特別研究報告書

SPARC アーキテクチャにおける
関数事前実行値再利用

指導教官　　富田 眞治 教授

京都大学工学部情報学科

正西申悟

平成 14 年 2 月 8 日

SPARC アーキテクチャにおける 関数事前実行値再利用

正西申悟

内容梗概

現在主流のマイクロプロセッサは、命令レベル並列性 (**ILP** :Instruction Level Parallelism) の抽出により、性能向上を図っている。しかしながら、ILP は命令間の制御依存関係およびデータ依存関係の存在によって左右される。前者による影響を取り除く方法には既に多くの研究があるものの、データ依存のうち特にフロー依存はILP を低下させる深刻なボトルネックと考えられてきた。近年、この影響を取り除く手法として、値予測に基づく投機的実行や値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、実際に計算を行って値を求める代わりに、将来の値を予測することによって計算を進めておく手法である。主に単一命令レベルでの研究が盛んに行われているものの、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始めている。ただし、主な問題点として、多くの命令間に依存関係が存在する場合、多くの投機的実行が必要となること、また間違った投機的実行をやり直す際のペナルティが大きいことが挙げられる。

一方、値再利用は、実際の計算を再現するために必要な入力データおよび出力データを再利用表に登録しておき、再度同じ計算を行う場合に、途中の計算を実行することなく出力データ値を直ちに得ることにより高速化を図る手法である。主な特長として、命令間の依存関係の多少は、機構の複雑さに影響を及ぼさないこと、また入力値および出力値の総数のみによってハードウェアコストが決定され、省略可能な命令数を制限しないことが挙げられる。しかし、一方で、以前に実行された結果のみを再利用するため、入力データが単調に変化する場合には、全く効果が無い。さらに、値再利用機構をハードウェアのみで実現するには、基本ブロックの検出が難しく、単一命令が対象となるため、大きな効果が期待できない。複数命令を対象とするためには、近年報告されているような、専用命令およびコンパイラによる支援を前提とする必要があり、既存のロードモジュールを高速化できないという深刻な問題が残ったままとなっている。

本論文では、SPARC ABI(Application Binary Interface) を利用することによ

り、専用命令を追加することなく、関数単位の値再利用を可能とし、さらに、並列事前実行により、入力データが単調に変化する場合についても高速化を図った。再利用を行うために、再利用表と呼ぶ機構を用意し、関数に対する入力データとして引数および主記憶読み出しデータ、また、出力データとして主記憶書き込みデータおよび返り値を登録しておく。また、並列事前実行を行うために、通常通り再利用を行いながら命令列を実行するプロセッサ (MSP) とは別に、先行して再利用表への登録のみを行うプロセッサ (SSP) を用意した。SSP は、再利用表に登録されている関数から、事前実行する関数を 1 つ選択し、Stride-based 予測を行って各引数を予測することによって事前実行を開始する。再利用の効果によって関数を 4 種類に分類し、それぞれの性質に合わせて、事前実行の停止および再利用表エントリ入れ換えアルゴリズムの切り替えを行った。

以上のような関数値再利用および事前実行機構を有し、命令レイテンシ、キャッシュミス、ウィンドウミス、および値再利用に伴うレイテンシを考慮する SPARC シミュレータを実装し、ベンチマークプログラムによる評価を行った。

測定の結果、Stanford-integer ベンチマークでは、並列事前実行を行わなければ全く再利用できないプログラムが、事前実行により最大 80%近くの命令実行ステップ数が削減された。しかし、SPEC FP95 では、事前実行の効果はほとんど現れず、事前実行を行わないで再利用するものの効果のみが現れた。事前実行による高速化は、最大でも 2%程度であった。これは、Spec FP95 には、引数が単調に変化する関数が少ないことに起因すると考えている。また、Standard-integer の Puzzle および Towers では、事前実行を行わない方が命令実行ステップ数の削減率が大きかった。これは、事前実行を行うことによって、本来再利用できるエントリを消去してしまったためである。また、再利用時のオーバーヘッドは、RB とキャッシュ内容との比較が大部分を占めることも分かった。

Stride-based 予測以外の方法を適用して事前実行の効果を上げられるかどうか、事前実行を行わない方が高速化されるプログラムの場合に、いかにして事前実行を停止するか、また、RB とキャッシュ内容との比較を高速化するために、いかに高速な RB を実現するかが、今後の研究課題として挙げられる。

Function Level Precomputation on SPARC Architecture

Shingo Masanishi

Abstract

Recently, most commercial microprocessors depend on Instruction Level Parallelism(ILP) to improve the performance . But, control dependences and data dependences are major hurdles to exploit the amount of ILP. The technique for removing the former barrier has already been researched in many cases, though the flow constraints have so far been considered to be the serious bottlenecks to gain ILP. However, recently, the speculative execution with value prediction and value reuse attract attentions as the technique for removing the bottlenecks. The speculative execution with value prediction is the new technique to overcome flow constraints by predicting the input value of the instructions. Although the major value prediction focus on instruction-level prediction, some researches which apply multi value predictions simultaneously with multi processors are also beginning to be reported. The main problems is that many speculative executions are needed, if the many dependences are exist between many instructions. Moreover, the penalties occured by the wrong speculative execution are too large.

On the other hand, in the case of value reuse technique, input data and output data are registered into the reuse buffer. If the same input data has been found in the reuse buffer when performing the same calculation again, the calculation can squashed. As main features, it is mentioned that the number of dependences during instructions doesn't affect the complexity of a mechanism, that the hardware cost is determined only by the amount of input data and output data, and that the number of reusable instructions are not restricted. However, because it reuses only the previos result, when input data changes sequentially, it is completely ineffective. Also, if the mechanism is implemented only by the hardware, the effectiveness will be smll because it is difficult to detect basic blocks instead of a single instruction. Recent researches reported some implementations based on special proposal instructions and support by the compiler. However the problem that the existing load module can not be

reused exists.

Considering these situations, this paper proposes a new reuse technique which doesn't need an special proposal instructions utlizing SPARC ABI and can improve the speed against the case that input data changes seaquentially, by parallel precomputation. As informations required to reuse, input data which are arguments and memory read data and output data which are memory write data and return values are registered into the reuse buffer. Moreover, to perform parallel precomputation, the additional processors which pre-executes functions and register to the reuse buffer was prepared. A function is selected from the functions registered in the reuse buffer, and precomputation is executed with predicted argument by Stride-based prediction. The functions are classified into four according to the characteristics and the timing of precomputations and the replacement algorithm of reuse buffer are controlled according to each function.

Considering instruction latency, cache miss penalty, windows miss penalty and reuse latency, I developed a cycle simulator and evaluated the precomputation mechnism with some benchmark programs.

On the Stanford-integer, about a maximum of 80% of the number of instructions executions was reduced for the program which is not reusable at all when precomputation is not performed. However, on the SPEC FP95, the effect of precomputation is small. The improvement in the speed by precomputation was at the maximum as 2%. It is because that there are few functions in which the arguments change sequentially in SPEC FP95. Moreover, in Puzzle and Towers, precomputation gains smaller effect than normal reuse. It is because the entry which should originally be reusable was eliminated by precomputation. Moreover, as for the overhead of reuse, comparison of RB with cache occupied many percentages.

These are future works whether methods other than Stride-based prediction can reduce more cycles or not, and how to stop precomputation in the case of program which gains larger performance under normal reuse than precomputation, and how to construct high-speed RB in order to accelerate comparison of RB with cache.

SPARC アーキテクチャにおける
関数事前実行値再利用

目次

第 1 章　はじめに

現在主流となっているプロセッサ開発では、命令レベル並列性(**ILP**:Instruction Level Parallelism) の抽出に多くのコストを投入している。プログラムに内在する ILP を利用した命令スケジューリング手法には、コンパイル時に行う静的命令スケジューリングや実行時に行う動的命令スケジューリングがある。ILP は制御依存関係やデータ依存関係の存在により低下する。前者の影響を取り除く手法には、分岐予測を用いた投機的実行があり、より精度の高い分岐予測機構を提案する研究が多数なされている [14, 15]。後者は、出力依存、逆依存、フロー依存に分類することができ、このうち、出力依存と逆依存による ILP の低下は、レジスタリネーミングにより抑えることができる。残るフロー依存に関しては、値予測に基づく投機的実行と値再利用により性能向上を図ることができる。現在のところ、数多く研究されている命令レベルでの値予測機構 [5, 6, 16] は、データ依存関係のある命令間において先行命令の実行結果を予測し、後続命令を投機的に開始するものである。先行命令の終了時に実際の値と予測値を比較し、一致していれば予測した結果を用いる。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 [8, 9, 10, 11, 12, 13]。

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。

1. 命令間に多くの依存関係があると、多くの投機的実行が必要となる。
2. 予測が誤った場合、予測に基づく一連の演算結果を全て無効化する必要があるため、多大なハードウェアコストを要する。

一方、値再利用 [3, 4] では、プログラムの部分的な実行結果を再利用表に保存しておく。同一部分を再度実行する場合、入力データが全て同じならば逐次実行を省略し、保存しておいた実行結果を再利用する。この方式の特長を列挙する。

1. 命令間の依存関係の多少は、再利用機構の複雑さに影響を及ぼさない。
2. 入力値および出力値の総数にのみハードウェアコストが決定され、省略可能な命令数が制限されない。

ただし、以前に実行された結果のみを再利用するため、入力データが単調変化する場合には、全く効果がない。また、近年報告されている値再利用機構 [7] は、専用命令およびコンパイラによる支援を前提としており、既存のロードモジュールを高速化することができない。

以上のことから、SPARC ABI を利用した値再利用を行うことにより、専用

命令を必要としないだけでなく、並列事前実行により、入力データが単調に変化する場合についても高速化を図った。2 章では従来の投機・再利用技術について述べ、3 章では投機的手法による関数事前実行方式の提案を行う。4 章では本方式の評価を行う。

