修士論文

区間再利用および事前実行のための効率の良いハードウェア構成法

指導教官　　富田 眞治 教授

京都大学大学院情報学研究科
修士課程通信情報システム専攻

尼嵜 央典

平成 15 年 2 月 7 日

区間再利用および事前実行のための効率の良いハードウェア構成法

尼嵜 央典

内容梗概

プログラムには，値の局所性が存在することが明らかになってきている．値の局所性を利用し，データ依存を緩和する手法として，投機的実行および区間再利用が提案されている．値予測に基づく投機的実行とは，過去の実行結果をもとに今後の実行結果を予測し，予測値を入力とする後続命令を投機的に実行することにより，後続命令の待ち時間を短縮する手法である．ただし，予測を誤った場合には，投機的な実行結果をすべて無効化し，再実行しなければならないという欠点がある．

一方，区間再利用とは，一連の命令列において，過去に出現した命令列の同一入力による実行の際に，記憶しておいた過去の実行結果を再利用して，命令列の実行自体を省略する高速化手法であり，実行結果の無効化といったペナルティが生じない．

本論文では，既存ロードモジュールに対し，専用命令を追加することなく，動的に関数およびループ構造を識別し，これらが多重構造を形成するような命令区間を再利用する機構を提案する．また，将来再利用すると考えられる命令区間を予測し，あらかじめ別のプロセッサに実行させて登録しておく並列事前実行機構を加えた．さらに，再利用機構を実現する際に問題となる，大規模な再利用表の構成に関して，ハードウェアの利用効率を高め，より少ない再利用表を用いて，同等の性能を得る工夫を施した．

以上のような再利用機構を有する SPARC サイクルシミュレータを開発し，Stanford および SPEC を用いて命令レイテンシ，キャッシュミス，ウィンドウミス，および，区間再利用のオーバヘッドに関する性能評価を行った．この結果，プログラムにより効果が異なるものの，Stanford ベンチマークでは最大 75%，SPEC95 ベンチマークでは最大 45%のサイクル数を削減できることが分かった．

さらに，ハードウェアの利用効率を高めるために関数管理補助表を導入した結果，再利用表の構成によって効果が異なるものの，同等の性能を維持しつつ，ハードウェア量を削減できることが分かった．

An Effective Hardware Architecture for Region Reuse and Precomputation

Hisanori AMASAKI

Abstract

Recent studies have demonstrated that value locality exists in many programs. Many researches on value speculation and region reuse are reported for absorbing data dependency by exploiting value locality. Value speculation is a technique that speculatively executes the following instructions with predicted input values, and eliminates the waiting time for the following instructions. Though, when the predicted values are revealed to be incorrect, the results should be squashed and the speculatively executed instructions should be re-executed.

On the other hand, region reuse is a speedup technique that memorizes the input and the output while executing a series of instructions, and reuses the result when the instructions with the same input reappear in a program.

I propose a hardware mechanism that dynamically picks up a region such as function and loop and that reuses the region, without adding special purpose instructions for the existing load modules. Furthermore, I add a mechanism that predicts the instruction region which must be reused in future, and memorizes the result while executing by another processors in advance.

I also propose the mechanism that enables the reuse of not only a simple region but also nested structure of functions and loops. Moreover, I achieve the same performance with less amounts of reuse table by increasing efficiency by introducing an additional small hardware.

I developed a cycle simulator considering instruction latency, cache miss penalty, window miss penalty and reuse latency, and evaluated the performance of the reuse mechanism with Stanford-integer and SPEC. I found that the maximum ratio of eliminated cycle reached to 75% in Stanford-integer programs, and 45% in SPEC95 programs respectively.

Moreover, though the effectiveness is different among the programs, I found the additional small hardware can maintain the original performance and can reduce the amount of hardware for large reuse table.

区間再利用および事前実行のための効率の良いハードウェア構成法

目次

第 1 章　はじめに　1

第 2 章　従来の投機的実行と区間再利用　3

2.1　アドレスに関する投機的実行 3

2.2　値に関する投機的実行 3

2.3　区間再利用 5

第 3 章　SPARC ABI の概要　8

3.1　レジスタウィンドウ 8

3.2　スタック 10

3.3　関数実行の流れ 11

第 4 章　提案する区間再利用機構の概要　12

4.1　SPARC ABI に基づく関数再利用 12

4.2　関数再利用機構 15

4.3　ループへの適用 20

4.4　並列事前実行機構の追加 23

第 5 章　再利用表縮小のための提案　26

5.1　RS の導入 27

5.2　RS の動作 27

5.3　RS エントリの置き換えアルゴリズム 28

第 6 章　評価　31

6.1　ハードウェアの仮定 31

6.2　Stanford Integer による評価 31

6.3　SPEC95 による評価 35

第 7 章　おわりに　43

謝辞　44

参考文献　45

第 1 章　はじめに

命令列実行の高速化を妨げる要因には，一般的に，制御依存とデータ依存がある．制御依存は，分岐予測および条件付き実行により緩和できる．一方，データ依存は，フロー依存，逆依存，および出力依存の 3 種類に分類できる．このうち，逆依存と出力依存については，レジスタリネーミングなどにより緩和することができるものの，フロー依存の緩和には，プログラム内に存在する「値の局所性」を利用した，値予測に基づく投機的実行 [2, 3, 4, 5] や，区間再利用 [6, 7, 8, 9, 10, 11, 12] といった技術を導入する必要がある．

さて，値予測に基づく投機的実行とは，過去の実行結果をもとに今後の実行結果を予測し， 予測値を入力とする後続命令を投機的に実行することにより，後続命令の待ち時間を短縮する手法である．先行命令の終了時に，実際の実行結果と予測された入力を用いて投機的に実行した結果を比較し，値が一致していれば，そのまま後続命令列の実行を続ける．さらに，複数のプロセッサを投入して，複数の値予測を同時に行う研究も報告されている．しかし，値予測に基づく投機的実行には，以下のような問題点がある．

- 命令間に多くの依存関係があると，それだけ多くの投機的実行が必要となる．
- 予測を誤った場合には，投機的に実行した結果を無効化して再実行しなければならないので，無効化処理を行うことによるペナルティが生じる．
- 前項と同じ理由により，ハードウェアコストが増大する．

これに対し，区間再利用とは，一連の命令列において，過去に出現した命令列の同一入力による実行の際に，再利用表に記憶しておいた過去の実行結果を再利用して，命令列の実 行を省略する高速化手法である．本手法の特長を挙げる．

- 命令間の依存関係の頻度は，再利用機構の複雑さに影響しない．
- 値予測に基づく投機的実行とは違い，失敗時にペナルティが発生しない．
- ハードウェアコストを決めるのは，入力値および出力値の総数のみであり，省略可能な命令数が制限されない．

現在提案されている区間再利用の実現方法は，ハードウェアのみによるもの，コンパイラ単独によるもの，ハードウェアとコンパイラが協調するもの，の 3 種に分類できる．再利用の単位としては，単一命令や，基本ブロックが一般的である．

さて，本稿では，既存ロードモジュールに対して，専用命令を追加すること

なく，動的に関数やループといった命令区間を識別し，再利用する機構を提案する．また，単純な命令区間だけではなく，各命令区間が入れ子になっているような複雑なプログラム構造についても再利用を可能とする機構を提案する．さらに，再利用機構を実現する際に問題となる，大規模な再利用表の構成に関して，ハードウェアの利用効率を高め，より少ない再利用表を用いて，同等の性能を得る工夫について詳述する．なお，本稿では，SPARC アーキテクチャへの応用を仮定している．プログラムが SPARC ABI に従うことを仮定し，プログラムにおける関数やループの構造を動的に把握することを狙う．

第 2 章 では，従来の投機的実行および区間再利用の技術について述べる．第 3 章 では，SPARC ABI の概要を述べる．第 4 章 では，本論文が提案する再利用機構の構成および動作について述べる．第 5 章 では，特に，ハードウェア量が問題となる再利用表に着目し，性能を落とすことなく，再利用表のサイズを大幅に縮小する手法について述べる．第 6 章 では，サイクルシミュレータを用いた定量的評価を行い，考察を加える．

第 2 章　従来の投機的実行と区間再利用

本章では，これまでに行われている，投機的実行および区間再利用による高速化技術に関連する研究について述べる．まず投機的実行は，アドレスに関するものと値に関するものに大きく分けることができる．これらについて簡潔に説明する．さらに，投機的実行と対照的な技術である区間再利用について述べる．最後に，関連研究の評価結果を例に挙げながら，投機的実行と区間再利用の比較を行う．

2.1　アドレスに関する投機的実行

アドレスに関する投機的実行は，分岐予測とオペランドアドレス予測に分類できる．さらに，分岐予測は大きく，静的分岐予測と動的分岐予測に分けることができる．

静的分岐予測　コンパイラがプログラムを解析して分岐予測を行い，命令中に予測した情報を埋め込む方法である．プログラム実行時に動的な分岐予測は行わず，コンパイラの指示に従う．

動的分岐予測　コンパイラは分岐予測を行わず，プログラム実行時に過去の分岐履歴を用いて分岐先を予測する方法である．命令分岐の方向には偏りがあり，過去 2 回程度の分岐履歴情報を用いて十分に予測可能であると言われている [20]．単純なハードウェア機構により実現でき，大きな効果が得られるため，Intel 社の Pentium4 や ARM 社の ARM11 など多くの商用プロセッサが採用している．

一方，オペランドアドレス予測の代表的なものとして，Next Line Prefetch がある．あるキャッシュラインが連続して参照されると，近い将来次のキャッシュラインを参照すると予測し，あらかじめ主記憶からプリフェッチしておく方法である．

2.2　値に関する投機的実行

近年，多数研究が行われているのは，アドレスではなくデータ値に基づく投機的実行に関するものである．具体的には，過去の履歴に基づいて先行命令の結果を予測し，予測値を入力として後続命令列を投機的に実行する．予測の単位は，命令レベル [3, 4, 14, 15] とスレッドレベル [16, 17, 18, 19] に大別できる．

命令レベルの予測方法には，次のものがある．

Last-value 予測　同じ命令アドレスにおける前回の演算結果をそのまま使う方法である．さらに，動的に予測可能かどうかを判断する CT (Classification Table) を用いて，予測ミスを減らす機構が提案されている [3]．

Stride-based 予測　最近の 2 回の演算結果の差分を $Stride$，最近の演算結果を $Base$ とした場合に，次の予測値を $Stride + Base$ とする方法である [4]．

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

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

ハイブリッド予測　Last-value 予測と Stride-based 予測，Stride-based 予測と two-level 予測を組み合わせた方法である [4]．

また，スレッドレベルの予測方法には，以下のものがある．

制御予測　複数のスレッドから次に実行するスレッドを選択するために用いる．また，各スレッドの分岐先を予測するためにも用いられる．

データ依存予測　異なるスレッド間のデータ依存関係を予測するために用いられる．

単一のプログラムを複数のスレッドに分割して，並列に実行する様子を図 1 に示す．プログラムは，A, B, C, D の 4 つのスレッドに分割されているとする．A と B，A と C，B と D，C と D にはそれぞれ制御依存関係がある．また，A と D には x を介したデータ依存関係がある．最初に実行するスレッドとして A がプロセッサに割り当てられる．次に実行されるスレッドは制御予測により B と判断され，B は別のプロセッサに割り当てられる．さらに，次に実行されるスレッドは D と予測される．ただし，A と D には x を介したデータ依存関係がある．命令レベルの値予測やトレースレベルの値予測を用いて x の値を予測し，A と D の並列実行を行う．

以上のような値予測に基づく投機的実行は，データ依存を軽減する [3]．したがって，データ依存が問題となるスーパスカラや VLIW と組み合わせることにより，大きな効果を発揮する [5]．ただし，予測が正しければ後続命令が先行命

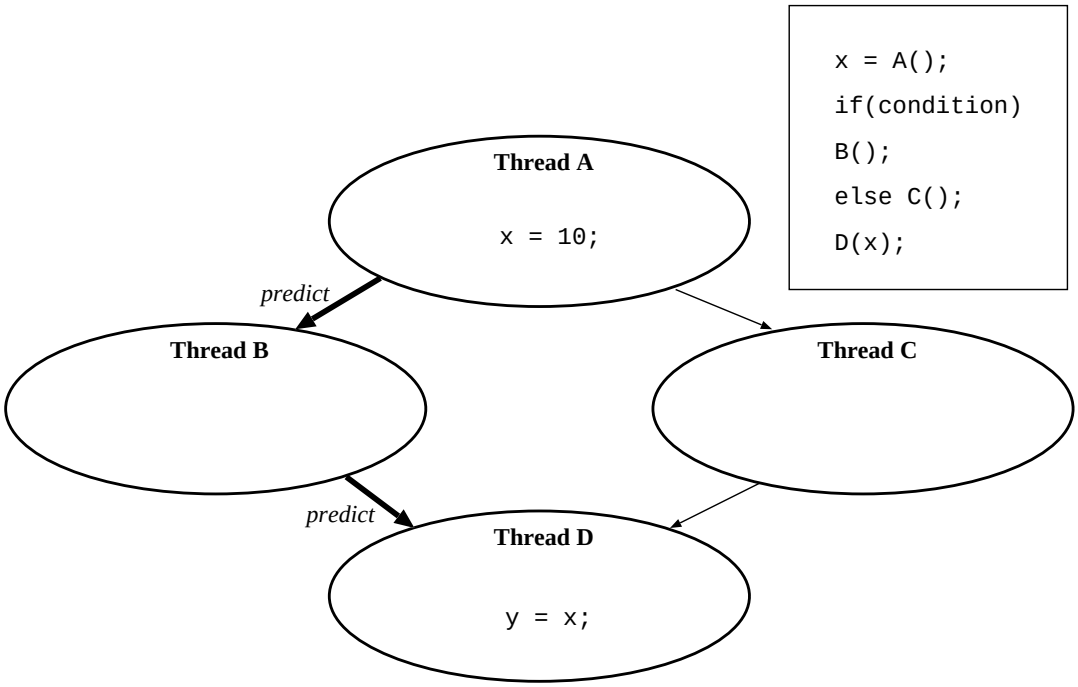


図 1: スレッドレベルでの値予測実行例

令を待つ時間が短縮される一方で，誤っていると，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，投機的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．

2.3 区間再利用

区間再利用とは，一度実行した命令列の入力と出力を記憶し，再び同じ入力を用いて同じ命令列を実行する際に，記憶しておいた出力値を用いて命令実行を省略する高速化手法である．値予測とは異なり，投機的な手法ではないため，予測の失敗による無効化を必要としないという特長がある．

現在提案されている実現方法は，以下の 3 つに分類できる．

ハードウェアのみによるもの　主に命令単位の再利用に用いられる．

コンパイラが既存の命令を用いて区間再利用を行うオブジェクトを生成するもの
　主記憶上に再利用表をもつ必要があるため，再利用表へのアクセスが遅く，再利用に時間がかかるという欠点がある．

