは図 3.12 に示すように，循環キューの構成であり，矢印の方法により新エントリが追加される。
- hist_index: RF.hist 表における最近 4 つの履歴記録のインデックス値を収める。-1 に初期化され，hist_index[0] → hist_index[3] の順により新しい履歴記録のインデックス値が格納される装置である。
- pred_dist: 本エントリに対応する命令区間の投機実行を行うスレッド ID のリストである。各ノードが-1 に初期化され，対応する命令区間の投機実行が投入される際に pred_dist にスレッド ID を書き込む。最初，対応する命令区間が MSP により実行されるため，pred_dist[4] に 0 を書き込み，投機実行が投入されると，pred_dist[5] から投機スレッドの ID を格納する。同時に pred_dist リストのインデックス値からの投機距離を計算し，その距離値

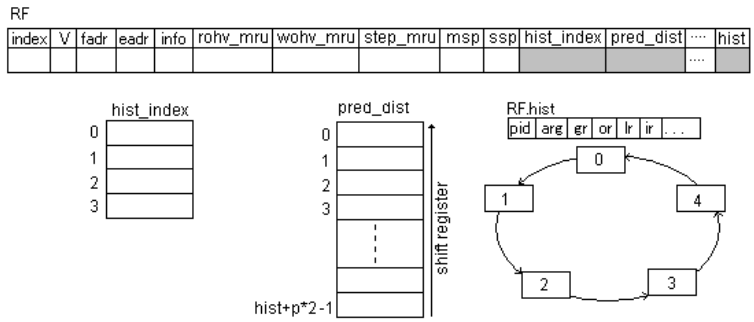



図 3.12: 命令区間表

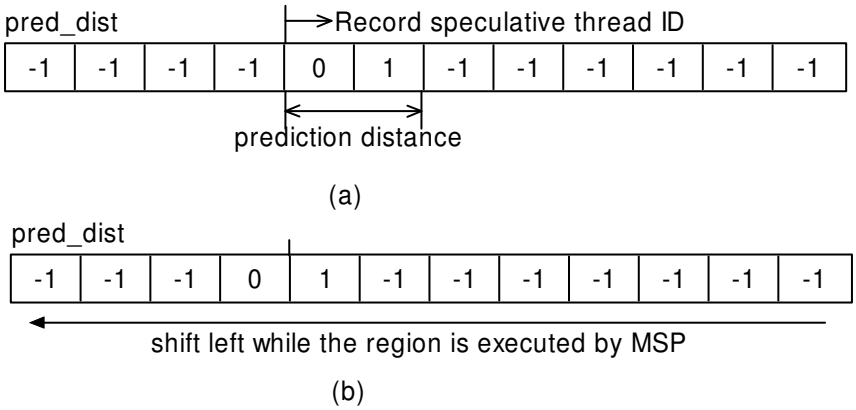



図 3.13: リスト pred_dist

に基づいて入力予測値を計算する。hist(歴史記録数)=4，p(スレッド数)=4 の場合を例にすると，リスト pred_dist の操作が図 3.13 に示すように行われる。投機実行スレッドにより命令区間が実行されると，図 3.13(a) に示すように，pred_dist[4] から空いている箇所 (-1) を探して投機スレッド ID が書き込む。図 3.13(b) に示すように本命令区間が MSP により実行されるとリスト pred_dist が全体的に一個分左へシフトされる。

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，SSPi 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。

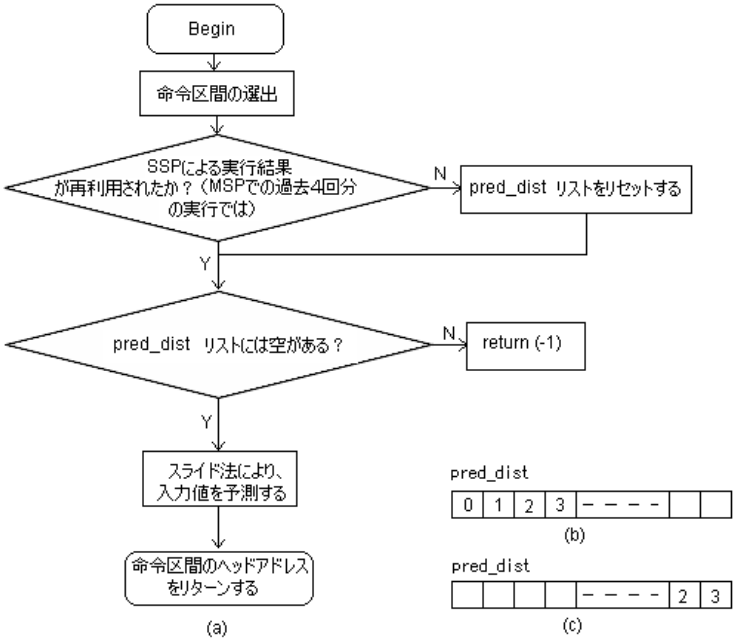



図 3.14: 命令区間の入力値予測処理プロセス

## 3.8 命令区間の入力値予測処理

投機スレッドは新たな命令区間を実行しようとする際は，図 3.14(a) に示す手順に従う。まず，RF 中の履歴により，サイクル数を最も削減できそうな命令区間を選出する（3.7 節を参照)。そして，図 3.14(b) に示すように，本命令区間が過去 4 回 (図 3.13(b) に示すように MSP が本命令区間を毎回通った後，pred_dist リストを一個分左へシフトする)MSP によって実行された際に，SSP による投機実行の結果が再利用されたかどうかを判断する。投機実行の結果が再利用されなかった場合は投機スレッドの入力値が正確に予測できなかったと判断し，pred_dist リストと本命令区間に関する入力予測値を全てリセットし，その時点から改めて入力値を予測する。pred_dist リストの内容をチェックし，空があると，投機スレッドの ID を書き込み，ストライド法により，入力値の予測を行う。空がないと，pred_dist リストが図 3.14(c) のようになり，本命令区間に対して投機スレッドは 8 度ほど事前実行が行ったが，その間にメインスレッドが本命令区間を一度も実行しなかったことが分かる，投機実行が行われなくかり，この命令区間については，入力値予測プロセスが中止される。

## 3.9 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び出される場合，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。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。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 $G_i$ ，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。

$$Gai = Gi * Ri/64 \quad Loi = Ti * (64 - Ri)/64$$

$Ga_i < Lo_i$ の場合は RB を参照すべきでないと判断する。本手法では，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 Ri の値を初期化して，再び再利用できるようにする。

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，再利用する。エントリが見つからなかった場合は，命令区間を実行する。命令区間を 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。

## 3.10 投機実行区間の選択

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，かつ，SSP が登録したエントリの再利用頻度が高いものが，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，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 Gi，MSP が実行後 RB に登録した回数を Xi，再利用回数を Ri，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 Si とする。Gi ＜ 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 Xi +Ri = 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 Xi +Ri であるため，SSP の貢献により MSP が全て再利用した場合は Gi × (Xi + Ri) の高速化が可能となる。すなわち評価式 Ei は，

$$Ei = Gi \times (Xi + Ri) \times Si$$

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，Gi には直前の実行結果をそのまま用いる。また，Xi，Ri，Si には，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，Si 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ことにより精度を高める。

## 3.11 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 3.15 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

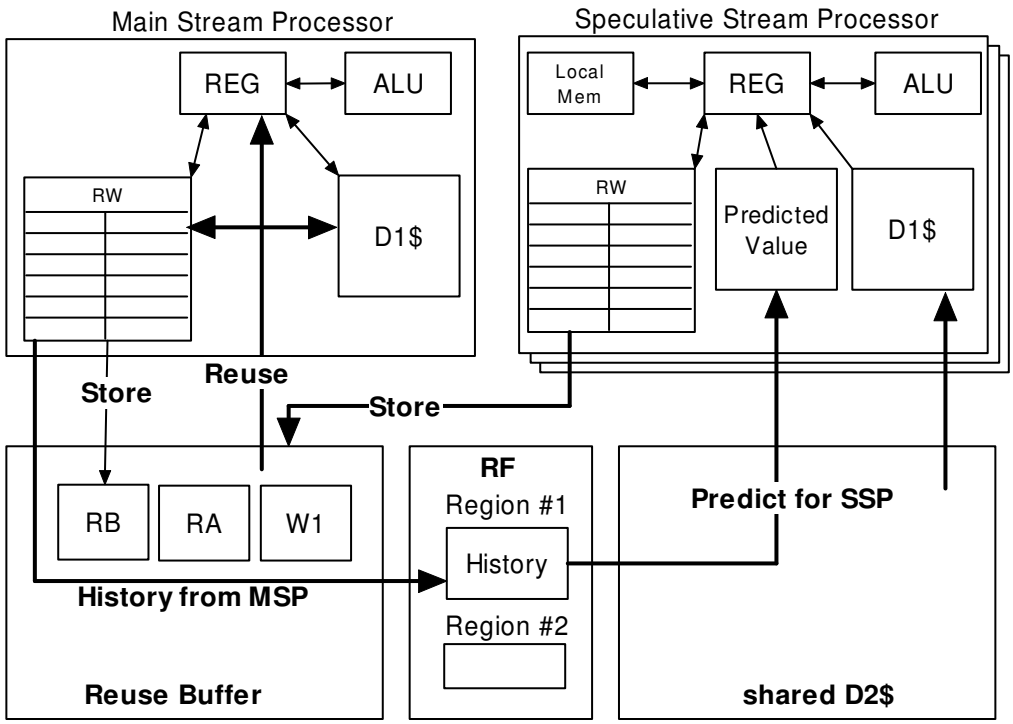



図 3.15: ハードウェアモデル

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。RWin を再利用表へ蓄積する際，同時に入力履歴として RF に格納する。入力履歴は RWin 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 RWin 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，RWin に登録する。以降は前述のように，(1) RWout，(2) RWin 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

