動的命令解析に基づく多重再利用および並列事前実行

中島康彦[†] 津邑公暁[††] 五島正裕[††]
森眞一郎[††] 富田眞治[††]

関数およびループ構造に対して，多重再利用や並列事前実行を適用することにより，SPARC Application Binary Interfaceに従って作成されたプログラムを高速化する手法を提案する．本提案の特長は，コンパイラによる専用命令の埋め込みを必要とせず，実行時に命令を解析することにより関数およびループの多重構造を検出して高速化を図る点にある．Stanford-Integerでは最大75%，SPEC95では最大45% のサイクル数を削減できることを示す．

Multilevel Reuse and Parallel Precomputation Based on Dynamic Instruction Analysis

Yasuhiko Nakashima,[†] Tomoaki Tsumura,[††]
Masahiro Goshima,[††] Shinichiro Mori[††] and Shinji Tomita[††]

This paper proposes a speed-up technique introducing multilevel reuse and parallel precomputation of functions and loops. We assume the target load modules obey SPARC application binary interface. The major point of this proposal is to detect the multilevel structure of functions and loops dynamically without any additional instructions controlled by the compiler. We show the maximum ratio of eliminated cycle reaches to 75% against Stanford-Integer and 45% against SPEC95 benchmark programs respectively.

1. はじめに

区間再利用（以下，再利用と略す）とは，命令区間において，過去に出現した同一入力による実行の際には，再度命令列を実行することなく，過去の実行結果の再利用により，高速化を図ることである．また，事前実行とは，あらかじめ予測した入力値に基づいて実行結果を登録しておくことにより，入力が単調に変化する場合など，過去の実行結果の単純な再利用では効果がない局面においても高速化を図ることである．

さて，後述するように，現在提案されている再利用は，コンパイラによる専用命令の埋め込みを前提としている．これは，プロセッサが，個々の命令から得られる情報のみを基に基本ブロックを切り出すことが難しいためである．残念ながら，専用命令を前提とする場合は，専用のコンパイラが生成したロードモジュールだけが高速化の対象となる．第1の課題は既存ロードモジュールをいかに高速化するかである．次に，再利用を行う場合，命令区間を大きくするほど効果が大きいものの，区間に局所的な変数が混入すると，記録すべき入出力数が膨大となるだけでなく，コンテクストに依存しない再利用が困難となる．第2の課題は，再利用区間に局所的な入出力を排除しながら，いかに大きな命令区間を確保するかである．また，過去の演算を記憶するだけの単純な再利用では，パラメータが単調変化する場合にまったく効果がない．第3の課題は，単調変化に対していかに追随するかである．このような場合，一般的に予測に基づく投機的実行機構の導入が考えられるものの，命令区間を大きくすると投機のキャンセルが困難となる．第4の課題は，いかにキャンセルを排除するかである．さらに，命令レベルの再利用では，プログラマが高速化機構の存在を意識して直接利用することが困難である．キャッシュの構成を意識してプログラミングするのと同様に，再利用機構を前提としてプログラミングすることができれば，より効率の良いプログラムを書ける可能性がある．このようなプログラミングを可能とするためには，再利用機構における入力と出力の関連が，ソースコード上のなんらかの構造と対応する必要がある．たとえば，関数に写像することにより，関数の入出力および値の

† 京都大学/科学技術振興事業団さきがけ研究21
Kyoto University/PRESTO, JST
†† 京都大学
Kyoto University

局所性を意識してプログラミングすることが，そのまま再利用機構を利用した高速化に結び付く．第5の課題は，いかにプログラミング時の直接利用を可能とするかである．

本提案では，この5つの課題を解決するために，一般的に，ロードモジュールがABI（Application Binary Interface）に従って作られることを利用する．特に，SPARC ABI[1]を利用して，関数および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，既存ロードモジュールへの適用を可能とする．さらに，関数およびループ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除外し，効率向上に貢献する．特に，関数については，関数の複雑さにかかわらず，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る再利用および事前実行が可能であることを述べる．また，関数再利用機構に対して若干の機能を追加することにより，ループ再利用も実現可能であることを述べる．

以下では，まず，単一の関数またはループを対象として，1レベルの再利用を行うために必要な機構について詳述し，さらに，関数およびループが多重構造を形成している場合に対応するための機能拡張について述べる．次に，多重構造における事前実行機構について説明し，最後に，Stanford-Integerおよび SPEC95を用いた評価を行う．

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[2),3)]．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[4)~6)]．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つね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[7)]．

一方，再利用[8)~10),15)]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，正しい出力値をただち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を用い，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の実行時間を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[9),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[17)]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ヘッドが問題となる．これに対し，我々の提案の大きな特長は，再利用区間の「入力値」を予測の対象としている点であり，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まったくない．

さて，関数に注目した再利用および並列事前実行技術[16)]は，本稿が提案する多重再利用に最も近い関連研究である．ただし，関数呼び出し命令の分岐先から復帰命令までを再利用区間としており，明示的な関数呼び出しがなければ，再利用による高速化が不可能である．本稿では，さらに，後方分岐命令の分岐先から，同一後方分岐命令までの命令区間を同様に再利用区間とし，関数とループの複雑な入れ子構造についても多重再利用を可能とする方法を提案している．

3. 1レベル再利用

本章では，単一の関数またはループを対象として，何が入力で，何が出力であるかを明らかにし，1レベルの再利用を行うために必要な機構について詳述する．プログラムにおいては，一般的に関数とループが多重構造を形成している．関数A（以下，A）が関数B（以下，B）を呼び出し，Bがループ C（以下，C）を実行する構造を**図1** (a) に示す．大域変数（Globals）は，

A，B，Cの入出力（$A/B/C_{in/out}$）となりうる．Aの局所変数（Locals-A）は，Aの入出力ではないものの，ポインタを通じてBおよびCの入出力（$B/C_{in/out}$）となりうる．また，AからBへの引数（Args）はBおよびCへの入力（B/C_{in}），返り値（Ret.Val.）はBおよびCからの出力（B/C_{out}）となりうる．同様に，Bの局所変数（Locals-B）は，Cの入出力（$C_{in/out}$）となりうる．