専用命令を追加し，ハードウェアとコンパイラが協調するもの　一般的に，プ

ロセッサが動的かつ効率良く基本ブロックを切り出すことは難しく，簡単化するためには，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，既存のロードモジュールを高速化できない欠点がある．

また，区間再利用の単位によって，以下の 2 つに分類できる．

単一命令を単位とする方法　各命令のオペランドと演算結果，ロード/ストア命令の場合はさらにオペランドアドレスを再利用表に格納しておく．命令デコード時に入力値と再利用表の入力値との比較を行い，再利用可能かどうかの判定を行い，再利用可能であれば再利用表の出力値の内容に従って，レジスタや主記憶へ結果を書き込む．この手法により，演算や主記憶読み出しを削減し高速化を図る．ただし，一度の再利用につき数サイクル程度の高速化にとどまり，大幅な高速化は期待できない．全ての命令を対象にしたもの [8] やロード/ストア命令のみを対象にしたもの [6] が研究されている．

命令区間を単位とする方法　命令列の入力値（レジスタの値および主記憶から読み出した値）および出力値（レジスタの値および主記憶へ書き込んだ値）を対にして再利用表に格納しておく．同じ命令列を再度実行する際に，入力値が全て一致しているかどうかを比較し，一致した場合に，再利用表に格納しておいた出力値に従ってレジスタおよび主記憶への書き込みを行う．単一命令を単位とする方法に比べて，一度に複数の命令の実行を省略できるという利点がある．

投機的実行と区間再利用を比較すると，区間再利用は適用範囲が狭いものの，投機的実行における予測失敗時のペナルティを考慮した場合，区間再利用のほうが有利であることが報告されている．引数，局所変数および大域変数の区別が容易である Java 仮想マシンおよび SpecJVM98 を用いた研究 [12] では，Last Value Prediction による投機的実行の場合で 3.8%から 29.1%（平均 17.0%），メソッド単位の区間再利用の場合で 1.1%から 47.0%（平均 16.7%）のサイクル数を削減できること，区間再利用の効果はプログラムによる偏りが大きいこと，予測失敗を考慮するとやはり区間再利用のほうが有利であることが報告されている．

また，投機的実行と区間再利用を組み合わせた方法としては，コンパイラが再利用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を予測して後続区間の実行を投機的に開始する研究が報告されている [13]．

第 3 章　SPARC ABI の概要

本章では，SPARC ABI (Application Binary Interface) [1] の概要について説明し，SPARC アーキテクチャにおいて関数呼び出しがどのように行われるかについて述べる．また，どのようにして再利用に必要な情報を収集するかについて述べる．

3.1　レジスタウィンドウ

プログラムは常に計 32 個の汎用レジスタを使うことができる．汎用レジスタは大きく分けて以下の 4 つに分類できる．

global レジスタ (%g0 ~ %g7) どの変数からも常にアクセスできるレジスタであり，大域変数の格納場所として用いられる．%g0 の値は常に 0 である．

out レジスタ (%o0 ~ %o7) %o0 ~ %o5 は，引数を渡すため，または作業用に使われる．特に%o0 は返り値を受け取るためにも用いられる．%o6 は 3.2 において詳述するスタックポインタとして用いられる．%o7 は関数呼び出しの際に，call 命令もしくは jmpl 命令のアドレスを保持する．

local レジスタ (%l0 ~ %l7) 局所変数を格納するため，または作業用に使われる．

in レジスタ (%i0 ~ %i7) %i0 ~ %i5 は関数が引数を受け取る際に用いられる．特に%i0 および%i1 は返り値の格納にも用いられる．%i6 は 3.2 において詳述するフレームポインタとして用いられる．%i7 は関数呼び出しの際に，関数を呼び出した call 命令のアドレスを保持する．

さて，レジスタウィンドウとは，SPARC アーキテクチャにおいて，関数呼び出し時のパラメータ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ものある．図 2 に概要を示す．各ウィンドウは上記の$\%\mathrm{o}_n$，$\%\mathrm{l}_n$，$\%\mathrm{i}_n$ ($0 \leq n \leq 7$) から構成され，$\%\mathrm{o}_n$は隣接するウィンドウの$\%\mathrm{i}_n$と同一，また，$\%\mathrm{i}_n$は反対側に隣接するウィンドウの$\%\mathrm{o}_n$と同一のレジスタとなっている．それに対して，$\%\mathrm{l}_n$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することができる．CWP の値は save 命令によってインクリメントされ，restore 命令によってデクリメントされる．save 命令と restore 命令は，一般に関数呼び出し時と関数終了時に実行される．CWP の値がインクリメントされ

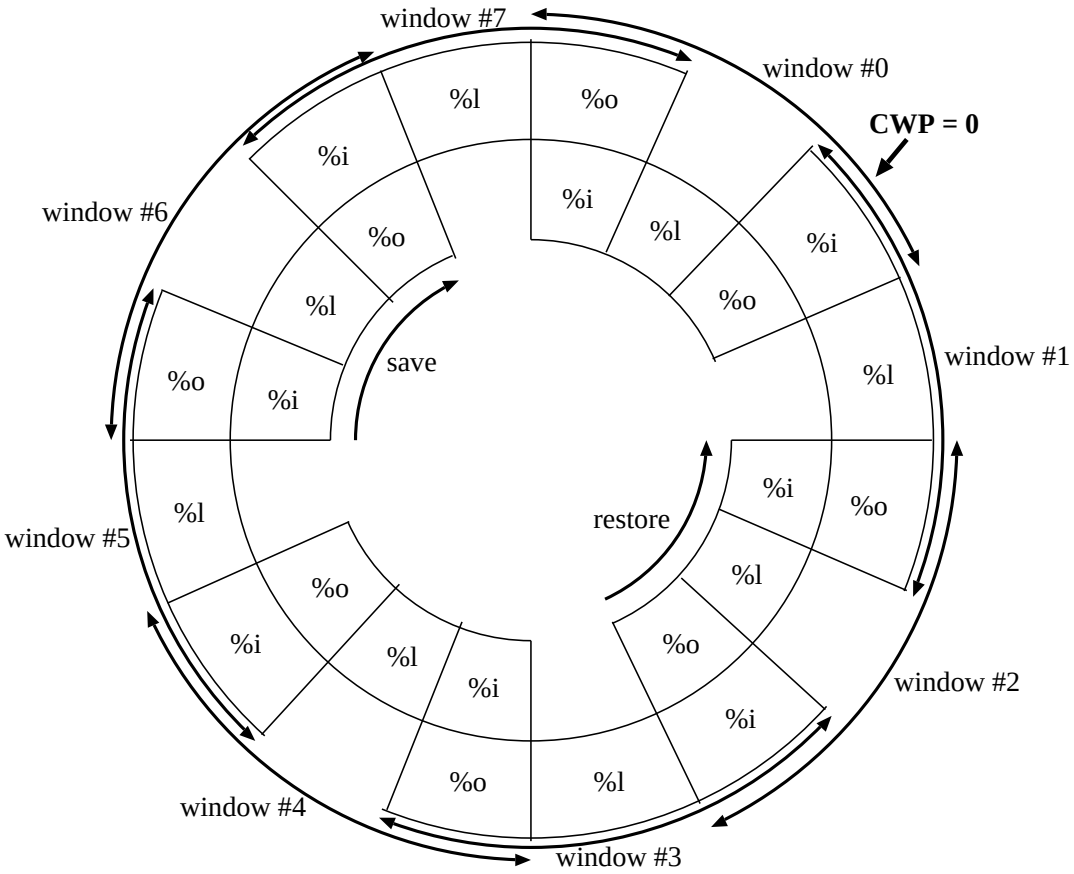


図 2: レジスタウィンドウ

るとウィンドウが 1 つ進められ，以前に%o として参照していたレジスタが%i となり，以前に%l および%i として参照していたレジスタは使用できなくなる．その代わりに，新しく%l および%o が割り当てられる．逆に CWP の値がデクリメントされるとウィンドウが 1 つ戻り，以前に%i として参照していたレジスタが%o となり，以前に%l および%o として参照していたレジスタは使用できなくなる．その代わりに，新しく%l および%i が割り当てられる．

save 命令により割り当てられたレジスタの内容は，restore 命令を実行するまでは保存される．しかし，save 命令が続くと，レジスタウィンドウの容量を超過し，新たなレジスタを割り当てることができなくなる．この場合にはウィンドウオーバフロー割り込みが発生し，レジスタの内容が 3.2 において詳述する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．第 4 章 で述べる再利用機構は，入れ子関数をまとめて再利用することにより，関数呼び出しを削減し，ウィンドウオーバフローやウィンドウアンダフローによる性能の低下を抑制することを狙った構成となっている．

3.2 スタック

SPARC におけるスタックは，以下の用途に使われる．

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

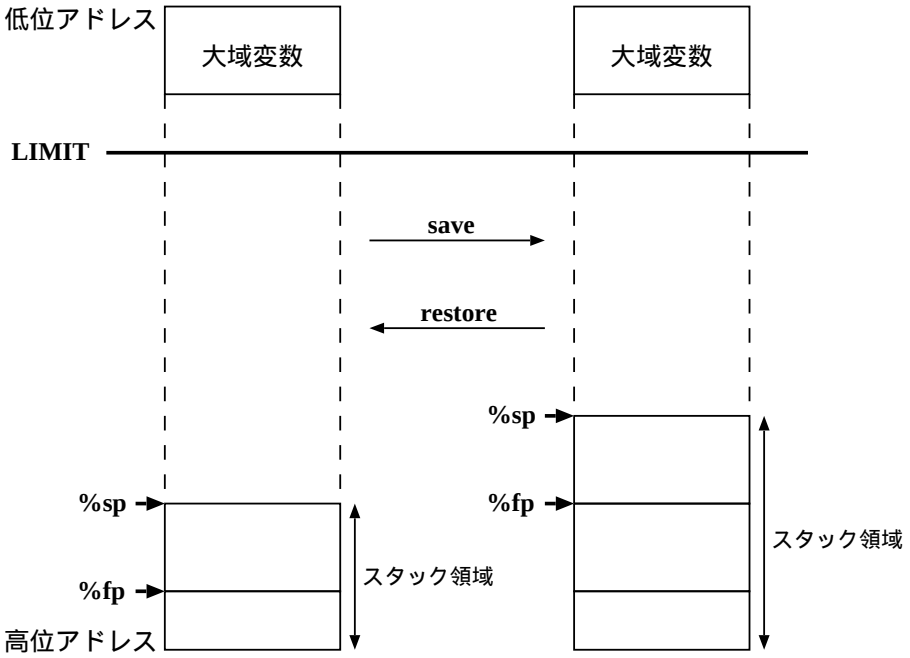


図 3: スタック

スタックは，主記憶の高位アドレスから低位アドレスに向けて伸びていく．現在有効なスタックの下限アドレスが，スタックポインタと呼ばれるレジスタ%o6 (%sp) に格納されており，図 3 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%o6

(%sp) を減少し，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 命令の場合はその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 になることはなく，LIMIT 以上%sp 未満の領域は無効データ領域である．

3.3 関数実行の流れ

関数は，call 命令，または，現在の PC を%o7 に書き込む jmpl 命令により呼び出される．現在のプログラムカウンタ (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 7 ワード以上ある場合には，前述のようにスタックに格納する．この場合，第 7 ワードは%sp+92 に，第 8 ワードは%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 call 命令自身のアドレスが見える．

非 Leaf 関数は，第 1 オペランドが%i7 $+\alpha(\alpha > 0)$，第 2 オペランドが%g0 である jmpl 命令，およびこれに続く restore 命令により終了する．このとき，%i7 には関数呼出し時の PC の値が格納されている．返り値は 1 ワードの場合%i0 に，2 ワードの場合は%i0 および%i1 に格納される．さらに restore 命令によって返り値は，%o0 および%o1 に見えるようになる．

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$+\alpha(\alpha > 0)$ 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．

第 4 章　提案する区間再利用機構の概要

本章では，まず，本論文が提案する関数再利用機構の構成と動作を説明する．次に，この関数再利用機構を拡張することによって，ループを関数と同様に再利用できることを説明する．また，単に過去の演算結果を再利用するだけでなく，その他の複数のプロセッサが入力データの予測に基づいて事前に命令区間を実行することにより高速化を図る，並列事前実行機構について説明する．

4.1　SPARC ABI に基づく関数再利用

本節では，実際に SPARC ABI を利用してどのようにして関数再利用を行うかについて説明する．具体例として，関数 A が関数 B を呼び出し，さらに関数 B が関数 C を呼び出す場合について考える．なお，関数 C は leaf 関数であるとする．図 4 に，この場合の引数および関数フレームを示す．

(a) は関数 A を実行中かつ下位関数 B の実行を開始していない状態である．LIMIT 未満の太枠部分に命令列および大域変数，また，%sp 以上の部分に有効な値が格納されている．%sp+64 から 1 ワードの領域には，関数 B が構造体を返り値とする場合の暗黙的引数として，構造体の先頭アドレスが格納される．関数 B に対する明示的な引数は，先頭の 6 ワードがレジスタ%o0 ~ %o5 に，第 7 ワード以降は%sp+92 以上に格納される．ベースレジスタを%sp とするオペランド%sp+92 が使用された場合，この領域は関数 B への第 7 引数，すなわち関数 B の局所変数である．一方，オペランド%sp+92 が出現しない場合，この領域は関数 A の局所変数である．このように，(a) の時点では，関数 A の局所変数と関数 B の局所変数を区別することができる．また，関数 A の入力データは，引数および大域変数である．

(b) は関数 B を実行中かつ下位関数 C の実行を開始していない状態である．関数 A と同様に引数および大域変数を入力とする．また，ポインタを通じて他の大域変数や関数 A の局所変数も入力となる可能性がある．save 命令が実行されると，%sp の値を%fp に格納し，必要な分だけ%sp の値を低位アドレスに移動する．関数 B の局所変数には，引数の先頭 6 ワードのアドレスを扱うために必ず確保される%fp+68 ~ 91，および，第 7 ワード以降が存在する場合に確保される%fp+92 以上の領域が含まれる．ただし，%fp 相対アドレスによって参照されるとは限らないため，一般に，%fp+92 以上の領域は，関数 A の局所変