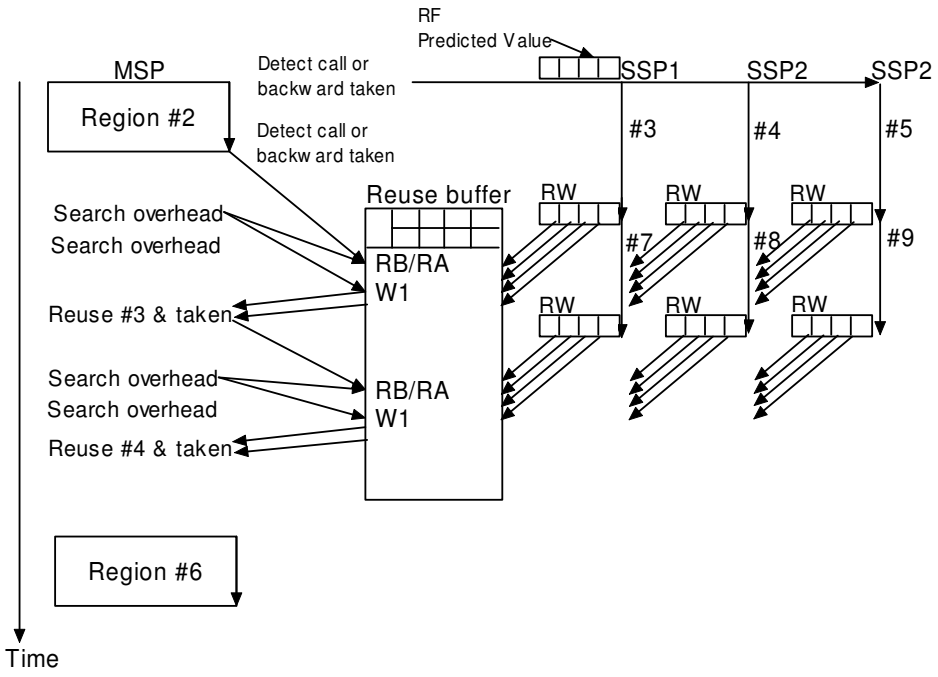



図 3.16: 評価モデル

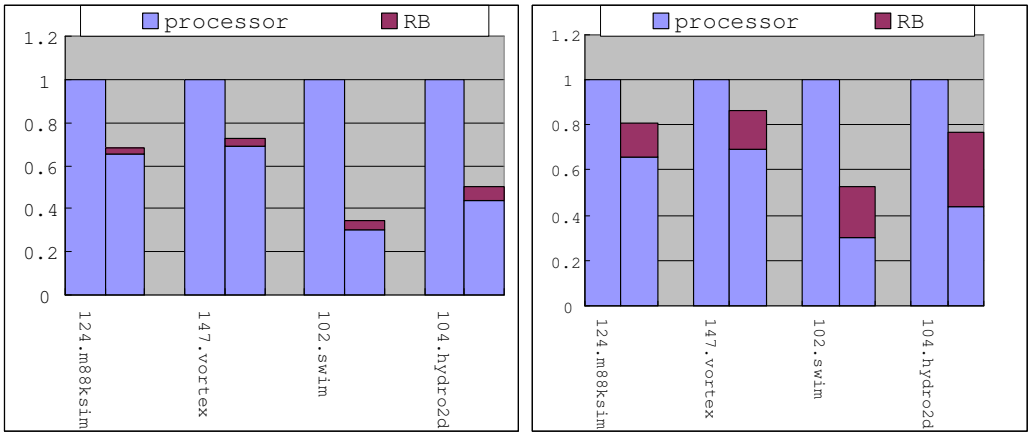
長レイテンシとする。図 3.16 に再利用表に関する評価モデルを示す。MSP が命令区間を検出すると SSP を起動する。また，MSP が命令区間を飛び越そうとする際には，再利用表の連想検索コストと，ヒット時の書き込みコストが生じる。再利用表の検索に必要な主記憶値を取得する際にキャッシュミスが発生した場合はサイクル数を加算する。SSP は前述のように投機実行対象区間を選択し，SSP が担当する命令区間の投機実行を完了した際には，RW から再利用表に対して書き込みを開始すると同時に，空き RW エントリを用いて次の担当命令区間の実行を開始できると仮定する。また SSP が RW から再利用表に記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　再利用表の分割とその評価

再利用表のエントリ数が多いほどアクセス時間が長くなり，再利用表全体をアクセス対象とすると，消費電力も多くなり。図 4.1（a）は，各評価プログラムを実行する際の，再利用を適用しない場合の実行ステップ数をベースとした実行ステップ比率を示している，赤は，再利用表へのアクセスサイクル数である。ただし，再利用表の動作速度がプロセッサの周波数よりも低い場合，この評価は正しくない。一方，4.1（b）はプロセッサの動作周波数を１ＧＨｚとし，再利用表（１Ｋエントリ）が 200MHz の場合のアクセス時間の比率を示している。4.1（a）よりも，性能への影響が大きい。この問題を解決するため，再利用表を分割し，各ブロックのエントリ数を数分の一にし，アクセス速度を向上させる方法を考えた。同時に，１ブロックのアクセスに必要な消費電力も，再利用表全体に比べ，大幅に削減できると考えられる。

## 4.1　分割の導入

図 4.2(a) のプログラムを例として再利用表の分割を導入する。対応する $RW_{in}$，RB，RA および W1 のレコードを図 4.2(b) に示す。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 (0001000C 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 に格納し手続きを終了する。RW の各レコードに 16 バイトのデータおよび対応する 16 ビットのマスクを格納できる。引数のアドレスが 0x0001000C であり，16 バイト境界をまたぎ，値”ABCDEFG”が二つのレコードに分かれることになる。すなわち，$RW_{in}$ 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 0001000C，第 2 レコードにアドレス 00010000 と 4 バイト値 ”41424344 ”，第 3 レコードにアドレス 00010010 と 4 バイト値 ”454600– ”，第 4 レコードに終端を記録する。RWout 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，RWin の先頭レコードから順に $RB_{in}$ へ登録する。分割前，RW における関数 strlen(str) の内




(a) Cycles (b) Time

図 4.1: ＲＢのアクセスサイクル数と時間

容が既に RBin に登録した内容と異なる場合，また，これまで登録しなかった場合は，全 RB 範囲内で空きエントリを探し，空きエントリがあれば，$RW_{in}$ の各レコードを（nextkey を 200 とする）新たなエントリとして追加する。分割後，$RW_{in}$ の各レコードを各ブロック ($RB_{0...}$) に振り分けることになる。

$RW_{in}$ のレコードを各ブロックに登録する際に，対応するブロックに空きエントリがなくなると，各ブロックでは登録しようとする命令区間の全てのレコードを削除する，そこで，分割後各ブロック ($RB_{0...}$) の利用率が RB 全体の利用率とほぼ同じであるとよい分割法と考える。

図 4.2(b) は $RW_{in}$ に格納する各レコード (16 バイトのデータおよび 16 ビットのマスク) を示す。$RW_{in}$ は 256 レコードを格納できる，各レコードの ID は 0〜255 となる。レコード 0 には命令区間の先頭アドレスを持つ，各命令区間の ID という役割を果たす。各レコード ID をブロック ID にマッピングすると横方向の分割，各命令区間の先頭アドレスをブロック ID にマッピングすると縦方向の分割となる。