3.1 関数再利用

さて，コンテクストに依存せずにBを再利用するには，B実行時に，$B_{in/out}$ のみを入出力として登録しなければならない．具体的には，図1 (b) に示すメモリマップにおいて，$B_{in/out}$ を含まない領域はLocals-Bのみであることから，$B_{in/out}$ を識別するには，GlobalsとLocals-Bの境界，および，Locals-BとLocals-Aの境界をそれぞれ確定しなければならない．前者については，一般的に，OSが実行時のデータサイズとスタックサイズの上限を決めることを利用し，OSからあらかじめ境界（LIMIT）が与えられるとし，後者については，Bが呼び出される直前のスタックポインタの値（SP in A）を用いることにより解決することができる．詳細は，末尾の付録 A.1 を参照されたい．

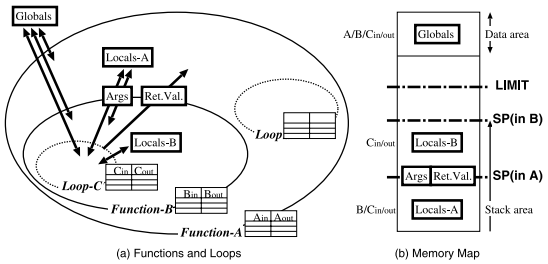

図1 多重入出力構造
Fig. 1 Multilevel I/O structure.

関数再利用を実現するために，関数管理表（RF）および入出力記録表（RB）を設けることにする．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**図2**に示す（網かけ部分は後述）．複数の関数を再利用可能とするには，本構成を複数組用意する．各表のVおよびLRUは，各々，有効エントリの表示およびエントリ入替えのヒントである．RFは，関数の先頭アドレス（Start）および参照すべき主記憶アドレス（Read/Write）を保持し，RBは，関数呼び出し直前の%sp（SP），引数（V：有効エントリ，Val.：値），主記憶値（Mask：Read/Writeアドレスの有効バイトを示す4ビット，Value：値），返り値（V：有効エントリ，Val.：値）を保持する．返り値は，%i0〜1（リーフ関数では%o0〜1に読み換える）または%f0〜1に格納され，%f2〜3を使用する返り値（拡張倍精度浮動小数点数）は対象プログラムには存在しないものと仮定する．Readアドレス（3）はRFが一括管理し，有効バイト位置を示すマスクおよび値（4）はRBが管理することにより，Read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している．

単一の関数を再利用するには，まず，関数実行時に，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．主記憶参照については，各オペラン

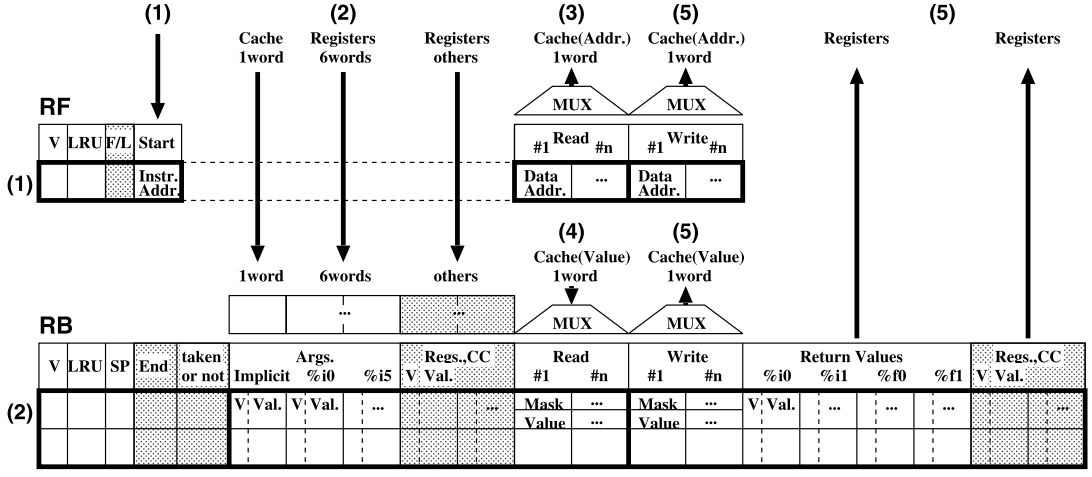

図2 1レベル再利用のための表構造
Fig. 2 Structure of onelevel reuse buffer.

ドアドレスが SP+64 と一致する場合は暗黙引数として登録処理を行い，LIMIT 以上かつ SP+92 未満の場合は局所変数として登録対象外とする．その他の場合は，登録対象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超えた，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擾乱が発生しなかった場合，復帰命令を実行した時点で，登録中の入出力表エントリを有効にする．ただし，これらの撹乱のうち，入出力数が入出力表の容量を超えた場合については，放置しておくと，以後まったく登録ができない状況が続くため，使用頻度の少ない RB エントリを解放することにより，引き続く登録を可能とする．以後，関数を呼び出す前に，図 2 に示した番号順に，(1) 関数先頭アドレスが一致する RF エントリを検索し，(2) 対応する RB の中から引数が完全に一致する 1 つまたは複数のエントリを選択し，(3) 各 Read アドレスごとに，選択した RB エントリのすべての Mask の論理和（4 ビット）を検査し，$0000_{(2)}$ 以外となるアドレス，すなわち少なくとも 1 バイトを比較しなければならないことが判明した Read アドレスに関して，順に 4 バイトずつをロードし，(4) 各 RB エントリごとに格納された Mask と値を用いて一致比較を行う．すべての入力が一致する RB エントリが存在した場合に，(5) 登録済の出力（返り値，大域変数，および，A 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．