第 2 章　従来の投機・再利用技術

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大別できる。分岐予測は大きく、静的分岐予測と動的分岐予測に分類できる。静的分岐予測とは、コンパイラがプログラムを解析し分岐予測を行い、命令中に予測した情報を埋め込む方法である。動的分岐予測とは、過去の分岐履歴を用いて分岐先を予測する方法である。オペランドアドレス予測の代表的なものとして Next Line Prefetch がある。これはキャッシュに関するもので、あるアドレスにアクセスされると、近い将来その次のキャッシュラインを参照すると予測するものである。

2.2　値に関する投機的実行

近年、値予測に基づく投機的実行に関する研究が多数行われており、命令レベルでの研究 [3, 5, 6, 16] とマルチスレッドレベルの研究 [8, 9, 10, 11] に大別される。

2.2.1　命令レベルの値予測

先行命令の結果を予測 (値予測) し、後続命令を投機的に実行する。そして、正しい結果が判明したとき、予測値を検証する。正しければそのまま実行を続けるものの、誤っていれば投機実行した演算を全て無効化するため、予測の確度向上が重要課題である。

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用いる方法である。さらに、Classification Table(**CT**) を用いて、予測ミスを減少させる機構が提案されている [6]。CT は、動的に予測可か不可かを判断する。

Stride-based 値予測　最近の 2 回の演算結果の差分 $Stride$ と最近の演算結果 $Base$ から、次の予測値を $Base + Stride$ とする方法である [5]。

2レベル値予測　さらに [5] は、動的に生成される値の多くは数種類、特に 4 種類であるという結果に着目し、2 レベル値予測機構を提案している。最近の 4 種類の演算結果と過去の予測結果を記憶しておく。1 段階目は、予測する命令のアドレスから、記憶している 4 種類の演算結果を出力する。2 段階目は、1 段階目で出力された 4 値から、過去の予測結果を利用して、1 つの値を選択することによって予測値を得る。

ハイブリッド予測　さらに [5] は、Last-value 予測と Stride-based 予測、Stride-based 予測と 2 レベル値予測のハイブリッド予測を提案している。

値予測機構の比較　[16] は、値予測のヒット率とミス率を測定している。表 2.1 に、CT を用いた Last-value 予測、Stride-based 予測、2 レベル値予測、Stride-based 予測と 2 レベル値予測のハイブリッド予測について、ヒット率とミス率をまとめた。

	LV+CT	Stride	2level	Hybrid
ヒット率	21.7%	29.8%	29.4%	39.9%
ミスヒット率	1.7%	2.9%	3.9%	5.9%

表 2.1: 全実行命令に対する予測ヒット率とミス率

2.2.2　スレッドレベルの値予測

スレッドレベルの値予測には、以下の 3 つが挙げられる。

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。各スレッドの分岐先を予測するためにも用いる。

データ依存予測　異なったスレッド間のデータ依存関係を予測するために用いる。

命令・トレースレベルの値予測　命令レベルの値予測機構や、[11] が提案している Increment Predictor のトレースレベルの値予測機構を用いて、スレッド間の、データ依存関係を解消するために用いる。

単一のプログラムを複数のスレッドに分割して、並列に実行する例を図 2.1 に示す。図 2.1 に示すプログラムは、A,B,C,D の 4 つのスレッドに区切られる。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

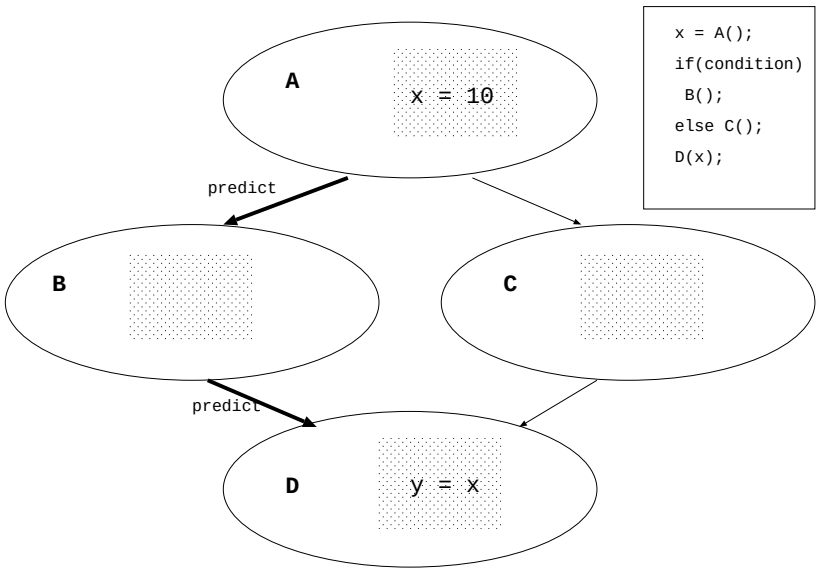

図 2.1: スレッドレベルでの値予測実行例

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。

2.3 値再利用

値再利用は、値予測とは違い投機的手法ではない。命令をフェッチし、再利用表を参照して、その命令の全ての入力データが再利用表に登録されている当該エントリの入力データと一致していれば再利用可能となり、命令実行が省略される。一致しなければ、通常の実行が行われ、その結果を RB に登録する。

近年報告されている具体的実現方法 [7] として、コンパイラが再利用を行う命令列を生成し、最初と最後に、再利用のための専用命令を追加する。さらには、値予測に基づく投機的実行と統合した研究 [13] も報告されている。

第 3 章 投機的手法による関数事前実行方式の提案

前述の値再利用は、過去の演算結果を再利用する技術であるため、入力データが単調に変化する場合には、全く効果が無い。入力データの単調変化を予測

して、演算結果を事前に登録することができれば、さらなる高速化を図ることができる。また、投機的演算結果を RB に閉じこめることにより、予測が外れた場合でも、投機実行のキャンセル動作を不要とすることができる。

3.1 SPARC ABI を利用した値再利用

値再利用を行うには、プログラムを構成する命令列から、入力データおよび出力データがそれぞれ明確に取り出せるような区間を取り出す必要がある。さらに、命令区間が多くの命令をふくんでいれば、再利用の効果が高まる。このような理由から、本論文では、関数を再利用単位とした。

SPARC ABI に準拠したプログラムでは、メモリ参照に関して以下の性質がある。

- 大域変数を格納するデータ領域と、局所変数を格納するスタック領域には、それぞれ上限がある。
- 命令列と大域変数は、低位アドレスと上記性質を利用してあらかじめ決めておく LIMIT の間に納められる。
- 有効なスタック領域は、スタックポインタ (以下%sp) から高位アドレスまでの範囲である。
- %sp は、LIMIT から高位アドレスの範囲を移動する。
- LIMIT から%sp-1 までの領域は無効である。
- save 命令が実行されると、%sp はより低位アドレスへ移動し、それまでの %sp は、フレームポインタ (以下%fp) となる。
- restore 命令が実行されると、%sp はそれまでの%fp へ移動する。
- %sp から高位へ 16 ワードまでは、レジスタの退避領域であり、関数の入出力には関係しない。
- %sp+64 から 1 ワードは、構造体を返り値とする場合の暗黙的引数のための領域である。
- %sp+68 から 6 ワードは、関数の引数の一時退避空間であり、関数の入出力には関係しない。
- 関数の明示的引数は、最初の 6 個はレジスタ%o0 から%o5 へ、7 個目以降は%sp+92 から高位アドレスへ格納される。
- 関数のローカル変数は、%fp- α ($\alpha > 0$) に格納される。

具体的に、メイン関数 A が関数 B を、さらに関数 B が関数 C を呼びだす場合

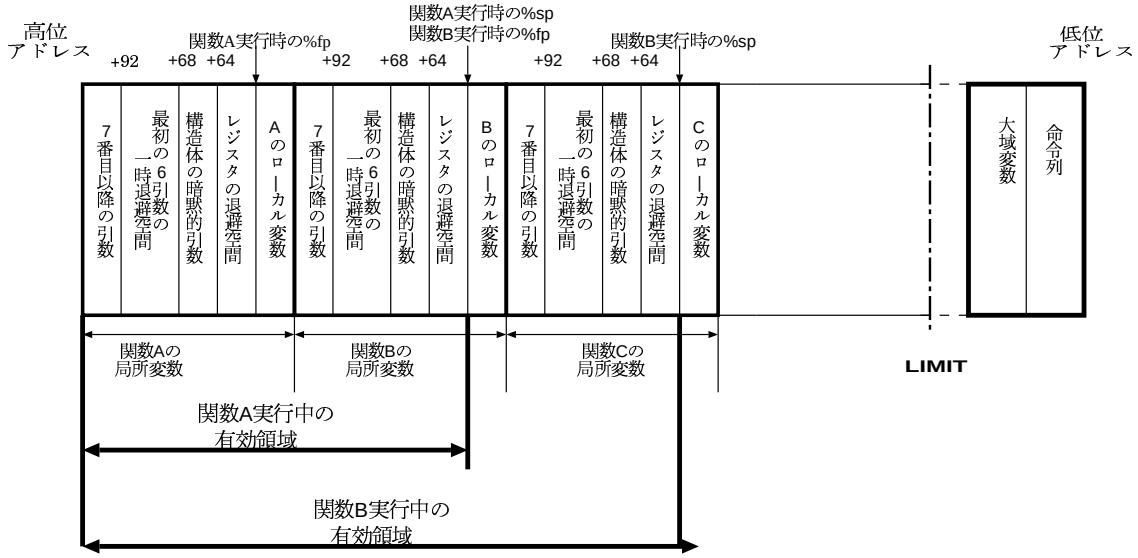

図 3.1: スタック概要

の例を示す。なお、関数 C はリーフ関数とする。図 3.1は、この場合のスタックの構成図である。

関数 A が実行中で関数 B がまだ実行されてない場合、有効な局所変数は、高位アドレスから%sp までの範囲である。B への引数は、最初の 6 個はレジスタ %o0〜%o5 へ、7 個目以降は%sp+92 以降へ格納される。なお、%sp+64 から 1 ワードの領域は、関数 B が構造体を返り値とする場合の構造体の先頭アドレスを格納する領域である。入力データは、大域変数と引数である。