以上のレコードＩＤと命令区間の先頭アドレスを用いて再利用表の分割法を以下の三通り提案する。

## 4.2 先頭アドレスに基づく分割 (row)

本分割法は，先頭アドレスの数ビットに基づいて入力データを各ブロックに振り分ける分割法である。８分割する場合は，ブロック ID は下式により計算

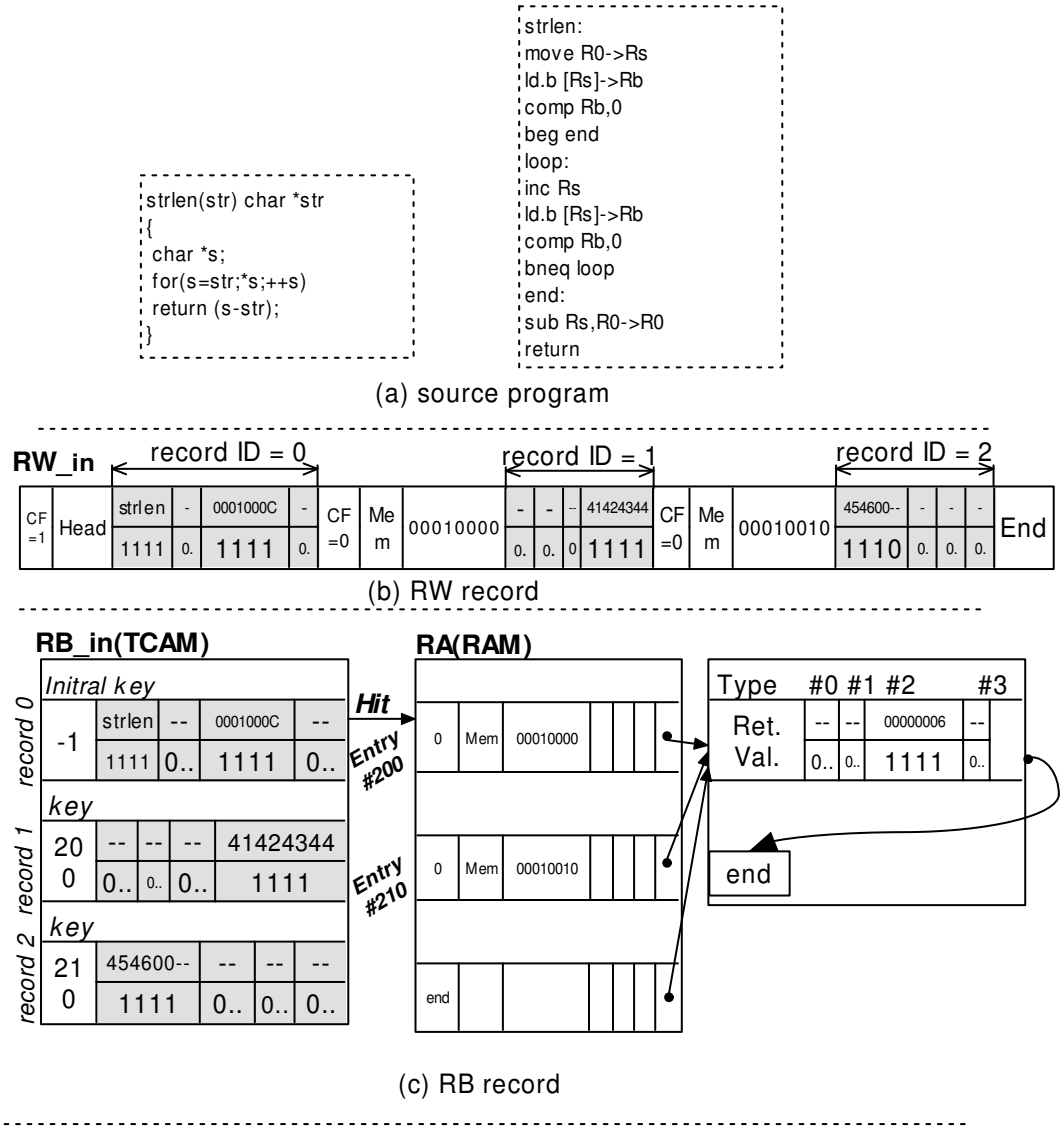



図 4.2: RB の構成．RWin の 1 行分の入力を木構造の 1 パスとして格納する。CAMPart が枝，RAM-Part がノードに対応する。

する。

$$\text{ブロック } ID = ((\text{先頭アドレス}) >> 2\,)\ \&\ (7)$$

命令区間の先頭アドレス中の第 2 ビットから数ビット（8 分割の場合は 2〜4 ビット目，16 分割の場合は 2〜5 ビットを用いる）を Block Number として利用し，再利用データの登録および検索処理は Block Number に対応するブロック内でのみ行う。図 4.3 では，先頭アドレスの 2〜5 ビットを使って 16 分割している。先頭アドレスが 0x0001a90c である場合は，この命令区間に関するすべての再利用

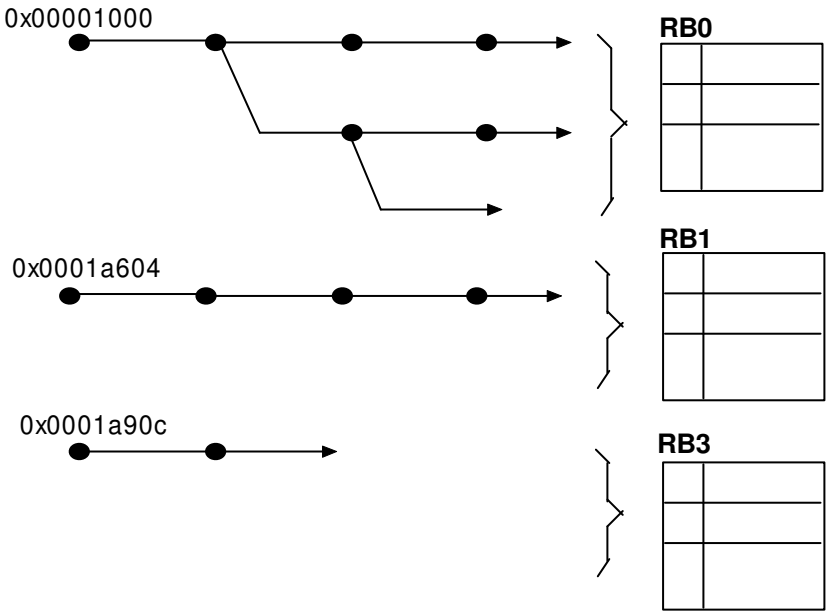



図 4.3: 先頭アドレスによる分割

データの格納と検索処理がブロック 3 で行われる。先頭アドレスは 0x0001a604 の場合は，この命令区間に関するすべての再利用データの格納及び検索処理がブロック１で行われる。

この方法の特徴は，同一命令区間の再利用データが同一ブロックに集中し，命令区間の競合を抑えることができる点である。再利用率が高い命令区間と低い命令区間が多くのエントリを占める場合に，命令区間を異なるブロックに分けることにより，再利用率が低い命令区間のデータによって再利用率の高い命令区間のデータを上書きされることがなくなると考えられる。また，各命令区間の入力エントリ数が均一である場合は，各ブロックの利用率と全体の利用率が均一となるため，この手法も有効であると考えられる。

## 4.3 レコード ID に基づく分割（col）

レコード ID を用いて入力データ各ブロックに振り分ける分割法である。図 4.4 に示すように，再利用区間が登録される際に，入力順，つまり木の根から順にレコード ID を付ける。入力セットはレコード ID に基づいてそれぞれのブロックに分散される。命令区間の先頭アドレスから参照順に０から１ずつレコードＩＤをインクリメントする。レコードＩＤがブロック数より大きくなると，折り返してブロック０から続いて処理を進める。この分割法に基づき，ブロック

