特別研究報告書

関数 / ループの再利用および事前実行による高速化

指導教官　　富田 眞治 教授

京都大学工学部情報学科

笠原 寛壽

平成 15 年 2 月 3 日

関数 / ループの再利用および事前実行による高速化

笠原 寛壽

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、および区間再利用がある。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。

これに対して区間再利用は、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。区間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ものの、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。本論文では、SPARC ABI に従うプログラムを対象とし、専用命令を追加することなく区間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。

関数およびループ構造を命令区間単位とし、区間再利用を適用するにあたり再利用表を用いた。再利用表とは、区間再利用に必要な情報を記憶しておく表であり、入出力記憶表 (RB)、関数管理表 (RF)、再利用ウインドウ (RW) から構成される。まず、命令区間の入力および出力の情報を登録する RB には、命令区間が関数の場合、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し、ループの場合には、さらに参照したレジスタも含め入出力情報として記憶し、エントリの入れ替えには、LRU アルゴリズムを用いた。また、各命令区間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に RF を設けた。さらに、関数やループの多重構造の一括再利用を可能とするために、どの命令区間がどの命令区間を呼び出したかの情報を記憶する RW を設けた。

また、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の並列事前実行機構を実現した。区間再利用を行うプロセッサ (MSP) とは別に、入力値を予測して命令区間を投機的に事前実行を行うプロセッサ (SSP) を複数個設け、RB エントリの生存時間よりも同一パラメタが出現する間隔が長い場合や、命令区間のパラメタが単調に変化し続ける場合においても RB を有効化した。数回の試行の結果、MSP による実行頻度が高く、かつ、SSP の事前実行により登録されたエントリの再利用頻度も高い命令区間を継続して SSP の実行対象とした。さらに、動的に変化する登録頻度や再利用頻度を把握するために、一定期間における登録および再利用の状況をシフトレジスタを用いて記録した。MSP と SSP では RB エントリを使い分けており、SSP の実行により登録された RB エントリは FIFO による入れ替えを行った。

以上のような機構を有し、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、Stanford-integer ベンチマークによる評価を行った。

評価の結果、最大 75%のサイクル数を削減できることがわかった。また、プログラムによっては再利用によってレジスタウィンドウミスが大幅に減らせることや、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が明らかになった。一方、再利用の効果はベンチマークプログラムによってばらつきがあり、入力値のパターンが膨大であるプログラムや、入力が不規則に変化し続けるプログラムについては、効果を上げるために更なる改良が必要であることがわかった。

A Speedup Technique with Function / Loop Reuse and Precomputation

Hirohisa KASAHARA

Abstract

Recent studies have demonstrated that value locality is observed in general programs. Many researches on region reuses and value speculation are reported for improving the performance of microprocessors by exploiting value locality. Value speculation is a technique that speculatively starts the following instructions with predicted input values, and collapses the waiting time by the intrinsic data dependencies among the series of instructions. Though, when the predicted values turn out to be incorrect, the results should be squashed and the instructions speculatively executed should be re-executed.

On the other hand, region reuse is a technique that memorizes the input and the output while executing the series of instructions, and reuses the result later. When the program counter encounters the same region with the same input as memorized before, the correct result is retrieved immediately without actual execution of the region. Thus, no penalties by the cancellation required in speculative executions are occurred. Three types of techniques are considered to implement the region reuse. First is hardware implementation. Second is the compiler implementation that generates the sequence of normal instructions for software reuse. Third is cooperation of hardware and compiler that generates the special purpose instructions for hardware reuse. In case of the hardware implementation, recognition of the basic block is very difficult. The instruction-level reuse may be easy to implement, but the effectiveness is relatively low. In case of the compiler implementation, general processors can be used, but the overhead of reuse becomes large. On the other hand, the cooperation of hardware and compiler can reuse large region like basic blocks, but can't speed the existing load modules. In contrast to these techniques, I exploit SPARC ABI (Application Binary Interface), and propose the hardware mechanism that enables the reuse of large regions without any special purpose instructions. Also I evaluate the effectiveness with some benchmark programs.

To implement the region reuse, I select functions and loops as reuse regions,

and assume a hardware structure so called reuse buffer. Reuse buffer consists of RB, RF, RW and memorizes the information required for reuse. RB holds the input and the output of the region. For function reuse, the arguments and the read data from memory are stored as input, and the return value and the write data for memory are stored as output. For loop reuses, the all referred registers in each iteration are stored as input or output. The replacement algorism for RB is based on LRU. I supposed a CAM structure to compare the all rows of RB entry with the values on the same registers or memory locations. RF holds the memory addresses to read/write. RW holds the relations between caller and callee in order to enable the multilevel region reuse.

Precomputation shows the effectiveness against such region that the simple reuse technique has no effects. With a combination of the simple reuse mechanism provided by MSP (Main Stream Processor) and the parallel precomputation mechanism provided by SSPs (Shadow Stream Processors), I make RB effective despite the interval of the argument reappearance is longer than the lifetime of RB entry or the arguments are linearly changed. The precomputation is continued on the region that the result of the trial executing shows good hit ratio of RB. To follow the dynamic behavior of the program, the status of memorization and reuse is stored in the shift register attached to each RF. We divide RB entry into two blocks and assign SSP's block with FIFO algorithm.

Considering instruction latency, cache miss penalty, window miss penalty and reuse latency, I developed a cycle simulator and evaluated the reuse mechanism with Stanford-integer. I found that the maximum ratio of eliminated cycle reached to 75% in Stanford-integer programs. Moreover, it is shown that multilevel reuse can reduce window miss overhead significantly on some programs and that reuse does not affect the cache miss overhead. On the other hand, the effects of reuse are not stable and it is necessary to develop the technique for programs whose inputs take various values, and in which the input pattern changes irregularly.

関数 / ループの再利用および事前実行による高速化

目次

第 1 章　はじめに **1**

1.1　背景 1

1.2　関連研究 2

第 2 章　区間再利用 **3**

2.1　再利用表の構成 3

2.2　関数再利用機構の動作 5

2.3　関数とループの類似性 6

2.4　RF および RB の拡張 6

2.5　ループ再利用機構の動作 7

第 3 章　事前実行 **7**

3.1　MSP と SSP 8

3.2　入力の予測 9

3.3　RB エントリの入れ替え 9

3.4　命令区間の選択 10

第 4 章　評価 **11**

4.1　プログラム単位の評価 11

4.2　命令区間単位のステップ数の評価と分析 14

4.3　考察 23

第 5 章　まとめ **24**

謝辞 **24**

参考文献 **24**

第 1 章　はじめに

1.1　背景

区間再利用とは、出現した命令区間に対し、入力および結果を保存しておくことにより、再度同一の入力に際に、命令を実行することなく保存された結果を用いて、高速化を行う手法である。区間再利用の実現には、まず、コンパイラによる専用命令の追加を行う方法があげられる。専用命令を使用するのは、専用でない命令から得られる情報のみでは、プロセッサが基本ブロックを切り出すことが難しいためである。この方式では、専用のコンパイラが生成するロードモジュールのみ、高速化することができる。本論文では、一般的にロードモジュールが ABI(Application Binary Interface) に従って作成されることを利用し、既存ロードモジュールへの適用を可能とする。SPARC ABI を使用し、関数やループの入出力を特定することにより、コンパイラによる専用命令の埋め込みを不要とした。さらに、命令区間の多重構造を動的に把握することにより、関数内局所レジスタやスタック上の局所変数を再利用における入出力値から排除し、効率向上に貢献する。特に、関数については、関数の複雑さに関わらず、最大 6 のレジスタ入力、最大 4 のレジスタ出力、および、局所変数を含まない最小限度の主記憶値の登録により再利用機構が実現できる [16]。

また、過去の入力および結果を参照するだけの再利用では、命令を実行するごとにパラメタが変化するときに、効果が期待できない。そこで、パラメタが単調に変化するときにも追随するために、将来の入力値予測を行い、事前実行し、結果を作成しておくことにより、再利用を有効化を図る。パラメタ単調変化に追随する高速化手法に、値予測に基づく投機実行があり多くの研究が行われている。値予測に基づく投機実行とは、後続命令の入力を予測し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。予測が正しければ実行時間を短縮するものの、予測を誤った場合、投機的に実行した結果を無効化し再び実行する必要があるため、失敗時にペナルティがあり、命令区間が大きくなると、このペナルティも大きくなる。これに対し本方式では、失敗時にペナルティが発生しないという特長がある。

以下では、まず、関数を単位とする再利用の実現方法を説明し、さらに、再利用をループに適用するための機能拡張について述べる。次に事前実行機構について説明し、最後に Stanford ベンチマークを用いて評価を行い、その評価に