関数 B を呼び出す時、save 命令が実行され、%sp はより低位アドレスへ移動し、それまでの%sp は、%fp となる。関数 B の入力データは、引数または大域変数であるが、ポインタを介して上位関数 A の局所変数に参照する場合もある。第 7 引数以降が存在する場合すると、%fp+92 以上に格納される。ただし、この領域には A のローカル変数も格納されているため、A の局所変数か B の局所変数かを区別することは難しい。A において、%sp+92 以降へのストアがある場合に第 7 引数が存在すると考える。出現頻度が低い事が予想される事、および簡単のため、第 7 引数以降が存在する関数の場合は再利用の対象外とすることにした。

関数 C は、リーフ関数であるため、save 命令が実行されず、%sp,%fp は変化しない。関数 C の入力データは、引数または大域変数であるが、B の場合と同様に、ポインタを介して上位関数 A、B の局所変数を参照する場合もある。ま

た、関数 C の出力は、大域変数、上位関数の局所変数、レジスタ%o0 か%o1 に格納されている返り値である。なお、浮動小数点の場合は、レジスタ%f0 か%f1 に格納される。ただし、返り値が構造体の場合は、%sp+64 からの 1 ワードに格納されているアドレスに書き込む。

関数 B を終了する場合には、restore 命令が実行され、%sp,%fp は元の値になる。出力は、大域変数とレジスタ%i0 か%i1 に格納されている返り値であるが、返り値が構造体である場合には、%fp+64 からの 1 ワードに格納されているアドレスに書き込む。

なお、関数 B 実行時において、関数 C の入力データが、再利用表に登録されているデータと全て一致すれば、値再利用され関数 C の実行が省略される。この時点では、関数 C の入出力データとして登録すべきデータは、関数 A、B の局所変数である。関数 C の局所変数を参照してはならない。この区別には、さきほど述べたように第 7 引数以降が存在する関数を再利用の対象外をすることで、関数 A、B の局所変数は%sp+92 以上、関数 C の局所変数は%sp+92 以下と区別できる。

同様に、関数 A 実行時においても、この時点では、関数 B,C の入出力データは関数 A の局所変数であり、関数 B、C の局所変数を参照してはならない。この区別にも、%sp+92 を利用する。すなわち、登録開始時の%sp を記憶することで登録すべきデータを特定でき、複数レベルの登録を動時に行うことができる。

3.2 事前実行機構

メモリからプログラムを読み込み命令を逐次実行しているプロセッサを Main Stream Processer(以下 **MSP**)、RB へ投機的に入力値及び出力値を登録するプロセッサを Shadow Stream Processer(以下 **SSP**) と呼ぶことにする．再利用表の論理構成を図 3.2に示す。再利用表は、再利用ウィンドウ (以下 **RW**)、関数管理表 (以下 **RF**)、再利用表本体 (以下 **RB**) から構成されている。RW は、関数の入れ子関係を表現しており、各関数の RF と RB へのリンクを保持している。RF は、1 エントリが 1 関数に対応する。それぞれの構成要素を以下に示す。v=有効エントリを表す; 関数アドレス=関数の先頭アドレスを表す; lru=一定期間内の各関数毎の総ヒット数を示すカウンタ; pred_dist=予測した値の進み具合を示すカウンタ; value_history=当該 RB で最近使用した RB の index4 個を表す;

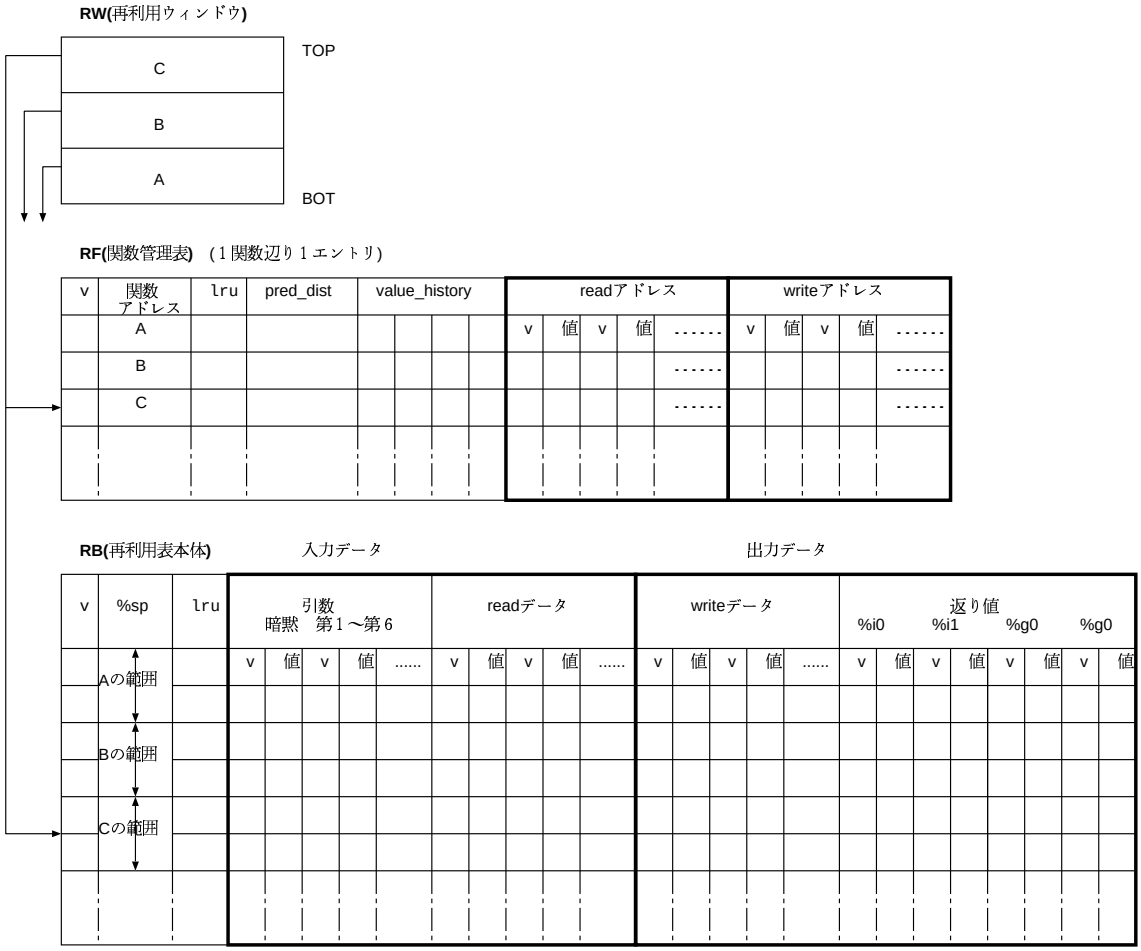

図 3.2: 再利用表論理構成

read アドレス=関数毎の主記憶読み出しアドレスを表す; write アドレス=関数毎の主記憶読み出しアドレスを表す。RB は関数毎のブロック構造となっており、構成要素を以下に示す。v =有効エントリを表す； %sp =スタックポインタを表す； lru =一定期間内のヒット数を表す； 引数=関数の引数の有効性を示す v と値を表す； read データ=関数の read データの有効性を示す v と値を表す； write データ=関数の write データの有効性を示す v と値を表す； 返り値=レジスタに格納される返り値を表す。なお、以降は RF の lru を rf_lru、RB の lru を rb_lru とする。関数呼び出しが再利用できるか否かを判断するには、関数の全ての入力データと、RB に登録済のエントリの入力データを比較する必要がある。なお、この機構では、RF で参照アドレスを一括管理し、RB では、その値のみを管理することで、複数のエントリを一度に比較可能としている。

次に、再利用表の物理構成を図 3.3に示す。引数および主記憶参照データと

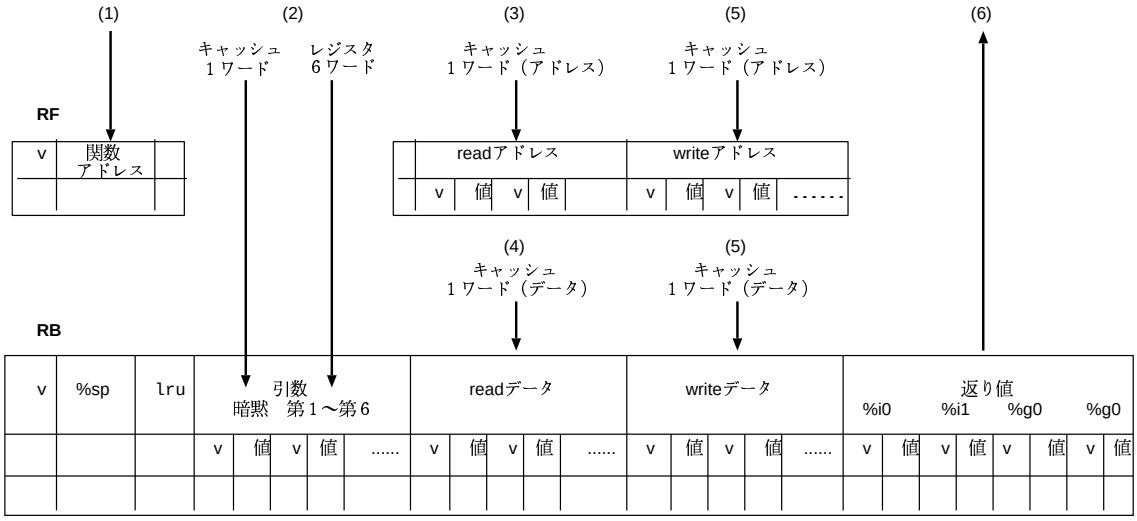

図 3.3: 再利用表物理構成

RB の内容との比較に CAM を使用する想定とした。まず、(1)RF の関数アドレスを参照して RF エントリを特定する。続いて、(2) 引数が全て一致する RB のエントリを特定する。そして、(3)RF の read アドレスを順々に取り出し、(4) 対応する RB の read データと比較する。全て一致するエントリが存在すれば、(5))RB の write データおよび (6))RB の返り値を主記憶およびレジスタに書き込む。再利用できなければ、入力データおよび出力データを再利用表に登録する必要がある。入力データは、引数と大域変数、さらにはポインタを介した上位関数の局所変数である。また、現在実行中の関数の局所変数と入れ子関数の局所変数とを区別するために、%sp を登録する必要がある。主記憶参照データは、参照アドレスを RF の read アドレスへ、値を RB の read データへ登録する。このとき、同じアドレスが write アドレスまたは read アドレスに登録されていたら、新たに登録する必要は無い。一方、出力データは、返り値または大域変数である。主記憶参照データは、参照アドレスを RF の write アドレスへ、値を RB の write データへ登録する。このとき、同じアドレスが write アドレスに登録されていたら、write データを更新する。