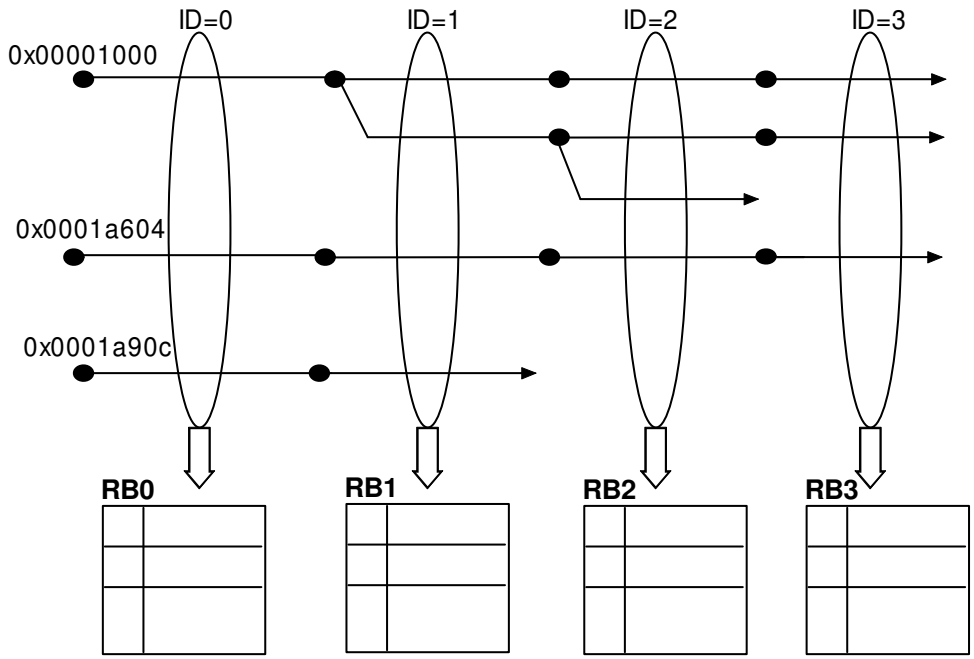



図 4.4: レコードＩＤによる分割

０中には全ての命令区間の先頭アドレスを格納し，ブロックｎには各命令区間の（ｎ＋Ｎ＊ブロック数）番目の入力を格納する（整数Ｎ≧０）。

この方法の特徴は，同一命令区間の入力列を各ブロックに分散できる点である。命令区間の始まりを格納する箇所が下式により事前に分かるため，ブロックの選択回路の制御ロジックが簡単になる。

$$\text{ブロック } ID = (\text{レコード } ID) mod (\text{ブロック数})$$

長い入力列を持つ命令区間に対しては各枝がそれぞれのブロックに分散でき，再利用表全体の利用率が高くなると予想されるものの，入力列の短い命令区間が多く存在する場合に対しては，入力データが番号の小さいブロックに集中するため，再利用表全体の利用率が悪くなり，再利用結果に悪影響を与えると考えられる。

## 4.4 先頭アドレスとレコード ID の組み合わせによる分割（cr）

図 4.5(a) に示すように，先頭アドレスの第 3 ビットからの数ビット分をレコード ID と加算し，結果をブロック番号として利用する分割法である。８分割の場

合，下式によりブロック ID を計算する：

$$\text{ブロック}ID = (((\text{先頭アドレス}) >> 2) \,\&\, (0x7) + \text{レコード}ID) mod (\text{ブロック数})$$

本分割法では，4.3.1 の方法と同様に，まずヘッドアドレス中の数ビット分に基づき，入力データを格納/検索する開始点を決め，そこから入力データの処理を進める。

本方法では，入力データのパターンに関わらず再利用データを分散できるため，前述の方法より再利用表の利用率が高くなると考えられる。

図 4.5（b）に，再利用表を１６分割する際に，入力数 4 つの命令区間 0x1abf8 と，入力数 5 つの命令区間 0x1efd4 それぞれの場合の登録の様子を示す。命令区間 0x1abf8 の入力数が 8 個以上になると，命令区間 0x1efd4 の入力データとブロック５から重なることになる。すなわち，重なっているところで競合により，有用なエントリを失う確率が高くなる。分割数を増やすと，重なる確率が少なくなるが，各ブロックごとのエントリ数が少なくなり，競合の可能性が高くなる。ブロック数とブロック当たりのエントリ数の両方の要素間のバランスが重要である。後章では，SPEC ベンチマークにより，各分割に対して評価を行う。

## 4.5 Stanford ベンチマークによる予備評価

### 4.5.1 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 4.1 に示す。命令レイテンシは，SPARC64-III[19] を参考にした。

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 16KB とした。

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

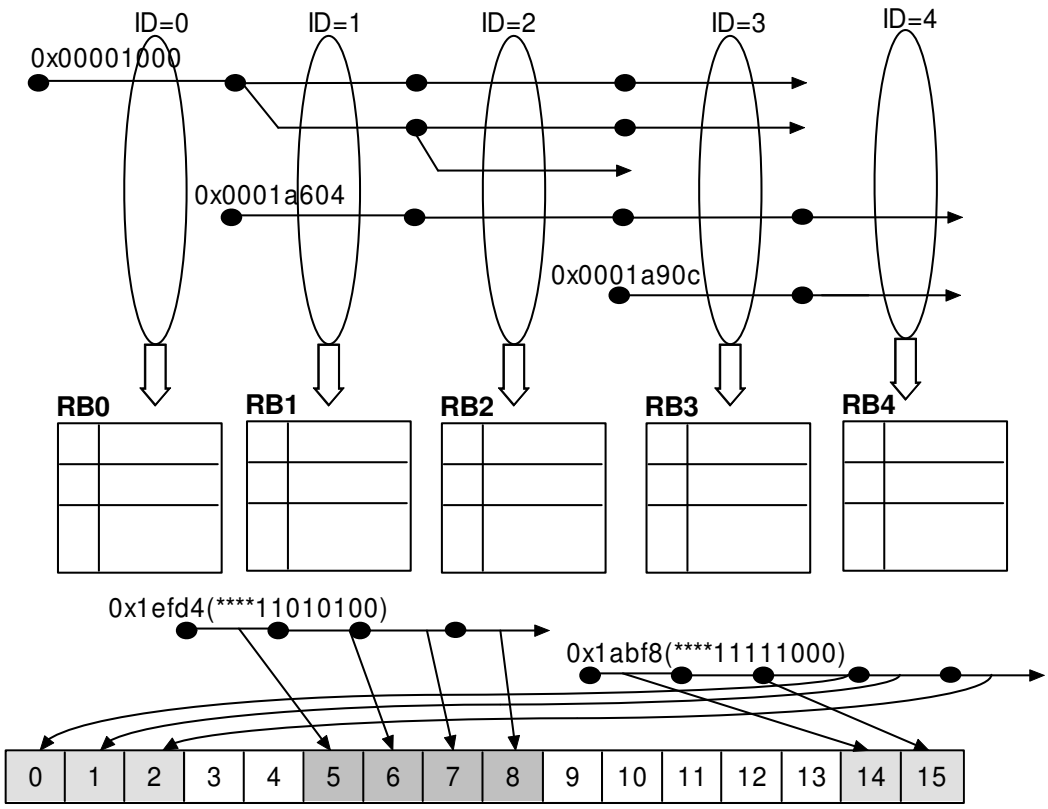



図 4.5: 先頭アドレスとレコード ID の組み合わせによる分割

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。ソフトウェアシミュレータを用い，Stanford ベンチマークを用いて，各分割法に対し，プログラムの実行ステップ数を実測した。図 4.6（a），(b)，(c)，(d）にはそれぞれ１ K エントリ容量の再利用表を８分割，１６分割，３２分割および６４分割した場合の評価結果を示す。縦軸は再利用適用時の実行時間を非適応時の実行時間により正規化したものであり，横軸は各ベンチマークプログラムを示す。ベンチマークごとの 4 本の棒グラフは左からそれぞれ再利用表を分割しない (nb)，先頭アドレスによる分割 (row)，レコード ID による分割法 (col)，先頭アドレスとレコード ID 併用する分割 (cr) を示す。グラフ中の凡例は時間の内訳を示す。exec は命令実行時間である。test，mem 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時間である。test と mem の二つの部分は CAM へ