対する分析を詳述する。

1.2 関連研究

命令間に依存関係が存在する場合でも、先行命令列の実行結果を予測し、後続命令列の投機的実行を開始することにより、命令レベルの並列度を確保する研究が数多く行われている [1, 2]。さらに、複数の予測値に基づき、複数のプロセッサを投入して高速化を図る投機的マルチスレッド実行に関する研究も報告されている [3, 4, 5]。しかしながら、値予測に基づく投機的実行を行う場合、予測が正しかったかどうかを常に検証する必要があるため、先行命令列の実行時間そのものを削減することはできない。このため、厳密な検証が必要となる値そのものを投機対象とするのではなく、投機的マルチスレッド実行機構を利用してロード命令を事前実行し、効果的なプリフェッチ機構として利用する研究が報告されている [6]。一方、再利用 [7, 8, 9, 10] は、プログラムの一部分に関する入出力値を表に登録しておく。同じ箇所を再度実行するとき、入力値が既知の場合には、正しい出力値を瞬時に求めることができる。本方式の特長は、入力値さえ一致すれば、実行結果を検証する必要がない点である。副次的な効果として、冗長なロード / ストア命令や消費電力を削減できることも報告されている [11, 12]。Connors ら [13] は、コンパイラが切り出した再利用区間も用い、記憶可能な入出力レジスタ数を各々 8 とする表を用意し、SPEC ベンチマークプログラムを 10%から 60%短縮している。ただし、主記憶上の値は再利用の対象外としているため、適用範囲が限られる。Huang ら [8, 14] は、再利用区間内に閉じたレジスタ (dead register) をハードウェアに伝達し、出力値としての登録を抑制するよう GCC を改良し、コンパイラの支援を受けた基本ブロックの再利用により、SPEC ベンチマークプログラムの実行時間を 1%から 14%短縮している。記憶可能なレジスタ値は、入力 5 、出力 6 、主記憶値は、入力 4 、出力 3 を仮定している。Wu ら [15] は、再利用と投機を組み合わせた方法として、同様にコンパイラが再利用の区間の切り出しを行い、実行時に再利用可能である場合には再利用を行い、再利用不可能である場合には再利用区間の出力値を予測して後続区間の実行を投機的に開始する研究を報告している。ただし、出力値の予測がはずれた場合、後続区間の投機的実行をキャンセルしなければならず、このための機構のコストやオーバーヘッドが問題になる。これらに対し、本論文では、再利用区間の 「入力値」を予測の対象としており、失敗した投機

的実行をキャンセルして再実行する必要がない。

第 2 章　区間再利用

本章では、まず、SPARC ABI に基づいて記述されたプログラムに対して、関数単位の再利用を適用する機構について説明する。関数再利用機構は、関数の入出力情報を再利用表を用いて記憶し、同一関数が同一入力値により呼び出された場合に関数の処理を省略する。このため、関数の入力や出力が何であるかの特定を要する。一般のプログラムにおいては、関数内で関数が呼ばれるなど、多重構造を形成することがある。関数 A が関数 B を呼ぶ場合、A から B への引数は B への入力、返り値は B からの出力になる。A の局所変数は、A の入出力ではないものの、ポインタを通じて B の入出力になり得る。B の局所変数は、A、B いずれのの入出力にもならない。また、大域変数は、A、B のどちらの入出力にもなり得る。

以下に、再利用表の構成と関数再利用機構の動作について述べる。また、関数再利用とループ再利用の類似性を紹介し、再利用機構のループへの適用法について説明する。

2.1　再利用表の構成

再利用表とは、関数の入力および出力の情報を記憶しておく表である。再利用ウインドウ (RW)、関数管理表 (RF)、入出力記憶表 (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。具体的には、現在実行中かつ登録中の RF および RB の各エントリをスタック構造として保持する。この様子を図 1に示す。なお、RW のエントリ数は有限であるものの、一度に登録可能な多重度を越えて関数が呼び出された際には、外側の関数から順次登録を中止し、より内側の関数を RW エントリに加えることにより、多重構造に追随する。また、ある関数の実行および登録中に、再利用可能な関数に遭遇した場合は、登録済みの入出力をそのまま登録中エントリに追加することにより、RW の深さを越える多重再利用を可能とする。

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 2に

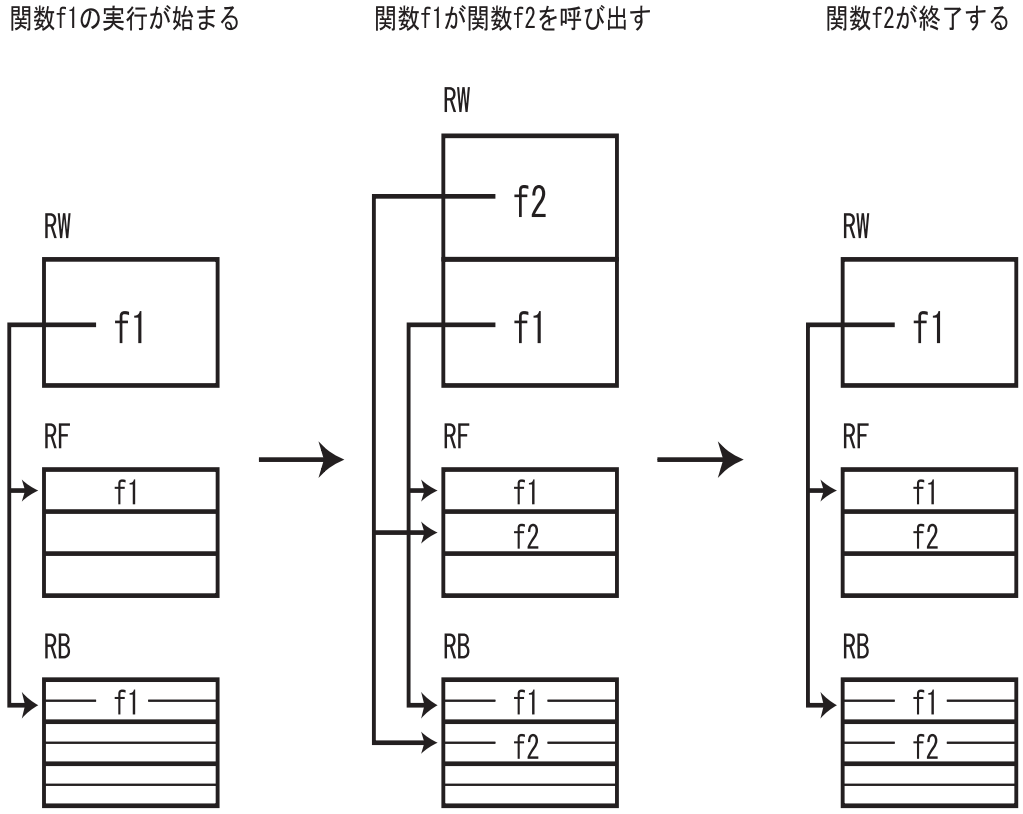

図 1: 再利用ウィンドウ (RW)

１つの関数を再利用するために必要なハードウェア構成を示す (関数再利用では、図中の網掛け部分を使用しない)。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。LRU カウンタは、RF エントリの入れ換えのためのヒントとして用いられる。

入出力記憶表 (RB) は、引数 (V:エントリ状況、Val.:値)、主記憶値 (Mask:読み出しデータ / 書き込みデータの有効バイト、Val.:値)、返り値 (V:エントリ状況、Val.:値) および、関数呼び出し直前の%sp の値を保持する。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。読み出しアドレスは RF が一括管理し、マスクおよび値は RB が管理するのは、読み出しアドレスの内容と RB の複数エントリを CAM により一度に比較する構成を可能とするためである。RB 各エントリ先頭のフラグ

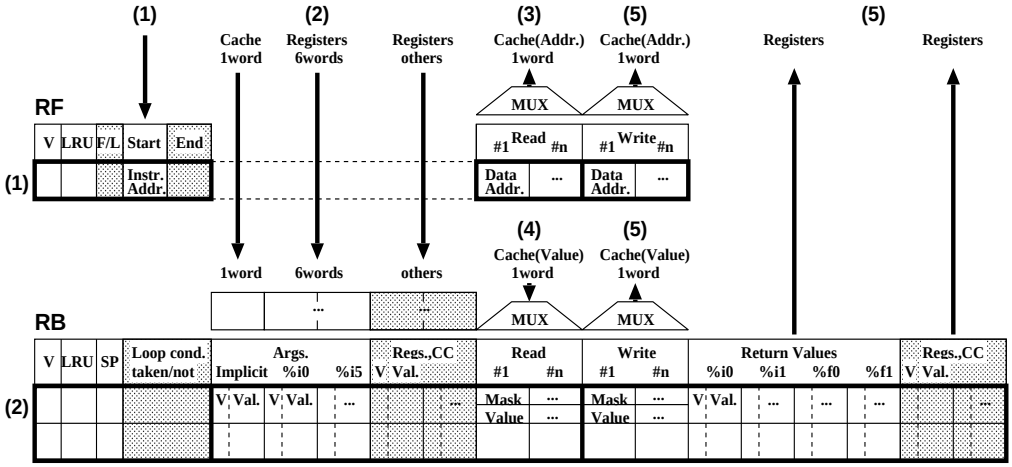

図 2: 関数管理表 (RF)、入出力記憶表 (RB)