3.2 ループ再利用

さて，次にループへの応用を試みる．**図 3** に関数とループの類似性を示す．関数に含まれる命令は，call/jmpl 命令の分岐先から ret 命令（厳密にはディレイスロットを含む）までであることから，call/jmpl 命令の検出時に再利用の可否判定を行い，再利用できない場合には，call/jmpl 命令の検出から ret 命令の検出までを再利用区間とすることができた．一方，ループは，一度後方分岐命令に到達し，後方分岐が成立した後に，再度同じ後方分岐命令に到達してはじめて，それまでの命令列がループを形成していたことが分かる．この方法では，本質的に，ループの第 1 回目のイタレーションを再利用することはできないものの，一般的にループは繰り返し実行されることを考えると，この制限は軽微なものであると考える．

以上述べたように，後方分岐命令の分岐先に始まり，同一の後方分岐命令に終わる命令区間について，この

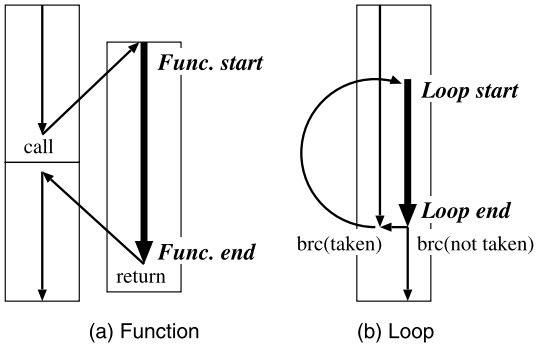

図 3 関数とループの類似性
Fig. 3 Analogy of function and loop.

間の入出力を登録しておくことにより，関数と同様にループを再利用することができる．ただし，多重ループなど，複数の異なるループが同じ先頭アドレスを共有する場合があるため，1 つの RF に属する複数の RB が，それぞれ後方分岐命令アドレスを記憶し，再利用後に引き続いて実行すべき命令アドレスを分ける必要がある．また，関数では局所変数の登録を除外することができるものの，ループでは除外することができない．これは，ループ内の局所変数が ABI では規定されないためである．すなわち，ループの再利用に必要な入出力は，参照したレジスタおよび主記憶アドレスのすべてである．このため，ループ再利用には，図 2 の網かけ部分に示す拡張が必要となる．具体的には，RF に，関数とループの区別（F/L），RB に，後方分岐命令アドレス（End），分岐方向（taken or not），引数や返り値以外のレジスタおよび条件コード（Regs, CC）を追加する．ループの場合，RB 中の SP の値を 0 に設定することにより，LIMIT 以上かつ SP+92 未満を登録対象外とする機構を生かしたまま，すべての主記憶参照を登録することができる．

同一の後方分岐命令に到達する前に関数から復帰したり，前節にあげた擾乱が発生するなど，ループの入出力登録が中止されなければ，登録中のループに対応する後方分岐命令を検出した時点で，登録中の入出力表エントリを有効にし，現ループの登録を完了する．さらに，後方分岐命令が成立する場合は，次ループが再利用可能かどうかを判断する．すなわち，後方分岐する前に，図 2 に示した番号順に，(1) 後方分岐先アドレスを検索し，(2) レジスタ入力値が完全に一致するエントリを選択し，(3) 関連する主記憶アドレスをすべて参照して，(4) 一致比較を行う．すべての入力が一致した場合に，(5) 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

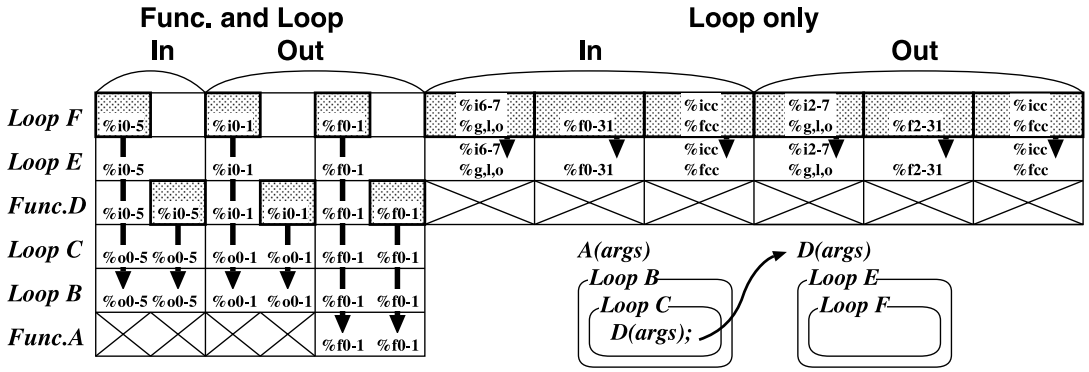

図 4 レジスタ入出力の影響範囲
Fig. 4 Scope of register I/O.

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，次ループの実行および RB への登録を開始する．

4. 多重再利用

前述した 1 レベル再利用機構を用いることにより，図 1 (a) に示したリーフ関数 B や最内ループ C をそれぞれ再利用することができる．これに対し，関数 A を一度実行しただけで，入れ子関係にある A，B，C のすべての命令区間が再利用可能となるよう登録を行う仕組みが多重再利用である．本章では，多重再利用に必要な機能拡張について述べる．**図 4** に，関数 A，D とループ B，C，E，F の入れ子構造において，内側の構造のレジスタ入出力（網かけ）が，外側の構造のレジスタ入出力となる影響範囲（矢印）を示す．たとえば，F 内部において入力として参照された%i0〜5 は，E および D に対する入力でもあり，さらに，D を呼び出した C，B に対する入力（ただし%o0〜5 に読み換える）でもある．A にとって%o0〜5 は局所変数であるため影響範囲は B までとなる．別の見方をすれば，D の内部で%i0〜5 が参照された場合には，B が直接的に%o0〜5 を参照しなくても，%o0〜5 を B の入力値として登録する必要がある．F 内部において出力された%i0〜1 についても同様である．

