特別研究報告書

SPARC アーキテクチャにおける
関数値再利用機構

指導教官　　富田 眞治 教授

京都大学工学部情報学科

緒方 勝也

平成 14 年 2 月 8 日

SPARC アーキテクチャにおける関数値再利用機構

緒方 勝也

内容梗概

最近の研究により、プログラムには値の局所性が存在することが指摘されている。値の局所性を利用し、データ依存を解消する高速化手法として、値予測に基づく投機実行、および値再利用がある。値予測に基づく投機実行とは、過去の実行結果を参考にして今後の結果を予測し、この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高速化手法である。ただし、予測を誤った場合に投機的に実行した結果を無効化し、再実行する必要があり、失敗時のペナルティが大きいという欠点がある。

これに対して値再利用は、一度実行した命令列の入力と出力を記憶し、再び同一命令列を実行する際に入力が同じである場合に、記憶しておいた値を用いて命令列の実行を省略する高速化手法であり、命令実行の無効化といったペナルティが発生しない。値再利用の実現方法としてはハードウェアのみによるもの、コンパイラによるもの、ハードウェアとコンパイラが協調するもの、の 3 つがある。ハードウェアのみで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、大きな効果を期待できない。コンパイラ単独によるものは、既存のハードウェアを用いることができるが、値再利用に時間がかかるという欠点がある。また、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は、基本ブロックなど、長い命令列を対象とすることができるものの、既存のロードモジュールを高速化できないという欠点がある。ただし、ハードウェアのみによる値再利用の場合でも、特定の ABI(Application Binary Interface) を仮定することにより、実行時に基本ブロックを切り出すことができると考えられる。

以上のことから、本論文では、SPARC ABI に従うプログラムを対象とし、関数を単位とすることにより、専用命令を追加することなく値再利用を実現するハードウェア機構を提案し、評価を行った。

値再利用のために必要な情報を登録するために RB(再利用表) を設け、入力値として引数と主記憶読み出しを、出力値として返り値と主記憶書き込みを記憶することとした。また、各関数ごとに主記憶から読み出した 1 つのデータと、

そのアドレスに対応するRBの全エントリとの比較を一度に行うCAM機構を想定し、主記憶読み出しと主記憶書き込みのアドレスを管理するためのRF(関数管理表)を設けた。さらに、入れ子の関数の一括再利用を可能とするために、どの関数がどの関数を呼び出したかの情報を記憶するRW(再利用ウィンドウ)を設けた。

RBエントリは、ヒット回数の小さいもの、RFエントリはヒット回数と省略できるステップ数の積の小さいものをそれぞれ優先して追い出すような置き換えアルゴリズムとした。また、ヒット回数が多いものの、しばらくの間使われていないエントリが残るのを防ぐために、ヒット回数はRBやRFの参照回数が一定に達するごとに初期化するようにした。

以上のような関数値再利用機構を有し、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したSPARCシミュレータを開発し、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においては、Permでは8.2%、Puzzleでは63.3%、Towersでは51.3%、SPEC CINT95においては、124.m88ksimでは23.1%、126.gccでは12.6%、130.liでは23.9%、134.perl(primes)では20.1%、147.vortexでは27.6%の高速化を実現できることが明らかになった。この結果より、値再利用は繰り返し処理を伴なうインタプリタやシミュレータ、値の局所性が存在するオブジェクト指向プログラムなどに適していることも分かった。

また、プログラムによっては値再利用によってウィンドウミスが最大で20%ほど減らせることや、値再利用がキャッシュミスに大きな影響を及ぼさないこと、ウィンドウミスが大幅に減少しているプログラムは全体としての値再利用の効果も高いという相関関係があることも明らかになった。その一方で、値再利用の効果はベンチマークによってばらつきがあり、効果が全くないプログラムもあること、また、入力値のパターンが膨大である画像処理や、入力が変化し続けるゲームなどのプログラムには効果が小さいという欠点も明らかになった。

過去の実行結果を記憶するだけの単純な再利用では高速化できないプログラムの処理を高速化するためには、入力値を予測して関数を投機的に実行し、その結果を前もってRBに登録しておくなどの手法が考えられる。このような改良については今後の課題である。

Function Level Reuse on SPARC Architechture

Katsuya OGATA

Abstract

Recent studies have demonstrated that value locality presents in many programs. Computation reuse and value prediction techniques are proposed for improving microprocessor performance by exploiting value localities and collapsing data dependencies. Value prediction is a technique that reduces the waiting time by executing next instructions speculatively with the predicted input values based on previous results. Though, if the prediction turns out to be incorrect, speculative results should be squashed and all instructions should be re-executed.

On the other hand, computation reuse is a technique that holds the inputs and outputs of previous execution instances of a computation. When the same computation is encountered later with same inputs, the held results are retrieved and used directly without actually executing the computation. Then, no penalties like cancellation are occured. For the implmentation of reuse, there are three ways. First, hardware implementation only. Second the compiler generates the objects for reuse. Third, compiler supports the special purpose instructions for reuse that are executed by hardware directly. By hardware implementation only, recognizing the basic block is very difficult. And since its target is a single instruction only, only few cycles is reused. By compiler, existing hardware can be used, but computation reuse takes long time. By compiler and hardware, instructions as basic block can be reused, but existing load modules are out of services. Even if by hardware implementation only, by supposing specific ABI(Application Binary Interface), the basic block would be recognized when executing.

For these reasons, this paper proposes function level reuse machanism on SPARC architecture without any special purpose instructions and evaluates its effects.

RB(Reuse Buffer) holds informations required for reuse. Arguments and memory read data are stored as inputs, and return values and memory write data are stores as outputs of each function. Then, I supposed CAM mechanism

for comparison all RB entry about one address in every function and memory read data. And RF holds memory read/write address. RW holds relations between caller and callee functions.

The minimum referenced and reused RB entry is replaced first. And the RF entry that the product of reused count and reduced steps is minimum is replaced first. In order to prevent the entry which is not used for a long time are remaining, hit counters are cleared when RB or RF access reaches to a constant.

Considering instruction latency, cache miss penalty, window miss penalty and reuce latency, I developed a cycle simulator and evaluated the reuse mechanism. On the Stanford-integer, 8.2% of cycles of Perm are reduced, 63.3% of Puzzle and 51.3% of Towers. And on the SPEC INT95, 23.1% of 124.m88ksim, 12.6% of 126.gcc, 23.9% of 134.perl(primes) and 27.6% of 147.vortex. From these results, computation reuse is suitable for interpreter or simulator programs which include repeting computing, or object-oriented programs which include value locality.

Moreover, the following facts were found. 1)20% of the window miss overhead will be reduced. 2)Reuse does not affect for cache miss overhead. 3)On the programs on which reuse can reduce many window miss overhead, many net cycles would be reduced.

On the other hand, the effects of reuse are not stable and on some benchmark, no cycle will be reduced. And it is declared that this implementation cannot reduce many cycles on image processing programs, whose imputs take various values, and game programs, in which the input pattern changes coutinuously.

In order to reduce cycles of these programs, executing functions speculatively based on input prediction and registering its result in RB will be effective. The evaluation of such technique is the future work.