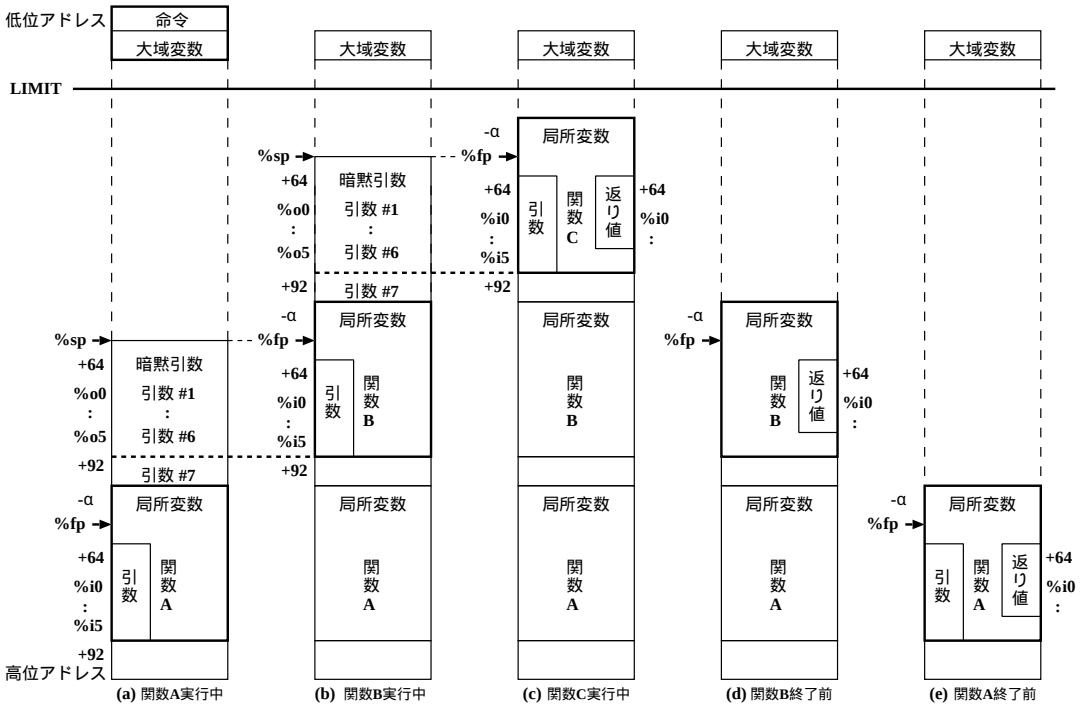


図 4: 引数およびフレームの概略

数か関数 B の局所変数かの区別ができない．局所変数を区別するためには，(a) の時点において引数の第 7 ワード以降を検出した関数呼び出しは再利用の対象外とし，検出しない関数呼び出しは，直前に%sp+92 の値を記録しておく必要がある．第 7 ワード以降を使用する関数呼び出しの出現頻度が低いと予想されることから，この制限による性能の低下は軽微なものと考える．以上の準備により，(b) における主記憶参照アドレスが，あらかじめ記録しておいた%sp+92 の値以上の場合は関数 A の局所変数，%sp+92 の値未満の場合は関数 B の局所変数であることが分かる．関数 B 実行時には，関数 B の局所変数を除外しながら，大域変数および関数 A の局所変数を再利用表へ登録する．

さて，再利用の際は，関数 B の局所変数は入出力から除外されるため，関数 B の局所変数のアドレスが一致している必要がない．このため，いかなるコンテクストであっても，入力さえ一致すれば，再利用することが可能である．ただし，関数 B が参照する大域変数や関数 A の局所変数については，アドレスおよびデータの両方が再利用表の内容と完全に一致する必要がある．関数 B を実行する前に，どのようにして，比較すべき主記憶アドレスを網羅するかが鍵になる．関数 B が参照する，大域変数や関数 A の局所変数のアドレスは，そもそ

も，関数 B において生成されるアドレス定数や，大域変数/引数を起源とするポインタに基づくため，まず引数が完全に一致する再利用表中のエントリを選択した後に，関連する主記憶アドレスをすべて参照して一致比較を行うことにより，関数 B が参照すべき主記憶アドレスを網羅することができる．すべての入力が一致した場合にのみ，登録済みの出力 (帰り値，大域変数，および関数 A の局所変数) を再利用することができる．

関数 C は，leaf 関数であるため，%sp および%fp は変化しない．関数 C の入力は，大域変数，および，レジスタ%i0 ~ %i5 (save 命令を含まない場合は，%o0 ~ %o5)，同様にアドレス%fp + α (save 命令を含まない場合は，%sp + α，$\alpha \geq 0$) に格納されている引数である．一方，関数 C の出力は，大域変数，および，レジスタ%i0 ~ %i1 (save 命令を含まない場合は，%o0 ~ %o1) に格納される返り値である．なお，返り値が浮動小数点数の場合は%f0 ~ %f1，構造体の場合は%fp+64 (save 命令を含まない場合は%sp+64) に格納されている構造体の先頭アドレスへ書き込まれる．ポインタを通じて，大域変数および上位関数 A または B の局所変数も入力となる可能性がある．(d) および (e) は同様に関数 B および関数 A の終了直前の状況を表している．

さて，(b) において，関数 C に対する入力が既知であり，対応する出力が計算済みであれば，再利用によって関数 C の実行を省略できる．ただし，この時点では関数 C の局所変数が場所として存在しないため，再利用の際に関数 C の局所変数を参照してはならない．関数 C の実行時に関数 C の入出力として登録すべきフレーム上のデータは，上位関数 A および B の局所変数である．この区別には，先ほど述べたように，引数の第 7 ワード以降を扱わない場合，関数 A および B の局所変数が%sp+92 以上，関数 C の局所変数が%sp+92 未満にそれぞれ対応することを利用する．

これと同様に，(a) において，関数 B に対する入力が既知であれば，(b) (c) および (d) を一度に省略することができる．関数 B の実行時に関数 B の入出力として登録すべきフレーム上のデータは，上位関数 A の局所変数であり，途中の関数 C の実行時に関数 B の入出力として登録すべきフレーム上のデータもまた，関数 A の局所変数である．以上のように，関数 C の実行中であっても，どの関数の入出力を登録しているかによって，関数 B の局所変数が入出力に含まれるか否かが異なる．この区別には，関数 B の局所変数が，関数 A 実行時は%sp+92 未満，関数 B 実行時は%sp+92 以上であることを利用する．つまり，

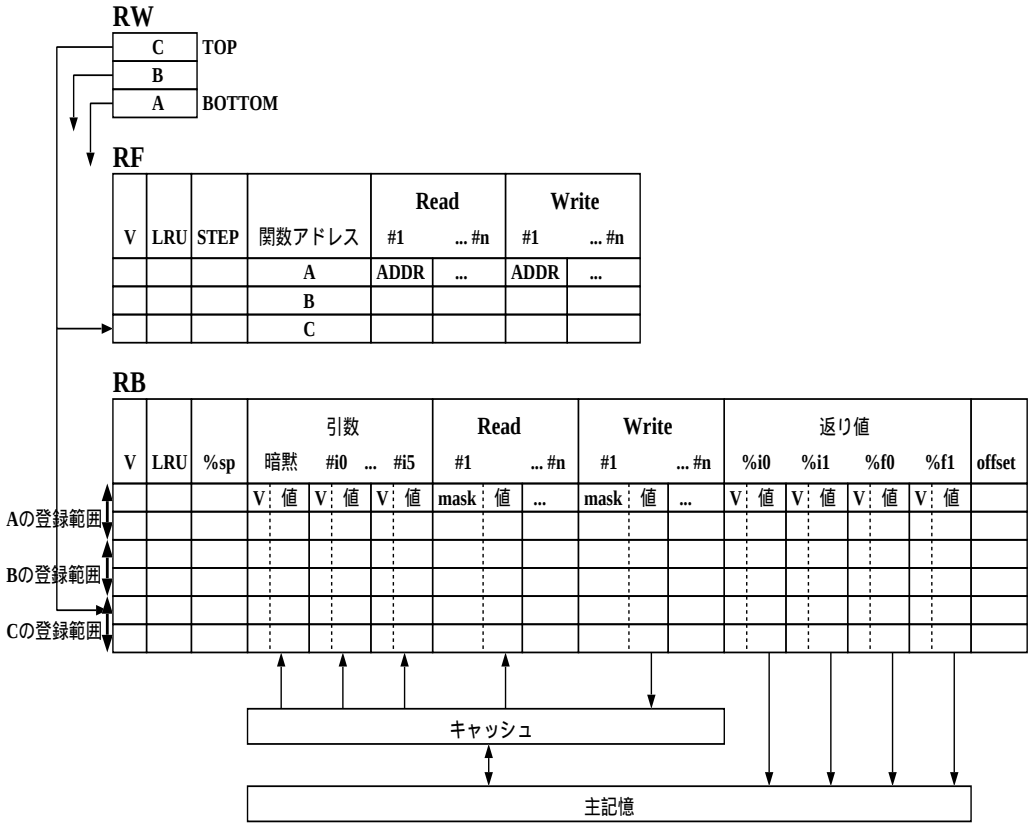


図 5: 従来の再利用表の論理構成

登録開始時の%sp を記憶することで登録すべき入出力を特定でき，複数レベルの登録を同時に行うことができる．

4.2 関数再利用機構

再利用を実現するための再利用表全体の論理構成を図 5 に示す．再利用表は，再利用ウィンドウ (RW)，関数管理表 (RF)，本体 (RB) から構成される．

再利用ウィンドウ (RW) は，現在実行中かつ登録中である関数呼び出しの入れ子関係を表現しており，各エントリは各々の関数呼び出しに対応する RF および RB の各エントリのインデクスを保持している．

関数管理表 (RF) の各エントリは，互いに異なる関数に対応しており，以下の情報を保持する．

V 有効エントリを表示する．

LRU LRU カウンタ．後述する RF エントリの置き換えのために用いられる．

STEP 該当する関数を再利用することにより削減できるステップ数を示す．こ

の値は，登録開始時に 0 に初期化され，登録完了時に更新される．

関数アドレス　関数の先頭アドレス．

Read　主記憶読み出しアドレス．各アドレスは，有効であるか否かを示すフラグ (V) を保持している．

Write　主記憶書き込みアドレス．Read と同様，各アドレスは，有効であるか否かを示すフラグ (V) を保持している．

再利用表本体 (RB) は，RF の各エントリに対応する複数エントリがブロック化されており，関数の並び順は RF と同一である．RB は以下の情報を保持する．

V　各エントリが「登録可能」「登録中」「登録済み」のどの状態にあるかを表す．

LRU　LRU カウンタ．RF と同様，エントリの置き換えのために用いる．

%sp　関数呼び出し時の %sp の値．4.1 でも述べたように，関数の局所変数と上位関数の局所変数を区別するために用いる．

引数　有効エントリを示すフラグ (V) および入力値．

Read　RF の各 Read アドレス 4 バイトの有効バイトを示すマスク (mask) および入力値．

Write　RF の各 Read アドレス 4 バイトの有効バイトを示すマスク (mask) および出力値．

返り値　汎用レジスタまたは浮動小数点レジスタ (%f0 ~ %f1) に格納される出力値および有効エントリを表示するフラグ (V)．

offset　関数からの復帰先アドレスのオフセット．復帰先アドレスは，関数呼び出しに用いた call 命令もしくは jmpl 命令のアドレスに，この値を加えたものとなる．なお，%f2 ~ %f3 を使用する返り値 (拡張倍精度浮動小数点数) は対象とするプログラムには存在しないものと仮定する．

Read アドレスの内容と RB の複数のエントリを一度に比較するために，Read アドレスは RF が一括管理し，RB はマスクおよび値だけを管理する方法を採用した．

次に再利用表に関する各パラメタについて説明する．

RFSIZE　RF が保持できるエントリ数の最大値．

RBSIZE　RB が 1 つの関数あたりに保持できるエントリ数の最大値．RB エントリの総数は，$RFSIZE * RBSIZE$ 個となる．

RFPURGETIMER　RF の LRU カウンタをリセットする間隔を表す定数．

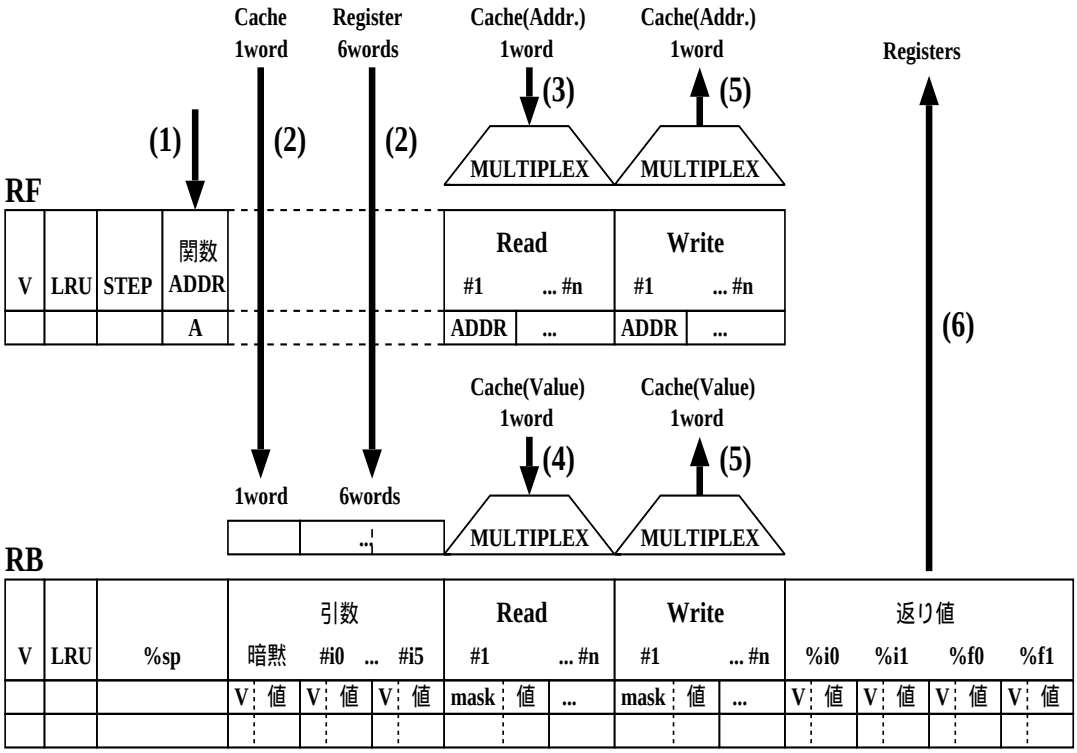


図 6: 従来の再利用表の物理構成

RF の参照回数がこの値に達するごとにカウンタがリセットされる．

RBPURGETIMER RB の LRU カウンタをリセットする間隔を表す定数．

RB の参照回数がこの値に達するごとにカウンタがリセットされる．

RBMMRMAX RB 1 エントリが保持する主記憶読み出しアドレスの最大数．

RBMMWMAX RB 1 エントリが保持する主記憶書き込みアドレスの最大数．

図 6 は，RF 1 エントリ分に対応する再利用表の物理構成を示している．引数および主記憶読み出しデータと RB の内容との比較には CAM を利用することを想定している．関数呼び出しが再利用可能か否かを判定するためには，まず，関数の先頭アドレスが一致する RF エントリ (1) を特定し，次に，引数が全て一致する RB エントリ (2) を特定する．さらに，少なくとも 1 つのマスクが有効である Read アドレス (3) を RF から順に選択し，主記憶から読み出した各 4 バイトのデータ (4) と RB の対応する列の全ての値とを一度に比較する．マスクが有効である全ての値が一致したとき，そのエントリに登録されている主記憶書き込みデータ (5) および返り値 (6) をレジスタおよび主記憶に格納する．