浮動小数点レジスタはレジスタウィンドウに含まれないため，出力された%f0〜1 は，A を含む全階層の出力となる．一方，その他のレジスタは，関数を越えて影響が及ぶことはないため，F 内部における入出力の影響範囲は E までとなる．主記憶に対する入出力については，前述した，関数呼び出し直前の%sp（SP）と比較する方法を入れ子の全階層に対して適用することにより，影響範囲を特定することができる．

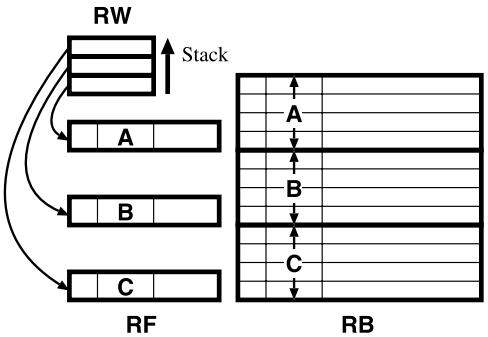

図 5 多重再利用のための表構造
Fig. 5 Structure of multilevel reuse buffer.

以上のことから，多重再利用を実現するには，前述した RF および RB を関数やループの入れ子構造と関連付ける機構が必要である．**図 5** に示すように，再利用ウィンドウ（RW）を装備することにより，現在実行中かつ登録中である 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．関数やループの実行中は，RW に登録されているすべてのエントリについて，これまでに述べた方法に基づき，レジスタおよび主記憶参照を登録していく．この際，あるエントリに関して，

(1) 登録可能項目数の超過
(2) 引数の第 7 ワードの検出
(3) システムコールの検出

により，再利用不可能であると判断した場合には，RW を用いて，そのエントリに対応する RB および上位の RB を特定し，登録を中止することができる．

なお，RW の深さは有限であるものの，一度に登録可能な多重度を超えて，関数やループを検出した場合には，外側の命令区間から順次登録を中止し，より内側の命令区間を登録対象に加えることにより，入れ子関係の動的変化に追随する．また，実行および登録中

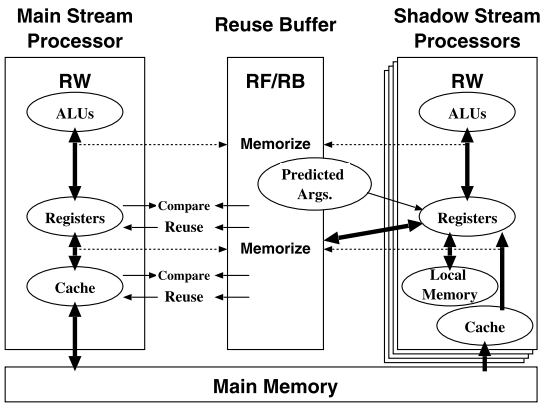

図 6 並列事前実行機構
Fig. 6 Structure of parallel precomputation.

（たとえば A）に，再利用可能な命令区間（たとえば D）に遭遇した場合には，登録済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 の深さを超える A の多重再利用も可能となる．多重再利用の詳細な手順については，末尾の付録 A.2 を参照されたい．

5. 並列事前実行

以上に述べた，関数やループの多重再利用では，RB エントリの生存時間よりも同一パラメータが出現する間隔が長い場合や，パラメータが単調に変化し続ける場合にまったく効果がない．我々は，多重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．並列事前実行機構の概要を**図 6** に示す．RW，演算器，レジスタ，キャッシュは各プロセッサごとに独立しており，RF，RB，主記憶は全プロセッサが共有する．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．さて，並列事前実行における課題は，

(1) どのように主記憶一貫性を保つか
(2) どのように入力を予測するか
(3) どのように RB エントリを入れ替えるか
(4) どの命令区間を実行するか

である．

5.1 主記憶一貫性

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である．特に，予測した入力パラメー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 とで異なる．これを解決するために，図 6 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，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超えた場合など，実行を継続できない場合は，事前実行を打ち切る．

ところで，SSP が局所メモリを参照するためには，SSP のスタックポインタを初期化しておく必要がある．局所メモリに関する参照は関数再利用時の入出力に含まれないことから，スタックポインタの値は MSP と同じである必要がなく，LIMIT の値に，局所メモリの容量を加えた値を初期値としてよい．一方，ループの場合は，前述のようにすべての主記憶参照を RB に登録する必要があるため，スタックポインタの初期値は MSP がループの実行を開始した時点の値に初期化しなければならない．

なお，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．

5.2 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[3]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．

5.3 RB エントリの入替え

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かが

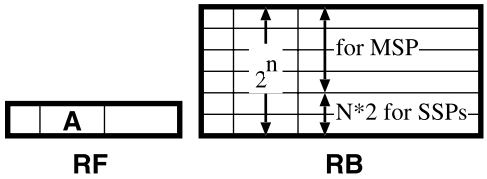

図 7 RB の分割
Fig. 7 Partitioning of RB.

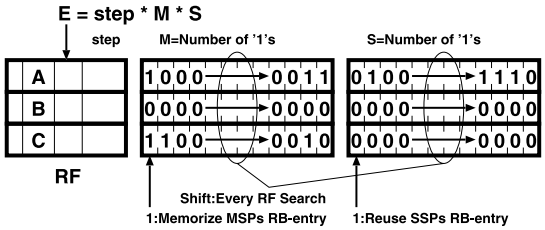

図 8 命令区間選択機構
Fig. 8 Structure of region selection.