SPARC アーキテクチャにおける関数値再利用機構

目次

第 1 章　はじめに　1

第 2 章　従来の投機実行と値再利用　2

2.1　値予測に基づく投機実行 2

2.2　値再利用 3

第 3 章　SPARC ABI(Application Binary Interface) の概要　4

3.1　レジスタウィンドウ 4

3.2　スタック 5

3.3　関数実行 6

3.3.1　関数呼び出し 6

3.3.2　関数本体の実行 7

3.3.3　復帰 7

3.4　SPARC ABI を利用した値再利用 7

3.4.1　引数 7

3.4.2　主記憶読み出し 8

3.4.3　主記憶書き込み 8

3.4.4　返り値 8

第 4 章　関数値再利用機構の提案　9

4.1　再利用表の構成 9

4.2　再利用機構の動作 11

4.2.1　関数呼び出し 11

4.2.2　関数本体の実行 11

4.2.3　復帰 11

4.2.4　再利用表のエントリ置き換えアルゴリズム 12

第 5 章　評価　12

5.1　ハードウェア構成の仮定 12

5.2　ベンチマークプログラムの概要と予想される結果 13

5.2.1　Stanford-integer 13

5.2.2 SPEC CINT95 15
5.3 測定結果 17
5.3.1 Stanford-integer 17
5.3.2 SPEC CINT95 18
5.4 考察 20
第 6 章 おわりに **21**
謝辞 **22**
参考文献 **22**

第 1 章　はじめに

プロセッサの高速化の妨げになる要因には、データ依存や制御依存がある。これらを解消するために、スーパースカラや VLIW が考案され、データ依存はレジスタリネーミングやレジスタ割り付けの工夫、また、制御依存は分岐予測や条件付き実行などにより軽減されてきた。一方、最近の研究では、プログラムには値の局所性が存在することが指摘されている [1, 2]。具体的には、load 命令が同じ値を取り出すことや、store 命令が同じ値を書き込むことである。このような命令は、本来、コンパイラにより削除されるべきであるものの、完全に検出することは、まだできていない。

値の局所性を利用してデータ依存を解消する高速化手法には、値予測に基づく投機実行 [1, 2, 3, 4, 5] や値再利用 [6, 7, 8, 9, 10, 11] があり、多くの研究が行われている。また、その両方を組み合わせた手法についても研究されている [12, 13]。

値予測に基づく投機実行とは、過去の実行結果を参考にして今後の結果を予測し、この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高速化手法である。予測値が正しければサイクル数が減少する。しかし、誤った場合には投機的に実行した結果を無効化し、再実行する必要があるため、失敗時のペナルティが大きい。

これに対して値再利用は、一度実行した命令列の入力と出力を記憶し、再び同一命令列を実行する際に入力が同じである場合、記憶しておいた出力値を用いて命令列の実行を省略し、高速化する。値予測に基づく投機実行と異なり、失敗時にペナルティが発生しない。値再利用の実現方法は、ハードウェアのみによるもの [8]、コンパイラが既存の命令を用いて値再利用を行うオブジェクトを生成するもの、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 [7, 9]、の 3 つに分類できる。再利用の単位としては、単一命令 [6, 8] や、基本ブロック [9, 11] が一般的である。

本論文では、SPARC アーキテクチャにおいて、関数を単位とする値再利用を専用命令なしに実現するハードウェア機構の提案および評価を行った。第 2 章では、従来の投機実行と値再利用について述べる。第 3 章では SPARC ABI の概要を述べる。第 4 章では SPARC ABI を利用する関数値再利用機構の構成および動作について述べ、第 5 章ではハードウェア構成の仮定、評価結果および

考察について述べる。

第 2 章　従来の投機実行と値再利用

本章では値予測に基づく投機実行と値再利用を比較しながら、従来技術を説明する。

2.1　値予測に基づく投機実行

命令アドレスに関して投機的手法を適用する高速化手法として、分岐予測が挙げられる。分岐命令がどちらに分岐するかを予測し、その予測に基づいて後続する命令を投機的に実行する。命令分岐の方向には非常に偏りがあり、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 [14]。単純なハードウェア機構により実現でき、大きな効果が得られるため、多くの商用プロセッサが採用している。

値予測に基づく投機実行とは、アドレスではなくデータ値そのものに着目する高速化手法である。具体的には過去の履歴に基づいて命令列の実行結果を予測する。予測値は後続命令の入力となり、後続命令列が投機的に実行される。予測の単位には、1 命令や基本ブロックが多く用いられる。予測方法には、最近の実行結果をそのまま使う Last-Value 予測 [4]、最近の実行結果の差分を用いて外挿するストライド予測、最近の 4 種類の演算結果と過去の予測結果を用いる 2 レベル値予測、これらを組み合わせたハイブリッド予測などがある [3]。

値予測に基づく投機実行は、データ依存を軽減する [2]。従って、データ依存が問題となるスーパースカラや VLIW などの技術と組み合わせることにより、大きな効果を発揮する [5]。予測値が正しければ、後続命令が先行命令を待つ時間が短縮される。しかし、予測が誤った場合には、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やり直す必要がある。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。また、予測値が正しかったかどうか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がある。

2.2 値再利用

値再利用とは、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しようとした時に、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。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。

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　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値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持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　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、値再利用の単位は、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比較を行って再利用可能かどうかの判定を行い、再利用が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書き込む。演算や主記憶読み出しを削減することによる高速化が期待できる。ただし、一度に数サイクル程度の高速化にとどまり、大幅な高速化は望めない。全ての命令を対象としたもの [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。

命令列を単位とする方法　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 (レジスタおよび主記憶へ書き込んだ値) を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、入力値が全て同じであるか否かを比較し、一致した場合、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と異なり、一度に複数の命令の処理を省略することができる。

ところで、命令列の単位には、基本ブロックが考えられる。しかし、ハードウェアによる基本ブロックの切り出しは難しい。本研究では専用命令を追加することなく、多くの命令を含む区間を容易に特定するために、SPARC ABI を利用し、関数を再利用の単位とすることにした。

第 3 章　SPARC ABI(Application Binary Interface) の概要

本章では、関数呼び出しに関する SPARC ABI[15] の概要について述べ、関数呼び出しがどのように行われるかについて説明する。また、いか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

プログラムは常に以下の 4 種類のレジスタを使用することができる。

global レジスタ (%g0–7)　どの関数からも常にアクセスできるレジスタであり、大域変数の格納に使われる。%g0 の内容は常に 0 である。

out レジスタ (%o0–7)　%o0–5 は引数を渡すため、また、作業用に用いられる。%o0 は返り値を受け取るために用いられる。%o6(%sp) 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、関数を呼び出した call 命令もしくは jmpl 命令のアドレスを保存する。

local レジスタ (%l0–7)　全てローカル変数や作業用に用いられる。

in レジスタ (%i0–7)　%i0–5 は関数が引数を受け取る時に用いられる。%i0 および%i1 は返り値の格納場所としても用いられる。%i6(%fp) 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、自関数を呼び出した call 命令のアドレスを保持する。

SPARC アーキテクチャでは、関数呼び出し時のパラメータの受け渡しに際して、主記憶を介す必要がないよう、レジスタウィンドウを規定している。図 1に示すように、各ウィンドウは%i、%l、%o から構成され、%i は隣接ウィンドウの%o と同一、また、%o は反対側の隣接ウィンドウの%i と同一のレジスタとなっている。%l 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。図 1では CWP がインクリメントされるとウィンドウが#0 から#1 に切り替わる。以前に%o として参照したレジスタは%i となり、以前の%l と%i は使用できなくなる。代わりに、新しく%l と%o が割り当てられる。CWP がデクリメントされると、逆に以前の%i が%o となり、以前の%l と%o の代わりに新しい%l と%i が割り当てられる。