V および LRU カウンタは RF の場合と同じように用いられる。

2.2 関数再利用機構の動作

関数を呼び出す call もしくは jmpl 命令が実行されると、再利用機構は関数の先頭アドレスが RF に登録されているかどうかを調査する。登録されていた場合には、その関数に対応する RB から、引数が完全に一致するエントリを選択し、関連する主記憶アドレスすなわち少なくとも 1 つの Mask が有効である読み出しアドレスをすべて参照し、一致比較を行う。すべての入力が一致した場合に、登録済の出力 (返り値、大域変数、上位関数の局所変数) を書き戻すことにより、関数の実行を省略することができる。

RB にすべての入力が一致するエントリが登録されていない場合、引き続いて関数本体の実行を開始する。局所変数を除外しながら、引数、返り値、大域変数および上位関数の局所変数に関する入出力情報を登録していく。読み出しが先行した引数レジスタは関数の入力として、また、返り値レジスタへの書き込みは関数の出力として登録する。その他のレジスタ参照は登録する必要がない。主記憶参照も同様に、読み出しが先行したアドレスについては入力、書き込みは出力として登録する。関数から復帰するまでに次の関数を呼び出したり、登録すべき入出力が入出力表の容量を越えた、引数の第 7 ワードを検出した、あるいは、途中でシステムコールや割り込みが発生したなどの問題が発生した場合、登録を中止する。復帰を示す jmpl 命令によって関数呼び出しが終了すると、RB の状態を 登録中」から 登録済」に変更する。最後にこの RB エント

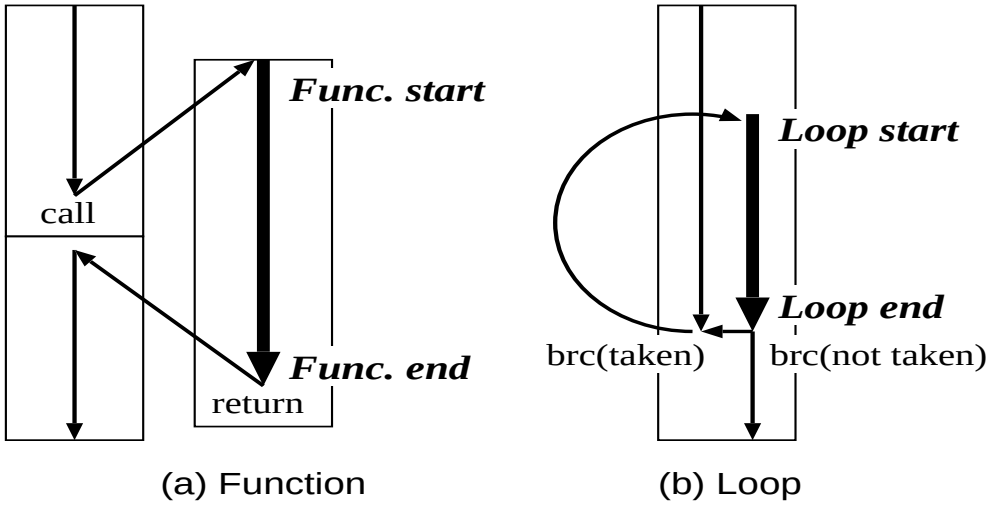

図 3: 関数とループの類似性

リのインデックスを RW から降ろし、RB への登録を完了する。

また、RF に関数アドレスが登録されていない場合は、RF エントリの登録から行う。RF や RB がすべて埋まっている場合には、LRU アルゴリズムに従って既存のエントリを追い出す。

2.3 関数とループの類似性

次に、再利用機構のループへの適用法について述べる。関数とループの類似性を図 3 に示す。関数の場合、関数呼び出しにより区間を認知し、呼び出し命令の分岐先から復帰命令までを区間単位とした。同様に、ループの際には、始まりをループ末の後方分岐命令により認知し、後方分岐命令の分岐先から同じ後方分岐命令までを区間とする。このように、ループは関数と区間の始まり方こそ異なるものの、ループ再利用を実現するためには、新しい機構を構成する必要はなく、関数と同様に命令区間の入出力を登録すればよい。

2.4 RF および RB の拡張

再利用をループに適用するにあたり、上述した関数とループの類似性により、再利用機構そのものの改良は不要である。しかし、RF および RB の拡張が必要である。ループ内の局所変数は ABI では規定されないため、関数再利用時には区分することができる局所変数の登録をループ再利用では除外することができない。これより、まず、ループ再利用においては、参照したレジスタおよび

主記憶アドレスのすべてを入出力として表に登録する必要がある。また、ループが再利用可能であった際に、引数や返り値以外のレジスタの値および分岐の評価を行うため条件コード (Regs、CC)、ループ終了時の分岐方向 (taken/not) の情報が必要となる。このため、関数管理表 (RF) および入出力記録表 (RB) に拡張を施す。図 2の網掛け部分がループの再利用のために追加した登録項目である。RF の項目 (F/L) は、関数とループの区別をするために使用し、(End) にはループの終了アドレスを格納する。

2.5　ループ再利用機構の動作

後方分岐命令が成立すると、再利用機構は、後方分岐する前に分岐先アドレスが RF に登録されているかどうか調べる。登録されていた場合には、対応する RB から、レジスタ入力値が完全に一致するエントリを選択し、関連する主記憶アドレスをすべて参照し、一致比較を行う。すべての入力が一致した場合に、登録済の出力 (レジスタおよび主記憶出力値) を書き戻すことにより、ループの実行を省略することができる。再利用した場合、RB に登録されている分岐方向に基づいて、さらに次ループに関して同様の処理を繰り返す。

一方、次ループが再利用不可能であれば、次ループの実行を開始する。参照したレジスタおよび主記憶アドレスを登録していく。ループが完了する以前に関数から復帰したり、関数再利用時にあげた問題が発生すると、登録を中止する。登録中のループに対応する後方分岐命令の検出によりループの完了を確認すると、ループ終了時の分岐方向の情報を登録し RB の状態を 「登録中」から 「登録済」に変更する。最後にこの RB エントリのインデックスを RW から降ろし、RB への登録を完了する。

第 3 章　事前実行

関数やループの再利用では、RB エントリの生存時間よりも同一パラメタが出現する間隔が長い場合や、パラメタが単調に変化し続ける場合に効果が期待できない。そこで、プロセッサを複数個設け、予測をした将来の入力値を用い、命令区間の事前実行をする。この結果を RB に登録しておくことにより、再利用の有効化を図る。

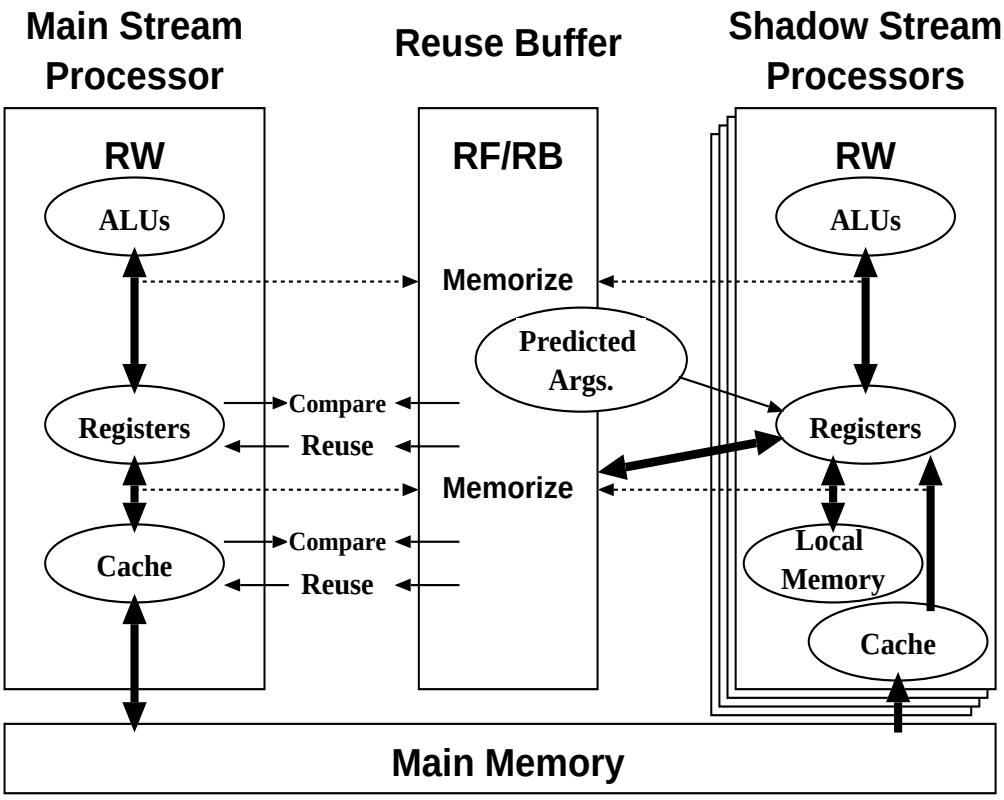

図 4: 並列事前実行機構

3.1 MSP と SSP

これまでに説明した再利用を行うプロセッサ (Main Stream Processor :以下 MSP と略する) とは別に、命令区間の事前実行により RB エントリを有効化するプロセッサ (Shadow Stream Processor :以下 SSP と略する) を複数個設けることにより、さらなる高速化を図った。事前実行機構の概要を図 4 に示す。RF、RB、主記憶は全プロセッサが共有する。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設ける。破線は、MSP および SSP が RB に対して入出力を登録するパスを示している。事前実行では、MSP および SSP に対して、いかに主記憶一貫性を保つかが課題となる。特に、予測した入力パラメタに基づいて命令区間を実行する場合、主記憶に書き込む値が MSP と SSP で異なる。これを解決するために、図 4 に示すように、SSP は、RB への登録対象となる主記憶参照には RB、また、その他の局所的な参照には SSP ごとに設けた局所メモリを使用することとし、キャッシュおよび主記憶への書き込みを不要とした。もちろん、MSP が主記憶に対して書き込みを行った場合には、