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、RB の read データを参照する。さらにそこでも読み込めなかった場合、キャッシュを参照する。

さて、図 3.4に、関数事前実行機構を示す。SSP は、MSP とは別に RW、演算器、レジスタ、キャッシュ、ローカルメモリを持つが、主記憶、RF、RB は全プロセッサで共有する。R は、レジスタからの読み出しおよび RB への登録を表す。R1 は、再利用表に登録済のアドレスから読み出す場合は、キャッシュからでは無く RB から読み出すことを示す。先に write データを参照し、無ければ read データから読み出す。R2 は、再利用表に未登録のアドレスから読み出す場合は、キャッシュから読み出し RB へ登録を行うことを表す。W は、レジスタおよびキャッシュへの書き込み、および RB への登録を示す。

SSP は、RF に登録されているエントリから、事前実行する関数を 1 つ選択し、対応する RB から引数を予測することによって事前実行を開始する。SSP が関数を選択し、引数を予測して関数を実行するとき、局所変数を格納するためのスタックが必要となる。MSP の場合なら、主記憶にスタック領域を確保で

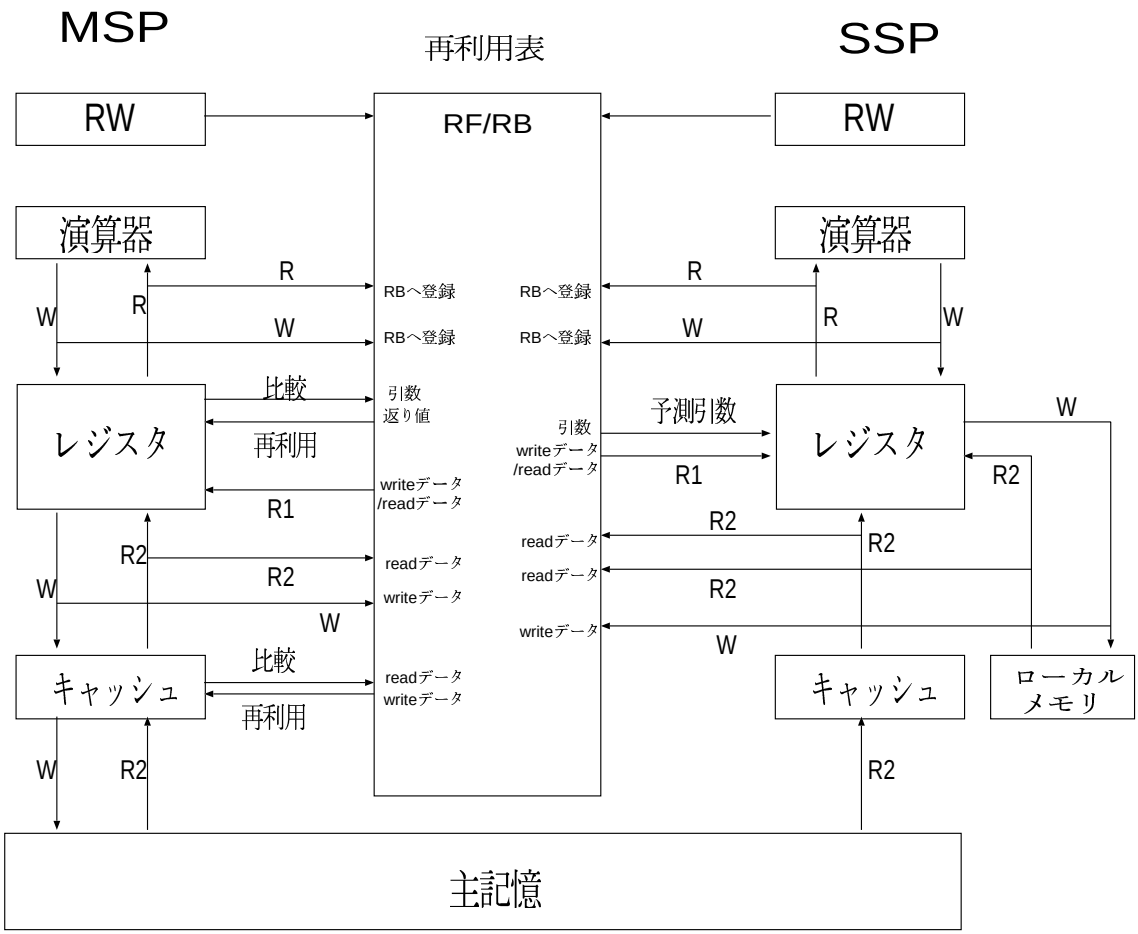

図 3.4: 関数値事前実行機構

きるが、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ローカルメモリが必要である。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中に作成したスタック領域に格納される。上位関数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、MSP と同様に write データ、read データ、キャッシュの順である。また、SSP は主記憶へ書き込んではならないので、大域変数が write データとなっている場合には、主記憶には書き込みせず、RB の write データのみに書き込みを行う。返り値は、直接 RB の返り値へ登録すればよい。関数の実行が終了すると、MSP と同様に登録中の RB を有効にし、RW から該当エントリを削除し、新しい関数の実行を開始する。ただし、ローカルメモリのサイズには上限があり、スタックサイズがその上限を超えた場合には、実行を打ち切る。

3.3 事前実行手順

3.3.1 関数選択

rf_lru を利用する。SSP は、RF に登録されている関数の中から一つを選択して、事前実行を行う。事前実行して最も高い効果が得られるのは、登録回数が多くヒット数が少ないものであると考えられる。また、関数実行にも時間局所性があるので、最近の登録数やヒット数を考慮するのが望ましい。

3.3.2 引数予測

選択された関数の引数を予測するには、RF の value_history と pred_dist を用いる。value_history は、関数ごとに最近使用した RB の index を 4 個保持している。value_history[0] が最も最近使用した RB の index として、2 番目は value_history[1], 3 番目は value_history[2],4 番目は value_history[3] とする。各々の引数に対して、図 3.5のような予測機構を行う。value_history[i]$(0 \leq i \leq 3)$ が指す RB の引数値を $history_i$ とする。$Base = history_0$、$Stride = history_0 - history_1$ として予測値 $= Base + Stride \times pred\_dist$ とする Stride-based 予測である。また、図 3.5の右側が示すように、$history_2 - history_3 = history_1 - history_2$ の場合には、$Stride = history_2 - history_3$ とする。SSP は空きがあれば、$Base$ から離れた値も予測させる。しかし、離れ過ぎた値を予測しても、無駄なエントリが RB に登録されるだけなので、制限する必要がある。pred_dist は、予測した値の進み具合を示すカウンタであり、その範囲は $2 \leq pred\_dist \leq$ プロセッサ

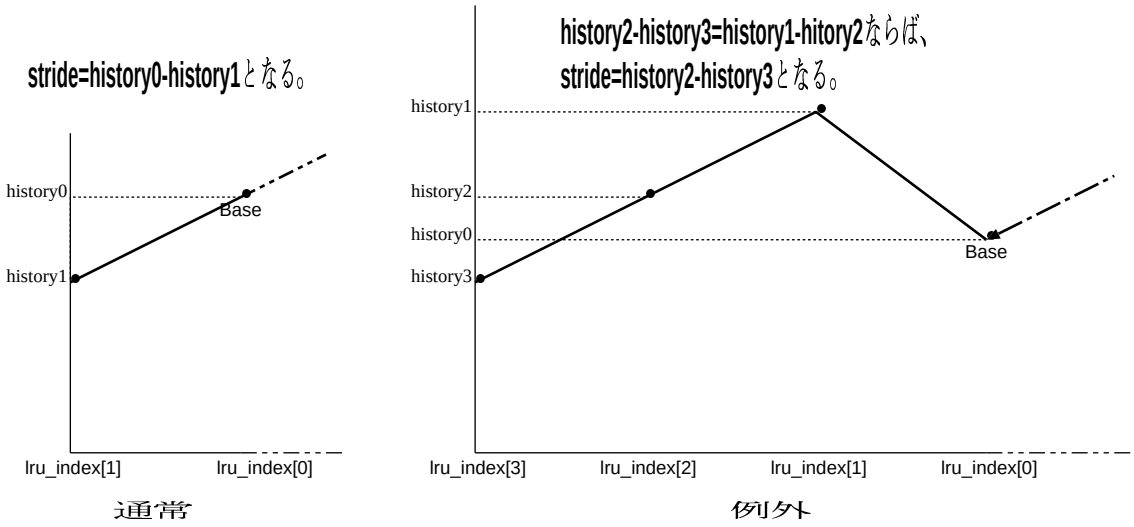

図 3.5: 引数予測

数 × 2 である。

3.3.3 エントリの入れ換え

まず、RF のエントリの入れ換えについて述べる。多くの命令をふくんでいる関数は再利用の効果が高いと言える。また、ヒット数と登録数が多いエントリも効果が高いと言える。よって、$value\_history[0]$ が指すエントリの実行命令数 × rf_lru を最小にするエントリを削除することにした。削除するエントリに対応する RB のエントリは全て初期化される。