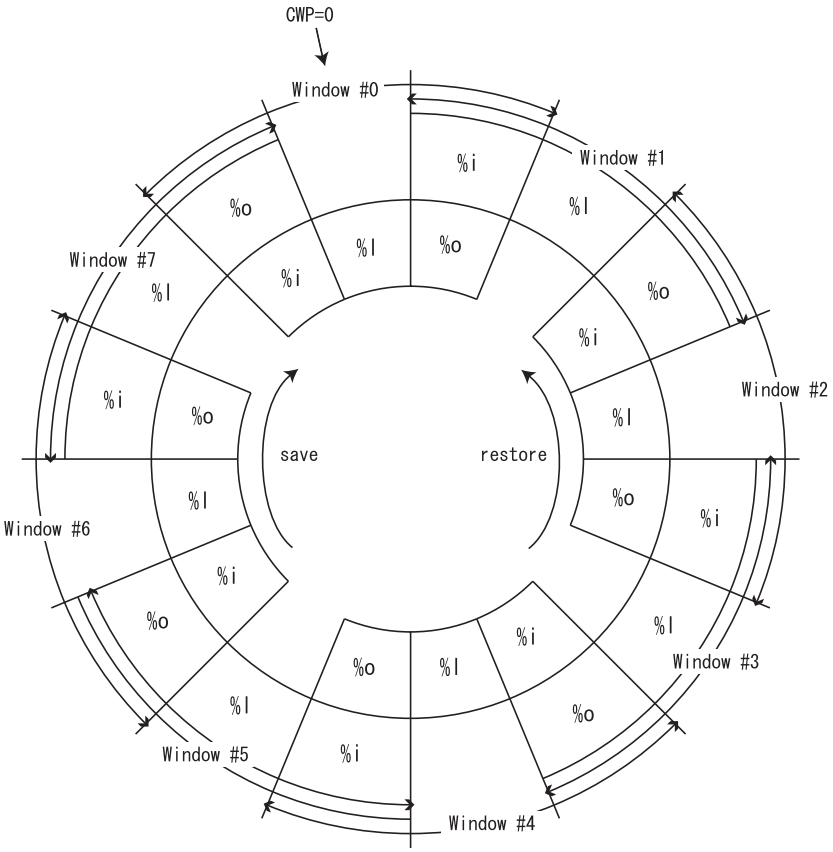

図 1: レジスタウィンドウの構成 (ウィンドウ数=8 の場合)

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があふれ、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込みが発生し、レジスタの内容が後述するスタックへ保存される。また、restore 命令が続くと、アンダフロー割込みが発生し、スタックに保存された値がレジスタに戻される。オーバフローとアンダフローに際しては主記憶参照が生じるため、プログラムの実行が遅れる。本論文では入れ子の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アンダフローによる性能の低下を抑えることを狙っている。

3.2 スタック

スタックは以下の用途に用いられる。

- ウィンドウオーバフローが起こった時にレジスタの値を保存する。
- 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。

- 引数の一時退避
- 関数呼び出しの際に第 7 ワード以降の引数を格納する。
- ローカル変数を格納する。

スタックは、主記憶の上位アドレスから下位アドレスに向かって伸びる。有効なスタックの下限アドレスは、スタックポインタと呼ばれる%o6(%sp) に格納される。図 2に示すように、save 命令が実行されると、積まれる関数フレームの大きさ分だけ%o6(%sp) は減少し、元の%o6(%sp) の値はフレームポインタと呼ばれる%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、大域変数のためのデータ域とスタックのためのデータ域には各々上限が設けられる。そのため、主記憶上においてこれら 2 つの領域には境界が存在する。この境界を LIMIT と呼ぶことにし、以後、大域変数と局所変数の区別のために用いる。

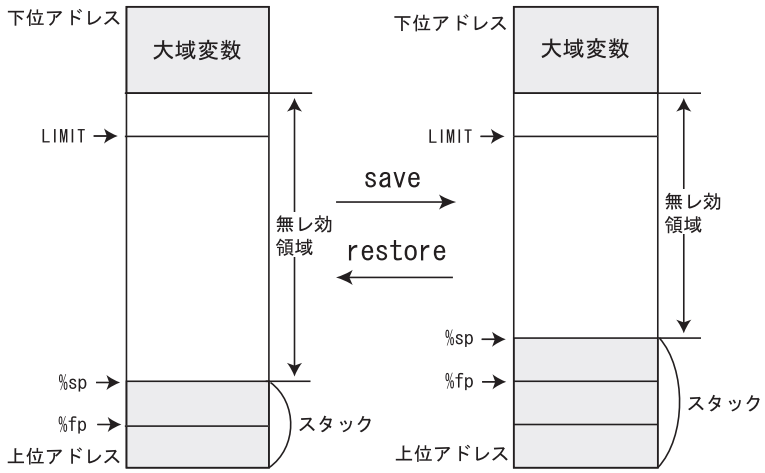

図 2: スタック

3.3 関数実行

関数単位の値再利用を説明するために、引数および命令アドレスの受け渡し方法について述べる。

3.3.1 関数呼び出し

関数は、call 命令、または、第 2 オペランドが%o7 である jmpl 命令により呼び出される。現在の PC(Program Counter) の値が%o7 に格納され、PC が関数の先頭アドレスに書き換えられる。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。命令の実行順は、1)call もしくは jmpl 命令 ;2)call もしくは jmpl の次アドレスの命令 ;3) 関数の先頭の命令 ;となる。引数は%o0–5 に入る。引数が 7 ワード以上ある場合には、スタックが使用される。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。以降の引数も同様にスタックに積まれる。

3.3.2 関数本体の実行

Leaf 関数を除く関数は save 命令を含む。save 命令の実行により、%i0–5 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格納される。第 7 ワード以降の引数は%fp+92 以降に格納される。

3.3.3 復帰

Leaf 関数を除く関数は、第 1 オペランドが%i7+α (α > 0) であり、第 2 オペランドが%g0 である jmpl 命令、および、それに続く restore 命令により終了する。この時、%i7 には関数呼び出し時の PC の値が格納されている。%g0 は常に 0 であるため、この jmpl 命令では現在の PC の値は保存されない。返り値は、1 ワードの場合%i0、2 ワードの場合%i0–1 に格納される。restore 命令の実行により、返り値は%o0 および%o1 に入る。なお、浮動小数点数の場合は%f0 および%f1。拡張倍精度浮動小数点数の場合はさらに%f2 および%f3 に入る。

ところで、関数呼び出しを持たない leaf 関数は、前述の関数とは構成が異なる。save 命令および restore 命令はなく、また、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%o0–5 のままで用いられ、返り値は%o0 および%o1 に格納される。

3.4 SPARC ABI を利用した値再利用

本節では、SPARC ABI を利用して、どのようにして値再利用を行うかについて述べる。ただし、以下は、対象となる関数が leaf 関数でない場合の記述である。対象関数が leaf 関数の場合には、%i を%o に、%sp を%fp にそれぞれ読み換える。