次に，実際に SPARC ABI を利用してどのようにして関数再利用を行うかについて説明する．以下では，関数 A が関数 B を呼び出し，さらに関数 B が関数 C を呼び出す場合について考える．

3.3 において述べたように，関数呼び出しの契機は，call 命令または現在の PC を%o7 に書き込む jmpl 命令である．RF を参照し，前述の手順により再利用を試みる．再利用した関数 (例えばこれを C とする) を含む上位関数 A および B が登録中である場合，C の再利用を行った RB エントリの主記憶参照に関する内容を，A および B の登録中 RB エントリに追加する．ただし 4.1 で述べたように，A および B の呼び出し時における%sp+92 未満に対する参照は対象外とする．

ところで，上記の方法によって入れ子の関数を登録していくと，より上位の関数 (例えば B) において登録すべき主記憶参照箇所が，RB の各エントリが収容できる上限を越える．この場合は，RB エントリの制限を越えた関数を含む上位関数 (この場合は A および B) それぞれに対応する登録中 RB エントリおよび RW エントリを無効化し，引き続き登録可能な C のみを，RB および RW に残して登録を続行する．

一方，再利用できなかった場合，その関数が RF に登録されていなければ，後述する LRU アルゴリズムに基づいて RF に無条件に新規登録する．さらに，RF の該当エントリに対応する RB のブロックに LRU アルゴリズムに基づいて新たなエントリを確保し，RW に，これから実行しようとする関数，すなわち，RF および RB の該当エントリを積む．このとき，RW に登録可能な上限値を越えた場合は，最も上位の関数に対応する RB エントリおよび RW エントリを無効化する．

関数再利用ができなかった場合は，さらに，関数本体の実行を開始する．また，以後の再利用に備えて，再利用に必要な関数の入力および出力を登録していく．登録は以下のようにして行う．

trap 命令　システムコールを含む関数は再利用できないと判断し，RW が保持している実行中の全ての関数について，RB エントリおよび RW エントリを無効化する．

レジスタ参照　レジスタ%i0 ~ %i5 (save 命令を伴わない関数では%o0 ~ %o5) には明示的引数の先頭 6 ワードが格納される．関数内においてまず読み出しを行った対象を，引数として RB に登録する．まず書き込みを行った対象

は引数ではないため比較対象外として登録する．ただし，%i0 ~ %i1 (save 命令を伴わない関数では%o0 ~ %o1) および%f0 ~ %f1 への書き込みは，返り値の可能性があるので，返り値としても RB に登録する．その他のレジスタ参照は，関数への入力から得られる中間結果であるため，登録する必要はない．

LIMIT 以上，呼び出し時%sp+64 未満　前述したように，この区間は無効領域または関数の局所変数が格納される領域なので，RB への登録は不要である．

呼び出し時%sp+64 ~ 67　暗黙的引数が格納される．関数内においてまずこの場所からの読み出しを行った場合，引数として RB に登録する．まず書き込みを行った場合は引数ではないため比較対象外として登録する．

呼び出し時%sp+68 以上　%sp+68 ~ 91 の領域は局所変数が格納されている領域である．%sp+92 以上への書き込みを検出した場合，引数の第 7 ワード以降が存在するため，現在実行している関数から復帰する前に次の関数が呼び出された場合，次の関数を含む上位関数の登録を中止する．

その他の主記憶参照　上記以外の場合，大域変数または上位関数内の局所変数であり，関数に対する入出力として登録する必要がある．RW に登録されている全ての RF エントリおよび RB エントリに対して以下の操作を行う．

- Limit 以上，各 RB の%sp+92 未満であるアドレスは無視する．
- 読み出しアドレスが Write または Read としてすでに登録されている場合，内容が上書きされたか，または，登録済みであるため，新たな登録は行わない．
- 書き込みアドレスが Write としてすでに登録されている場合，登録内容を更新する．
- 登録されていない場合，登録数の上限を越えていなければ，Read/Write に応じて登録を行う．一方，上限を越えた場合は，その関数を含む上位関数の登録を中止する．

なお，大域変数ならびにポインタを介した上位関数の局所変数を読み込む場合，最初に参照するのは，RB の Write データである．そこで読み込めなかった場合，次に参照するのは，RB の Read データである．さらにそこでも読み込めなかった場合は，キャッシュを参照する．

%g0 へ現在の PC を書き込み，%i7 (save 命令を伴わない関数では%o7) の内

容へ無条件分岐する jmpl 命令によって関数から復帰すると，RB への登録が完了する．登録が完了した RB エントリの状態を 登録中」から 登録済み」に変更し，最後に RW から該当エントリを削除する．

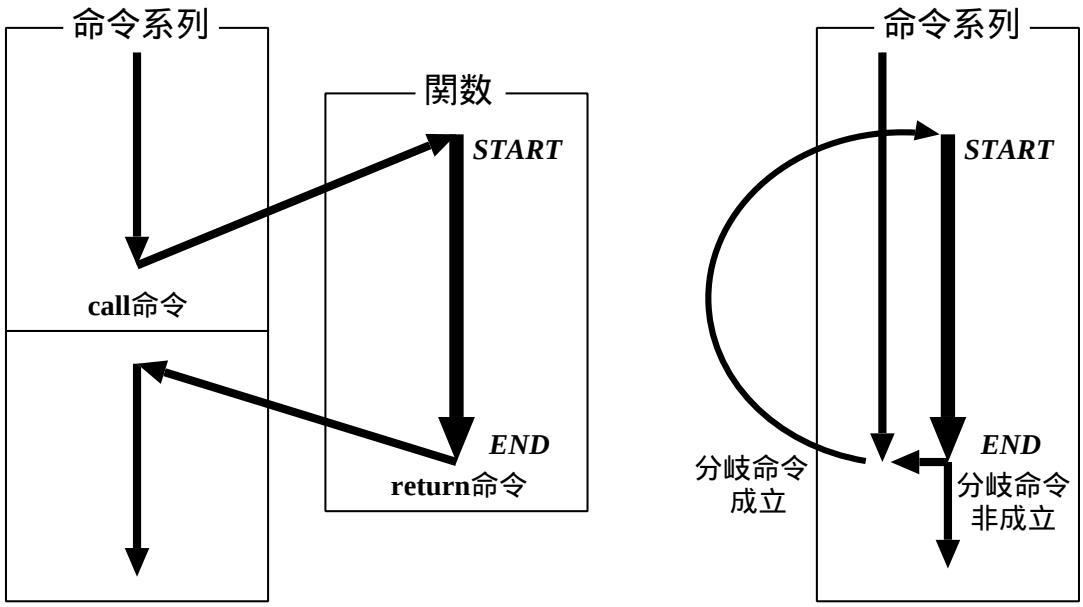
RF および RB に，再利用に必要な関数の入出力を次々に登録していくと，いずれ全ての使用可能なエントリが 登録済み」の状態になる．さらに登録するためには，既存のエントリから将来再利用される可能性の低いエントリを選択して追い出す必要が生じる．そこで，以下に述べるエントリ置き換えアルゴリズムを適用する．

全ての RF エントリには LRU カウンタを備えている．LRU カウンタの初期値は 0 であり，その RF エントリに登録されている関数に対応する RB エントリが登録完了時またはヒットした際にインクリメントされる．また，先ほど述べたように，RF エントリは，該当する関数の再利用によって削減できるステップ数を表す STEP を保持している．RB ヒット回数が少なく，かつ，STEP の値が小さいエントリが，登録されているエントリのなかで最も不要であると考えられるため，LRU カウンタの値と STEP の値との積が最小となるエントリを優先して追い出す．

同様に，全ての RB エントリにも LRU カウンタを備えている．LRU カウンタの初期値は 0 であり，その RF エントリに登録されている関数に対応する RB エントリが登録完了時またはヒットした際にインクリメントされる．RB ヒット回数が最も少ないエントリが最も不要であると考えられるため，LRU カウンタの値が最小のエントリを優先して追い出す．

4.3 ループへの適用

図 7 に，関数とループの類似性を示す．関数に含まれる命令は，call 命令の分岐先から return 命令 (厳密にはディレイスロットを含む) までである．同様に，ループに含まれる命令は，後方分岐命令の分岐先から，同じ後方分岐命令までである．このように，区間の最初と最後を示す命令の種類は異なるものの，開始と終了の場所が明確であり，特定は容易であるため，前述の関数再利用機構に対して拡張を施し，ループについても入出力を登録可能とすることにより，関数と同様にループの再利用を行うことができる．ただし，関数では局所変数の登録を除外することができるのに対して，ループではこれを除外できない．これは，ループ内の局所変数が ABI では規定されていないためである．したがっ



(a) 関数　　**(b)** ループ

図 7: 関数とループの類似性

て，ループ再利用においては，ループ内で参照した全てのレジスタおよび主記憶アドレスを入出力として再利用表に登録しておく必要がある．また，関数にはないループ特有の情報を格納するために，関数のみの再利用機構の RF および RB に対して，以下のフィールドを追加する．

F/L (**RF**) 格納されている情報が関数，ループのいずれのものであるかを区別するために用いる．

End (**RF**，**RB**) ループの終了アドレス．ループの再利用表への登録が完了した際に，RB エントリの End の値を，対応する RF エントリの End に代入する．

taken/not (**RB**) ループ終了時の分岐方向．

Regs，**CC** (**RB**) 引数や返り値以外のレジスタおよび分岐の評価を行うための条件コード．

ループが完了する前に関数から復帰したり，関数再利用と同様の擾乱が発生するなどの理由により，ループの入出力の登録が中止されなければ，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，エントリの状態を 登録中」から 登録済み」に変えて登録を完了する．後方分岐命令が成立する場合は，次のループが再利用可能かどうかを判断する．つまり，後方分岐する前に以下に

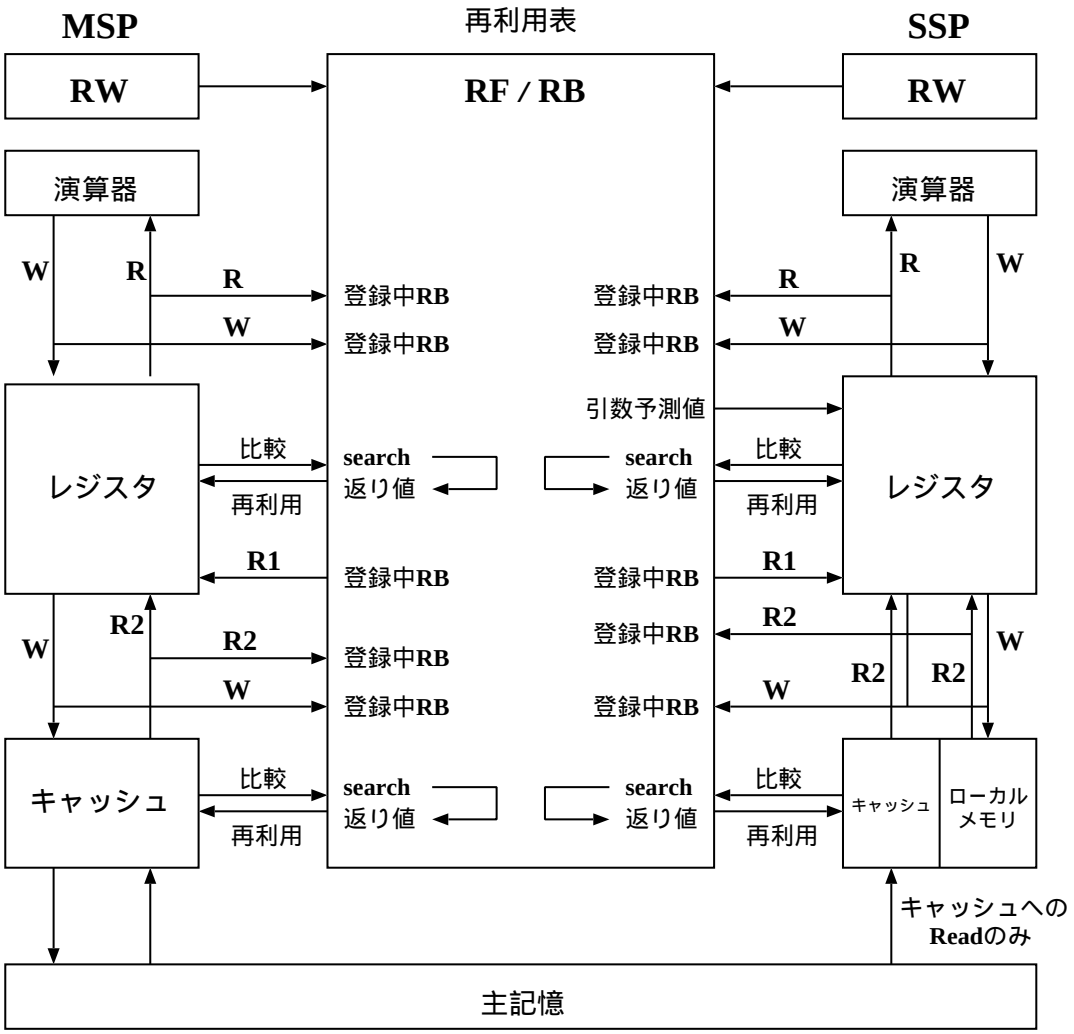


図 8: 並列事前実行機構

示す処理を行う．

(1) RF においてループの先頭アドレスを検索する．

(2) レジスタの入力値が完全に一致する RB エントリを選択する．

(3) 関連する全ての主記憶アドレスを参照し，順に比較する．

(4) 全ての入力値が一致した場合に，登録済みの出力値 (レジスタおよび主記憶書き込みデータ) を書き戻す．

再利用できた場合，RB 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．次のループが再利用できない場合は，ループの中身を実際に実行し，同時に RB への登録を開始する．

4.4 並列事前実行機構の追加

これまでに述べた，関数やループの再利用は，RB エントリの生存時間よりも同一入力データが出現する間隔が長い場合や，入力データが単調に変化する場合には，全く効果がない．そこで，通常どおり再利用を行いながら命令を逐次実行しているプロセッサ（Main Stream Processor: 以下 MSP と略する）とは別に，先行して RB エントリへ入力値および出力値を登録するプロセッサ（Shadow Stream Processor: 以下 SSP と略する）を複数個設けることにした．図 8 は，並列事前実行機構の概要を示したものである．RW，演算器，レジスタ，キャッシュは各プロセッサごとに独立しており，RF，RB，主記憶は全プロセッサが共有する．また，SSP には，引数を予測して関数やループを実行する際に，関数やループ内の局所変数を格納するためのスタックが必要になる．MSP の場合は，主記憶にスタック領域を確保することができるものの，SSP は，主記憶へ書き込んではならないので，このスタック領域を確保するためにローカルメモリを設ける．予測された明示的引数はレジスタに格納され，暗黙的引数はローカルメモリ内に格納する．また，上位関数内の局所変数を参照する場合は，ローカルメモリを参照する．大域変数を読み出す場合は，4.2 に示した MSP の場合の手順と同様である．また，SSP は主記憶へ書き込むことができないため，大域変数への Write データについては，主記憶ではなく，RB の Write データ登録領域へ書き込みを行う．返り値は，直接 RB の返り値へ登録する．関数またはループの実行が完了すると，MSP と同様に RB エントリの状態を「登録中」から「登録済み」に変更し，RW から該当エントリを削除した上で，新たな命令区間の実行を開始する．ただし，ローカルメモリのサイズには上限があるため，これを越えた場合には，関数またはループの事前実行を打ち切る．また，0 番地の参照など，実行を継続できない例外が発生した場合も，実行を打ち切る．なお，前述のように，事前実行の結果は主記憶に書き込まれないため，事前実行の結果を使って，さらに次の事前実行を行うことはできない．