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分かれると考えられる．前者であれば，LRU による入替え，後者であれば，FIFO による入替えが有効である．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ただ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 がただ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 7** に示す．

5.4 命令区間の選択

次に，どの命令区間を SSP に事前実行させるかが課題である．同一パラメータが出現する間隔が長い命令区間や，パラメー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，事前には分からない．このため，RF に新規に登録された命令区間については，ただちに 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) $*$ (MSP登録回数) $*$ (SSP登録エントリの再利用回数) 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 8** に示す．RF の各エントリごとに 2 つのシフトレジスタを設け，MSP が RF を検索するとき，すなわち，関数呼び出し命令または後方分岐命令を検出したときに，全エントリのシフトレジスタを右に 1 ビットシフトする．検索の結果，再利用不可であった場合には，MSP が命令区間の実行を開始し，RB への登録が完了したときに，対応する RF エントリの M（MSP 登録回数）を表すシフトレジスタの左端に 1 をセットする．一方，検索の結果，SSP が生成した RB エントリが再利用可能であった場合には，対応する RF エントリの S（SSP 登録エントリの再利用回数）を表すシフトレジスタの左端に 1 をセットする．事前実行機構が効率良く動作している場合には M および S が 1 以上となり，削減ステップ数や再利用頻度の高い区間が選択される．一方，そもそも MSP による登録ができない区間は M が 0，SSP による事前実行結果がまったく再利用されない区間は S が 0 となり，事前実行の対象外となる．

表 1 シミュレーション時のパラメータ
Table 1 Simulation parameters.

D-Cache	64 Kbytes
Line Size	64 bytes
Ways	4
Cache Miss	20 cycles
Register Window	4 set
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 byte/cycle
RB(Reg.)→Register Write	1 cycle
SSP Local Memory	64 Kbytes

6. 性能評価

評価には，これまでに述べた機構を搭載した単命令発行の SPARC-V8 シミュレータを用い，MSP および SSP のサイクルベースシミュレーションを行った．各パラメータを**表 1** に示す．キャッシュ構成や命令レイテンシは HAL の SPARC64 [18] を参考にした．

さて，本論文の狙いは，理想的な再利用により，どこまで高速化が可能であるかを示すことにあるものの，連想検索のシミュレーションには多大なコストを要するため，連想検索の上限，すなわち，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 1,024 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 1,024 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 1,024 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．

6.1 Stanford-Integer

まず，Stanford-Integer を gcc-3.0.2（-msupersparc -O2）によりコンパイルし，スタティックリンクにより生成したロードモジュールを用いた．ただし，FFT と Queens において，単に同じ処理をそれぞれ 20 回と 50 回繰り返している最外ループは，再利用の効果が無意味に高く現れないよう，各 1 回に変更した．

図 9 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．凡例は，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 通りの測定条件を表す．Quick および Bubble では，再利用および事前実行にループを加えることにより，はじめて命令ステップ数が 15%程度減少した．ループ内に関数呼び出しがない FFT も同様に，ループを加えることにより事前実行の効果が大幅に高まった．Trees では事前実行にループを加えると性能が低下しているものの，表 1 に示した RW の深さを 4 から 8 に増加させたところ，Trees についてのみ「MSP+SSP*3 F+L」の命令ステップ数比が 0.84 から 0.59 に大幅に減少し，ループを加えた効果が現れた．ループ中に再帰呼び出しがある Queens でも，ループを加えることにより約 50%の命令ステップ数削減に成功している．さらに，**図 10** に示すように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，90%近くの命令ステップ数削減に成功している．一方，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，最内ループが関数に括り

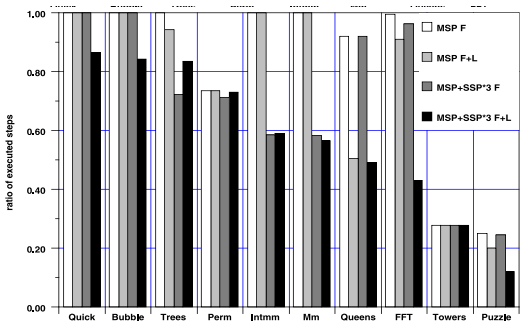

図 9 MSP が実行した命令ステップ数（Stanford-Integer）
Fig. 9 Executed steps on MSP (Stanford-Integer).

```
T(j): loop  if (F(i, j)) {
              k = P(i, j);
              if (T(k) || k == 0 ) return true;
              else R(i, j);
            }
      return false;
```

図 10 Puzzle の構造.
Fig. 10 Structure of Puzzle.

出されているため，ループを加えた効果はみられない．同様に，そもそも MSP のみによる関数再利用の効果が高い Perm および Towers でも，ループを加えた効果はみられない．命令ステップ数の削減率を平均すると，「MSP F」および「MSP+SSP*3 F」ではそれぞれ 18%および 30%に留まるのに対し，「MSP F+L」および「MSP+SSP*3 F+L」ではそれぞれ 24%および 43%に達している．

図 11 に，「MSP+SSP*3 F+L」の構成において，再利用可能であった命令区間を入れ子の深さ（Level 1〜4 以上）ごとに分類して，

（上段） 通常実行時に対する削減ステップ数の比
（中段） 再利用 1 回あたりの削減ステップ数
（下段） 入出力ワード数を入力レジスタ（in-regs），Read アドレス（read），Write アドレス（write），出力レジスタ（out-regs）ごとに平均したもの

を示す．深さ 4 以上では削減ステップ数の比がほぼ 0 となり，Stanford-Integer では入れ子の深さは 3 までを考慮すればよいことが分かる．また，再利用 1 回あたりの削減ステップ数は，深さ 2 以上の多重再利用では数百に達し，特に Puzzle では，貢献度の大きい Level2 および Level3 の削減ステップ数が 400 に達することが分かる．さらに，入力レジスタ数は数個であること，また，削減ステップ数の比を 2%以上に限っ