続いて、RB の入れ換えの場合について述べる。関数は次の 4 種類に分類できると予想される。

第 1 種関数 パラメータが単調に変化し、MSP だけの再利用では全く効果が無いが、SSP による高速化が可能と考えられるもの。

第 2 種関数 パラメータの変化が小さく、MSP だけの再利用でも効果はあるが、SSP でさらなる高速化が可能と考えられるもの。

第 3 種関数 MSP だけの再利用でも効果があると考えられるもの。

第 4 種関数 パラメータの変化が不規則で、MSP だけでは全く効果が無いが、SSP でも全く効果が無いと考えられるもの。

入れ換え方法として、rb_lru を利用する。rb_lru の最も小さなエントリを消去する方法 (以下 LRU) と、rb_lru の最も大きなエントリから消去 (以下 FIFO) する方法を考えた。また、MSP で登録されたエントリを **m_entry**、SSP で登録されたエントリを **s_entry** とする。

LRU は、繰り返し再利用されるエントリが残し、再利用されていないエントリから消去するので、特に第3種関数には好ましい方法である。しかし、次の例の場合に好ましくない。

```
for ( i = 1; i <= 100; i++ )
 for ( j = 1; j <= 100; j++ )
    function(i,j、&a[i][j]);
```

関数 function は、第1種関数である。s_entry でありかつ MSP で再利用されたエントリは、これ以上は再利用されることは無いため rb_lru は 1 のままである。ところが、s_entry でありかつ MSP で再利用されていないエントリは、rb_lru は 0 である。LRU に従って入れ換えを行うと、これ以上再利用されないエントリを優先的に残す一方で、s_entry でありかつ MSP で再利用されるはずのエントリを、再利用する前に消去してしまう。一方、FIFO は最近あまり利用していないエントリを残し、最も多く利用したエントリから消去する。第1種関数には好ましい方法であるが、繰り返し再利用されるエントリから消去してしまうため、第3種関数には好ましくない方法である。

よって、LRU と FIFO を便宜に合わせて切り換える事にした。前述した様に SSP は LRU との相性が悪いため、LRU に従っている場合は SSP による登録は行わないことにした。また、新たなパラメータとして、fmsp,fssp を定義する。fmsp は、$rb\_lru \geq 1$ の m_entry が存在すれば 1、存在しなければ 0 として、fssp は、$rb\_lru \geq 1$ の s_entry が存在すれば 1、存在しなければ 0 とする。$fmsp = 1$ ならば、MSP によって登録されたエントリは少なくとも 1 度は再利用されており、$fmsp = 0$ ならば、MSP によって登録されたエントリは全く再利用されていない。

$fmsp = 0$ かつ $fssp = 0$ の場合、第4種関数と判断する。$fmsp = 0$ かつ $fssp = 1$ の場合は、第1種関数と判断する。よって、入れ換えは FIFO として SSP を稼働する。$fmsp = 1$ かつ $fssp = 0$ の場合は、第3種関数と判断する。よって、入れ換えは LRU として SSP による登録を行わない。$fmsp = 1$ かつ $fssp = 1$ の場合、第2種関数と判断する。よって、SSP を稼働するが、単純に FIFO による入れ換えを行うと、繰り返し再利用されるはずの m_entry を優先的に削除してしまう。よって、この状況に対処するために、RB を 2 分割し、m_entry、s_entry が使用できる領域を区分した上で、m_entry は LRU に、s_entry は FIFO に従って入れ換えを行う。

第 4 章　評価

本章では、第 3 章で提案した投機的手法による関数値事前実行方式をシミュレータにより実現し、ベンチマークプログラムを用いた性能評価を行う。

4.1　プロセッサモデル

本論文では、事前実行による値再利用機構を搭載した SPARC-V8 アーキテクチャ・シミュレータを作成した。本シミュレータでは、表 4.1 に示す仮定をした。キャッシュ構成や命令レイテンシは HAL の SPARC64 を参考にした [2]。

なお、再利用表参照時に要するサイクル数を説明する。再利用可能かどうか

キャッシュサイズ	64KB
ラインサイズ	64Byte
ウェイ数	4
キャッシュミスペナルティ	20 サイクル
Register-Window	6 セット
Window ミスペナルティ	20 サイクル
ロードレイテンシ	2 サイクル
整数乗算レイテンシ	8 サイクル
整数除算レイテンシ	70 サイクル
浮動小数点乗算レイテンシ	4 サイクル
単精度浮動小数点レイテンシ	16 サイクル
倍精度浮動小数点レイテンシ	19 サイクル
RW の深さ	6
RF のエントリ数	32
RB(引数) とレジスタの比較	1 サイクル
RB(read) とキャッシュの比較	1 サイクル
RB(write) からキャッシュの書き込み	1 サイクル
RB(返り値) からレジスタの書き込み	1 サイクル
SSP のローカルメモリ	64KB

表 4.1: 本シミュレータの仮定

調べるには、入力データがRB と一致しているか確認する必要がある。さらに、再利用されると、返り値をレジスタへ、write データをキャッシュに書き込む必要がある。例として、引数が6 個、主記憶読み出しデータが10 個で、返り値が1 個、主記憶書き込みデータが10 個の関数を再利用する場合には、RB との比較に16 サイクル、書き込み作業に10 サイクルかかると仮定した。

4.2 Stanford-integer による測定

測定対象には、Stanford-integer ベンチマークを gcc-2.7.2(-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使用した。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1 回とした。Stanford-integer は、次の10 個のサブルーチンから構成されている。以下にこれらの概要および予想される再利用効果を述べる。

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第1 種関数だけである。しかし、削減される命令ステップ数は大変小さいため、あまり高速化されないと予想される。

Intmm、Mm　それぞれ、整数、浮動小数の行列積の計算を行う。最内ループが要素数40 の積和を求める第1 種関数を呼び出しているため、MSP だけでは再利用の効果は望めない。しかし、事前実行によって大きく高速化できると予想される。

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第3 種関数が繰り返し呼び出されている。MSP だけでも再利用の効果が得られると予想される。

Puzzle　$5 \times 5 \times 5$ の立方体空間を4 種類のピースを使って埋め込む。$1 \times 2 \times 4$ のピース13 個、$1 \times 1 \times 3$ のピース3 個、$1 \times 2 \times 2$ のピース1 個、$2 \times 2 \times 2$ のピース1 個を適宜使う。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が全て埋まるまで繰り返す。これらの関数は皆、第3 種関数と考えられ、MSP だけでも十分な高速化が期待される。

Queens　8×8 の碁盤目に、8 つのクイーンをお互いに取れない位置に置く8 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。以下のことから、この関数は第4種関数に近いと考えられる。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 データが、MSP によって再利用される時には変化されているため、事前実行の効果は小さいと予想される。また、今回の碁盤は大変小さいため、再利用される機会は少ないと予想される。

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が、同じ数列をソートすることは無いため、MSP だけでは再利用できないと考えられる。また、入力データの変化が不規則であるため、事前実行の効果は小さいと考えられる。よって、この関数は第4種関数と考えられ、再利用があまりできないと予想される。

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ことから、これらの関数は第3種関数と考えられ、MSP だけでも十分な高速化が期待できる。

Trees　5000 個の整数に対して、1個ずつ要素を挿入して二分探策木を作成する。これは、ある節点の要素を 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、その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個ずつ順に挿入しているため、MSP だけでは再利用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、この関数は第1種関数と考えられ、SSP での高速化が期待される。

図 4.1 に、再利用を適用しない場合の総実行ステップ数に対するステップ数削減率を縦軸に表す。横軸には、RF1 エントリあたりの RB エントリ数を表す。MSP1 台をふくむプロセッサ数を 2、4、8 台として測定した。なお、RF に登録可能な read/write アドレス各々のエントリ数は 4096 とした。ただし、Bubble、FFT、Queens、Quick は予想通りステップ削減率は小さく、最大でも 10%未満であるため省略した。Intmm、Mm、Trees は、プロセッサ数が 1 台だけの場合ではほとんど再利用されていないが、プロセッサ数が増えるにつれて効果が大きくなり、削減率は最大約 20〜75% となっている。これは、予想通り積和計