図 8 の **R** はレジスタやキャッシュからの読み出しおよび RB への登録，**W** はレジスタやキャッシュへの書き込みおよび RB への登録をそれぞれ表している．**R1** は，すでに RB に登録されているアドレスからの読み出しは，RB からデータを直接得ることを表していて，これはキャッシュを汚さないための工夫にもなっている．**R2** は，RB にまだ登録されていないアドレスは，従来通りキャッ

シュを参照することに対応する．

さて，前述のように，MSP はレジスタや主記憶に対する参照を RB に登録しながら通常どおり命令列を実行し，可能であれば関数呼び出し時および後方分岐命令成立時に再利用を試みる．これに対して SSP は，特定の関数のみを実行し，SSP が有するローカルメモリおよび RB への読み書きは行うものの，キャッシュおよび主記憶への書き込みは一切行わない．このような違いのため，SSP のふるまいは MSP とは異なる点がいくつかある．

事前実行に際しては，RB の使用履歴に基づいて将来の入力を予測し，SSP に渡す必要がある．このために， RF の各エントリに小さなプロセッサを設け，MSP および SSP とは独立に予測値を求める．具体的には，最近の 2 回の入力値の差分を $Stride$，最近出現した入力値を $Base$ とした場合に，次の予測値を $Stride + Base$ とする Stride-based 値予測である．ただし，$Stride + Base$ に基づく命令区間の実行は，予測値が求まった時点で MSP がすでに開始しているものと考える．つまり，N台の SSP を使用する場合，用意する入力予測値は，$Stride * 2 + Base$ から $Stride * (N + 1) + Base$ の範囲となっている．

4.2 で述べたように，各 RF エントリは互いに異なる命令区間に対応しており，入力と出力の対応関係が RB に登録される．これについては，関数だけでなくループについても同様のことが言える．このとき，MSP と SSP が RB エントリをどのように使い分けるかが問題となる．命令区間は，MSP のみでも再利用の効果があるものと，配列を扱うループのように MSP による効果がないものに，大きく分けることができると考えられる．RB エントリの入れ替えは，前者であれば LRU に，後者であれば FIFO に基づいて行うのが有効である．しかし，ある命令区間がどちらの性質のものであるかを動的に判断するのは難しいため，各々の RF に対応する RB エントリを MSP 用と SSP 用に分割し，それぞれに LRU と FIFO を適用する仕組みとした．上述のように，入力予測値は N 組であり，SSP が RB への登録完了後，ただちに MSP がそのエントリの情報を利用することを想定して，SSP 用に割り当てる RB エントリ数を $N * 2$ とした．この様子を図 9 に示す．

また，SSP に事前実行させる命令区間をどのように選択するかが問題となる．同一パラメタが出現する間隔が長い命令区間や，パラメタが単調に変化し続ける命令区間に対して事前実行の効果があることが予想されるものの，命令区間の性質および実際の事前実行の効果は事前には分からない．そこで，RF に新

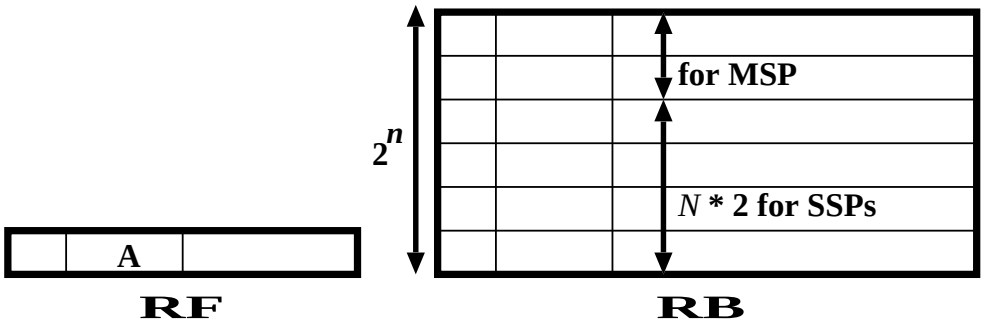


図 9: RB の分割

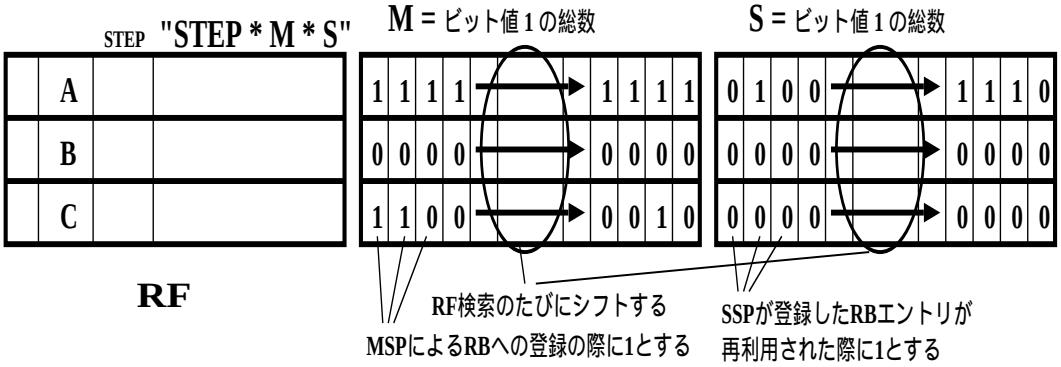


図 10: 命令区間選択機構

規に登録された命令区間については，ただちに SSP による数回分の事前実行を試み，その結果，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の頻度も高い RF エントリを，継続して SSP の実行対象とした．また，動的に変化する登録頻度および再利用頻度を随時把握しておくために，一定時間における登録および再利用の状況をシフトレジスタに記録する．具体的には，図 10 に示すように，一方のシフトレジスタでは，MSP が RB エントリに登録された際に，その RB エントリの属する RF エントリに対応するシフトレジスタの最上位ビットを 1 とする．もう一方のシフトレジスタでは，SSP が登録した RB エントリが再利用された際に，その RB エントリの属する RF エントリに対応するレジスタの最上位ビットを 1 とする．また，ともに RF 検索がなされるごとにビットをシフトする．それぞれのシフトレジスタ内のビット値 1 の総数をそれぞれ M, S とする．RF に設けた小さなプロセッサが，これらの値と RF. STEP の値をもとに $STEP * M * S$ を計算し，各 SSP が，この値が最大となる RF エントリを選択し，該当する命令区間の実行を継続する．

第 5 章　再利用表縮小のための提案

4.2節に述べたように，再利用表全体の大きさは，$RFSIZE$，$RBSIZE$，$RBMMRMAX + RBMMWMAX$ の 3 つの値の積により決定される．このうち，$RBSIZE$，$RBMMRMAX$ および $RBMMWMAX$ については，個々の関数の性質によって必要量が決定されるのに対し，$RFSIZE$ は，登録可能な関数の個数である．

さて，前述した再利用機構では，次に実行する命令区間が RF に登録されていなければ，LRU アルゴリズムに基づいて RF に無条件に新規登録する．RF エントリを無条件に置き換える仕組みのままでは，将来再利用される頻度が高いと考えられる命令区間のデータも，短期間のうちに別のデータに上書きされてしまうため，このまま RF のエントリ数を削減すると，再利用の効果が著しく損なわれることが予想される．

本論文では，登録しても効果のない関数をあらかじめ特定し，RF のエントリを消費しないよう工夫することにより，再利用表全体の大きさを縮小しつつ，同等の性能を維持できるのではないかと考えた．

登録を開始しても効果がない関数には以下が挙げられる．

- システムコールを含むもの
- RW に登録可能な入れ子の深さを超過するもの
- 第 7 ワード以降の引数が検出されるもの
- Read/Write アドレスの登録数が上限を超えるもの

ある命令区間が，そもそも再利用の効果が期待できない命令区間か否かを判定するために，RF エントリ内の STEP (以下，RF.STEP) の値を利用する．STEP は，該当する RF エントリに新たな命令区間が登録されるときに，0 に初期化しておく．命令区間の実行および RB への登録が完了した時に，その命令区間を実行するのに要するステップ数を RB エントリ内の STEP (以下，RB.STEP) に記録し，同様に最後に得られた RB.STEP の値を RF.STEP に格納する．そもそも再利用できない命令区間と判断された場合は，RB への登録が完了しないため，RF.STEP の値も更新されず 0 のままとなる．本論文では，この情報を RS と呼ぶ，RF とは別の構造に記憶することにより，再利用の効果がある RF エントリが追い出されない工夫をした．

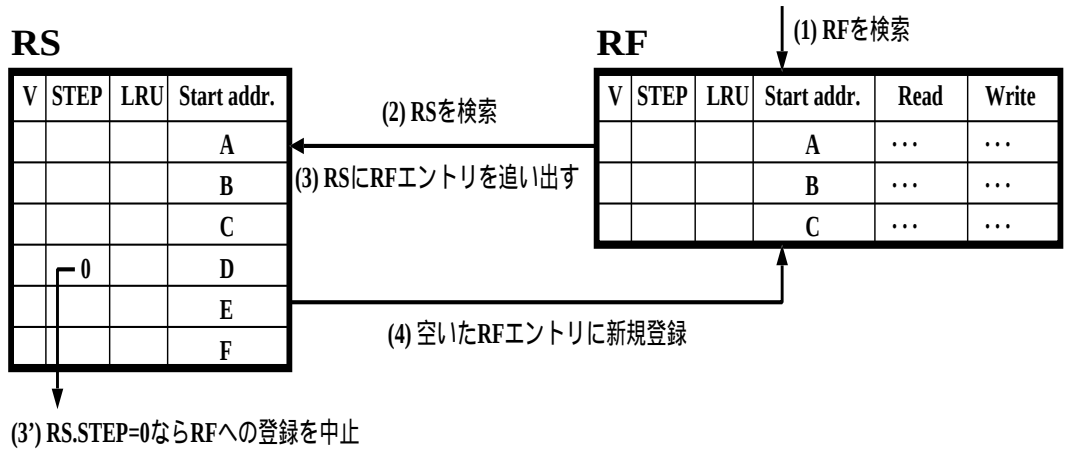


図 11: RS の動作

5.1 RS の導入

RS は，RF と同様，各エントリが互いに異なる関数またはループに対応している．ただし，主記憶読み出しアドレスおよび主記憶書き込みアドレスは保持していない．つまり，RF が保持する項目のうち，必要最小限の情報である有効エントリ表示 (V)，再利用により削減できるステップ数 (STEP)，関数またはループの先頭アドレス，後述の LRU カウンタのみから構成される．したがって，RS エントリのサイズは RF に比べて非常に小さい．RF エントリに比例して，再利用表本体のサイズが増加するのに対し，RS エントリの追加は，再利用表全体のサイズに対して，さほど影響を及ぼさない．

5.2 RS の動作

図 11 に，RS の動作を示す．命令区間の実行が開始されると，まず従来通りに (1) RF を検索する．命令区間の先頭アドレスが一致する RF エントリが見つからなければ，次に (2) RS を検索する．関数またはループの先頭アドレス (Start addr.) が一致する RS エントリがあった場合，STEP の値が 0 であれば，前述のように，そもそも再利用できない命令区間であるので，(3') 再利用表への登録を中止する．一方，STEP の値が 0 でなければ，(3) 4.2 節において説明した LRU アルゴリズムに基づいて選択した RF エントリの内容を RS エントリへ追い出す．このとき，RF から追い出されるエントリの関数またはループの先頭アドレスと一致するアドレスが RS エントリに見つかれば，その RS エントリに RF エントリを追い出す．こうすることにより，RS エントリの

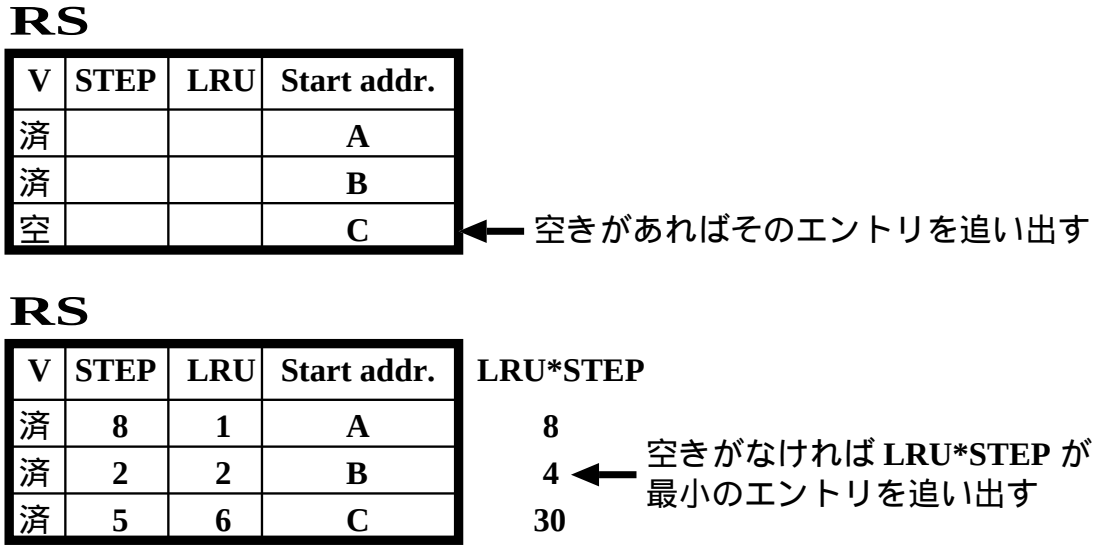


図 12: RS エントリの置き換えアルゴリズム

内容が重複することを回避している．さらに，(4) 以上の操作によって不必要になった RF エントリに新規登録を開始する．また，実行しようとしている命令区間の先頭アドレスが RS でも見つからなかった場合は， 5.3 で説明するアルゴリズムに基づいて選択した RS エントリに RF エントリを追い出し，RF に新規登録を行う．