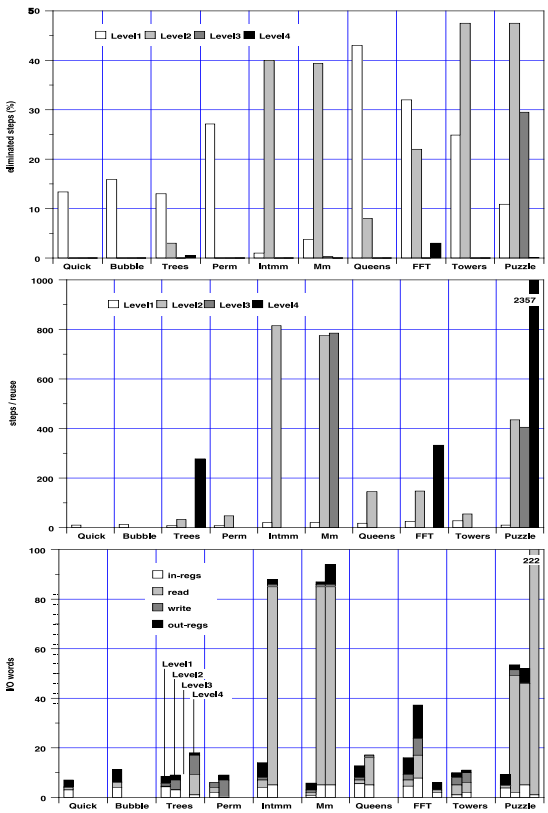

図 11 深さごとの，削減ステップ数比，再利用 1 回あたりの削減ステップ数，入出力ワード数（Stanford-Integer）．
Fig. 11 Ratio of eliminated steps, steps per reuse, I/O words in each depth (Stanford-Integer).

ても，Read/Write アドレスの平均個数は 80/8 を超えないことが分かる．

さて，実際に高速化を達成するには，再利用にともなうオーバヘッドを見極める必要がある．**図 12** は，命令ステップ数（exec）に，表 1 に示した RB(Reg.) ⇔ Register Compare および RB(Read) ⇔ Cache Compare（test），RB(Write) ⇒ Cache Write および RB(Reg.) ⇒ Register Write（write），キャッシュミス（cache），レジスタウィンドウミス（window）の各オーバヘッドを加えた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．Quick，Bubble および Perm では，残念ながら，命令ステップ数の減少分が，test オーバヘッドのために帳消しとなっている．一方，前述のように RW の深さを 4 から 6 に増加させた場合の Trees は，0.80 にサイクル数が減少することが分かっている．ループを加えることにより大きな効果があったのは，Queens，FFT および Puzzle である．サイクル数の削減率を平均すると，「MSP+SSP*3 F」では

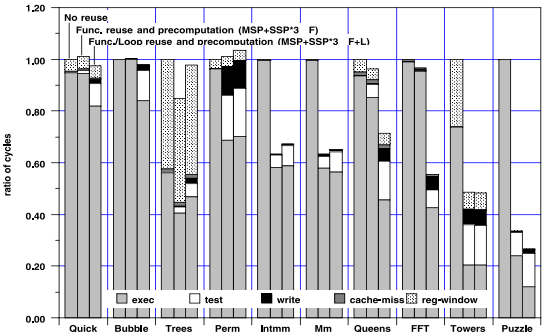

図 12 MSP が実行したサイクル数（Stanford-Integer）．
Fig. 12 Executed cycles on MSP (Stanford-Integer).

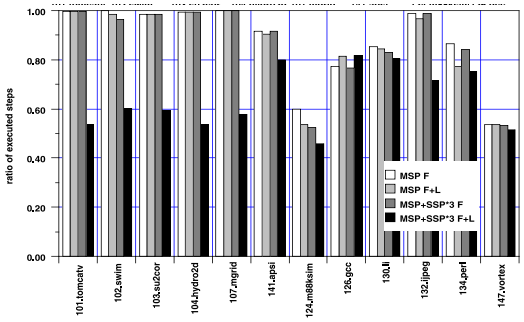

図 13 MSP が実行した命令ステップ数（SPEC95）．
Fig. 13 Executed steps on MSP (SPEC95).

20%にとどまるのに対し，「MSP+SSP*3 F+L」では 30%に達している．また，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．全般的に，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いえる．

6.2 SPEC95

次に，より大きなプログラムである SPEC95（train）を用いて評価を行った．まず，**図 13** に，図 9 と同じ方法による測定結果を示す．126.gcc および 147.vortex を除くプログラムについて，ループを加えたことによる明らかな命令ステップ数の減少がみられる．特に，101.tomcatv から 107.mgrid については，ループを含む事前実行によって，はじめて命令ステップ数が 40〜45%程度減少している．なお，SPEC95 の場合，RW の深さを 4 から 8 に増加してもほぼ同じ結果が得られることが分かっている．命令ステップ数の削減率を平均すると，「MSP F」および「MSP+SSP*3 F」ではそれぞれ 12%および 14%にとどまるのに対し，「MSP F+L」および「MSP+SSP*3 F+L」では

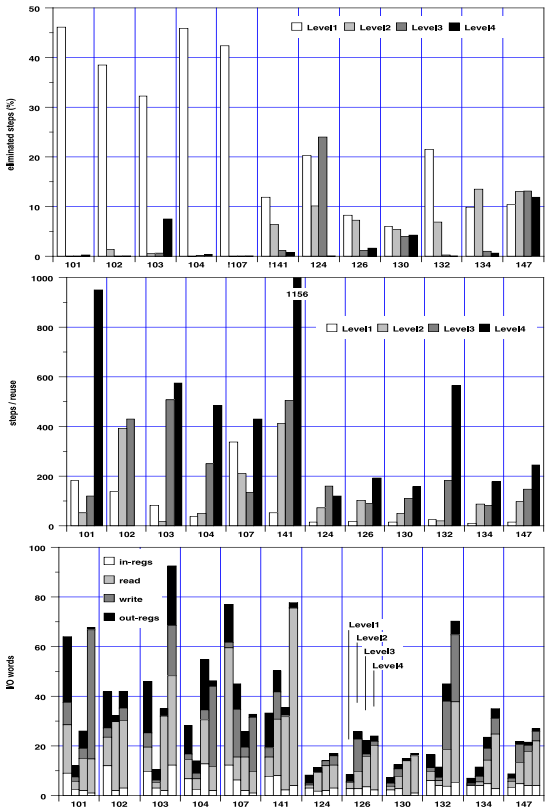

図 14 深さごとの，削減ステップ数比，再利用 1 回あたりの削減ステップ数，入出力ワード数（SPEC95）．
Fig. 14 Ratio of eliminated steps, steps per reuse, I/O words in each depth (SPEC95).