3.4.1 引数

呼び出された関数側から見ると、第 6 ワードまでの引数は%i0–5 に、第 7 ワード以降の引数は%fp+92 以降に入っている。また、返り値が構造体である場合には、ポインタが%fp+64 に入っている。関数内において、該当レジスタやスタックからの読み出しを行った場合、引数の可能性がある。一方、値が読み出され

る前に書き込んだ場合は引数ではない。ただし、第 7 ワード以降の引数の参照は%fp 相対で行われるとは限らないため、上位関数の局所変数との区別がつかない。そこで上位関数によって%sp+92 以降に書き込まれた場合に第 7 ワード以降の引数が存在すると判断する。本論文では、簡単のためと、引数が 7 ワード以降に及ぶ場合は少ないとの考えから、引数が 7 ワード以上の場合は関数値の再利用を適用しないこととした。

3.4.2 主記憶読み出し

一般に、主記憶からの読み出しデータは関数の実行に影響を及ぼす。しかし、以下に示すアドレスに対する主記憶読み出しデータは、再利用にとって不要である。

LIMIT 以上、呼び出し時%sp+92 未満のアドレス 自関数や下位関数の局所変数、または返り値である構造体へのポインタである。前者は再利用自体に不要な情報であり、後者は引数として登録するので、いずれも主記憶読み出しデータとしての登録は不要である。なお、この判別を可能とするために、RB は各関数呼び出し時の%sp の値を記憶しておく必要がある。

一度書き込みがあったアドレス 読み出された値は関数の実行によって生成された値であり、不要である。

3.4.3 主記憶書き込み

一般に、主記憶への書き込みは関数の実行結果である。しかし主記憶読み出し同様、以下に示すアドレスに対する書き込みは再利用にとって不要である。

LIMIT 以上、呼び出し時%sp+92 未満のアドレス 主記憶読み出し同様、自関数や下位関数の局所変数、または返り値である構造体へのポインタであり、主記憶書き込みデータとしての登録は不要である。

3.4.4 返り値

本論文では、返り値が倍精度浮動小数点数である関数は存在しないと仮定した。この場合、呼び出された関数からの返り値は%i0–1、または%f0–1 に格納される。該当するレジスタへの書き込みがある場合、返り値の可能性がある。返り値が構造体の場合にはポインタが%fp+64 に格納されるが、この値は関数呼び出し時に渡されたものであり、引数として登録する。また、構造体の本体は主記憶上に存在しているため、書き込みが起こった際に、主記憶書き込みデータとして登録される。

第 4 章　関数値再利用機構の提案

本論文では 3 章に述べた SPARC ABI に基づいて記述されたプログラムに対し、関数単位の値再利用を適用する機構を提案する。本機構は関数の入出力情報を再利用表に記憶し、同一入力値により同一関数が呼び出された場合に関数の処理を省略する。本章では、再利用表の構成と再利用機構の動作について述べる。

4.1　再利用表の構成

関数の入力および出力情報を記憶しておく表が再利用表である。再利用表は、再利用ウィンドウ (RW)、関数管理表 (RF)、再利用表本体 (RB) から構成される。

再利用ウィンドウ (RW) は、入れ子の関数を再利用するために、どの関数がどの関数を呼び出したかを保持する。各エントリは RF および RB の各エントリのインデックスを保持する (図 3)。

関数管理表 (RF) は、RB に登録されている関数のアドレス、RB エントリに共通する主記憶読み出しアドレスおよび書き込みアドレスを保持する (図 4)。各エントリ先頭のフラグ V は、各エントリの空き状況を表す。LRU カウンタは、後述する RF エントリの置き換えのために用いられる。

再利用表本体 (RB) は、入力である引数と主記憶読み出しデータ、出力である

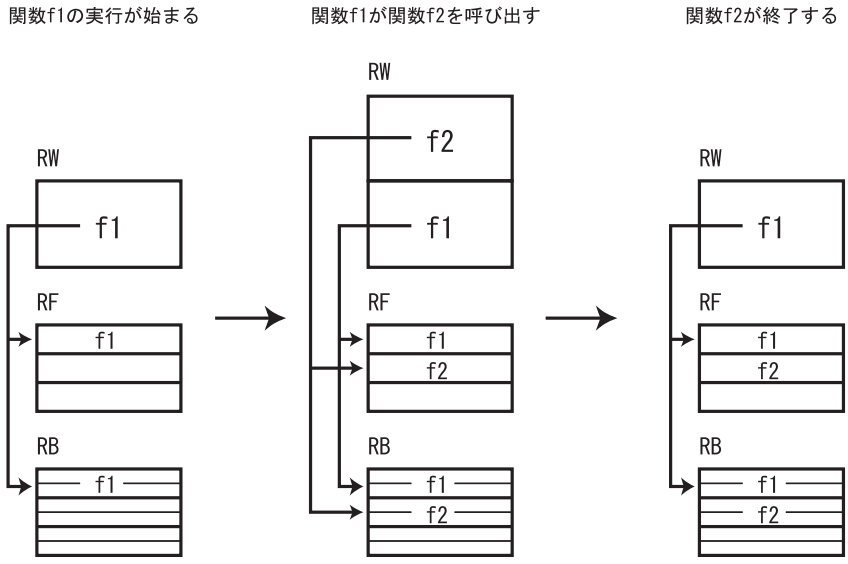

図 3: 再利用ウィンドウ (RW)

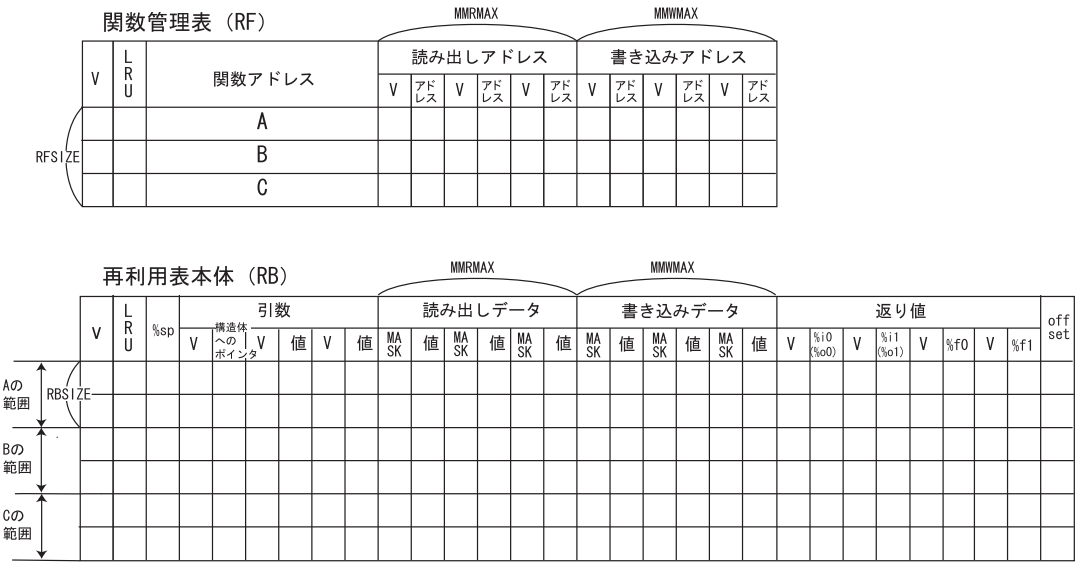

図 4: 関数管理表 (RF) と再利用表本体 (RB)