5.3 RS エントリの置き換えアルゴリズム

上述のように，RS エントリに RF エントリを追い出す際に，全ての RS エントリがすでに登録済みの場合，RF エントリの追い出し先として，内容を上書きする RS エントリを 1 つ選択する必要がある．このとき，RS の各エントリに設けた LRU カウンタ (以下，RS.LRU) および以下の値を利用する．

pt パージタイマ．MSP が関数呼び出しやループを検出して再利用表を検索するたびにインクリメントされる．初期値は 0 ．

RSPURGETIMER pt の値を 0 にリセットするタイミングを決定する定数．本論文の評価では，この値を RS エントリ数の 2 倍とした．

delay_line RS の各エントリが保持する 32 ビットの値であり，過去に再利用表が参照された際に RS への RF エントリの追い出しがどの程度行われたかを表す．具体的には，ビット値 1 のある位置が上位ビットであるほど，最近行われたことを意味する．

RS を検索して RF エントリの追い出し先エントリが決定したときに，そのエントリの delay_line の最上位ビットが 1 でなければ，RS.LRU の値をインクリメントし，delay_line の最上位ビットを 1 に変える．また，pt の値が RSPURGETIMER に達した際に，delay_line の最下位ビットが 1 であれば，対応するエントリの RS.LRU の値をデクリメントする．さらに，RS の各エントリの delay_line をそれぞれ右に 1 ビットシフトする．つまり，RS.LRU の値は，常に delay_line 内のビット値 1 の総数を表すようになっている．この時点で RS.LRU の値が 0 であれば，そのエントリが最近全く参照されていないと考え，状態を 「登録済み」から 「登録可能」に変更する．

このように，delay_line および pt を用いて LRU カウンタに工夫を施すことにより，長期間にわたる過去の RS の参照頻度を把握することができるようになり，これによって，従来よりも正確に最も不要な RS エントリを特定することができる．この工夫は，RF エントリ置き換えにおける LRU カウンタ，および事前実行時の命令区間選択に使用する値 M, S にも使用している．

また，RS は，RF と同様，再利用によって削減できるステップ数を表す STEP を保持する．RF エントリの追い出し先となる RS エントリは，RS に空きエントリがある場合は，最初に見つかった空きエントリとし，空きエントリがない場合は，LRU カウンタの値と STEP の値の積が最小となる RS エントリとする．これは，RF エントリの追い出し先として参照される回数が最近少なく，STEP の値が小さいエントリが最も不要であると考えるためである．

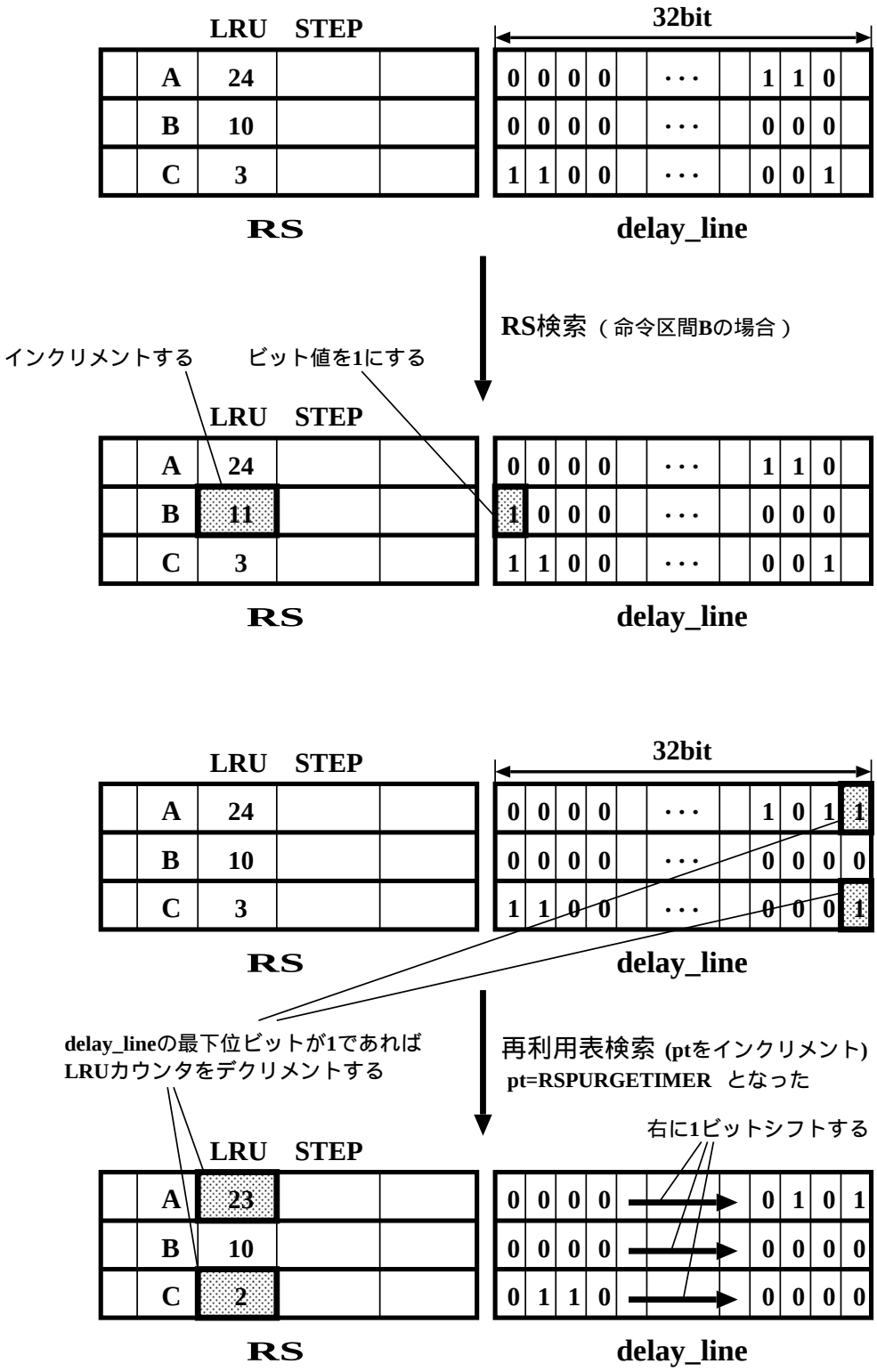


図 13: RS の LRU カウンタ

第 6 章　評価

本章では，第 4 章 で説明した再利用機構を搭載したシミュレータを実現し，ベンチマークプログラムを用いて性能評価を行い，考察を加える．

6.1　ハードウェアの仮定

本論文では，第 4 章 で説明した再利用機構を搭載した SPARC-V8 サイクル・シミュレータを使用した．実行しようとしている命令区間が再利用可能か否かの判定および再利用自体の動作に必要なサイクル数を表 1 のように仮定した．キャッシュ構成や命令レイテンシは HAL の SPARC64 [21] を参考にした．

さて，本論文の目的は，理想的に再利用が行われることによって，どこまで高速化が可能であるかを示すことにある．しかし，連想検索のシミュレーションには多大なコストを要するため，連想検索の上限，すなわち，RF1 エントリに対応する RB エントリ数を，現実的な値 256 とした．一方，RB の横幅に関わる主記憶アドレス数については，シミュレーションが可能な限り大きくし，読み出しと書き込みそれぞれを 1024 アドレスとして測定した．レジスタの内容と RB 内のレジスタ入力データとの比較には 1 サイクル，キャッシュの内容と RB 内の主記憶入力データ (最大 1024 アドレス) の比較は 1 ワードあたり 1 サイクルと仮定した．なお，比較の際にキャッシュミスを検出した場合には，通常のキャッシュミスと同じペナルティが発生する．再利用の際の書き込みは，RB 内の主記憶書き込みデータから MSP のレジスタへは 1 ワードあたり 1 サイクル，RB 内のレジスタ出力データから MSP のレジスタへは 1 サイクルを要すると仮定した．

測定には，Stanford Integer，SPEC INT95 および SPEC INT2000 を gcc-3.0.2 (-supersparc -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．

6.2　Stanford Integer による評価

Stanford Integer は，次の 10 個のサブルーチンから構成されている．ただし，FFT と Queens において，単に同じ処理を 20 回と 50 回繰り返している最外ループは，再利用効果が無意味に高く現れないよう，各 1 回に変更した．以下に，これらの概要と予想される再利用の効果を述べる．

表 1: シミュレータの諸元

データキャッシュ	64 Kbytes
ラインサイズ	64 bytes
ウェイ数	4
キャッシュミスペナルティ	20 cycles
レジスタウィンドウ	4 sets
ウィンドウミスペナルティ	20 cycles/set
ロードレイテンシ	2 cycles
整数乗算レイテンシ	8 cycles
整数除算レイテンシ	70 cycles
浮動小数点数加減乗算レイテンシ	4 cycles
単精度浮動小数点数除算レイテンシ	16 cycles
倍精度浮動小数点数除算レイテンシ	19 cycles
RW の最大深さ	4
RF エントリ数	32/16
読み出しアドレス数	1024/RF
書き込みアドレス数	1024/RF
RF 1 エントリあたりの RB エントリ数	256
RS エントリ数	0/256
RB (引数) とレジスタの比較	1 cycle
RB (Read) とキャッシュの比較	4 bytes/cycle
RB (Write) からキャッシュへの書き込み	4 bytes/cycle
RB (返り値) からレジスタへの書き込み	1 cycle

Quick　5000 個の整数に対してクイックソートを行う．ソートする関数を繰り返し呼び出すものの，同じ数列をソートすることはないため，MSP のみによる再利用はできないと考えられる．また，入力データの変化が不規則なため，事前実行の効果は小さいと考えられる．

Bubble，FFT　それぞれバブルソート，高速フーリエ変換を行う．関数呼び出しは乱数計算を行う関数だけである．しかし，削減できるステップ数は大変小さいため，効果は微小であると予想される．

Trees　5000 個の整数に対して，1 個ずつ要素を挿入して二分探索木を作成する．これは，ある接点の要素を x とするとき，その左部分木内の要素は全て x より大きく，右部分木内の要素は全て x より小さいという条件を満たす．挿入する場所を探す関数を再帰的に呼び出す．木の状態が全て一致すれば再利用されるが，1 個ずつ順に挿入するため，MSP のみによる再利用はできないと考えられる．また，木の状態はある程度予測可能と考えられることから，事前実行による高速化が期待できる．

IntMm，Mm　それぞれ，整数および浮動小数点数の行列積の計算を行う．最内ループが要素数 40 の積和を求める関数を呼び出しているため，MSP のみによる高速化は期待できないものの，事前実行の効果は大きいと予想される．

Perm　配列上の値を再帰的に入れ換える．2 値を入れ換えるだけの関数が繰り返し呼び出されているため，MSP のみでも再利用の効果が得られると予想される．

Queens　8×8 の碁盤目に，8 つのクイーンをお互いに取れないように配置する，いわゆる「クイーン問題」を解く．クイーンが配置できるかどうかを判断する関数が繰り返し呼び出される．配置できない位置が全て一致しなければ再利用できない．引数は単調に変化するが，事前実行のために読み込んだ read アドレスが，MSP によって再利用される時には変化しているため，事前実行による効果は小さいと予想される．また，碁盤が大変小さいため，再利用される機会は少ないと予想される．

Towers　18 個のディスクによる「ハノイの塔問題」を解く．ディスクを持ち上げる関数および塔に配置する関数を繰り返し読み出す．操作しようとするディスクの大きさおよび塔の状態が一致すれば再利用できる．ディスクの数が少ないため，MSP のみでも再利用の効果が得られると予想される．

Puzzle　$5\times5\times5$ の立方体空間を 4 種類のピースを使って埋め尽くす．$1\times2\times4$ のピース 13 個，$1\times1\times3$ のピース 3 個，$1\times2\times2$ のピース 1 個，$2\times2\times2$ のピース 1 個を適宜使用する．まず，ピースを埋め込もうとする場所が空いているかどうかを調べる関数を呼び出す．空いていればピースを埋め込む関数を呼び出し，空いていなければピースを取り外す関数を呼び出す．以上の動作を，立方体空間が全てピースで埋められるまで繰り返す．これらの関数はすべて，MSP のみによる再利用の効果が得られると予想される．

図 15 に，再利用を適用しない場合の総実行命令ステップ数（レイテンシが 2 以上の命令はレイテンシを加算）を 1 とする，MSP の実行命令ステップ数の比を示す．凡例は，MSP のみを用いる場合（MSP），MSP および 3 台の SSP を用いる場合（MSP+SSP*3）と，関数再利用のみの場合（F）と，再利用をループにも適用した場合（F+L）を組み合わせた 4 通りの測定条件を表す．なお，RF エントリ数を 32，RS エントリ数を 256 とした．

Trees では事前実行にループを加えると性能が低下することを除き，おおむね良好な結果が得られている．最内ループが配列要素の積和計算である IntMm および Mm では事前実行，ループ中に再帰呼び出しがある Queens ではループ再利用，ループ内に関数呼び出しがない FFT ではループ事前実行，関数の再帰呼び出しが多い Towers では関数再利用が，それぞれ高速化に大きく貢献している．また，ループと再帰呼び出しが複雑な入れ子構造となっている Puzzle では最大 90% 近くの命令ステップ数削減に成功している．

図 14 に，MSP に 3 台の SSP を付加し関数とループに対して再利用を行った場合（MSP+SSP*3 F+L）において，再利用可能であった命令区間の入れ子の深さ（Level1，Level2，Level3，Level4 以上）ごとに分類して，再利用を適用しない場合に対する削減ステップ数の比（上段），再利用 1 回当たりの削減ステップ数（中段），入出力ワード数を入力レジスタ（**in-regs**），Read アドレス（**read**），Write アドレス（**write**），出力レジスタ（**out-regs**）ごとに平均したもの（下段）を示す．入れ子の深さが 4 以上では削減ステップ数の比がほぼ 0 となり，Stanford では入れ子の深さ 3 までの命令区間を考慮すればよいこと，再利用 1 回あたりの削減ステップ数が，深さ 2 以上の多重再利用では数百に達すること，入力レジスタ数は数個であること，また，削減ステップ数の比を 2%以上に限定しても，Read/Write アドレスの平均個数は，80/8 を越えないことが分かる．

さて，実際に高速化を達成するためには，再利用に伴うオーバヘッドを考慮した評価を行う必要がある．図 16 は，命令ステップ数 (exec) に，表 1 に示した RB (引数) とレジスタの比較および RB (Read) とキャッシュの比較 (test)，RB (Write) からキャッシュへの書き込みおよび RB (返り値) からレジスタへの書き込み (write)，キャッシュミス (cache)，レジスタウィンドウミス (window) の各オーバヘッドを加えた，MSP が実行したサイクル数の内訳である．左側棒グラフは再利用を適用しない場合，右側棒グラフは「MSP+SSP*3 F+L」の場合の内訳である．

Towers では，再利用により関数の再帰呼び出しが減少し，その結果，レジスタウィンドウミスが大幅に減少している．再利用のオーバヘッドの大部分は，test が占めている．RB (Read) とキャッシュの比較のスループットを 4bytes/cycle から 8bytes/cycle に増加させるなど，比較の高速化が重要な課題であると言える．