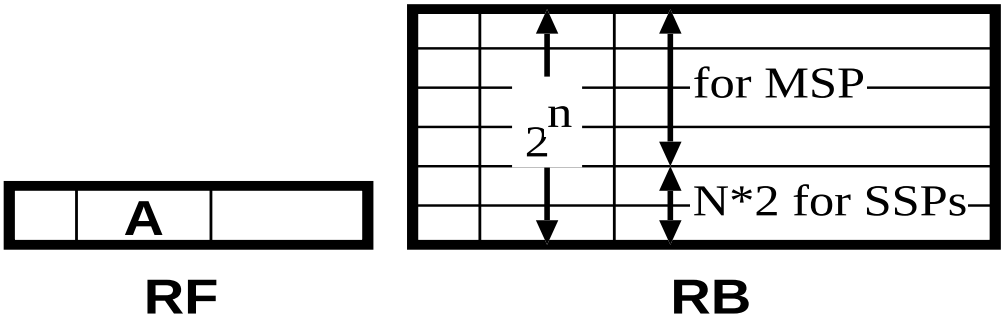

図 5: RB の分割

対応する SSP のキャッシュラインが無効化される。具体的には、RB への登録対象のうち、読み出しが先行するアドレスについては、主記憶を参照し、MSP と同様にアドレスおよび値を RB へ登録する。以後、主記憶ではなく RB を参照することにより、他のプロセッサからの上書きによる矛盾の発生を避けることができる。局所的な参照については、読み出しが先行することは、変数を初期化せずに使うことに相当し、値は不定でよいことから、主記憶を参照する必要はない。なお、局所メモリの容量は有限であり、関数フレームの大きさが局所メモリを越えた場合など、実行を継続できない場合は、事前実行を打ち切る。また、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。

3.2 入力の予測

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 へ渡す必要がある。このために、RF の各エントリごとに小さなプロセッサを設け、MSP や SSP とは独立に入力予測値を求めることにする。具体的には、最後に出現した引数 (B) および最近出現した 2 組の引数の差分 (D) に基づいて、ストライド予測を行う。なお、$B + D$に基づく命令区間の実行は MSP がすでに開始していると考える。SSP が N 台の場合、用意する入力予測値は、$B + D * 2$ から $B + D * (N + 1)$ の範囲とした。

3.3 RB エントリの入れ替え

各 RF エントリが 1 つの命令区間に対応し、入力と出力の対応関係が RB に登録される。このとき、MSP と SSP が RB エントリをどのように使い分ける

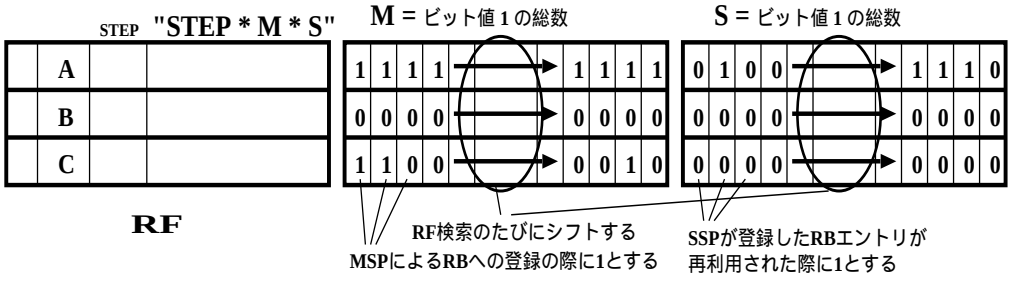

図 6: 命令区間選択機構

かが課題となる。命令区間は、大きく、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 では効果がないものにわかれると考えられる。前者であれば、LRU による入れ替え、後者であれば、FIFO による入れ替えが有効である。しかし、ある命令区間の性質がいずれであるかを動的かつ直ちに判断することは難しいため。個々の RF に属する RB エントリを MSP 用と SSP 用とに分割し、それぞれ LRU と FIFO により入れ替えることにする。前節において述べたように、入力予測値は N 組であり、SSP が登録後、MSP が直ちに利用することを想定して、SSP 用に割り当てるエントリ数は $N * 2$ としておく。この様子を図 5 に示す。

3.4 命令区間の選択

次に、どの命令区間を SSP に実行させるかが課題である。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効果があることが予想されるものの、各々の命令区間の性質や実際の効果の有無は、事前にはわからない。このため、RF に新規に登録された命令区間については、直ちに SSP による数回分の事前実行を試みることにした。数回の試行の結果、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頻度も高い RF を継続して SSP の実行対象とする。動的に変化する登録頻度や再利用頻度を把握するために、一定期間における登録および再利用の状況をシフトレジスタに記録する。RF ごとに付加した小さなプロセッサが E =(過去の削減ステップ数)*(登録回数)*(再利用回数) を計算し、各 SSP が、Eが最大となる RF を選択する。この様子を図 6に示す。

第 4 章　評価

評価には Stanford-integer を用いた。Stanford-integer は 10 種類のプログラムから成る。それぞれのプログラムについて測定を行った。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、評価を行った。各パラメタを表 1 に示す。RB 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、RF あたりの RB エントリ数を現実的な 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 内のレジスタ入力値の比較には 1 サイクル、キャッシュと RB 内の主記憶入力値 (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、比較時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生じる。再利用時の書き込みは、RB 内の主記憶出力値 (最大 1024 アドレス) からキャッシュへは 1 ワードあたり 1 サイクル、RB 内のレジスタ出力値から MSP のレジスタへは 1 サイクルを要すると仮定した。

4.1　プログラム単位の評価

MSP のみ (MSP)、MSP および 3 台の SSP (MSP+SSP*3) と、関数のみ (F)、関数およびループ (F+L) を組み合わせた 4 通りの測定条件により評価を行った。図 7 に通常実行時の総実行命令ステップ数 (レイテンシが 2 以上の

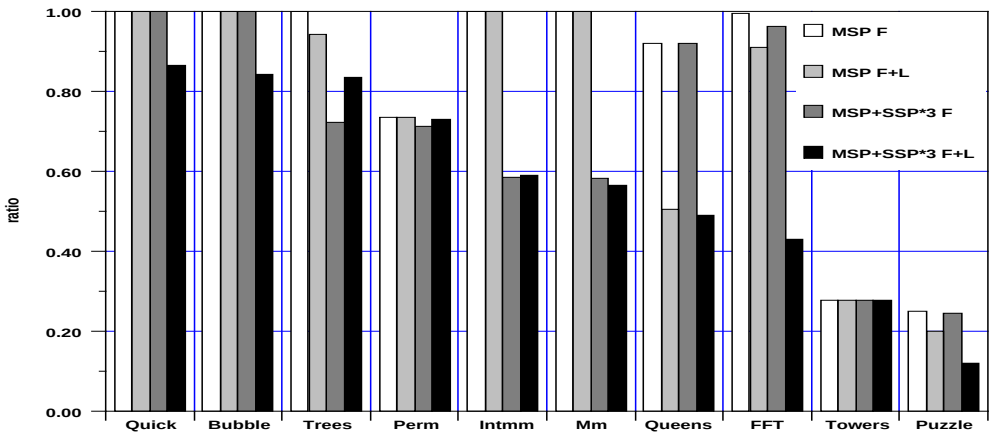

図 7: MSP が実行した命令ステップ数

表 1: シミュレーション時のパラメータ

D-Cache	64 Kbytes
Line Size	64 bytes
Ways	4
Cache Miss	20 cycles
Register Window	4 sets
Register Window Miss	20 cycles/set
Load Latency	2 cycles
Integer Mult.	8 cycles
Integer Div.	70 cycles
Floating Add/Mult.	4 cycles
Single Div.	16 cycles
Double Div.	19 cycles
RW Depth	4
RF Entry	32
Read Address	1024/RF
Write Address	1024/RF
RB Entry	256/RF
RB(Reg.)-Register Compare	1 cycle
RB(Read)-Cache Compare	4 bytes/cycle
(Additional 20 cycles on each cache miss.)	
RB(Write)→Cache Write	4 bytes/cycle
RB(Reg.)→Register Write	1 cycle
SSP Local Memory	64 Kbytes