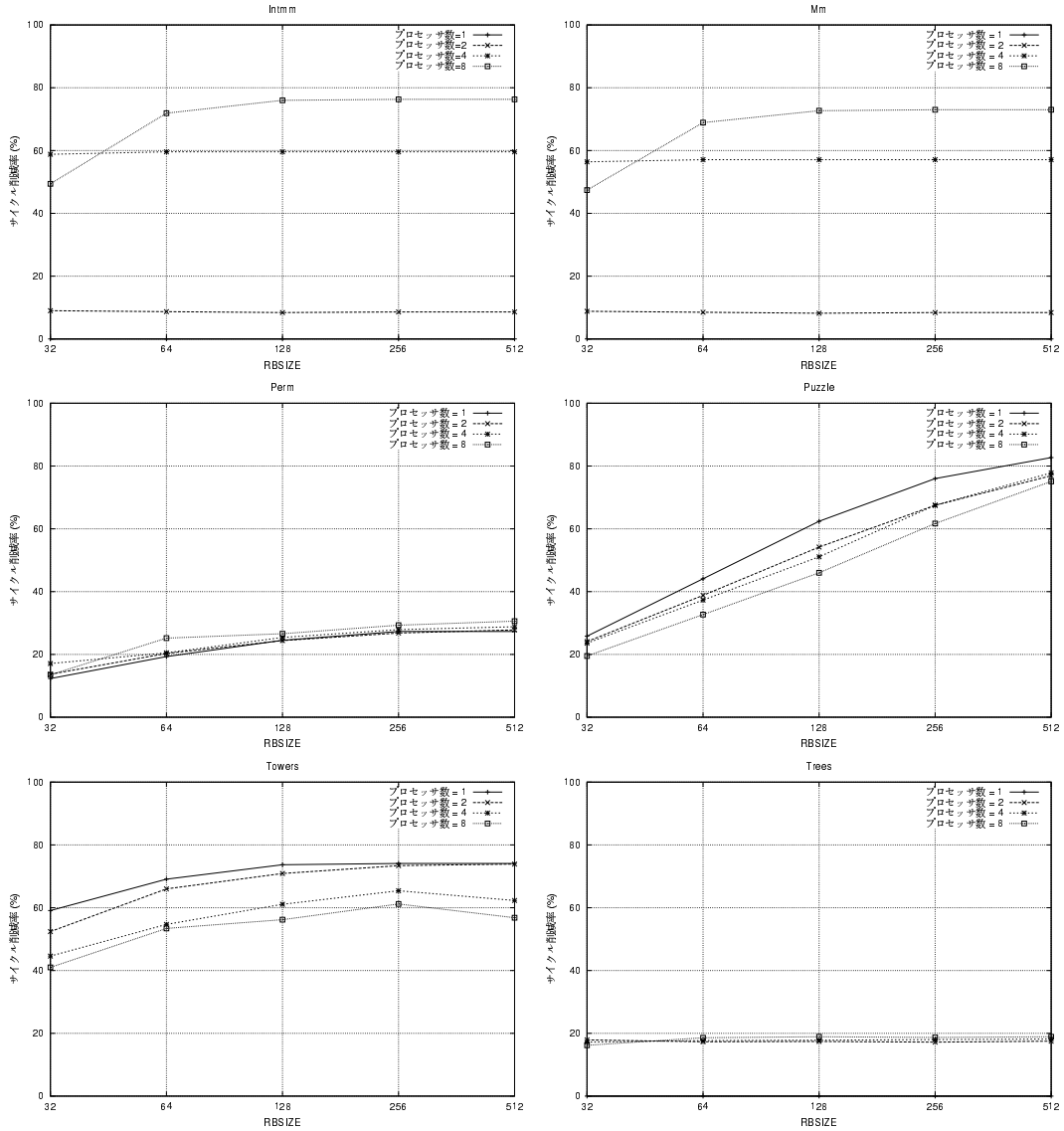

図 4.1: Stanford における実行ステップ削減率

算を行う第 1 種関数によるものである。また、Puzzle および Towers は、MSP だけの方が削減率は大きく、プロセッサ数が少ない方が削減率は大きく、一部 Towers では、RB エントリ数が 256 から 512 に増えたにも関わらず削減率が減少している。

図 4.2は、プロセッサ数=8、RB エントリ数=256 の構成において、再利用された関数を入れ子の深さごと (L1〜L6) に分類し、再利用を適用しない場合の命令実行ステップ数からの命令実行ステップ削減率を折れ線グラフで、再利用 1 回あたりの命令実行削減数を棒線グラフで表したものである。なお、上端を超

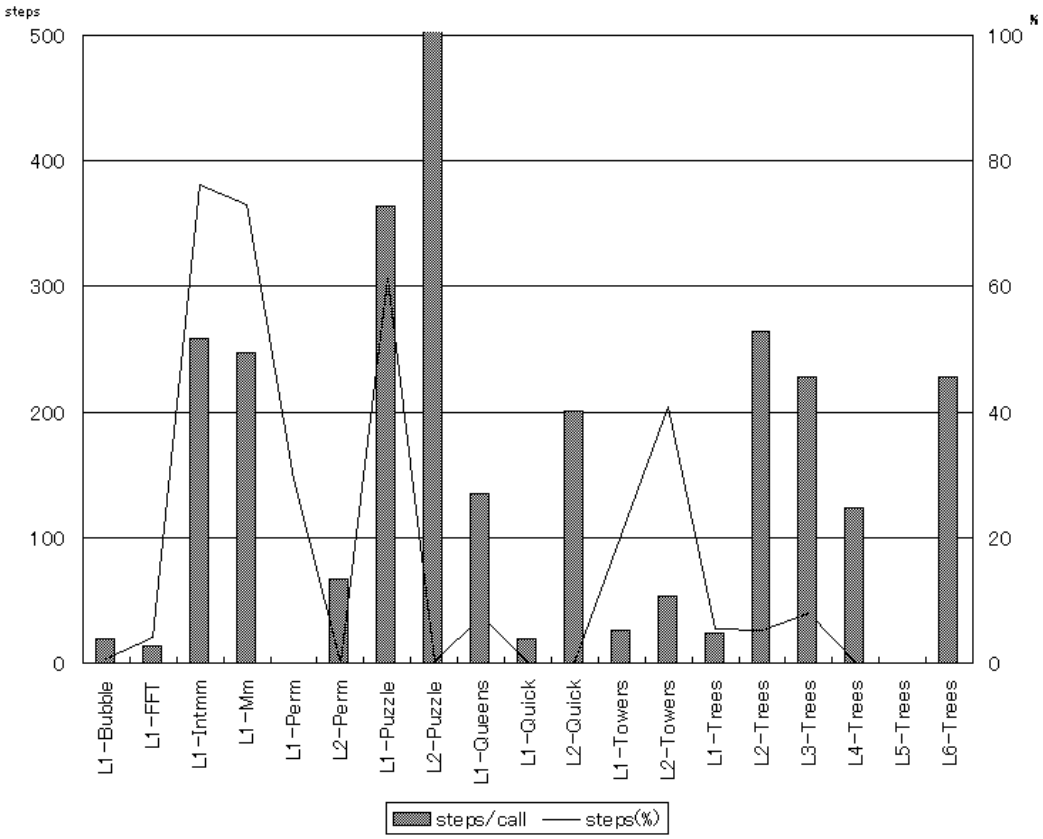

図 4.2: Stanford における深さごとの再利用ステップ数および実行ステップ削減率

える L2-Puzzle の命令実行削減数は 2068.4 である。深さ 4 以上の関数のステップ削減率はほぼ 0 であるため、深さ 3 までの関数を考慮にいれればよいこと、ステップ削減率 5% 以上に限定した場合でも、ステップ数 200 以上の関数は頻繁に再利用されていることが分かる。

図 4.3は、図 4.2と同様に再利用された関数を入れ子の深さごとに分類し、再利用 1 回あたりの入出力数の平均を表す。args は引数、mmrs は read データ、mmws は write データ、retvs は返り値を表す。なお、上端を超える L2-Puzzle は mmrs=226、mmws=1、retvs=1 である。引数はほとんど 2 個以下であること、ステップ削減率 5%以上に限定した場合でも、read アドレス/write アドレスは 44/8 以下であること、入出力総数は大概 30 以下であることが分かる。

さて、実際のプロセッサ高速化を評価するには、再利用時におけるオーバーヘッドを考慮する必要がある。図 4.4は、表 4.1で示したオーバーヘッドを元にした命令サイクルの内訳を示す。exec は命令ステップ数、その他は、cache は

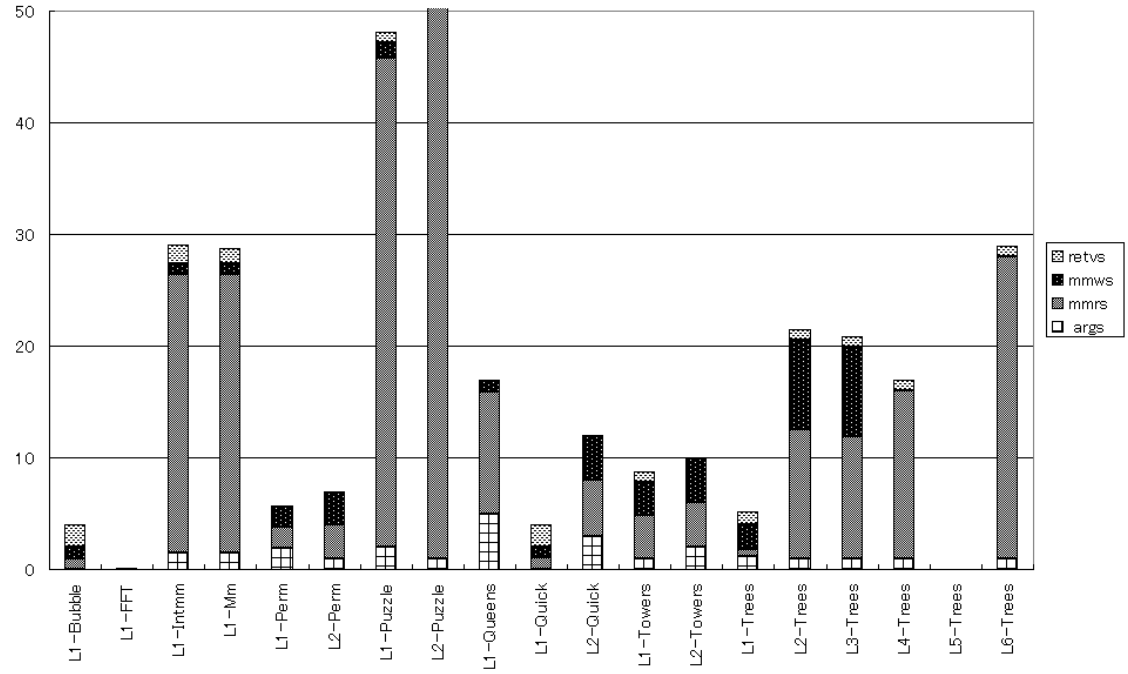

図 4.3: Stanford における深さごとの入出力数

キャッシュミス、window はレジスタウィンドウミス、test は RB(read データ) → キャッシュ比較、write は RB(write データ) →キャッシュ書き込みの各々のオーバーヘッドを示す。左側の棒グラフは再利用を適用しない場合、右側は適用した場合の各々の内訳を示す。Towers では、レジスタウィンドウミスのオーバーヘッドが再利用によって削減されている。また、再利用によるオーバーヘッドの大半は RB →キャッシュの比較が占めている。