返り値と主記憶書き込みデータ、および、関数が呼び出された時の%sp の値を保持する (図 4)。各エントリ先頭のフラグ V は各エントリが 登録可能」 登録中」 登録済」のどの状態にあるかを表す。1 つの関数あたりの RB エントリ数は一定であり、関数の並び順は RF と一致する。また、主記憶読み出しおよび書き込みデータのアドレスは RF が保持しており、同一関数におけるアドレスは列ごとに同一である。それぞれのデータのサイズは 4 バイトであり、主記憶読み出しデータと書き込みデータの有効バイトを MASK 値によって表す。offset に復帰先アドレスのオフセットを保存する。復帰先アドレスは関数を呼び出した call もしくは jmpl 命令のアドレスに offset 値を加えたものとなる。LRU カウンタは後述する RB エントリの置き換えのために用いられる。

再利用表に関するパラメータを以下に列挙する。それぞれ、RFSIZE:RF が保持するエントリ数の最大 ;RBSIZE:RB が 1 関数あたり保持するエントリ数の最大。RB のエントリの総数は RFSIZE*RBSIZE 個となる ;RFPURGETIME:RF の LRU カウンタをクリアする間隔。RF の参照回数がこの値に達するごとにカウンタがクリアされる ;RBPURGETIME:RB の LRU カウンタをクリアする間隔。RB の参照回数がこの値に達するごとにカウンタがクリアされる ;MMRMAX:RB1 エントリが保持する主記憶読み出しデータの最大数 ;MMWMAX:RB1 エントリが保持する主記憶書き込みデータの最大数 ;である。

4.2　再利用機構の動作

本節では、再利用機構の動作、および、RF と RB エントリの置き換えアルゴリズムについて述べる。

4.2.1　関数呼び出し

関数を呼び出す call もしくは jmpl 命令が実行されると、再利用機構は関数アドレスが RF に登録されているかどうか調べる。関数アドレスが登録されていない場合は、RF に登録する。全ての RF が埋まっている場合には、4.2.4に示す置き換えアルゴリズムに従って既存のエントリを追い出す。登録されていた場合には、入力値条件である引数および主記憶読み出しデータが一致する RB エントリが存在するかどうかを調べる。該当するエントリがあった場合には、RB に登録されている返り値および主記憶書き込みデータに従って、レジスタおよび主記憶の値を書き換え、PC の値を復帰先アドレスに書き換える。

再利用できない場合には、まず、状態が　登録可能」である RB を探して初期化し、状態を　登録中」に変える。全てのエントリが　登録済」の場合は、4.2.4 に示す置き換えアルゴリズムに従って既存のエントリを追い出す。次に、%sp の値を RB に登録する。最後に、登録を開始した RB エントリのインデックスを RW に積む。RW が一杯になった場合は、最も上位の関数を RW から削除し、その関数の登録を打ち切り、RB エントリの状態を　登録可能」に変える。

4.2.2　関数本体の実行

関数の再利用ができない場合、引き続いて関数本体の実行を開始する。レジスタや主記憶を参照するたびに、3.4 に示した判別方法に従って入力値や出力値を RB へ登録する。ただし、主記憶の参照は上位関数の入出力でもあるので、RW にインデックスが保持されている全ての RB について、登録が必要かどうかの判別を行い、必要であれば登録する。また、第 7 ワード以降の引数がある場合や、trap 命令によってシステムコールが起こった場合には、RW にインデックスが保持されている全ての RB エントリの登録を打ち切り、それぞれのエントリの状態を　登録可能」に変える。

4.2.3　復帰

第 1 オペランドが%i7(leaf 関数の場合は%o7) であり、第 2 オペランドが%g0 である jmpl 命令によって関数呼び出しが終了すると、RB への登録は完了する。まず、RB の状態を　登録中」から　登録済」に変える。次に、登録が完了した

RB エントリのインデックスを RW から降ろす。この手続きが完了すると登録したエントリは有効となり、以降、再利用に用いることができる。

4.2.4　再利用表のエントリ置き換えアルゴリズム

RB もしくは RF にエントリを追加していくと、いずれ RF や RB は一杯になる。さらに登録するためには、既存のエントリを追い出す必要が生じる。本節では、再利用機構における置き換えアルゴリズムについて述べる。

全ての RF エントリには LRU カウンタが付いている。LRU カウンタの初期値は 0 であり、その RF エントリが保持する関数の RB のヒットによってインクリメントされる。また、最後に登録もしくは使用した RB エントリのステップ数も保持している。ヒット回数の多いエントリや省略できるステップ数の多いエントリは有用であると考え、LRU カウンタとステップ数の積が最小のものを優先して追い出す。しかし、しばらくヒットしていない関数は今後もヒットする可能性が低いと判断し、RF への参照が一定回数 (RFPURGETIME) に達すると LRU カウンタをクリアする。

同様に、全ての RB エントリにも LRU カウンタがある。LRU カウンタの初期値は 0 であり、RB エントリのヒットおよび登録によってインクリメントされる。ヒット回数の多いエントリは有用であると考え、LRU カウンタの値が最小のものを優先して追い出す。しかし、RF の場合と同様の判断から、同一関数である RB エントリの参照が一定回数 (RBPURGETIME) に達すると LRU カウンタをクリアする。

第 5 章　評価

5.1　ハードウェア構成の仮定

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 を用いることを想定し、ハードウェア構成を図 5のように仮定した。呼び出されようとする関数が再利用可能かどうかの判定および再利用の手順は以下の通りである。

1. 関数アドレスが一致する RF エントリを探す。
2. 引数が全て一致する RB エントリを探す。
3. 少なくとも一つの MASK 値が有効である RF の読み出しアドレスについて、RB の値と主記憶の比較を行う。

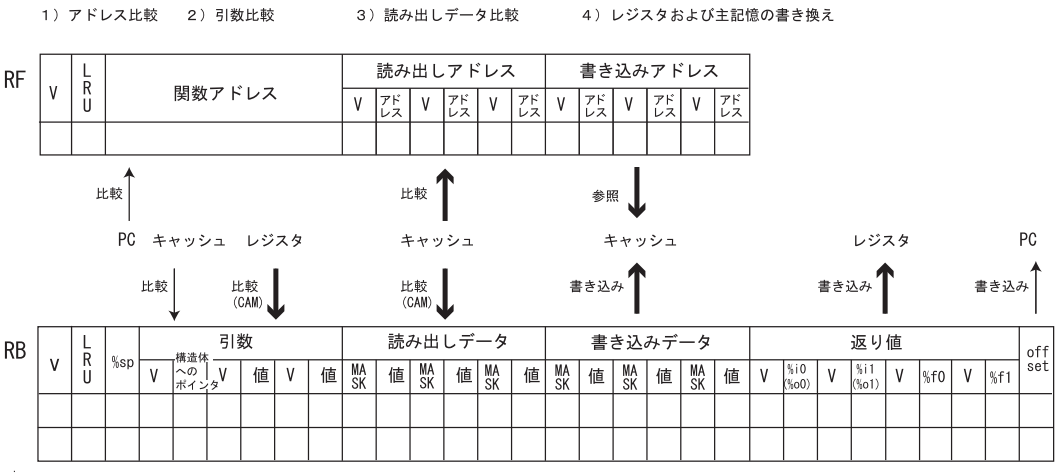

図 5: 再利用表のハードウェア構成

4. MASK 値が有効である全ての値が一致した場合には、書き込みデータと返り値を主記憶およびレジスタに格納する。
5. PC の値を関数を呼び出した call もしくは jmpl 命令のアドレスに offset の値を加えたものとする。

以上より、これらの動作に要するサイクル数を表 1のように仮定した。このようなシミュレータを作成し、評価を行った。

5.2 ベンチマークプログラムの概要と予想される結果

評価には Stanford-integer および SPEC CINT95 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。以下では、各プログラムの概要および予想される再利用の効果について述べる。