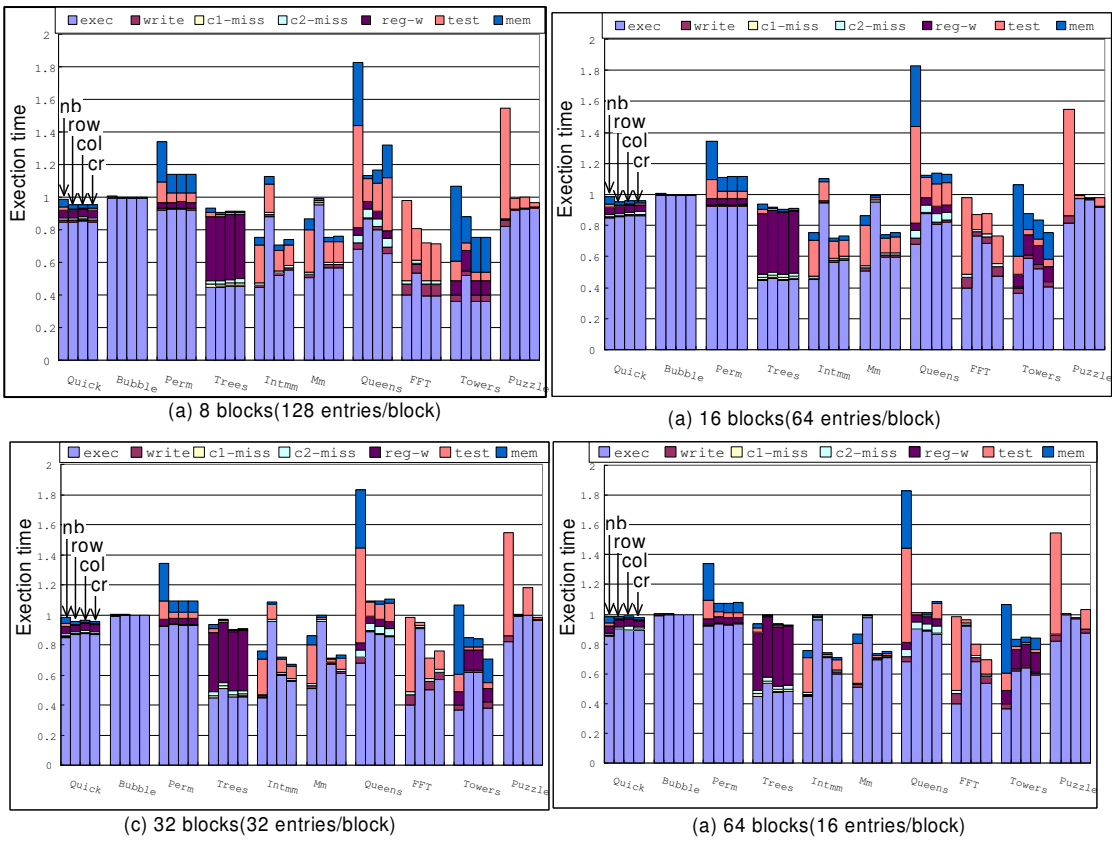



図 4.6: Stanford ベンチマークによる分割法の評価

のアクセス時間であり，表 4.1 に示す CAM のサイクル周波数により計算した。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時間である。また，c1_miss，c2_miss，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。

評価の結果，exec の部分に対して，row は 他の方法に比べて結果が著しく悪い。理由は命令区間のサイズが千差万別であり，大きな命令区間が分割したブロックに納まらず，再利用率が悪化したことが原因である。一方，col と cr の exec はそれほど悪くなく，全体的に col と cr の差異は大きくない。分割数が少なくブロックごとの容量が多い場合には，cr と col による分割の exec がほぼ同じであるが，分割数を増やすと，col 分割の exec が cr より悪くなった。表 4.2 に row，col および cr 全体の評価結果をまとめる，cr 分割がより高い性能を得られることが分かった。

表 4.1　シミュレーション時のパラメータ

| | |
|---|---|
| RW 深さ | 6 |
| RF エントリ数 | 256 |
| RB エントリ数 | $1K$ |
| プロセッサの速度 | $1ns/cycle$ |
| CAM の速度 (1024 エントリ/ブロック) | $5.6ns/cycle$ |
| CAM の速度 (128 エントリ/ブロック) | $2.6ns/cycle$ |
| CAM の速度 (64 エントリ/ブロック) | $2.2ns/cycle$ |
| CAM の速度 (32 エントリ/ブロック) | $1.9ns/cycle$ |
| CAM の速度 (16 エントリ/ブロック) | $1.7ns/cycle$ |
| ラインサイズ | $32Byte$ |
| 1 次 cache 容量 | $32KByte$ |
| 1 次 cache ウェイ数 | 4 |
| 1 次 cache ミスペナルティ | $10cycles$ |
| 2 次 cache 容量 | $2MByte$ |
| 2 次 cache ウェイ数 | 4 |
| 2 次 cache ミスペナルティ | $100cycles$ |
| Register-Window | 4 $set$ |
| Window ミスペナルティ | $20cycles/set$ |
| 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 $cycles$ |

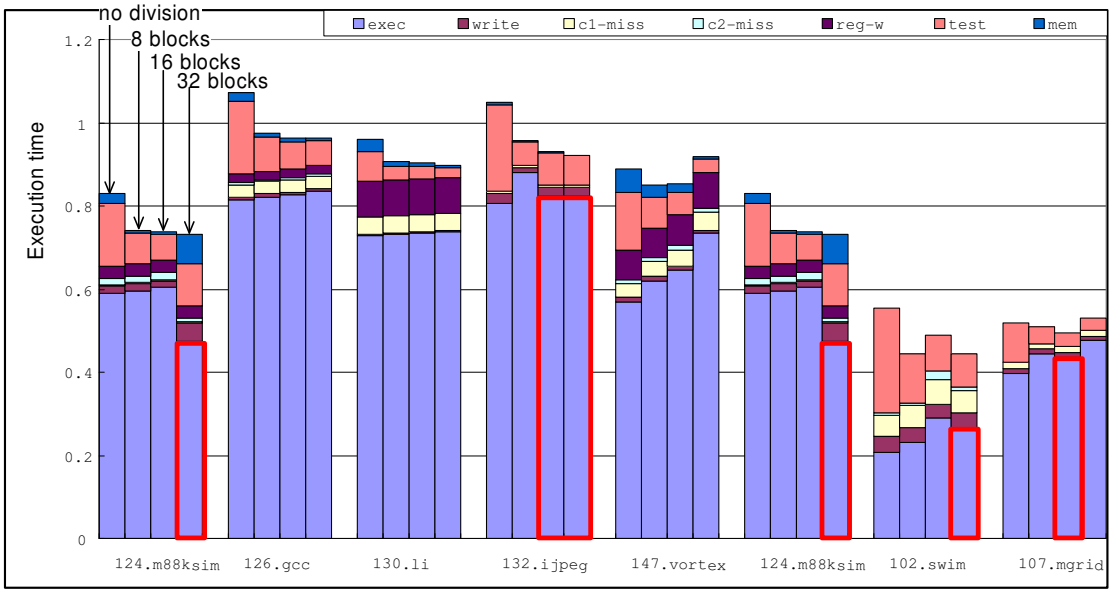



図 4.7: SPEC ベンチマークによる評価

表 4.2　平均削減実行時間

| | 8 分割 | 16 分割 | 32 分割 | 64 分割 |
|---|---|---|---|---|
| row | 0.7% | 0.7% | 0.1% | 1.5% |
| col | 8.9% | 7.3% | 7.8% | 8.9% |
| cr | 7.4% | 9.3% | 10.8% | 9.1% |

## 4.6　SPEC ベンチマークによる評価

再利用表を分割すればするほど再利用効果が悪化すると考えられる。Spec95 ベンチマークを用いて，再利用表を分割したときの結果を図 4.7 に示す。縦軸は再利用表を分割しない場合により正規化した実行時間であり，横軸は各ベンチマークプログラムを示す，ベンチマークごとの 4 本の棒グラフは左からそれぞれ各プログラムに対し，再利用表を分割しない，8 分割，16 分割，32 分割の結果である。全体的に，再利用表を分割するほど性能が悪くなっているが，赤の枠のところでは分割数が多いほうが高い性能となり，矛盾した結果となっている，このようなケースについて次章において原因を究明する。なお，これまでの実験により cr 分割法が最適と分かった。分割数については，次章の結論を待って考察する。

# 第５章　投機対象区間選択機構の改良

本章では，前節の中で最も矛盾した評価結果を残したベンチマークプログラム m88ksim の矛盾点を説明した上で，投機機構について改善案について述べる。

## 5.1　m88ksim の評価結果の矛盾点とその分析

図 4.7 に示す評価結果を見てわかるように，m88ksim では RB が 32 分割した時の性能が 16 分割よりよくなっている。これは RB が 16 分割される際の再利用率が 0.348 なのに対し，32 分割される際の再利用率が 0.487 となっているためである。分割数が多いほど RB の利用効率が落ち，再利用率が低くなるという予想に反する。この原因を分析し，対策を考える。

表 5.1　m88ksim の命令区間再利用状況

| | １６分割 | | ３２分割 | |
|---|---|---|---|---|
| 命令区間 | サイクル数 | 回数 | サイクル数 | 回数 |
| 0x316b8(関数) | 15559201 | 101209 | 22568010 | 146746 |
| 0x32fd8(関数) | 9522556 | 190883 | 7996925 | 160312 |
| 0x1abf8(ループ) | 0 | 0 | 6313132 | 901876 |
| 0x25498(関数) | 6007014 | 375367 | 5001888 | 312575 |
| 0x2bb0c(関数) | 4228280 | 302020 | 4485348 | 320382 |
| 0x1cc0c(ループ) | 1813736 | 453434 | 1874180 | 468545 |

表 5.1 に，再利用表を 16 および 32 分割した際の，m88ksim における再利用状況を示す（削減できたステップ数順）。この中で，命令区間 0x1abf8 (ループ) に対しては，32 分割の場合に合計 901876 回，6313132 ステップ再利用できたが，16 分割の場合には一度も再利用できなかった。これは両者の大きな違いである。