6.3 SPEC95 による評価

次に，Stanford よりもプログラムサイズの大きいベンチマークである SPEC95 を用いて評価を行った．SPEC INT95 および SPEC FP95 の各プログラムには test, train, ref の 3 種類が用意されており，実行時間は ref が最も長く，test が最も短い．本論文では，train を用いた．SPEC INT95 および SPEC FP95 のプログラムのうち，本論文で評価対象としたものの内容を以下に述べる．

124.m88ksim シミュレータ上で実行するプログラムにデータ依存があった場合に，後続命令がどれだけ待つ必要があるかを計算する関数と，命令実行ごとにブレークポイントの判定を行う関数が，実行命令数の半分以上を占める．前者は，入力が実行中の命令の状態であり，出力が待つべきサイクル数であるので，再利用が可能である．後者は，ブレークポイントが存在した場合に標準出力もしくはファイルへの出力が発生するので，再利用できない．ただし，ブレークポイントが存在しない場合は再利用が可能である．シミュレータ上で実行させるプログラムにループが多く含まれる場合に再利用の効果が高くなることが予想される．

126.gcc GNU C コンパイラのバージョン 2.5.3．呼ばれる関数に偏りはないものの，同じ文字列のコンパイルを繰り返す場合があり，再利用の効果がある程度現れることが予想される．

130.li　Lisp インタプリタ．処理の半分近くが文字列の読み込みと式評価で占められる．文字列読み込みは trap 命令を伴うので再利用できない．式評価は同じ式を繰り返し評価する場合があるため，全体として，再利用の効果がある程度現れることが予想される．

132.ijpeg　メモリ上での画像圧縮および伸長を行う．実行の大部分は，画像の量子化およびコサイン変換，ダウンサンプリング，RGB から YCC への変換，ハフマン符号化，逆量子化および逆コサイン変換を行う各関数によって占められる．これらの関数は 8×8 ピクセルを対象とし，色の深度は 16 ビットであるため，入力値が一致する可能性は低く，再利用の効果は薄いことが予想される．

134.perl　プログラミング言語 Perl のインタプリタ．与えられた一連の文字列の綴りの順番を変えたものの中から，英語辞書に記載されている単語と一致するものを全て探し出す scrabbl.pl スクリプトを読み込み，実行する．

147.vortex　オブジェクト指向のデータベース．オブジェクト指向プログラムにおいては，定数もオブジェクト化するために，値の局所性が生じるため，再利用の効果が期待できる．

101.tomcatv　ベクトル化されたメッシュ生成．2 次元のメッシュを境界に沿って生成する．

102.swim　差分近似による浅瀬式の求解．1024×1024 の行列を使用する．

103.su2cor　モンテカルロ法．素粒子の質量を QCD を利用して求める．

104.hydro2d　ナヴィエ・ストークス方程式．銀河ジェット問題を，流体のナヴィエ・ストークス方程式を利用して解く．

107.mgrid　3 次元ポテンシャル場．3 次元ポテンシャル場のマルチグリッド法による求解．

141.apsi　気象予測．気温，風向き，風速，汚染物質の拡散に関する式の求解．

図 17 に，図 14 と同じ方法による測定結果を示す．再利用 1 回あたりの削減ステップ数が，入れ子の深さ 2 以上の多重再利用では数百に達すること，入力レジスタ数は数個であること，Read/Write アドレスの平均個数はともに 100 未満であることなどが分かる．

図 18 に，図 15 と同じ方法による測定結果を示す．126.gcc および 253.perlbmk (SPEC INT2000) では，再利用をループに適用すると性能が低下することを除き，良好な結果が得られている．また，124.m88ksim および 147.vortex の，関

数再利用によるステップ数削減が著しい．

また，図 19 に，図 16 と同じ方法による測定結果を示す．本提案では，RB の内容と主記憶の比較にキャッシュを経由しており，キャッシュミスの増加による性能低下が懸念されたものの，再利用を適用しない場合と本提案の間にキャッシュミスの差がほとんどないことが分かる．また，130.li と 147.vortex では，Stanford の Towers と同様，レジスタウィンドウミスが減少していることが分かる．

次に，再利用表に RS を付加する場合と付加しない場合の MSP の実行したサイクル数の比較を行った．用いたプログラムは，124.m88ksim である．図 20 にその測定結果を示す．最も左側のグラフは，再利用を適用しない場合の，図 16 と同様のオーバヘッドを考慮に入れたサイクル数の内訳である．その他の棒グラフは左側から，『MSP F』，『MSP F+L』，『MSP+SSP*3 F』，『MSP+SSP*3 F+L』の内訳であり，それぞれの 4 本の棒グラフは，(RF エントリ数，RS エントリ数) = (8，256)，(16，0)，(16，256)，(32，0) の内訳である．理想的には，サイクル削減率について，(8，256) ≥ (16，0)，(16，256) ≥ (32，0) が成り立っていれば，RS の役割を十分に果たしていると言える．この評価結果では理想的なものはないが，理想に最も近いものは 『MSP F』の構成の場合である．その他の構成では，RS による効果が薄いことが分かる．RS エントリ数をさらに増やした場合にサイクル削減率がどう変化するか，その他のプログラムでは RS 付加による効果はどうか，RF エントリ数をさらに変化させた場合はどうかなどを調査し，また，RS を付加することによって性能向上が一切ない場合については，性能向上しない理由を明らかにすることが今後の課題である．

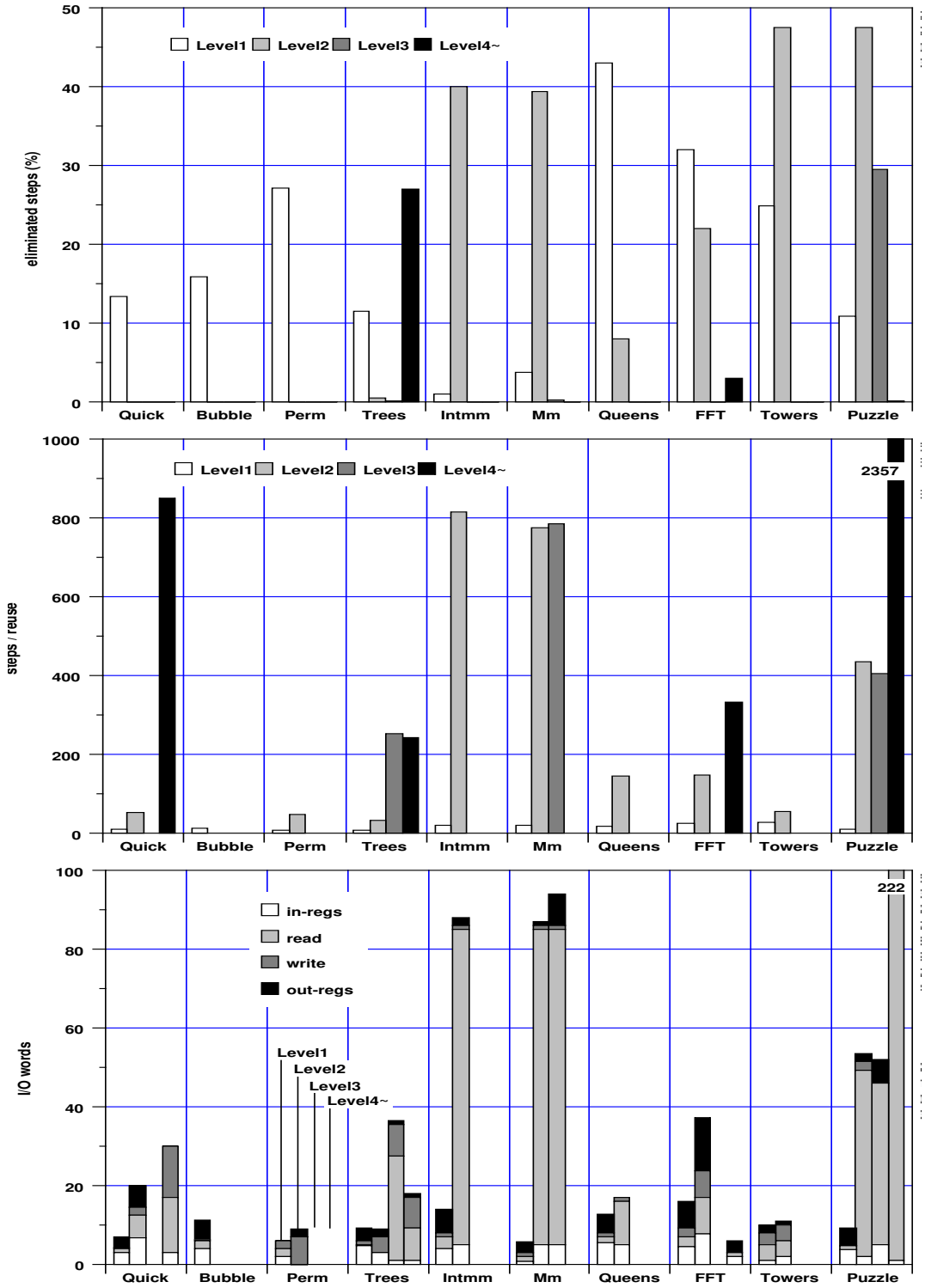


図 14: 深さごとの，削減ステップ数比，再利用 1 回あたりの削減ステップ数，入出力ワード数 (stanford)

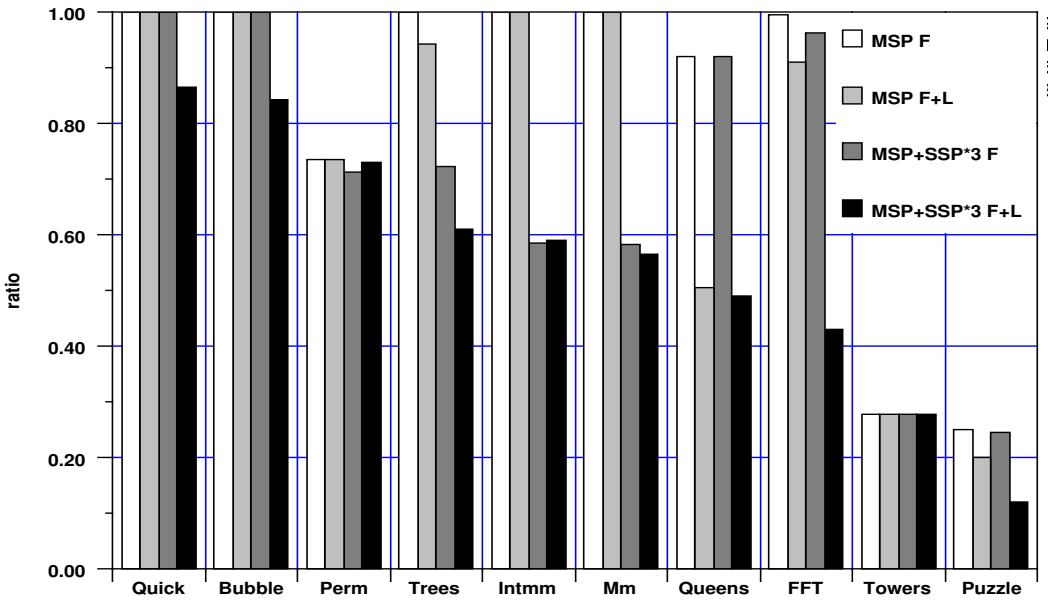


図 15: MSP が実行した命令ステップ数 (Stanford)

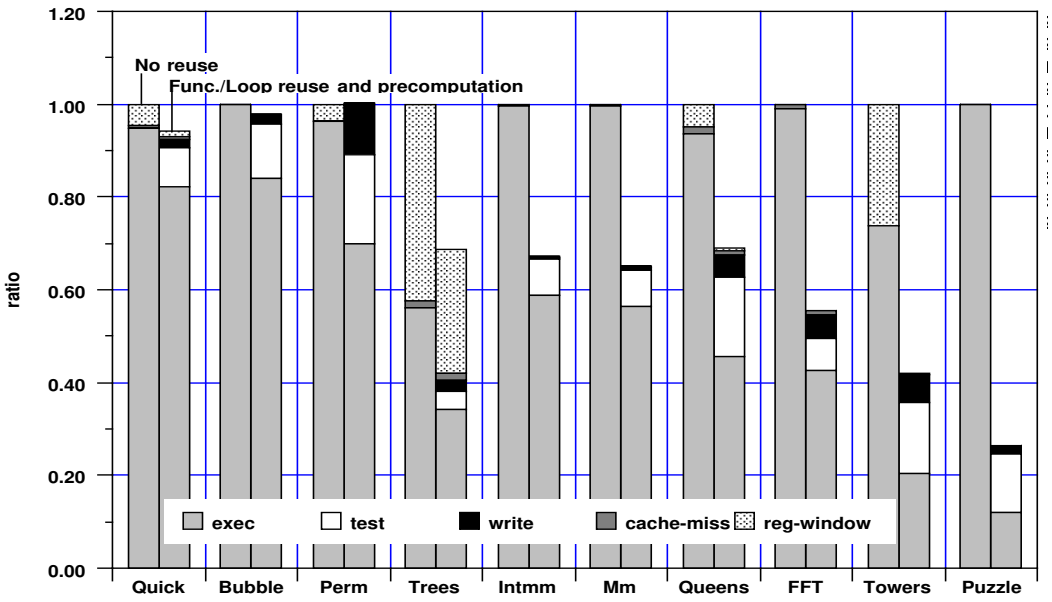


図 16: MSP が実行したサイクル数 (Stanford)