5.2.1 Stanford-integer

Stanford-integer は以下の 10 種類のサブルーチンから成るプログラムである。Stanford-integer のそれぞれのサブルーチンを一つのプログラムとしてコンパイルし直し、10 個のプログラムとして測定を行った。ただし、FFT と Queens には同じ処理をそれぞれ 20 回と 50 回繰り返すループがあり、そのままでは無意味に再利用の効果が高くなるため、両方とも処理を 1 回だけ行うようにプログラムを変更した。以下にそれぞれのプログラムの概要および予想される再利用

表 1: シミュレータの各パラメータ

データキャッシュ	
容量	64Kbyte
ラインサイズ	64byte
ウェイ数	4
キャッシュミスペナルティ	20cycle
レジスタウィンドウ	
ウィンドウ数	6
ウィンドウミスペナルティ	20cycle
命令レイテンシ	
ロードレイテンシ	2cycle
整数乗算 〃	8cycle
整数除算 〃	70cycle
浮動小数点加減演算 〃	4cycle
単精度浮動小数点除算 〃	16cycle
倍精度浮動小数点除算 〃	19cycle
再利用表	
RW の最大数	6
引数比較	1cycle
主記憶読み出しデータとキャッシュの比較	4byte/cycle
主記憶書き込みデータからキャッシュへの書き込み	4byte/cycle
返り値書き込み	1cycle

の効果について述べる。

Perm　再帰的に配列上の値の入れ替えを行う。配列上の 2 値を入れ替える関数が何度も呼ばれる。2 値の場所と値が一致すれば再利用が可能であり、ある程度は再利用の効果があると予想される。

Tower　ハノイの塔問題を解く。ディスクを置く関数および持ち上げる関数が何度も呼ばれる。ディスクの大きさと操作しようとする塔の状態 (一番上にあるディスクの大きさ、積まれているディスクの数) が一致すれば再利用が可能であり、十分に再利用の効果が現れることが予想される。

Queen　8 つのクイーンをお互いが取れない位置に置く 8 クイーン問題を解く。クイーンを置けるかどうかの判定する関数が何度も呼ばれる。置けない場所の状況が一致すれば再利用が可能である。しかし、チェス盤はさほど大きくないので、再利用ができたとしても大きな効果は望めないと予想される。

Intmm,Mm　それぞれ整数、浮動小数の行列積を計算する。計算結果は新しい 2 次元配列に格納されていくため、再利用の余地はないと予想される。

Puzzle　立方体空間の中に与えられたピースをつめていくパズルを解く。成功するまで試行錯誤するため、ピースをつめる関数と外す関数が何度も呼ばれる。ピースの種類とつめようとする場所の状況が一致すれば再利用が可能である。同じ操作が繰り返され、再利用の効果が大きく現れることが予想される。

Quick　クイックソートを行う。ソートは関数を再帰的に呼び出すことによって行われるが、同じ数列を繰り返しソートすることはなく、再利用の効果はないと予想される。

Tree　ヒープソートを行う。ノードをツリーの根に追加してソートする関数が繰り返し呼ばれる。しかし、ノードを追加する時点でのツリーは常にソートされた状態であり、同じ状態に同じ値のノードを追加することがありえず、再利用の効果はないと予想される。

Bubble　バブルソートを行う。関数呼び出しを伴わずにソートを行うため、再利用の効果はないと予想される。

FFT　高速フーリエ変換を行う。一つの関数が呼ばれるだけであり、再利用の効果はないと予想される。

5.2.2　SPEC CINT95

SPEC CINT95 は以下の 8 種類のプログラムから成る。それぞれのプログラムは ref,train,test の 3 つから成り、ref が最も実行時間が長く、test が最も短い。本論文ではシミュレーション時間短縮のために test を用いた。

099.go　国際ランクの囲碁のプログラム。隣接する空白領域や安全な領域、危険な領域、隣接する領域、目などの状況をリストで記憶しており、そのリストの操作 (追加、削除、全消去、コピー、マージ) が処理の半分以上を占める。再利用のためにはリストの状態が、追加や削除では一部、それ以外では全部一致する必要があるため、再利用できる局面が少なく、効果は小さいと予想される。

124.m88ksin　モトローラ 88100 マイクロプロセッサのシミュレータ。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。前者の入力は実行中の命令の状態であり、出力は待つべきサイクル数であるため、再利用が可能である。後者の入力は命令のアドレスであり、出力は判定結果である。ブレークポイントが存在した場合は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。存在しなかった場合は再利用が可能である。シミュレータ上で実行するプログラムにループが多い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が、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。

129.compress　メモリ上でのファイルの圧縮および伸張を行う。主記憶の初期化が実行命令数の大部分を占める。この関数は 69000 個の大域変数を一度に初期化するため、RB に登録することができず、再利用できない。次に実行命令数が多いのは、コードを一つ読み込んでそのコードが EOF かどうかを判定し、EOF でなければ入力をそのまま返す関数である。入力となるコードの種類はさほど多くないため、この関数に対しては十分に再利用の効果があると思われる。全体としてある程度の効果が予想される。

130.li　Lisp インタープリタ。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。文字列の読み込みは trap 命令を伴うため、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り、再利用が可能である。全体としてはある程度の効果が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張を行う。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。その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と予想される。

134.perl　Perl のインタプリタ。複数の文字列が与えられ、それぞれの文字列の文字の順番をランダムに入れ替える jumble スクリプトと素数を求める

prime スクリプトを実行する。malloc 関数が多く呼ばれる。malloc 関数は繰り返し呼ばれた場合であっても結果は大域変数の領域の状態に依存するため、引数が同じでも返り値は一致せず、再利用できない。プログラム全体の再利用の効果はインタプリタ上で実行するスクリプトの種類に依る。jumble では主記憶上の値の入れ替えが頻繁に起こるが、単一文字列が対象でないため、あまり再利用の効果はないと予想される。prime では入力に対して正規表現を用いて数字の部分だけを抽出してから素数の判定を行う。入力される文字列は全て同じようなパターンであるために、数字の抽出の部分で再利用が可能であり、多少の効果が予想される。

147.vortex オブジェクト指向データベース。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。よってある程度の効果が予測される。

5.3 測定結果

5.3.1 Stanford-integer

まず、サイズの小さい Stanford-integer を用いて計測を行った。FFT,Mm,Intmm, Bubble,Quick,Tree については予想した通り全く効果がなかったため、グラフは省略した。