命令区間 0x1abf8 の入力値は単調変化型であり，投機スレッドによる並列事前実行に対して適応できるはずである。この命令区間は，再利用表が 16 分割される場合には，MSP が命令区間 0x1abf8 を実行し始めると同時に，SSP が再利用履歴に基づいて選択した命令区間を実行している（3.10 節を参照）。図 5.1 に RB を 16 分割した際の 0x1abf8 のコンテキスト分析の結果を示す。すべての投機スレッドが長い命令区間 0x1efa4 を実行している。MSP が命令区間 0x1abf8 の何回実行を終わっても投機スレッドの状態が常にビジーであるため，この命令

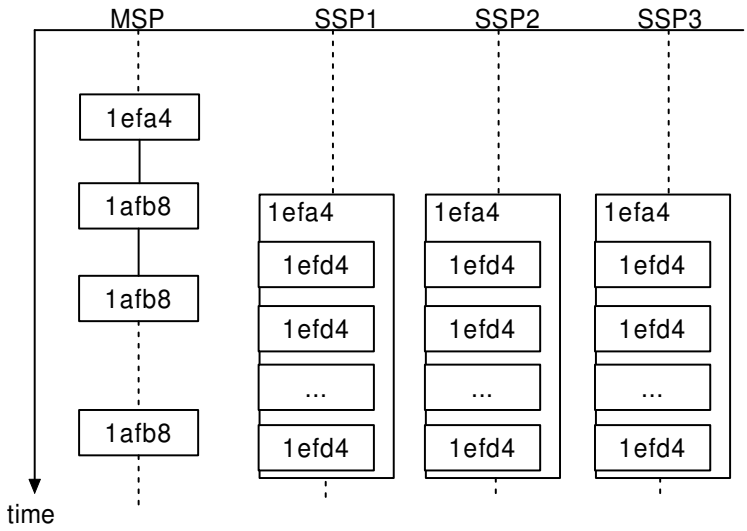



図 5.1: 0x1abf8 ループのコンテキスト分析

区間の事前実行が投入できない。その上，オーバーヘッド評価機構（5.3 節を参照）により，命令区間 0x1abf8 の再利用しないケースが続いたため，$R_i$（ヒット率）の値が小さくなり，$Ga_i$ の値も小さくなり，再利用効果が現れるにも関わらず，再利用表の検索を中止することになる。このことにより，命令区間 0x1abf8 の再利用回数が 0 となる。一方，RB が 32 分割される際には，MSP が命令区間 0x1abf8 を実行し始める時点で，履歴表内容が 16 分割の場合と違い，図 5.1 に示す状況にならなかったため，命令区間 0x1abf8 の再利用回数が 0 ではなかった。m88ksim には命令区間 0x1abf8 の実行がおよそ 110 万回あり，32 分割の場合ではその内 90 万回が再利用できた。

すなわち，以下が原因であると考えられる。

- 16 分割，32 分割両方の RB のブロック容量が違うため，再利用履歴も異なり，投機実行にされる命令区間が違う。
- 以下の評価式で，再利用検索が行われる前に，履歴中の各命令区間に対して再利用性能を評価する。

$$G_i * R_i/64 \quad > \quad T_i * (64 - R_i)/64$$

- 投機スレッドが大きな命令区間を実行すると，MSP で実行されている命令区間の再利用効果があるにも関わらず，SSP が空いていないために事前実行に投入できない。同時に本命令区間に対して，再利用評価のミス率が上がり，再利用効果のない命令区間と判断されてしまう。

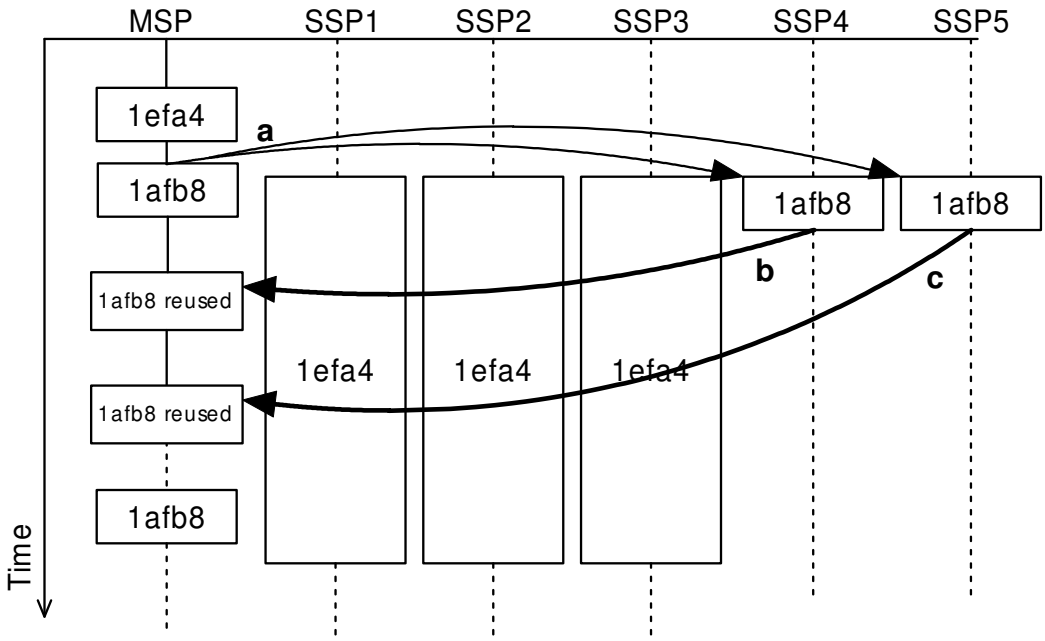



図 5.2: 改良案１の例

## 5.2 改良案

図 3.14(c) に示すケース（pred_dist リストに空がなくなり，本命令区間に対して投機スレッドは８回ほど事前実行が行われたが，その間にメインスレッドが本命令区間を一回も通っていなかったため，入力値予測プロセスが中止される。）が発生すると，3.10 節に述べた命令区間選出法に基づいて他の命令区間が選択されるまでの間に，投機スレッドは投機実行を行うことができず，無駄になる，同時に 5.4 節に示されたケースとあわせて以下の改良手法を提案する：

【改良案】１：二つの投機スレッドを増やし，五つの投機スレッドを二種類に分け，第１の投機スレッドは 3.10 節に述べた区間選出方法に基づき，履歴から命令区間を選んで投機実行する。第２の投機スレッドが MSP の動きを捉え，そこで実行されている命令区間に対して投機実行をする。図 5.2 には図 5.1 の改良後の動作を示す。図 5.1 には投機スレッドの状態が常にビジーであり，0x1abf8 を投機実行に投入できないの状況に対し，増やされた二つの SSP が MSP と同時に同じ命令区間を実行する。図 5.2a の時点で，MSP が 0x1abf8 区間に入れ，SSP1～SSP3 が 3.10 節に述べたアルゴリズムで投機実行命令区間を選択する，SSP4 と SSP5 が MSP と同様に 0x1abf8 を投機実行する。SSP4 と SSP5 の実行結果が MSP に再利用でき，図 5.1 に示すケースにならない。

【改良案２】: 3.10 節に述べた方法により，一つの命令区間だけを選出するのではなく，上位二つの命令区間を選出し，一番上位の命令区間の pred_dist に空がない場合に，第二位の命令区間に対して事前実行を起動する，同時に改良案１も実装する。

## 5.3 改良案の評価

以上に述べた改良手法を用いて，ソフトウェアシミュレータを開発し評価を行った。なお，本実行モデルのうち，特に，連想検索装置により構成された再利用表 (入力側) に対して要求される機能は次の通りである：

- 命令区間の登録：再利用ウインドウの入力側 ($RW_{in}$) の内容を $RB_{in}$ に登録する際に，$RB_{in}$ の連想検索を行い，既登録であるかどうかを判断し，既登録である場合には登録を行わず，未登録の場合には，連想メモリ (CAM) の空きエントリを選択して書き込む。
- 命令区間の再利用テスト：MSP が $RB_{in}$ を検索する際には，$RB_{in}$ の連想検索を行い，既登録であるかどうかを判断し，既登録である場合には該 $RB_{in}$ エントリに属する連想メモリ (CAM) および通常メモリ (RAM) を読み出し，未登録の場合には，検索を終了する。

以上の機構を持つ専用連想メモリ (CAM) の製作により，毎ブロックの容量が 1024 エントリ，128 エントリ，64 エントリ，32 エントリ，16 エントリの場合に，それぞれの速度が $5.6ns/cycle$，$2.6ns/cycle$，$2.2ns/cycle$，$1.9ns/cycle$，$1.7ns/cycle$ であることがわかっている。