命令はレイテンシを加算）を 1 とした場合の MSP の実行命令ステップ数の比を示す。

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，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となっている Puzzle では最大で 90%近くの命令ステップ数削減に成功している。

さて実際に高速化を達成するには、再利用に伴うオーバヘッドを見極める必要がある。図 8 は、命令ステップ数（exec）に、Register Compare および Cache Compare 時にかかるサイクル数（test）、Cache Write および Register Write 時にかかるサイクル数（write）、キャッシュミス（cache-miss）、レジスタウィンドウミス（reg-window）の各オーバヘッドを加えたサイクル数の内訳である。左側棒グラフは通常実行時、右側棒グラフは『MSP+SSP*3 F+L』の内訳である。

Towers では再利用により関数再帰呼び出しが減少した結果、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) ⇔ Cache Compare のスループットを 4byte/cycle から 8byte/cycle に増加させるなど、比較の高速化が重要課題であると言える。また、キャッシュミスの増加による性能低下はほとんどなかった。

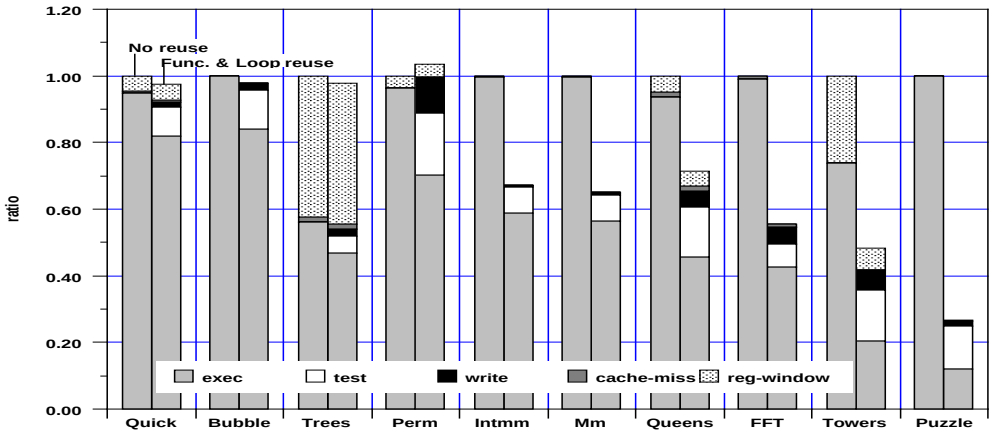

図 8: MSP が実行したサイクル数 (Stanford).

4.2 命令区間単位のステップ数の評価と分析

プログラムごとの削減ステップの比率には、大きな違いが存在した。また、一部のプログラムでは、事前実行にループを加えた場合、性能低下が生じた。以下では、プログラム内の命令区間単位での MSP の実行命令ステップ数を評価することにより、問題の原因を調査する。以下にプログラムごとに図示した、命令区間単位での MSP の削減ステップ数の評価は、前節と同様、MSP のみ (MSP)、MSP および 3 台の SSP (MSP+SSP*3) と、関数のみ (F)、関数およびループ (F+L) を組み合わせた 4 通りの測定条件で行っている。縦軸は削減したステップ数を表す。命令区間が関数の場合は、(関数名).F、ループの場合は、(ループを保持する関数名).L と表記した。

Intmm 整数要素からなる 2 つの行列の積を計算する。Stanford-integer では、行、列ともに 40 の行列を用いる。計算結果は新しい 2 次元配列に格納されていくため、事前実行を用いなければ、再利用の効果はない。解行列の (i,j) 要素を求める関数 rInnerproduct が、i、j がそれぞれ 1 から 40 まで推移する 2 重ループの中で呼ばれる。しかし、関数 rInnerproduct において、効果を十分に出すためには、多くの読み込みアドレスおよび書き込みアドレスの格納場所が必要になり、前節のシミュレーションでは 40%のステップ数削減にとどまった。ただし、アドレス格納の箇所を 1024 から 2048 に

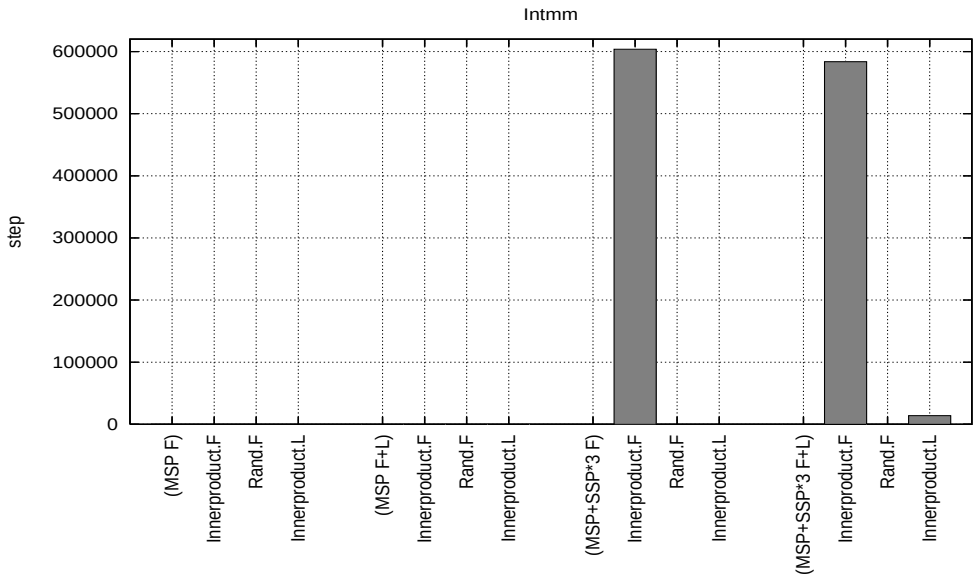

図 9: 命令区間別削減ステップ数 (Intmm)

増やすと、この問題は解決され、60%のステップ数削減が可能であった。これは、Intmm 中の 90%のステップ数を担う関数 rInnerproduct が呼び出された際に、3 回中 2 回は再利用されることによるものである。SSP を 3 台設けている本評価では、理想的には 4 回中 3 回再利用されるべきであるものの、入力値の予測が正確に行われるまでに時間を要するため、3 回中 2 回の再利用にとどまっている。パラメタが線形に変化する期間が長くなれば、ほぼ 4 回中 3 回再利用されると考えられる。また、図 9に示す通り、関数およびループの事前実行を行うと、関数 rInnerproduct 内のループが再利用されるものの、rInnerproduct はこのループのみから成り、関数が再利用されるときと、内部のループが再利用されるときでは、ほぼ同等のステップ数が削減されている。

Mm　浮動小数点数からなる 2 つの行列の積を計算する。要素の型が異なるだけで、プログラムの本質は Intmm と同様である。やはり、アドレス格納の箇所を 1024 から 2048 に増やすと、プログラム全体から見て 60%のステップ数を削減することができる。図 10に示す関数 Rand は、計算に用いる 2 つの行列の構成に使用される関数であり、事前実行を用いて再利用を適用することが可能である。具体的には、Rand は大域変数を読み出し、値を変更したのちに書き込む関数であり、次回に呼び出す Rand に限り、事前

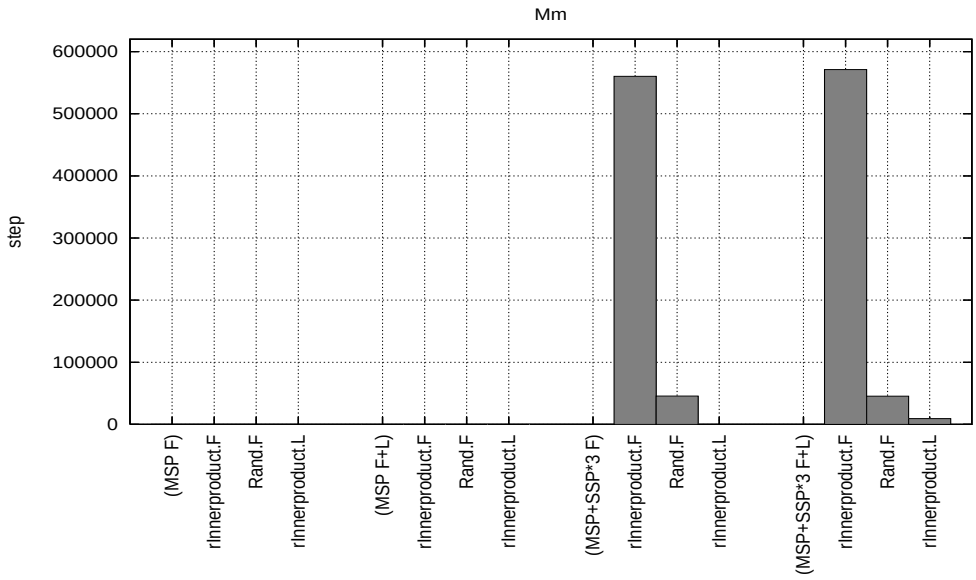

図 10: 命令区間別削減ステップ数 (Mm)