まず、RBSIZE および RBPURGETIME と削減率の関係について示す。RB 内に記憶できるエントリ数である RBSIZE、および置き換えアルゴリズムにおいて LRU カウンタをクリアする間隔である RBPURGETIME と削減率の関係について評価を行った。結果を図 6–9に示す。RBSIZE,RBPURGETIME 以外のパラメータは、RFSIZE=32,MMRMAX=1024,MMWMAX=1024 とした。評価には、再利用を行わなかった場合に要するサイクル数に対する、省略できるサイクル数の比を用いた。

Stanford-integer では、RBSIZE が 256 あれば十分な効果がある。また、この結果から RBPURGETIME は RBSIZE の 2 倍とすることにした。以下では RBPURGETIME は RBSIZE の 2 倍とする。この結果に倣い、RFPURGETIME も RFSIZE の 2 倍とした。

次に MMRMAX,MMWMAX と削減率の関係を調べた。MMRMAX と削減率の関係を図 10に、MMWMAX と削減率の関係を図 11に示す。MMRMAX, MMWMAX 以外のパラメータは RFSIZE=32,RBSIZE=512 とした。

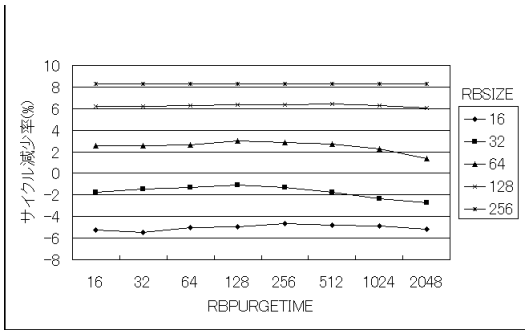

図 6: Perm

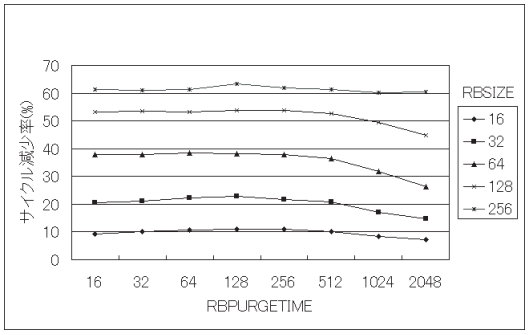

図 7: Puzzle

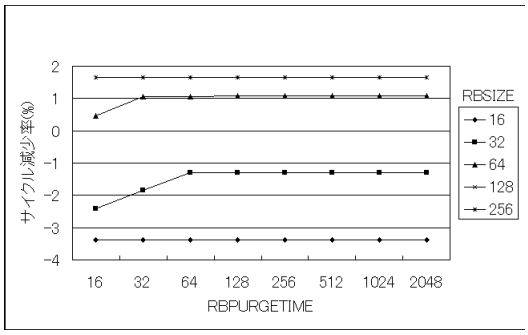

図 8: Queens

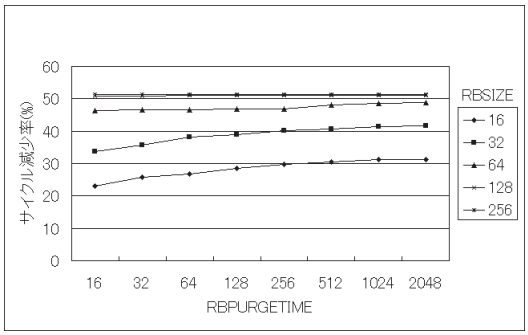

図 9: Towers

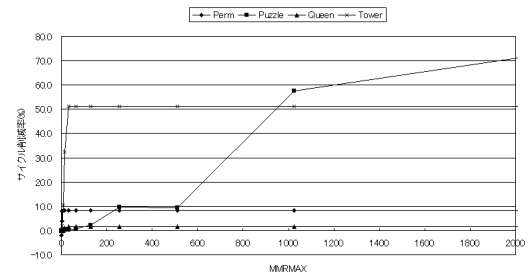

図 10: MMRMAX と削減率の関係

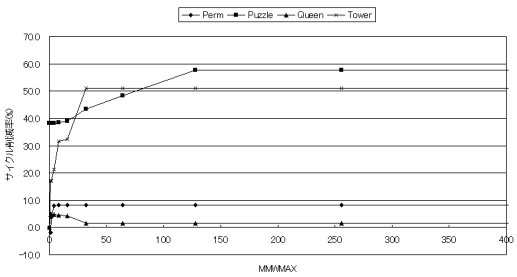

図 11: MMWMAX と削減率の関係

Stanford-integer においては MMRMAX は 150 ほどあれば十分な効果がある。また、MMWMAX は Puzzle 以外では 50 ほどあれば十分であるが、Puzzle 関数において十分な効果を出すためには、1000 ~ 2000 ほど必要となることが分かった。

5.3.2 SPEC CINT95

続いて SPEC CINT95 による評価を行った。Stanford-integer の結果を踏まえ、RFSIZE=32,RBSIZE=256 とした。MMRMAX と MMWMAX を等しくして変化させた時の削減率との関係を図 12に示す。値は Stanford-integer の場合

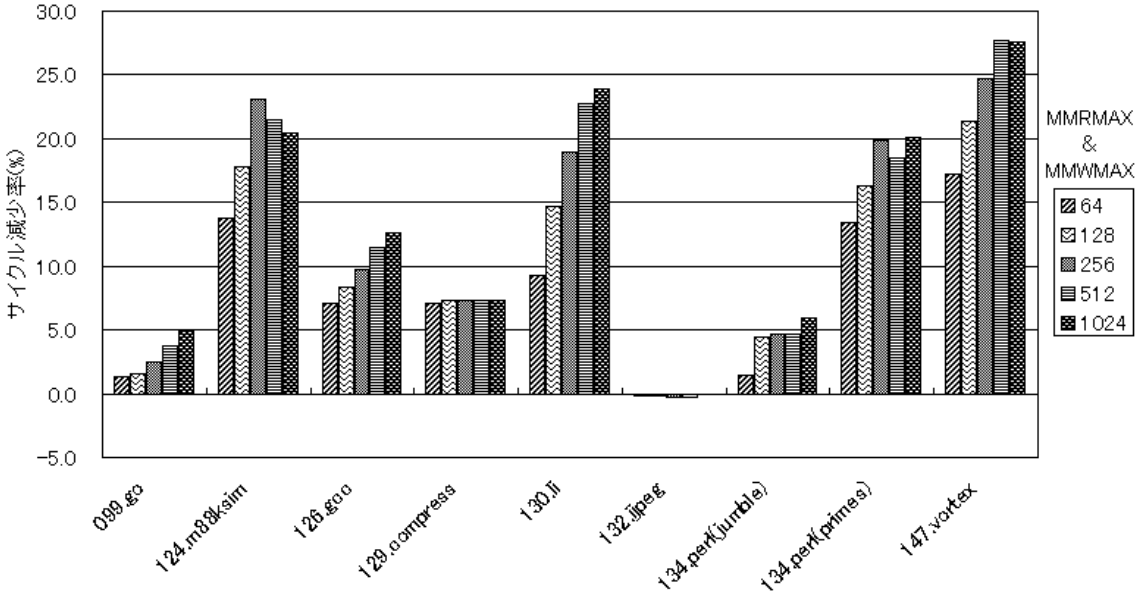

図 12: CINT

表 2: MMRMAX および MMWMAX とウィンドウミス回数の関係

	再利用なし	64	128	256	512	1024
126.gcc	897076 (100)	792804 (88.4)	786106 (87.6)	780362 (87.0)	775004 (86.4)	771388 (86.0)
130.li	5512368 (100)	5378576 (97.6)	5284514 (95.9)	5083052 (92.2)	4757450 (86.3)	4625798 (83.9)
134.perl (prime)	27383 (100)	20961 (76.5)	20929 (76.4)	20925 (76.4)	20923 (76.4)	20939 (76.5)
147.vortex	16841217 (100)	15391291 (91.4)	14757091 (87.6)	13967097 (82.9)	13826801 (82.1)	13829533 (82.1)