### 5.3.1 SPEC による評価結果

図 5.3,5.4,5.5 に，評価結果を示す。縦軸は再利用しない場合で正規化した実行時間であり，横軸は各ベンチマークのプログラムを示す。ベンチマークごとの 6 本の棒グラフは左から：再利用しない場合，RB を分割しない場合，RB を各分割 (8,16,32,64) した場合の実行時間である。また，グラフ中の凡例は時間の内訳を示す (図 4.6 と同様)。

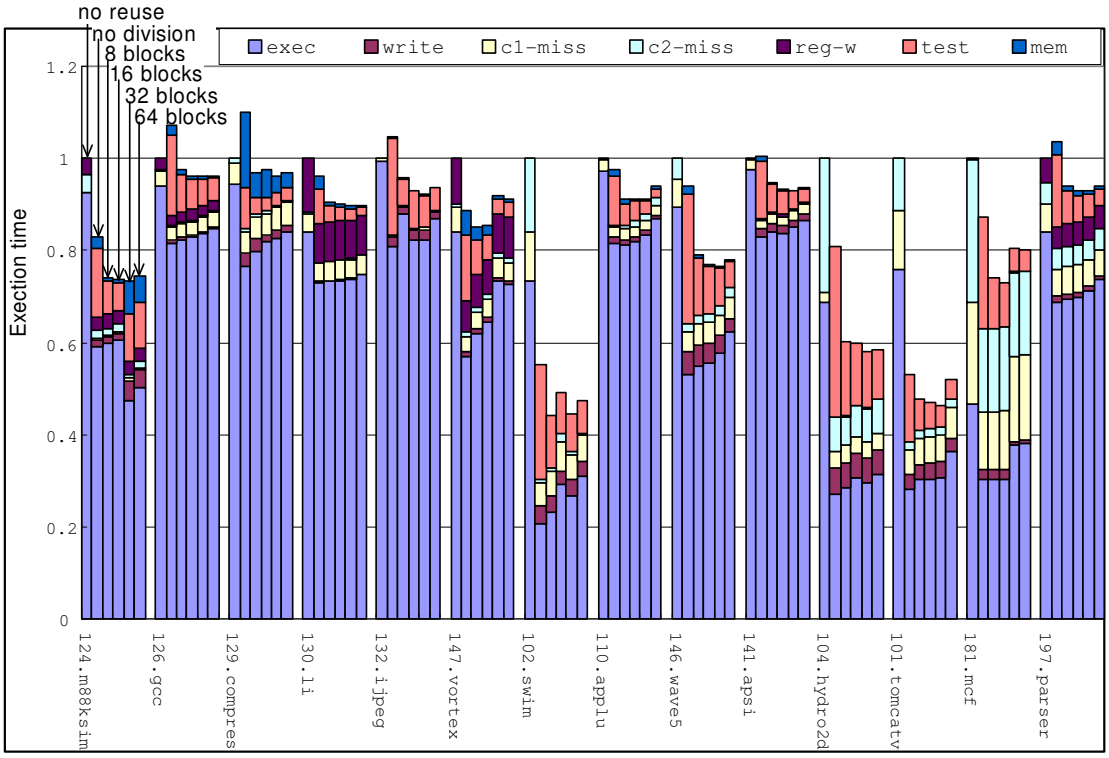



図 5.3: 改良前のモデルの評価結果

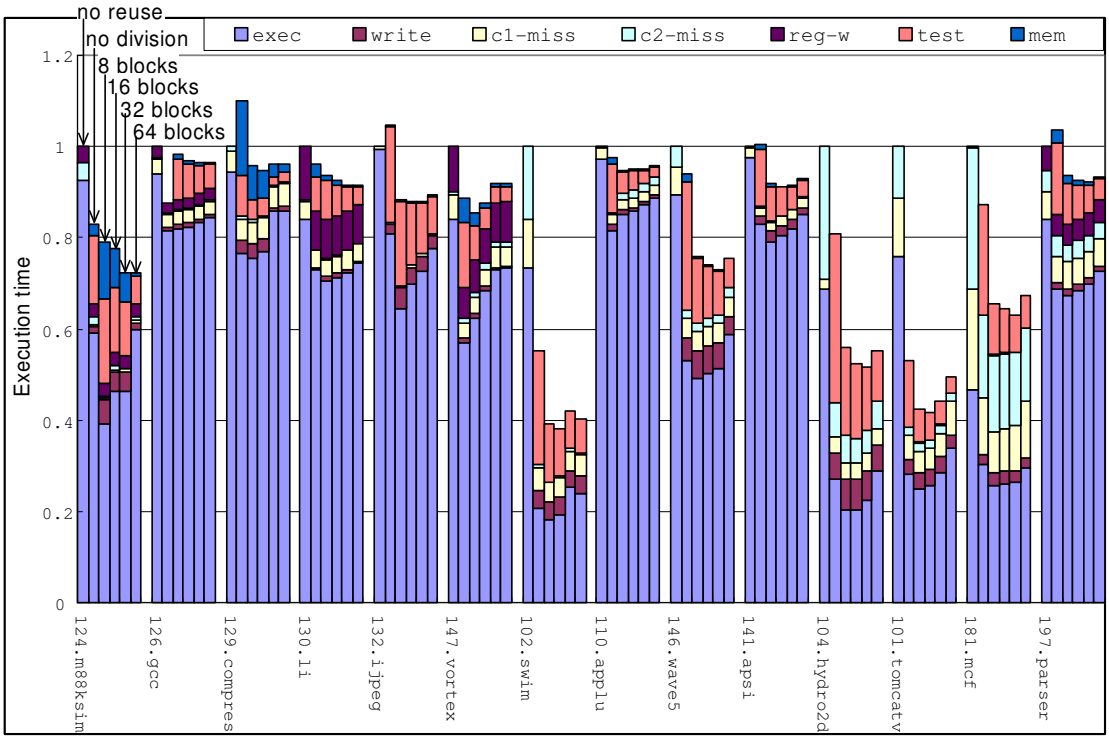



図 5.4: 改良案 1 の評価結果

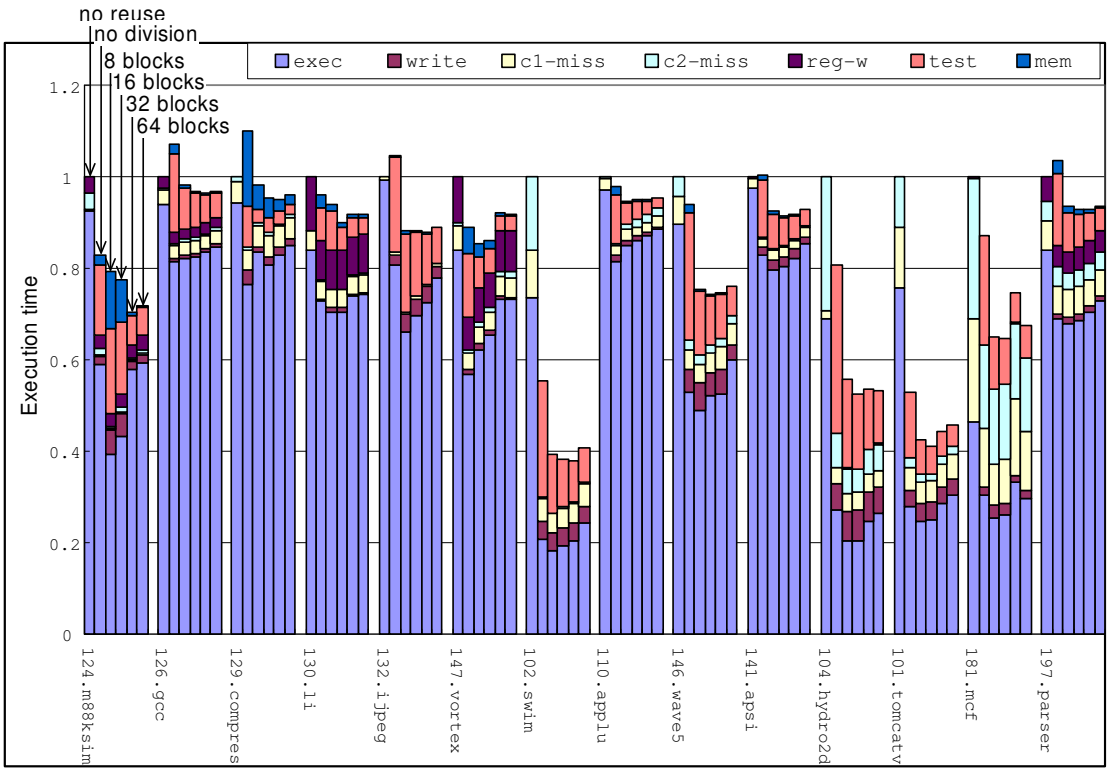



図 5.5: 改良案 2 の評価結果

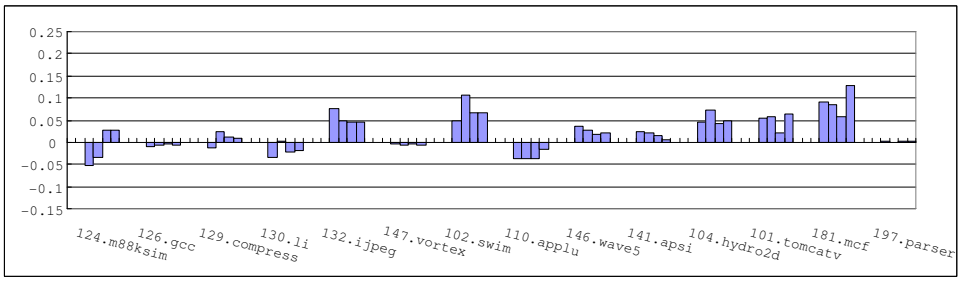
表 5.2　削減できたサイクル数の平均値

| | 8 分割 | 16 分割 | 32 分割 | 64 分割 |
|---|---|---|---|---|
| 改良前 | 20.7% | 21.3% | 21.6% | 20.1% |
| 改良案 1 | 22.8% | 24.5% | 24.4% | 23.0% |
| 改良案 2 | 22.7% | 24.8% | 23.9% | 23.6% |
| 分割しない | 9.9% | | | |

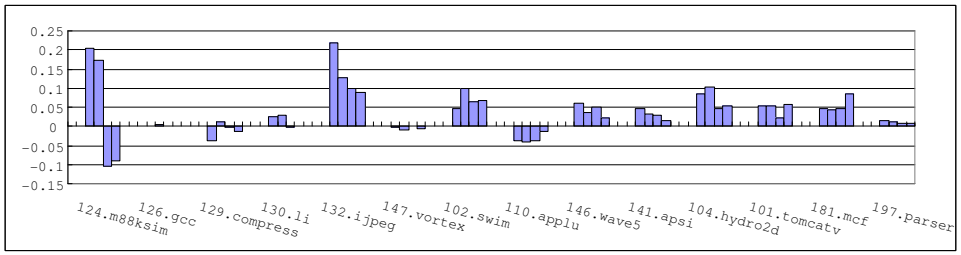
表 5.3　削減できたサイクル数の最大値

| | 8 分割 | 16 分割 | 32 分割 | 64 分割 |
|---|---|---|---|---|
| 改良前 | 55.7% | 53.1% | 55.5% | 52.6% |
| 改良案 1 | 61.9% | 61.8% | 57.8% | 59.6% |
| 改良案 2 | 60.7% | 61.6% | 62.1% | 59.2% |
| 分割しない | 47.0% | | | |

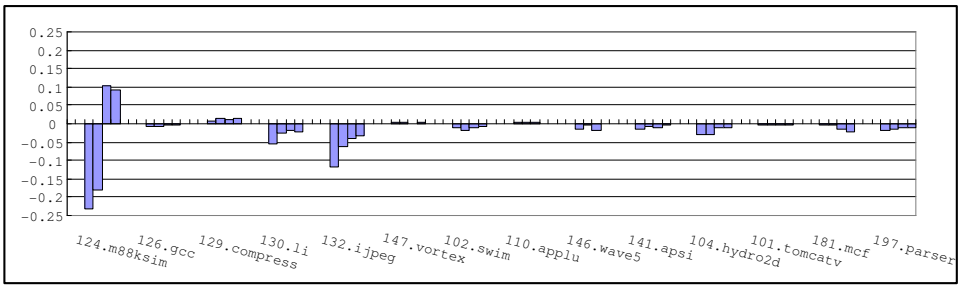
表5.3 と表5.4 の結果により，削減できたサイクル数は，改良案 2 で最大 24.8% となった。改良案 2 を採用し，再利用表を 16 分割することで，平均的に最大の




(a) Time_total(before) - Time_total(proposal2)




(b) Time_instruction(before) - Time_instruction(proposal2)




(c) Time_RB(before) - Time_RB(proposal2)

図 5.6: 改良案 2 と改良前の結果対照

効果が得られることがわかった。

図 5.6 は，改良前と改良案 2 の結果の差分を示す。縦軸は図 5.3 と同じ，横軸は各ベンチマークのプログラムを示す。ベンチマークごとの 4 本の棒グラフは RB を各分割 (8,16,32,64) した場合に改良前と改良案 2 の差分を示す。(a) は改良前の総実行時間と改良案 2 の実行時間の差分を示す。(b) は改良前の命令実行時間と改良案 2 の命令実行時間の差分を示す。(c) は改良前の RB アクセス時間と改良案 2 の RB アクセス時間の差分を示す。(b) に示すように多くのプログラム (102,104,124,132) の命令実行時間が 10%以上削減できた。(c) は，再利用率が上がると同時に，RB アクセス時間が増えたことを示している。

# 第6章　おわりに

再利用表および投機機構の改良による区間再利用の高速化手法を提案し，ソフトウェアシミュレータを用いて評価した。全ての機構を搭載し，再利用表を16分割 (64エントリ/ブロック) した場合，SPECの実行時間を平均24.8%，最大で61.6%削減できた。また，分析により，命令実行時間を削減すると同時に再利用表のアクセス時間が増え。今後，再利用表の動作周波数の性能向上に伴い，区間再利用の実行効果は，より向上することがわかった。

# 謝辞

本稿執筆に際し，多くの方々から大変貴重な助言を頂きました。この場をお借りして感謝を申し上げたいと思っています。

何よりもまず，指導教官である富田眞治教授，中島康彦助教授，森真一郎助教授，五島正裕助手，嶋田創助手に感謝の念を申し上げたいと思っています。私は，三年前に私費留学生として日本に留学してきて，研究生として富田研究室に所属して，修士課程在籍の二年間を含めた三年間にわたって本当に様々な場面でご指導，ご鞭撻を賜りました。

さらに，日頃暖かく御鞭撻下さった富田研究室の諸兄に心より感謝いたします。

# 参考文献


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