それぞれ 14%および 36%に達している．

図 14 に，図 11 と同じ方法による測定結果を示す．Stanford-Integer 同様，再利用 1 回あたりの削減ステップ数が，深さ 2 以上の多重再利用では数百に達し，特に 103.su2cor，124.m88ksim および 147.vortex では，貢献度の大きい Level3 や Level4 の削減ステップ数がそれぞれ，600 弱，200 弱および 200 前後に達することが分かる．さらに，入力レジスタ数は数個であること，Read/Write アドレスの平均個数は 100 未満であることなどが分かる．

また，**図 15** に，図 12 と同じ方法による測定結果を示す．ループを加えた効果がみられた 101.tomcatv から 107.mgrid については，サイクル数でも 30〜40%程度減少している．サイクル数の削減率を平均すると，「MSP+SSP*3 F」では 10%にとどまるのに対し，「MSP+SSP*3 F+L」では 25%に達している．ところで，本提案では，RB の内容と主記憶の比較を行う際にキャッシュを経由しており，キャッシュミスの増加による性能低下が懸念されたものの，通常

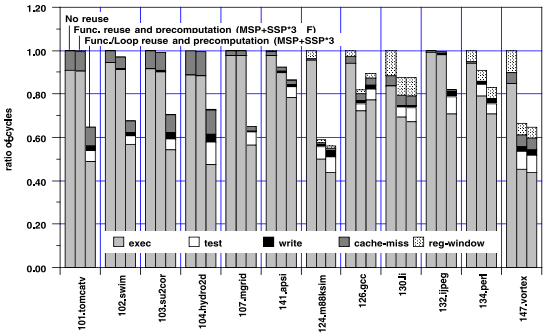

図 15 MSP が実行したサイクル数（SPEC95）．
Fig. 15 Executed cycles on MSP (SPEC95).

実行と本提案の間にキャッシュミスの差がほとんどないことが分かる．また，li と vortex では，Towers と同様にレジスタウィンドウミスが減少していることが分かる．全般的に，再利用のオーバヘッドのうち test が占める割合は，Stanford-Integer に比べて小さい．

7. おわりに

本稿では，ABI を利用して，関数やループからなる命令区間に多重再利用および並列事前実行を適用する高速化手法を提案し，

(1) 既存ロードモジュールの高速化
(2) 局所的な入出力の排除
(3) 単調変化に追随
(4) キャンセルの排除
(5) プログラミング時の直接利用

を可能とした．Stanford-Integer および SPEC95 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，Stanford-Integer では最大 75%，SPEC95 では最大 45%のサイクル数をそれぞれ削減できることが分かった．サイクル数の削減率を平均すると，関数のみを対象とした再利用および事前実行ではそれぞれ 20%および 10%にとどまるのに対し，ループを加えた場合には，それぞれ 30% および 25%に改善されることが分かった．また，RB の内容と主記憶の比較にキャッシュを用いたものの，キャッシュミスの増加による性能低下はほとんどないことが分かった．

参考文献

1) Paul, R.P.: *SPARC Architecture, Assembly Language Programming, and C*, Prentice-Hall (1999).
2) Lipasti, M.H. and Shen, J.P.: Exceeding the Dataflow Limit via Value Prediction, *29th MI-*

CRO, pp.226–237 (1996).
3) Wang, K. and Franklin, M.: Highly Accurate Data Value Prediction using Hybrid Predictors, *30th MICRO*, pp.281–290 (1997).
4) Codrescu, L., Wills, D.S. and Meindl, J.: Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications, *IEEE Trans. on Comp.*, Vol.50, No.1, pp.67–82 (2001).
5) Sohi, G.S. and Roth, A.: Speculative Multithreaded Processors, *IEEE Comp.*, Vol.34, No.4, pp.66–73 (2001).
6) Marcuello, P. and González, A.: Thread-Spawning Schemes for Speculative Multithreading, *8th HPCA* (2002).
7) Collins, J.D., Wang, H., Tullsen, D.M., Hughes, C., Lee, Y.F., Lavery, D. and Shen, J.P.: Speculative Precomputation: Long-range Prefetching of Delinquent Loads, *28th ISCA*, pp.14–25 (2001).
8) Sodani, A. and Sohi, G.S.: Dynamic Instruction Reuse, *24th ISCA*, pp.194–205 (1997).
9) Huang, J. and Lilja, D.J.: Exploiting Basic Block Value Locality with Block Reuse, *5th HPCA* (1999).
10) González, A., Tubella, J. and Molina, C.: Trace-Level Reuse, *ICPP* (1999).
11) Yang, J. and Gupta, R.: Load Redundancy Removal through Instruction Reuse, *ICPP* (2000).
12) Yang, J. and Gupta, R.: Energy-efficient load and store reuse, *ISLPED*, pp.72–75 (2001).
13) Connors, D.A., Hunter, H.C., Cheng, B.C. and Hwu, W.W.: Hardware Support for Dynamic Activation of Compiler-Directed Computation Reuse, *9th ASPLOS*, pp.222–233 (2000).
14) Huang, J. and Lilja, D.J.: Extending Value Reuse to Basic Blocks with Compiler Support, *IEEE Trans. on Comp.*, Vol.49, No.4, pp.331–347 (2000).
15) 重田大助，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，No.SIG5(HPS1), pp.28–38 (2000).
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，No.SIG6(HPS5), pp.1–12 (Sep. 2002).
17) Wu, Y., Chen, D.Y. and Fang, J.: Better Exploration of Region-Level Value Locality with Integrated Computation Reuse and Value Prediction, *28th ISCA*, pp.98–108 (2001).
18) FUJITSU/HAL SPARC64-III User's Guide (1998). http://www.sparc.com/standards/