実行が可能である。プログラム Intmm においても関数 Rand が存在したにもかかわらず、再利用が適用されなかったのは、関数 Rand が短い間隔で呼ばれたため、SSP の事前実行の完了前に、Rand に対応する RB エントリの検索が行われたという理由からである。プログラム Mm の場合は、値が浮動小数点数になったことにより、Rand が呼ばれる間の計算に多くのステップ数を要したため、SSP による事前実行が有効となっていた。

Quick　クイックソートを行う。ソートは関数 Quicksort を再帰的に呼び出すことによって行われる。同じ数列を繰り返しソートすることはなく、関数 Quicksort に再利用が適用されることはない。関数 Quicksort 内においても、前出の関数 Rand が 5000 個の要素からなる数列を構成する際に呼ばれる。関数 Rand の実行が占めるステップ数の割合は、関数 Quicksort 全体の 10%に達し、Rand の実行回数が事前実行により 4 分の 3 削減された結果、プログラム Quicksort 全体の 7.5%のステップ数を削減することができた。関数 Rand が関数のみの事前実行では高速化できない理由は、SSP が Rand の事前実行を完了する前に、Rand に対応する RB エントリの検索が行われるためである。ループ事前実行による、Rand の事前実行が有効であったのは、Rand を含むループ再利用に伴うオーバヘッドにより、次回の Rand 呼び出し以前に、SSP が事前実行を完了したためである。ここで、

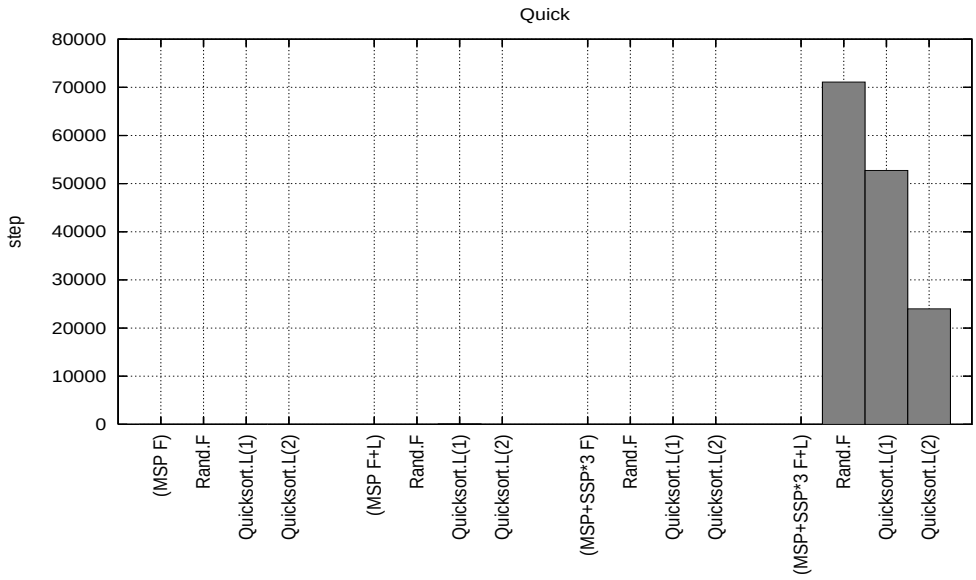

図 11: 命令区間別削減ステップ数 (Quick)

Rand は、大域変数を読み出し、値を変えて書き込むだけの関数であり、次回の呼び出しまでに SSP の事前実行が終了しさえすれば、毎回の呼び出しにおいて、再利用を適用できるはずである。しかしながら、4 分の 3 の削減にとどまっているのは、命令区間の実行完了時の 4 回に 1 回だけ、SSP の事前実行が開始されないためであり、今後、改善すべき課題である。

また、図 11に示す Quicksort.L(1)、Quicksort.L(2) は、それぞれパラメタのインクリメント、デクリメントを行うループであり、事前実行によるループ再利用が適用できる。以上のように、Quick はループ事前実行により高速化されるものの、再利用が有効ではないループや関数が存在するため、命令ステップ数の削減率は 15%にとどまる。

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、関数再利用の効果はない。図 12に示すループ Bubble.L(2) の区間では、数列の隣同士を比較し、その大小により数値を入れ替える動作を行う。数列内の同じ箇所での比較が、並びの等しい数列に対して実行されることはなく、再利用のみでは実行を省くことはできないが、事前実行を使用すれば、前回のループにおいて数列の並びに変化がないときのみ、削減が可能になる。本評価では、ランダム値の要素から成る、長さ 500 の数列を用いて、区間 Bubble.L(2) の実行回数が 6 分の 1 削減されている。最外ループ Bubble.L(1)

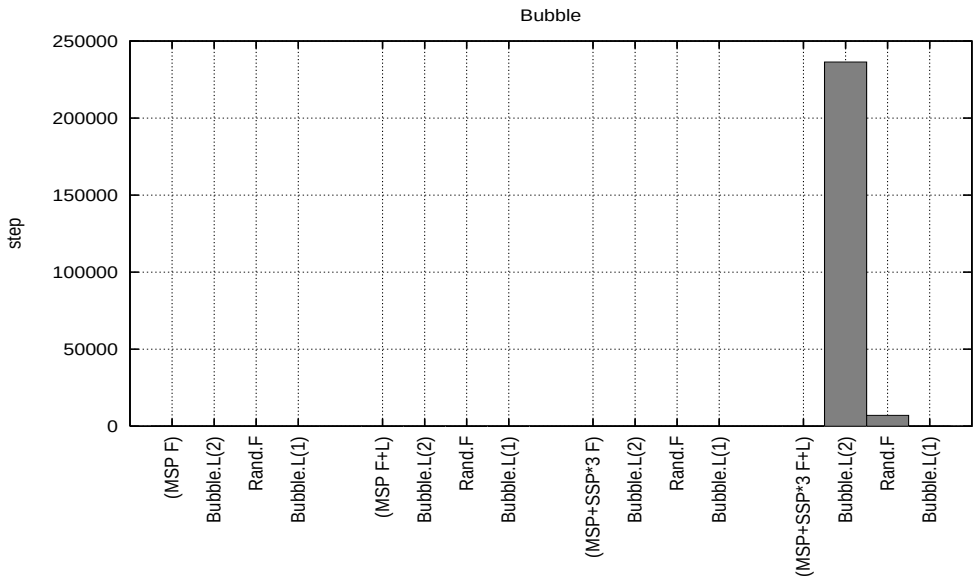

図 12: 命令区間別削減ステップ数 (Bubble)

は、並びの等しい数列で実行されることがほぼないため、再利用の効果はない。また、関数 Rand の実行は、Quicksort のときと同様、ループ事前実行により高速化されている。

Perm　再帰的に配列上の値の入れ替えを行うプログラムである。配列上の 2 値を入れ替える関数 Swap が何度も呼び出される。2 値の場所と値が一致すれば再利用が可能となる。本プログラムでは、配列の長さが短く、2 値の場所と値は 252 パターンしか存在しない。従って、RB エントリの入れ替えは不要であり、一度 RB に登録したパターンは、以降必ず再利用される。関数 Swap は 50000 回以上呼ばれるものの、252 回しか実行されない。関数 Swap を毎回実行したときのステップ数は、プログラム全体のステップ数の 27%に相当し、この割合だけ、高速化できたことになる。また、事前実行を用いると、実質 MSP が使用できる RB エントリ数がわずかに減少するので、RB にすべてのパターンを登録しておくことができなくなり、Swap の実行回数が若干増加する。

また、再帰呼び出しのある関数 Permute に関しては、再帰されない条件のときのみ、事前実行を用いて再利用できる。ただし、再帰されない場合、Permute 自体のステップ数が小さくなるため、大きな削減はできない。