同様、再利用を行わなかった場合に要するサイクル数に対する、省略できるサイクル数の比である。

132.ijpeg で値が負になっているのを除けば、全てのベンチマークにおいて効果が出ている。特に 130.li,134.perl(prime),124.m88ksim,147.vortex では最大で 20%以上の高い効果が得られた。ほとんどのベンチマークでは MMRMAX および MMWMAX の値が大きくなるに従って再利用の効果は高くなっている。

compress は MMRMAX および MMWMAX の大きさに影響を受けていない。CINT 全体としては MMRMAX および MMWMAX の値は 512 ほどあれば十分な効果が得られることが分かった。

また、この時のキャッシュミスおよびウィンドウミスの回数も計測した。キャッシュミスについては再利用を行った場合と行わなかった場合に大きな差はなかった。ウィンドウミスについての結果を表 2に示す。上段の数値は MMRMAX および MMWMAX の値であり、カッコ内は再利用を行わなかった場合の値に対する割合である。なお、変化のなかった 129.compare,132.ijpeg,134.perl(jumble),099.go および元々のウィンドウミス数が少なかった 124.m88ksim については省略した。

示した全てのベンチマークにおいて、MMRMAX および MMWMAX が増加するに従ってウィンドウミス数がおおむね減少しており、最大で 20%前後のウィンドウミスが省略されている。

5.4 考察

Stanford-integer においては、再利用の効果のあるプログラムとないプログラムの結果に大きな差が生じた。これはそれぞれのプログラムのサイズが小さく、関数が少ないためであると考えられる。

Stanford-integer の他のベンチマークに比べて、Puzzle では十分に効果を出すために必要な MMRMAX の値が極端に大きい。これは、Puzzle に多くのアドレスからの主記憶読み出しを伴う関数が存在し、それぞれの関数が異なったアドレスの値を読み出すためである。このような関数では、RBSIZE を大きくしたとしても、RF に登録できるアドレス数に限りがあるために、再利用の効果は上昇しない。

SPEC CINT95 において、MMRMAX および MMWMAX の値を大きくしすぎると再利用の効果が悪化する場合がある。これは登録できる主記憶読み出しおよび書き込みの数が増加することに伴って RB に登録できる関数の数が増えるために、それらの登録および主記憶値比較の回数が増えるのに対し、再利用できる関数の数が増加していないためである。このことから、MMRMAX および MMWMAX の値はあまり大きくしすぎない方がよいと考えられる。129.compress では、MMRMAX および MMWMAX の値を変化させても再利用の効果に変化は生じていない。これは関数の数が少なく、MMRMAX および MMWMAX が増加しても登録できる関数の数が増えないためと考えられる。

図 2に示したように、再利用によってウィンドウミスが減少するベンチマークは全て 20%前後の減少を見せており、再利用にウィンドウミスを減らす効果があるということが言える。また、ウィンドウミスが大幅に減少しているベンチマークは全体としての再利用の効果も高くなっているという相関関係が見られる。これに対し、再利用はキャッシュミスに大きな影響を及ぼさない。

以上の結果より、関数単位の値再利用はプログラムの実行を高速化できると言える。しかし、プログラムによって効果はまちまちであり、安定した効果を得られるとは言い難い。繰り返し処理を伴うインタプリタやシミュレータなどのプログラムや、値に局所性の存在するオブジェクト指向プログラムについては再利用の効果が高いが。入力値のパターンが膨大である画像処理や、状況が刻一刻と変化し、入力値が絶え間なく変化するゲームなどのプログラムでは再利用の効果は低くなっている。これらのプログラムを高速化するためには、入力値を予測して関数を投機的に実行し、その結果を RB に前もって登録しておく、などの手法を組み合わせていく必要があると考えられる。

第 6 章　おわりに

本論文では、SPARC アーキテクチャにおいて、関数を単位とする値再利用を専用命令なしに実現するハードウェア機構の提案および評価を行った。値再利用を行うことにより、サイクル数にして Stanford-integer では最大で 63.3%、SPEC CINT では最大で 27.6%の高速化が可能であることが明らかになった。また、再利用によって最大で 20%近くのウィンドウミスを減らせるということ、値再利用はキャッシュミスに影響を及ぼさないということ、ウィンドウミスが大幅に減少しているプログラムは全体としての値再利用の効果も高いという相関関係があることも分かった。

特に、繰り返し処理を伴うインタプリタやシミュレータなどのプログラムや、値に局所性が存在するオブジェクト指向プログラムでの再利用の効果は高く、この種のプログラムが再利用に適しているということも分かった。

しかし、再利用の効果はプログラムによってまちまちであり、安定した効果を得られるとは言い難い。入力のパターンが膨大である画像処理のようなプログラムや、入力が変化し続けるゲームのようなプログラムに対しては、パラメータをどのように設定しても再利用の効果は上がらず、逆に余計な RB アクセス

が増えるために、実行速度は低下する。

過去の実行結果を記憶するだけである現在の再利用機構の構造では、これらのプログラムの高速化は望めない。しかし、入力値の予測を行って関数を事前実行し、その結果を前もって再利用表に記憶することによって高速化できる可能性があり、これは今後の課題である。

謝辞

本研究の機会を与えてくださった、富田眞治教授に深く感謝の意を表します。

また、本研究に関して適切なご指導を賜った中島康彦助教授、北村俊明助教授、森眞一郎助教授、五島正裕助手に深く感謝いたします。

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

参考文献

[1] 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).

[2] Lipasti, M. H. and Shen, J. P.: Exceeding the Dataflow Limit via Value Prediction, *International Symposium on Microarchitecture*, pp. 226–237 (1996).

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

[4] Lipasti, M. H. and Shen, J. P.: Exceeding the Dataflow Limit via Value Prediction, *International Symposium on Microarchitecture*, pp. 226–237 (1996).

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

[6] Onder, S. and Gupta, R.: Load and Store Reuse Using Register File Contents, *International Conference on Supercomputing*, pp. 289–302 (2001).

[7] Connors, D. and Hwu, W.: Compiler-Directed Dynamic Computation

Reuse: Rationale and Initial Results, *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, *HPCA5*, 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 Techniques* (2000).

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

[12] Sodani, A. and Sohi, G. S.: Understanding the Differences Between Value Prediction and Instruction Reuse, *International Symposium on Microarchitecture*, pp. 205–215 (1998).

[13] Wu, Y., Chen, D.-Y. and Fang, J.: Better Exploration of Region-Level Value Locality with integrated Computation Reuse and Value Prediction, *ISCA28, July 2001* (2001).

[14] 富田眞治: コンピュータアーキテクチャ, 丸善 (1994).

[15] Paul, R.: SPARC Architecture, Assembly Language Programming, & C (1994).