[2] Wang, K. and Franklin, M.: Highly Accurate Data Value Prediction UsingHybrid 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] Sodani, A. and Sohi, G.S.: Dynamic Instruction Reuse, Proc. 24th ISCA, pp.194-205（1997）.

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

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

[9] Kevin B. Theobald, Guang R. Gao and Laurie J. Hendren: Speculative Execution and Branch Prediction on Parallel Machines,ICS-7/93, pp.122-124,(1993).

[10] Jayanth Gummaraju, Manoj Franklin: Branch Prediction in Multi-Thread Processors, Proceedings of the 2000 International Conference on pararrel Architectures and Compilation Techniques 0-7695-0622-4/00,IEEE,pp.32-33,(2000).

[11] Freddy Gabbay: Using Value Prediction to Increase the Power of Speculative Execution Hardware, ACM,pp.47-48,(1998).

[12] Jose Gonzalez, Antonio Gonzalez: Speculative Execution via Address Prediction and Data Prefetching, ICS 97,pp.197-198,(1997).

[13] Taku Ohsawa, Masamichi Takagi, Shoji Kawahara, Satoshi Matsushita: Speculative Multi-Threading Architecture Exploiting Parallelism over a wide range of Granularities,IEEE,pp.82-83,(2005).

[14] Mohamed Zahran, Manoj Franklin: Dynamic Thread Resizing for speculative MultiThreaded Processor, Proceeding of the 21st International Conference on Computer Design,IEEE,pp.313-314,(2003).

[15] Mauricio J. Serrano: Performance Tradeoffs in Multistreamed Superscalar Architectures, Ph.D. Dissertation, University of California, Santa Barbara, pp.195-196,(1994).

[16] A. Sodani, G. Sohi: Understanding the Difference between Value Prediction and Instruction Reuse, Proc. of the 31st Annual International Symp. on Microarchitecture, pp.205 -215, Dec. (1998).

[17] D. Connors and W.M. Hwu.: Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. Proc. of the 32nd Annual International Symp. on Microarchitecture,pp.158-159,Nov. (1999).

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

[19] HAL COmputer System/Fujitsu:SPARC64-III User's Guide,(1998)

[20] 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).

[21] 津邑公暁, 中島康彦, 五島正裕, 森真一郎, 富田真治: 並列事前実行機構における主記憶値テストの高速化, 情報処理会論文誌：コンピューティングシステム Vol.45 No.SIG 1(ACS 4),pp.31-32,(2004).

[22] 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).