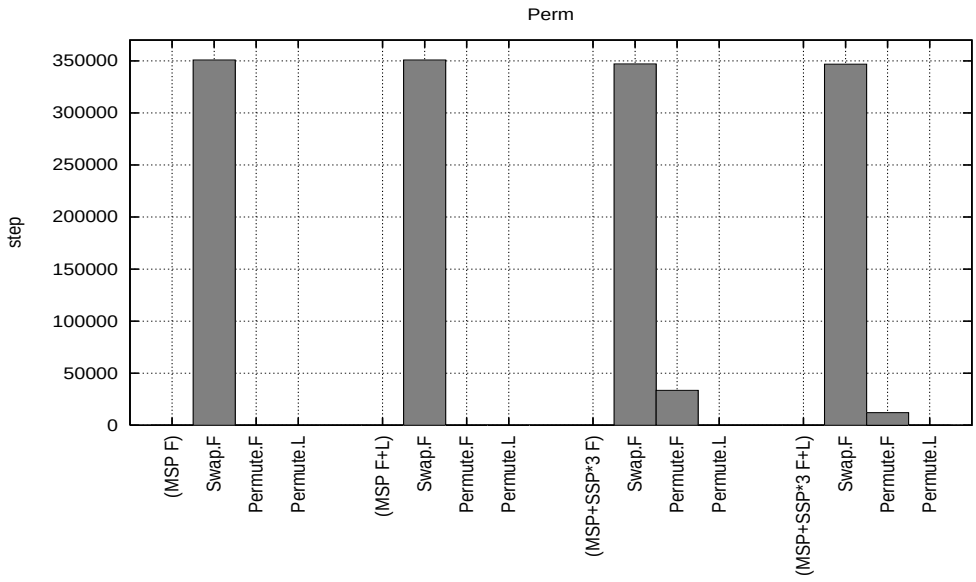

図 13: 命令区間別削減ステップ数 (Perm)

Towers　ハノイの塔問題を解くプログラムである。ループが存在しないために、ループ再利用の効果はない。主に Towers では、ディスクを持ち上げる関数 Pop およびディスクを置く関数 Push が呼び出される。これらの関数は、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) が引数である。この入力値のパターン数は限られており、再利用の効果は大きい。評価に使用した塔の高さを 14 とする Towers では、関数 Pop および関数 Push がそれぞれ 16400 回呼び出されるのに対し、入力パターンはそれぞれ 160 程度であるため、両関数の実行にかかるステップ数を 99%削減することができる。さらに事前実行が関数 Pop に対して有効である。ただし、関数再利用により、呼び出された関数 Pop の 99%は実行されないため、全体から見た事前実行の効果は非常に小さい。また、図 14に示す Move は、関数 Pop および関数 Push を呼び出してディスクの移動を行う関数であり、ハノイの塔問題では同じ状況での同じ動作は存在しないため、再利用によりステップ数を削減することは不可能である。事前実行機構を用いた際も、入力値の予測が困難であるために、ステップ数を削減できない。関数 Move を呼ぶ関数 tower においても、再利用による高速化は不可能である。図 14中の Getelement は、各ディスクが保持している、１つ上に存在するディスクへの情報を書き換え

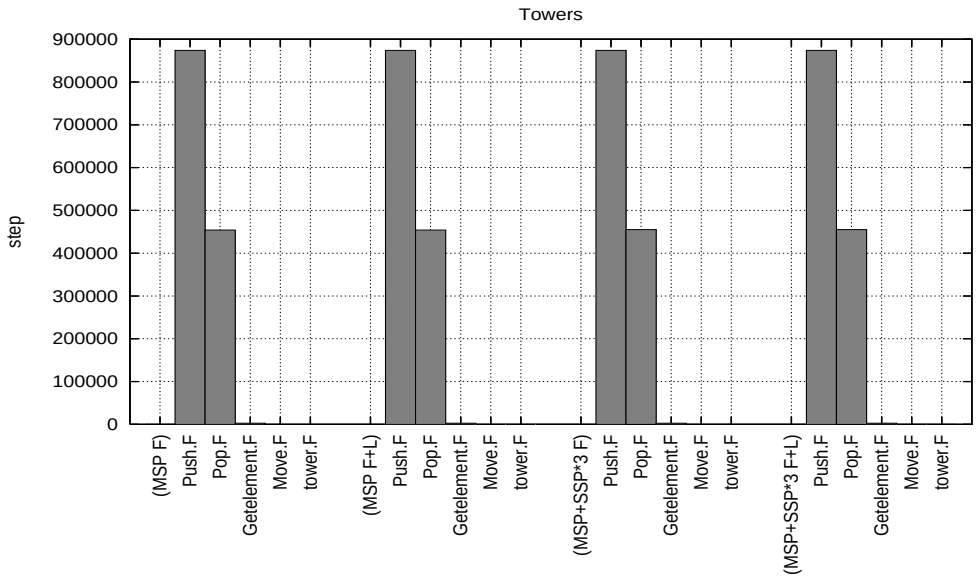

図 14: 命令区間別削減ステップ数　(Towers)

る関数である。入力値のパターンが限られているため、再利用を適用できるものの、実行ステップ数は小さい。

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 Place と外す関数 Remove および、ピースをつめることが可能であるか否か調査する関数 Fit 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。これらの操作が繰り返されるため、関数再利用のみでも効果が大きく現れ、75%の命令ステップ数を削減することができる。

ピースをつめる試行を行う関数 Trial 内のループでは、計算は実行されず、以上の 3 つの関数が呼び出される。まず、関数 Fit が呼ばれ、その結果により関数 Place および関数 Remove の呼び出しが決定する。Place および Remove の呼び出しが実行されず、Fit が再利用可能の場合のみ、Trial 内のループを区間単位とする再利用が可能である。これにより、Trial 内のループを再利用する場合に削減される命令ステップ数は、関数 Fit のみを実行する際のステップ数にほぼ等しい。このため、Trial 内のループに再利用を適用することによる効果はほとんどない。関数 Fit では、立方体空間内の情報を参照していく際にパラメタが単調変化するループ (図 15中、Fit.L) が使用され、ループ事前実行の効果が大きい。この結果、Puzzle の実行ス

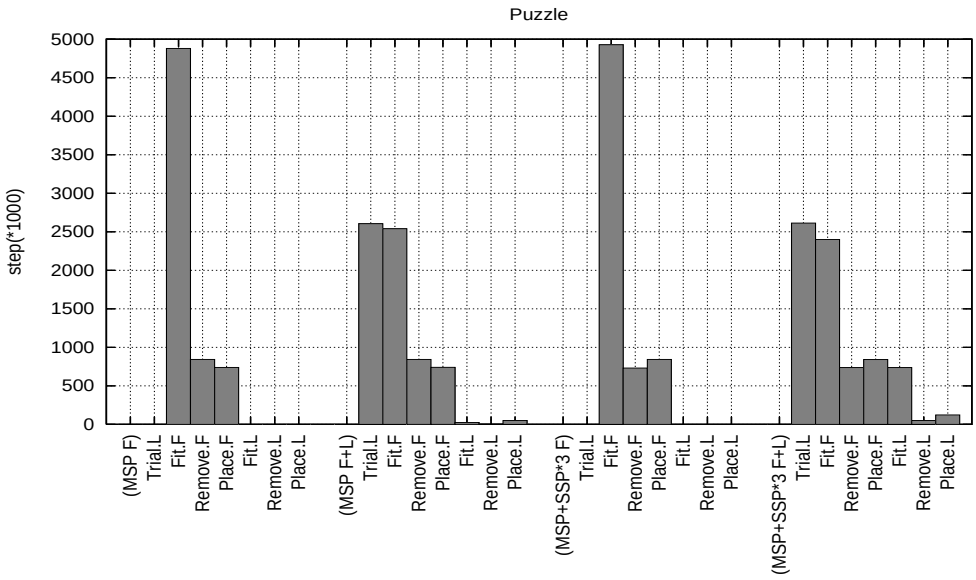

図 15: 命令区間別削減ステップ数　(Puzzle)

テップ数を 90%削減したすることができた。また、関数 Place および関数 Remove においても同様のループが使用されるものの、随時、立方体空間内の情報を書き換えるため、ループ事前実行の効果は小さかった。

Queens 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 Try が、再帰的に呼び出される。置けない場所の状況が一致すれば再利用が可能である。関数 Try 内のループを区間単位とすれば、同一パラメタの出現が多くなり、再利用の効果も大きくなる。ただし、いずれの場合も、SSP の実行の際に、入力値を正しく予測することは困難であり、事前実行によるステップ数削減は不可能である。図 16に示すループ Doit.L は初期設定時に動作し、パラメタが単調変化するために事前実行を用いてループ再利用が適用できるものの、命令ステップ数は小さい。

Trees 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、同じ状態かつ同じ値のノード追加が発生することはありえず、再利用の効果はない。ただし、プログラム内で呼び出される関数 malloc に再利用および事前実行が適用できている。

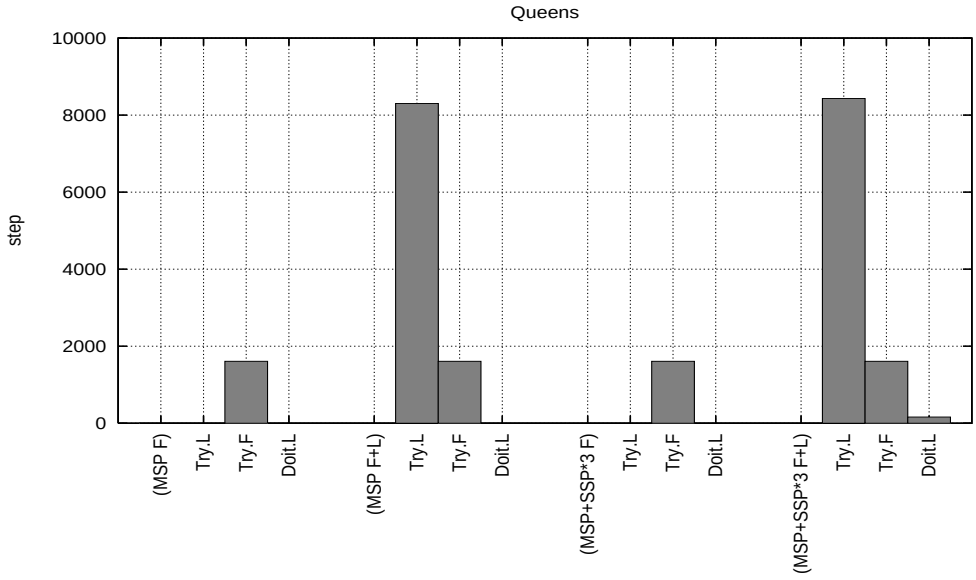

図 16: 命令区間別削減ステップ数 (Queens)