4.3 SPEC FP95 による測定

Stanford ベンチマークによる測定から、RB エントリ数は 256 程が適切であると考えた。また、プロセッサ数は 4 台あれば事前実行の効果が確認できることが分かった。SPEC FP95 では状況が異なると予想されるが、シミュレーション時間を短縮するため、RF に登録可能な read/write アドレス各々のエントリ数は 1024、RB エントリ数=256、プロセッサ数=1 および 4 として測定を行った。

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そった形で生成する。

102.swim 差分近似による浅瀬式の求解。1024x1024 行列使用 。

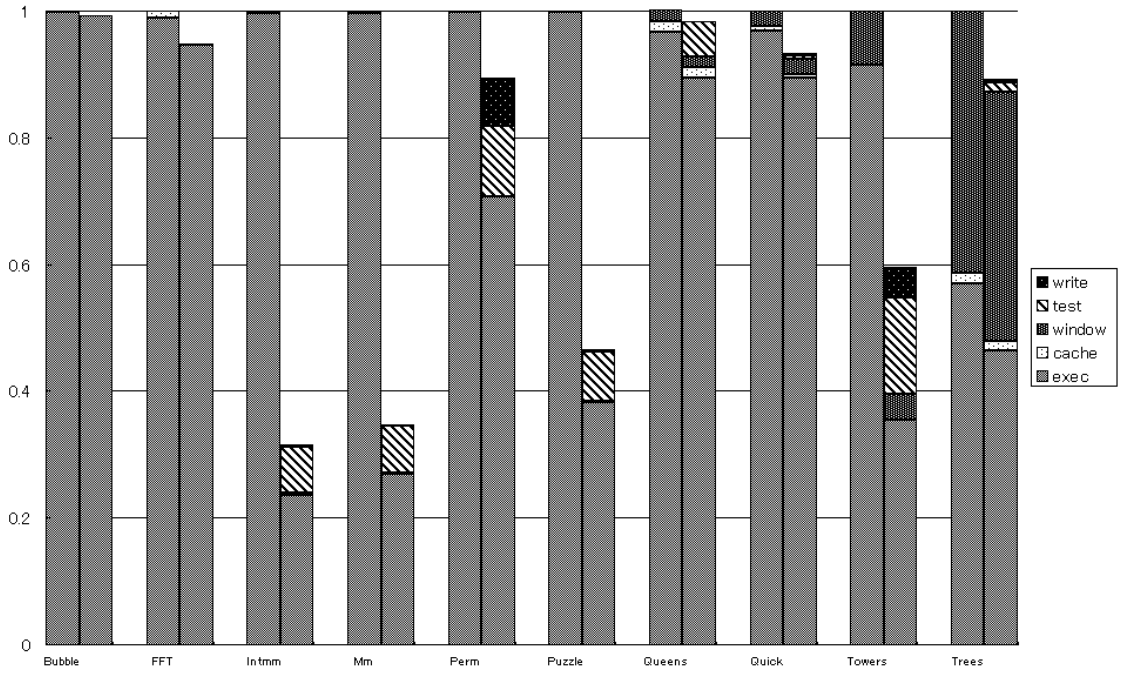

図 4.4: Stanford における実行サイクル数の内訳

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。銀河ジェットの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。

110.applu　放物型／楕円型の偏微分方程式の求解。

125.turb3d　乱流のモデル。立方体中の等方均質乱流のモデルを 1 次元ＦＦＴを利用して解く。

145.fpppp　Gaussian シリーズから抜き出した量子化学計算のベンチマーク。Gaussian92 の Link702 の 2 電子積分の導関数を求めるルーチン。

146.wave5　マクスウェル方程式。電場中での荷電粒子の運動のシミュレーション。

図 4.5は、再利用を適用しない場合と比較した場合の命令実行ステップの削減率を示す。左側の棒グラフはプロセッサ数=1、右側はプロセッサ数=4 の場合を表す。Stanford とは異なり、プロセッサ数=1 の場合と 4 の場合の削減率の差は小さく最大でも 3%未満であった。以下、図 4.6、4.7、4.8においてはプロセッ