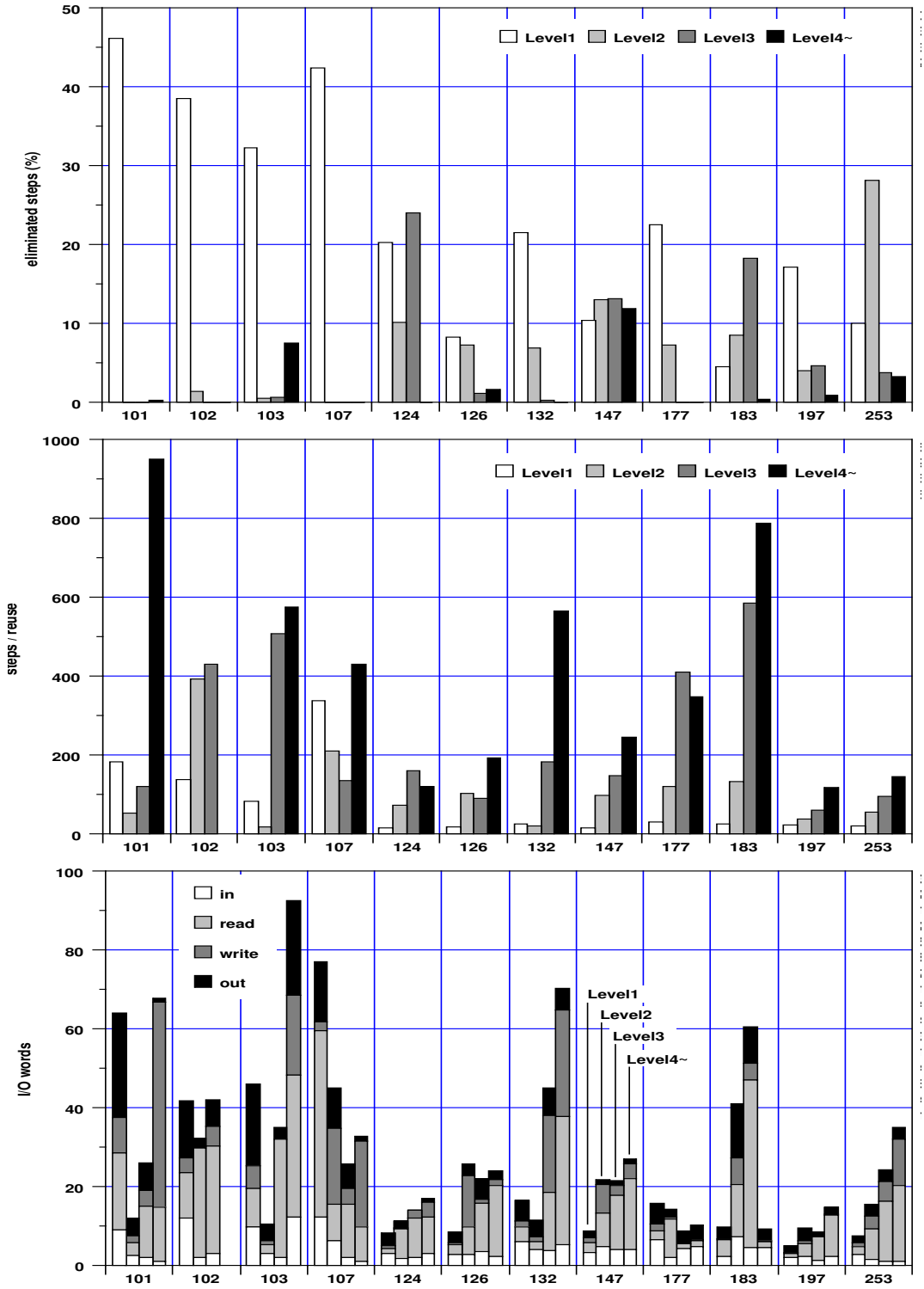


図 17: 深さごとの，削減ステップ数比，再利用 1 回あたりの削減ステップ数，入出力ワード数 (SPEC95)

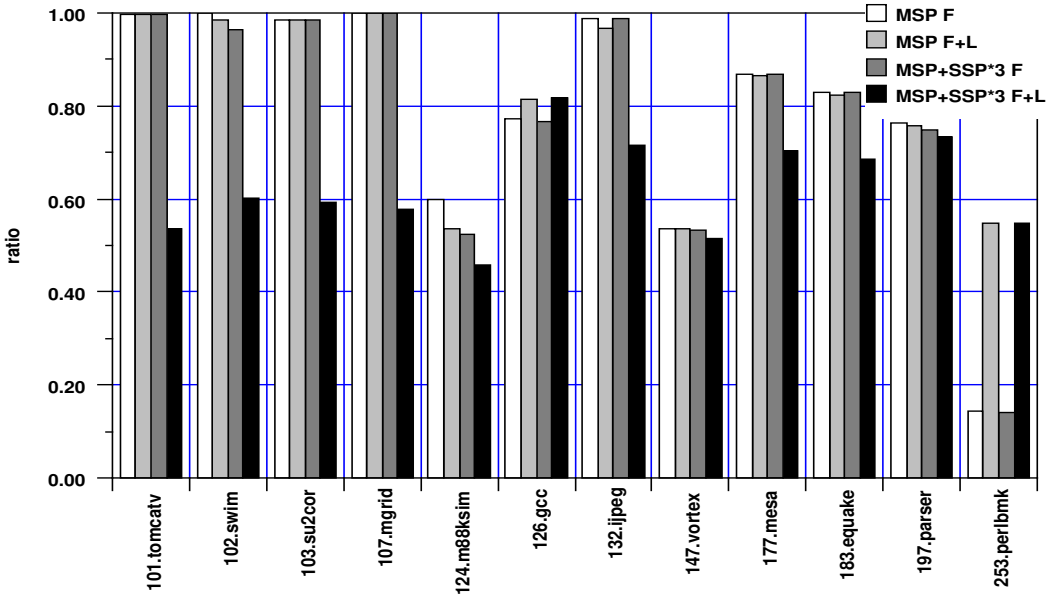


図 18: MSP が実行した命令ステップ数 (SPEC95)

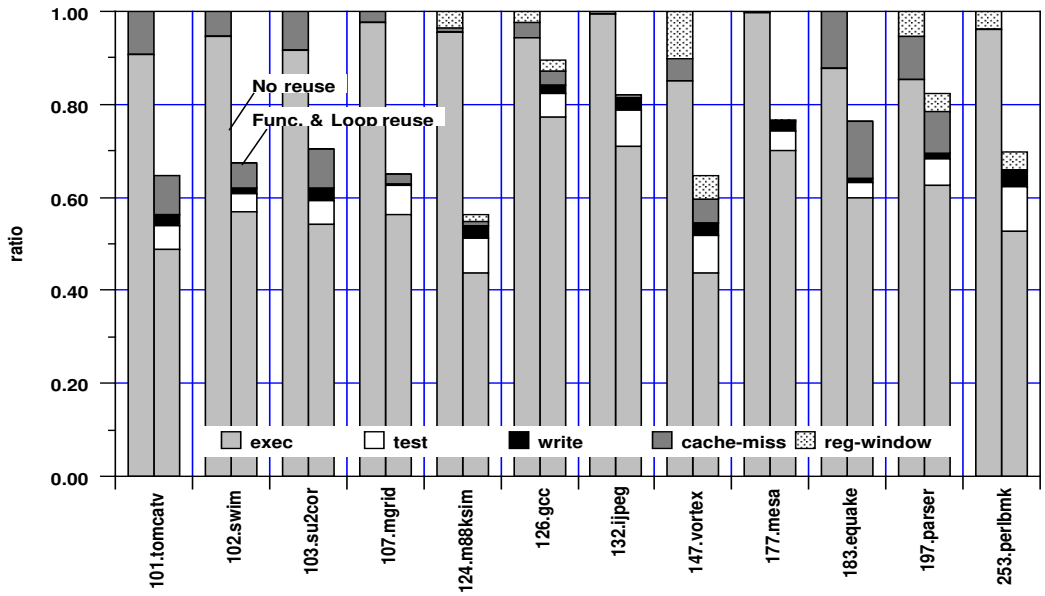


図 19: MSP が実行したサイクル数 (SPEC95)

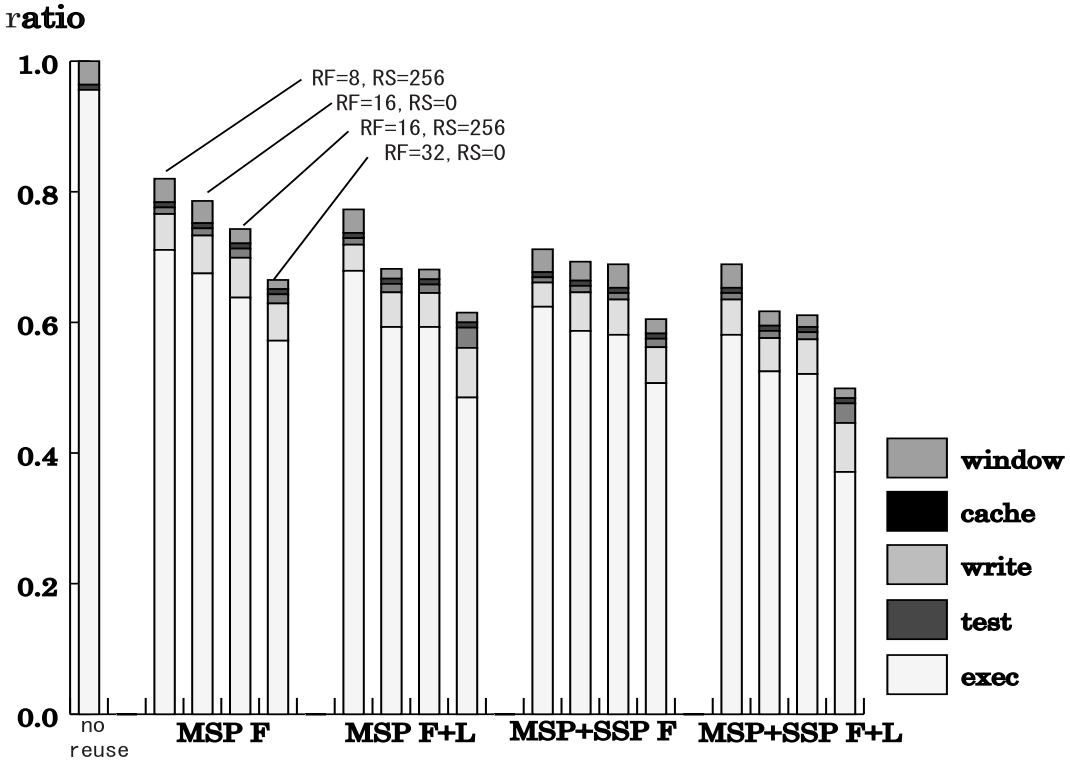


図 20: RS 付加による再利用の効果 (124.m88ksim)

第 7 章　おわりに

本論文では，既存ロードモジュールに対し，専用命令を追加することなく，関数やループからなる命令区間を動的に識別し，多重再利用および並列事前実行を適用する高速化手法を提案した．また，大規模な再利用表を構成するにあたって，ハードウェア使用効率を高め，より少ない再利用表で同等の性能を得る工夫を提案した．

本論文の狙いは，理想的な再利用により，どこまで高速化できるかを示すことにあるものの，連想検索のシミュレーションには多大なコストを要するため，現実的な仮定を行い，連想検索の上限，すなわち RF あたりの RB エントリ数を 256 エントリとした．一方，RB の横幅に関わる主記憶アドレス数については，理想的条件に近付けるために，シミュレーションが可能な限り大きくし，読み出しと書き込みそれぞれを 1024 アドレスとして評価した．

Stanford Integer ベンチマークを用いた評価では最大 75%，SPEC INT95 ベンチマークを用いた評価では最大約 45% のサイクル数を削減できることが分かった．また，RB の内容と主記憶の比較にキャッシュを用いたものの，キャッシュミスの増加による性能低下はほとんどないことが分かった．また，再利用が可能な命令区間を，入れ子の深さごとに分類して調査した結果，Stanford では入れ子の深さは 3 までを考慮すればよいこと，再利用 1 回あたりの削減ステップ数が，深さ 2 以上の多重再利用では数百に達すること，入力レジスタ数は数個であることが分かった．

また，再利用表に RS を付加することによって再利用の効果がどの程度上がるかについて，プロセッサ数と再利用区間を変化させ，MSP が実行したサイクル数を調査した．SPEC95 の 124.m88ksim に対して，RF エントリ数を 16 および 8 とした場合において調査した結果では，少し効果がある構成が 2 種あったが，それ以外の構成では効果はほとんどなかった．今後の課題として，RF エントリ数をさらに変化させた場合，他のプログラムに適用した場合，RS エントリ数をさらに増やした場合などを調べる必要がある．

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚なる謝意を表します．また，本研究に関して適切な御指導を賜わった中島康彦助教授，森眞一郎助教授，五島正裕助手，津邑公暁助手に心から感謝いたします．
さらに，日頃から御助力いただいた京都大学大学院情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝いたします．

参考文献

[1] Paul, R. P. SPARC Architecture, Assembly Language Programming, and C, Prentice-Hall, (1999).

[2] Lipasti, M., Wilkerson, C. and Shen, J. Value Locality and Load Value Prediction, *Senvnth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147 (1996).

[3] Lipasti, and Shen, J. P. Exceeding the Dataflow Limit via Value Prediction, *International Symposium on Microarchitecture*, pp. 226-237 (1996).

[4] Wang, K. and Franklin M. Highly Accurate Data Value Prediction Using Hybrid Predictors, *International Symposium on Microarchitecture*, pp. 281-290 (1997).

[5] Nakra, T., Gupta, R. and So, M. Value Prediction in VLIW Machines, *International Symposium on Computer Architecture*, (1999).

[6] Onder, S. and Gupta, R. Load and Store Reuse Using Register File Contents, *International Conference on Supercomputing*, pp. 289-302 (1999).

[7] Connors, D. and Hwu, W. Compiler-Directed Dynamic Computation Reuse, *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pp. 158-169 (1999).

[8] Sodani, A. and Sohi, G. S. Dynamic Instruction Reuse, *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 194-205 (1997).

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

[10] Costa, A., Franca, F. and Filho, E. The Dynamic Trace Memoization Reuse Technique, *International Conference on Parallel Architectures and Compilation Technigues*, (1997).

[11] Gonzalez A., Tubella, J. and Molina, C. Trace-Level Reuse, *International Conference on Parallel Processing, September 1999*, (1999).

[12] 山田克樹，中島康彦，富田眞治，投機的手法を用いたデータ再利用による Java 仮想マシンの高速化，情報処理学会研究報告 ARC, Vol.139, No.29,

pp. 169-174 (1999).

[13] Wu, Y., Chen, D. Y., Fang, J. Better Exploration of Region-Level Value Locality with Integrated Computation Reuse and Value Prediction, *28th Annual International Symposium on Computer Architecture*, pp. 98-108 (2001).

[14] Sodani, A. and Sohi, G. S. Understanding the Differences between Value Prediction and Instruction Reuse, *International Symposium on Microarchitecture*, (1998).

[15] 吉瀬謙二，坂井修一，田中英彦，2 レベル ・ストライド値予測機構の可能性検討，情報処理学会論文誌 Vol.41, No.5, pp. 1340-1350, May, (2000).

[16] Sohi, G. S. and Amir Roth, Speculative Multithreaded Processors, *IEEE Computer*, Vol.34, No.4, pp. 66-73 (2001).

[17] Pedro Marcuello, Antonio Gonzalez, Jordi Tubella, Speculative Multithreading Processors, ICS '98, (1998).

[18] Pedro Marcuello, Antonio Gonzalez, Jordi Tubella, Value Prediction for Speculative Multithreaded Architecture, *International Symposium on Microarchitecture*, pp. 269-280 Dec. (2000).

[19] L. Condrescu, S. Wills, J.Meindl, Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications, *IEEE Transactions on Computers*, vol.50, No.1, Jan. pp. 67-82 (2001).

[20] 富田眞治，コンピュータアーキテクチャ，丸善，(1994).

[21] FUJITSU/HAL SPARC64-III User's Guide, www.sparc.com/standards/ (1998).