図 17中の関数 Rand や関数 malloc を除く命令区間は、いずれも関数 malloc 内で呼び出され、ループ再利用や事前実行が適用できている。関数 malloc に事前実行を用いて関数のみの再利用を適用した場合、理想的に 4 分の 3 の実行回数を削減できる。ループ事前実行を用いた場合に削減数が減少する原因は、関数 malloc 内に多重ループが存在し RW の深さを越えるためである。多重ループが実行されない場合のみ、malloc に多重再利用が適用できる。このため、Trees に対して事前実行のループ構造への適用を行うと、図 7に示したとおり、命令ステップ数が増加する。RW が保持する多重構造の限界をより深くすると、ループ事前実行の場合においても、関数 malloc の再利用が可能になり、再利用を適用しない場合に対し、プログラム全体で命令ステップ数を 40%削減できる。また、関数 Rand の実行は Quicksort のときと同様に、ループ事前実行により削減される。

FFT　高速フーリエ変換を行う。プログラム内には関数が存在せず、主にループを用いて計算が実行される。ループ事前実行を適用することにより、最外ループ Fft.L(1) とその内部ループ Fft.L(2) の実行回数はそれぞれ 50%程度に削減される。バタフライ演算を行う最内ループ Fft.L(3) も事前実行により、削減ステップ数が大きくなる。プログラムの特性上、ループのパラメタの変化が線形的であり、パラメタの単調変化に追随する本事前実行に

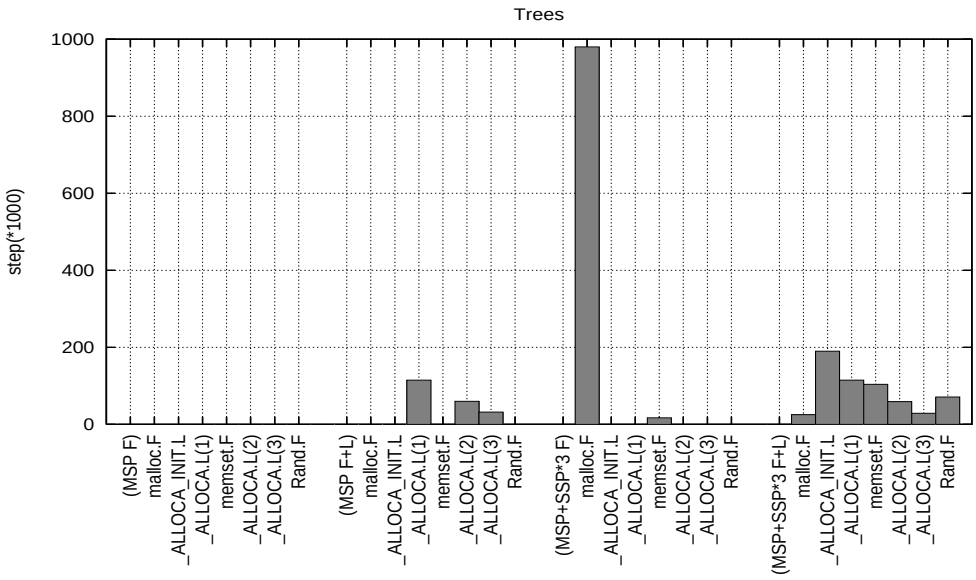

図 17: 命令区間別削減ステップ数 (Trees)

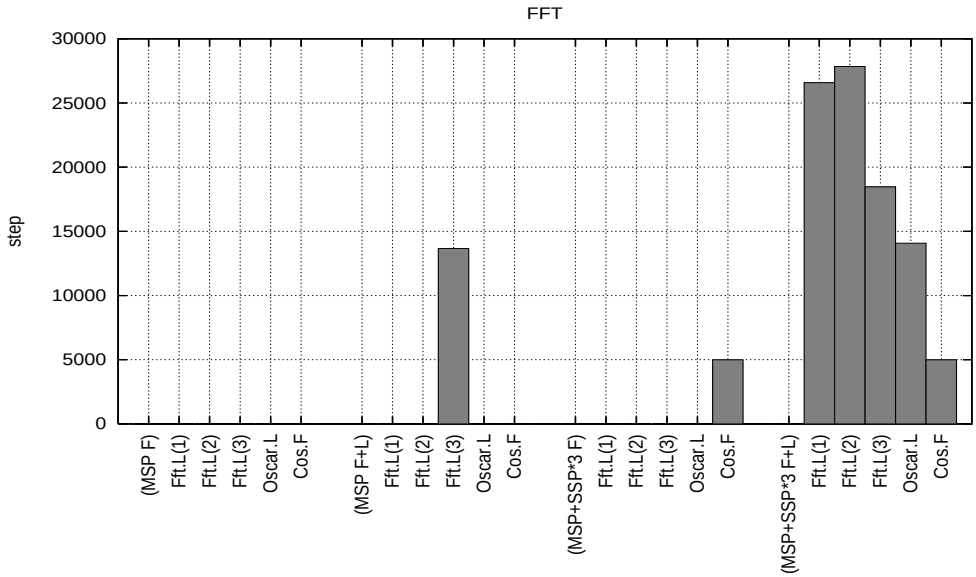

図 18: 命令区間別削減ステップ数 (FFT)

おいて、この削減率は妥当な結果である。

また、図 18に示す Oscar.L は初期パラメタを設定する際に用いられるループであり、関数 Cos は、余弦の値を求める際に使用される。それぞれ、パラメタが単調変化するために、事前実行の効果が大きい。

4.3 考察

大域変数の書き換えなどにより、次回分の事前実行しか効果がない場合に、いくつかの問題点があげられる。まず、命令区間が呼ばれる間隔が短いために、事前実行が間に合わないことがある。例えば、Intmm 内の関数 Rand の場合、SSP が正しく事前実行を行っているものの、事前実行完了以前に関数が呼び出され、RB エントリの調査が実行される。このため無駄になっている SSP 実行をいかに有効なものとするかが課題である。

また、次回の呼び出しまでに SSP の事前実行が完了し、再利用の適用が可能である場合においても、Quicksort および Bubblr 内の関数 Rand のように、命令区間の実行完了時の 4 回に 1 回、SSP の事前実行が開始されない例が存在する。これらの問題における、事前実行機構の改善が今後の課題である。

第 5 章　まとめ

本論文では、ABI を利用した関数やループからなる命令区間に再利用および事前実行を適用する高速化手法を提案し、既存ロードモジュールの高速化、単調変化に追随、投機のキャンセルの排除、プログラミング時の直接利用を可能とした。Stanford-integer を用いてサイクルシミュレータによる性能評価を行った結果、プログラムにより効果が異なるものの、最大 75% のサイクル数を削減できることがわかった。今後の課題として、SPEC2000 での評価、また、理想どおりに再利用されない部分の、事前実行機構の再考などによる改善があげられる。

謝辞

本研究の機会を与えてくださった、富田眞治教授に深く感謝の意を表します。

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

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

参考文献

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

[2] K.Wang and M.Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. 30th MICRO,pp.281-290,1997.

[3] L.Codrescu,D.S.Wills and J.Meindl. Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications. IEEE Trans. on Comp.,Vol.50,No.1,pp.67-82,2001.

[4] G.S.Sohi and A.Roth. Speculative Multithreaded Processors. IEEE Comp.,Vol.34,pp.66-73,2001.

[5] P.Marcuello and A.Gonzalez. Thread-Spawning Schemes for Speculative Multithreading. 8th HPCA 2002.

[6] J.D.Collins,H.Wang,D.M.Tullsen,C.Hughes,Y.F.Lee,D.Lavery and J.P.Shen.

Speculative Precomputaion: Long-range Prefetching of Delinquent Loads. 28th ISCA,pp.14-25,2001.

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

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

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

[10] 重田大介, 小川洋平, 山田克樹, 中島康彦, 富田眞治: 命令畳み込み, データ投機および再利用技術を用いた Java 仮想マシンの高速化, 情報処理学会論文誌: ハイパフォーマンスコンピューティングシステム,HPS1,pp.13-18,2000.

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

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

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

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

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

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