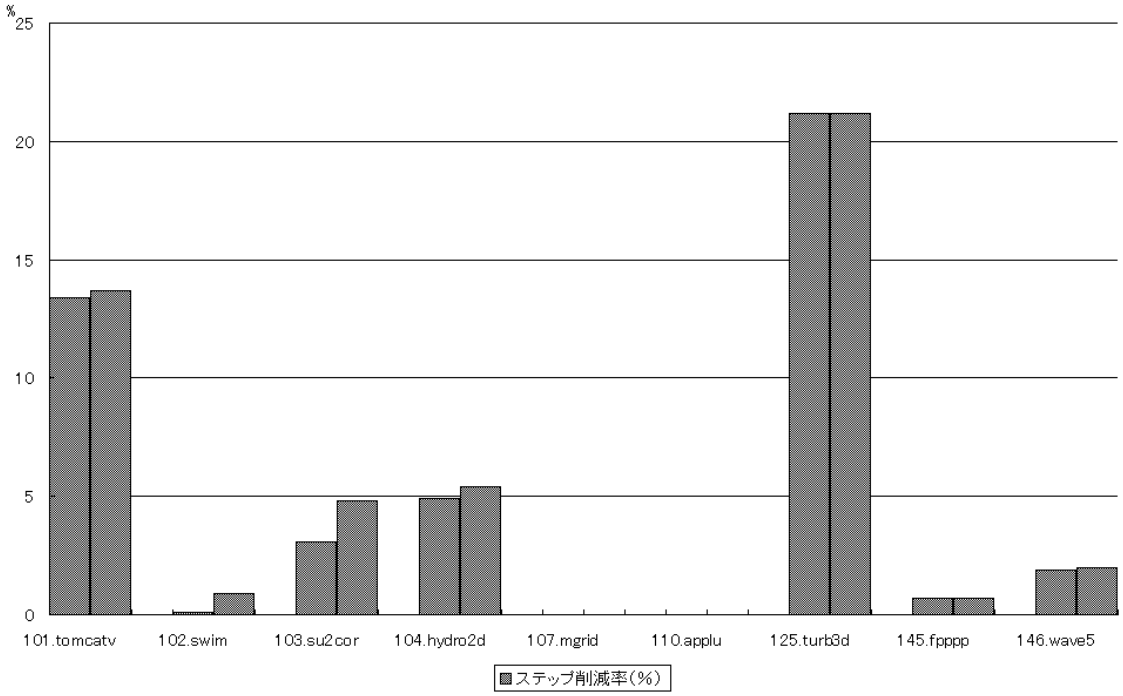

図 4.5: SPEC FP95 における実行ステップ削減率

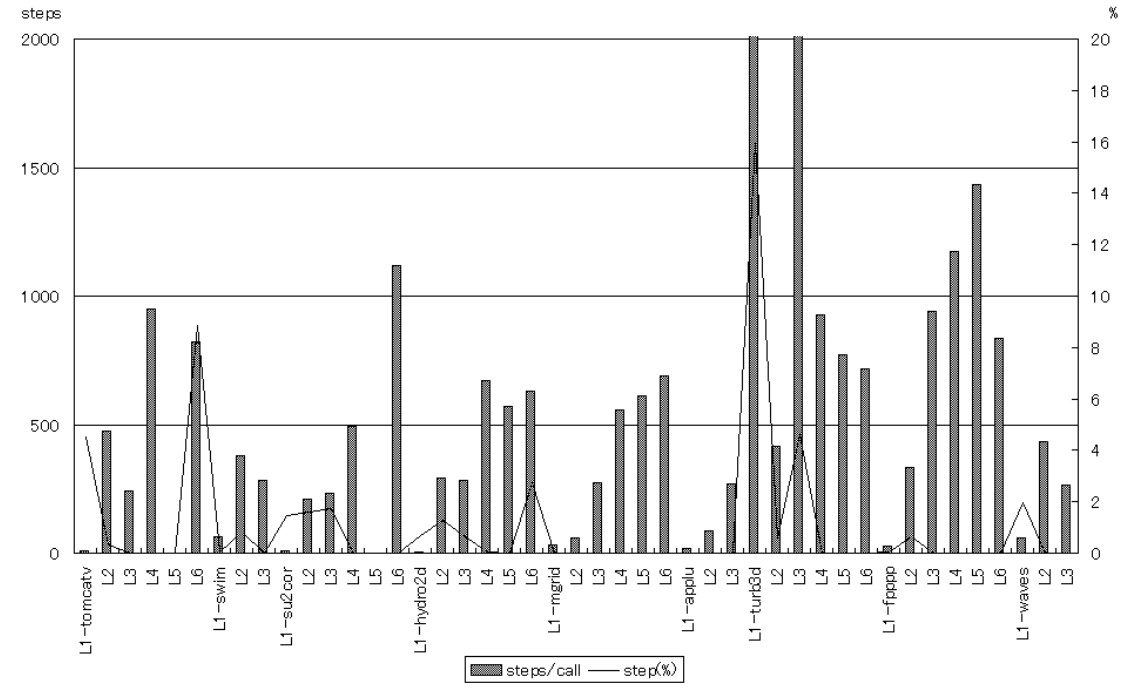

図 4.6: SPEC FP95 における深さごとの再利用ステップ数および実行ステップ削減率

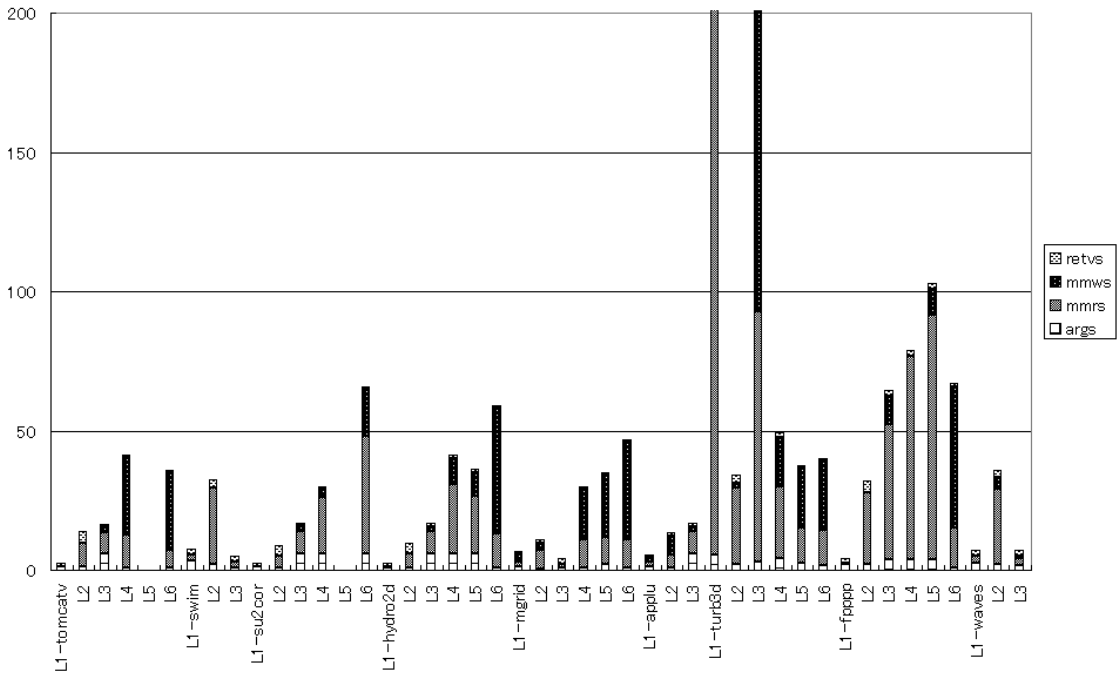

図 4.7: SPEC FP95 における深さごとの入出力数

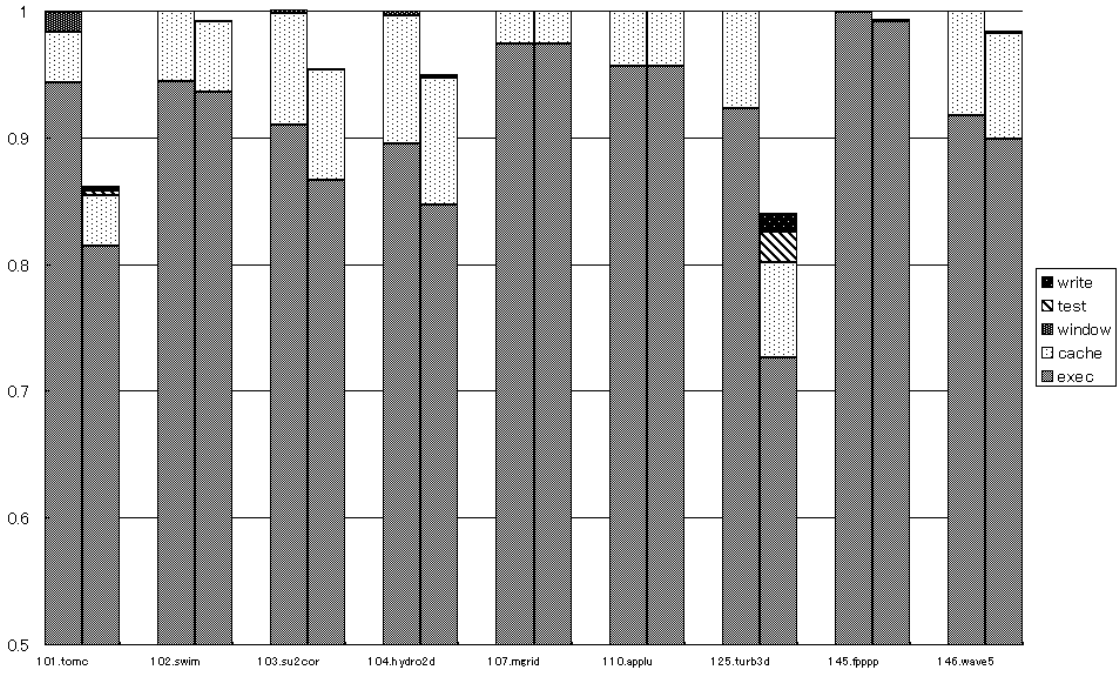

図 4.8: SPEC FP95 における実行サイクル数の内訳

サ数=4 の場合を述べる。図 4.7において、上端を越えている L1/L3-turb3d は、2545/40130 である。Standord とは異なり、深さ 4 以上の関数の削減率も大きな割合を占めていることが分かる。図 4.7において、上端を越えている L1/L3-turb3d は、mmrs=282/90、mmws=243/210、retvs=2/2 である。また、図 4.8 はオーバーヘッドを加えた評価である。Stanford と比較すると、test の割合が小さくなっていること、cache の割合が大きくなっていることが分かる。

4.4 考察

Puzzle および Towers のように、プロセッサ数が増えるにつれサイクル数が削減する関数がある。これは、第 3 種関数のヒット率が減少したことによる。この第 3 種関数は s_entry も再利用されるため誤って第 2 種関数と判断される。もし、第 3 種関数と判断していれば MSP は全ての RB エントリに登録できるが、第 2 種関数と判断したため、RB は m_entry、s_entry が使用できる範囲に半分ずつ分けられる。m_entry は LRU、s_entry は FIFO に従って入れ換える。s_entry よりも、それによって削除された m_entry の方が再利用効果が高かったことが原因である。また、第 3 種関数は同じエントリを繰り返し再利用する。繰り返し再利用されるはずの s_entry が FIFO によって優先的に入れ換えられたことも原因として挙げられる。ところで、一部 Towers において RB エントリ数が 256 から 512 に増えたにも関わらず、削減率が減少している原因は、RF に登録可能な read/write アドレスのエントリ数が尽きたことによる。read/write アドレスが極めて多い RB エントリが長く存在することによって、有効なエントリが減少したためである。さらに、read/write アドレスが極めて多い RB エントリが多数存在すると、RB$\Longleftrightarrow$ キャッシュ間の比較、書き込みによるオーバーヘッドが大きくなる。よって、一概には RB エントリ数は大きければよいとは言いきれないことが分かる。Stanford ベンチマークではプログラムごとに、MSP だけで再利用できるもの、SSP により高速化できるものに明確に分けることができたのに対し、SPEC FP95 では MSP だけで再利用されるものの効果しか現れなかった。これは、SPEC FP95 に第 1 種関数が少なく、第 3 種関数および第 4 種関数が多かったためと推測している。

第 5 章　おわりに

本稿では、値再利用技術が以前に演算された実行結果しか再利用できないことから、入力データが単調に変化する場合も並列事前実行による高速化手法を提案した。Stanford では、並列事前実行を行わなければ全く再利用できないプログラムも、事前実行により最大 70%近くのサイクル数が削減された。しかし、SPEC FP95 では、事前実行の効果はほとんど現れず MSP だけで再利用されるものの効果のみが現れた。これは、引数が単調に変化する第 1 種関数がほとんど現れなかったことが原因である。また、Standard では、第 3 種関数を誤って第 2 種関数と判断することにより、事前実行を行わない方が高速化されるプログラムもあった。また、再利用時のオーバーヘッドは、RB →キャッシュの比較が大部分を占めていた。

今後の研究課題として、Stride-based 予測以外の方法を適用して事前実行の効果を上げられるかどうか、事前実行を行わない方が高速化されるプログラムの場合にはいかにして事前実行を停止するか、RB →キャッシュの比較を高速化するためにいかにして高速な RB を実現するかが挙げられる。

謝辞

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

参考文献

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

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

[3] Avinash Sodani, Gurindar S Sohi:Understanding the Differences Between

Value Prediction and Instruction Reuse, MICRO31 (1998)

[4] Avinash Sodani, Gurindar S Sohi:Dynamic Instruction Reuse, 24th ISCA, pp.194-205 (1997)

[5] Kai Wang, Manoj Franklin:Highly Accurate Data Value using Hybrid Predictors, MICRO31, pp.281-290 (1998)

[6] Lipasti, M.H., Shen, J.P.: Exceeding the Dataflow Limit via Value Prediction, MICRO29, pp.226-237 (1996)

[7] Connors, D.A., Hwu, W.W.: Compiler-Directed Dynamic Computation Reuse; Rationale and Initial Results, Proc. MICRO32 (1999)

[8] Gurindar S Sohi and Amir Roth: Speculative Multithreaded Processors, IEEE Computer, Vol.34, No.4, pp.66-73, Apr. (2001)

[9] Pedro Marcuello, Antonio Gonzalez, Jordi Tubella: Speculative Multhithreading Processors, ICS'98 (1998)

[10] Pedro Marcuello, Antonio Gonzalez, Jordi Tubella: Value Prediction for Speculative Multhithreaded Architecture, MICRO33, pp269-280, Dec. (2000)

[11] 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. (2001)

[12] Jeffey T.Oplinger, David L.Heine, Monica S.Lam: In Search of Speculative Thread-Level Parallelism, PACT'99 (1999)

[13] Youfeng Wu, Dong-Yuan Chen, Jesse Fang: Better Exploration of Region-Level Value Locality with Integrated Computation Reuse and Value Prediction; ISCA'01, pp.98-108 (2001)

[14] S.Manne, A.Klauser and D.Grunward: Branch Predition Using Selective Branch Inversion, PACT'99 (1999)

[15] C.-C.Lee, I.-C.K.Chen, and T.N.Mudge: The Bi-Mode Branch Preditor, Micro30 (1997)

[16] 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)