付　　録

A.1 ABI に基づく主記憶アドレスの識別

本節では，与えられた主記憶アドレスが，大域変数であるか，または，どの関数の局所変数であるかを識別する方法について詳述する．ロードモジュールは，SPARC ABI に規定されている以下の条件を満たすと仮定する．なお，%fp はフレームポインタ，%sp はスタックポインタを意味する．

- %sp 以上の領域のうち，%sp+0〜63 はレジスタ退避領域，%sp+68〜91 は引数退避領域であり，いずれも関数の入出力ではない．
- 構造体を返す場合の暗黙的引数は%sp+64〜67.
- 明示的引数はレジスタ%o0〜5，および，%sp+92 以上の領域に置かれる．

さらに，大域変数と局所変数を区別するために，一般的に，OS が実行時のデータサイズとスタックサイズの上限を決めることを利用し，以下を仮定する．

- 大域変数は，LIMIT 未満の領域に置かれる．
- %sp は LIMIT 以下になることはなく，LIMIT〜%sp の領域は無効である．

以上の条件を満たしながら，関数 A が関数 B を呼び出す場合の，引数およびフレームの概要を**図 16** に示す．次に，A の局所変数と B の局所変数を区別する方法について述べる．

（a）は A 実行中の状態である．LIMIT 未満の太枠部分に命令および大域変数，また，%sp 以上に有効な

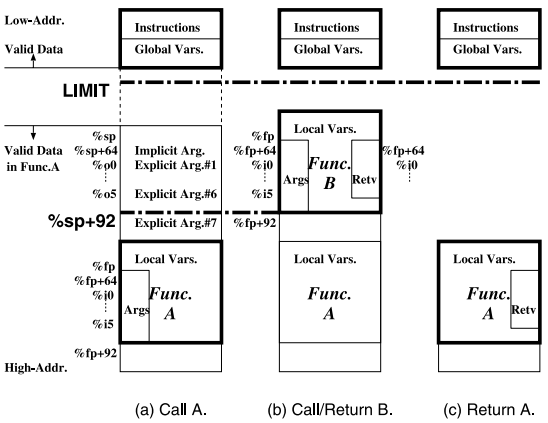

図 16 スタックフレームの概要
Fig. 16 Overview of stack frames.

値が格納されている．%sp+64 には，B が構造体を返り値とする場合の暗黙的引数として，構造体の先頭アドレスが格納される．B に対する明示的引数の先頭 6 ワードはレジスタ%o0〜5，第 7 ワード以降は%sp+92 以上に格納される．ベースレジスタを%sp とするオペランド%sp+92 が出現した場合，この領域は引数の第 7 ワードすなわち B の局所変数である．一方，オペランド%sp+92 が出現しない場合，この領域は A の局所変数である．このように，(a) の時点では，A の局所変数と B の局所変数を区別することができる．

一方，(b) は B 実行中の状態である．引数が入力，返り値が出力，大域変数および A の局所変数が入出力となりうる．ただし，B は可変長引数を受け入れる場合があるため，一般に，%fp+92 以上の領域が A の局所変数か B の局所変数かの区別ができない．局所変数を区別するには，(a) の時点において引数の第 7 ワード以降を検出した関数呼び出しは再利用の対象外とし，検出しない関数呼び出しは，直前に%sp+92 の値を記録しておく必要がある．第 7 ワードを使用する関数呼び出しの出現頻度が低いと予想されることから，この制限による性能低下は軽微なものと考える．以上の準備により，(b) における主記憶参照アドレスが，あらかじめ記録した%sp+92 の値以上の場合は A の局所変数，小さい場合は B の局所変数であることが分かる．B 実行時には，B の局所変数を除外しながら，大域変数および A の局所変数を表へ登録する．

さて，再利用の際は，B の局所変数は入出力から除外されるため，B の局所変数のアドレスが一致している必要がない．このため，いかなるコンテクストであっても，入力さえ一致すれば，再利用することが可能である．ただし，B が参照する大域変数や A の局所変数については，アドレスおよびデータの両方が表の内容と完全に一致する必要がある．B を実行する前に，どのようにして，比較すべき主記憶アドレスを網羅するかが鍵になる．B が参照する，大域変数や A の局所変数のアドレスは，そもそも，B において生成されるアドレス定数や，大域変数/引数を起源とするポインタに基づくため，まず引数が完全に一致する表中のエントリを選択した後に，関連する主記憶アドレスをすべて参照して一致比較を行うことにより，B が参照すべき主記憶アドレスを網羅できる．すべての入力が一致した場合にのみ，登録済の出力（返り値，大域変数，および，A の局所変数）を再利用することができる．

A.2 多重再利用の詳細手順

本節では，C 言語を模した疑似コードにより，多重再利用全体の動作手順を詳述する．なお，再利用機構が動作する契機は分岐命令であるため，命令語の解析結果を基準に説明する．

A.2.1 関数呼び出し命令

Call 命令や%o7 へ現 PC を書き込む jmpl 命令を関数呼び出し命令と判断する．ただし，ディレイスロットに restore 命令が記述されている場合は，関数呼び出しとは見なさない．

```
if (後述の手順により第 7 引数が検出されていた) {
  /* 該関数は登録不可能 */
  RW に積まれている RB エントリをすべて無効化;
  プログラムカウンタを関数の先頭へ進める;
}
else if (該関数の先頭アドレスおよび入力が
  RF/RB にない) {
  /* 該関数は再利用不可能 */
  if (該関数のための既存または空き RF/RB
      エントリがない) {
    /* 登録を開始しない */
    RW に積まれている RB をすべて無効化;
    RW を空にする;
  }
  else {
    /* 登録を開始する */
    既存または空き RF/RB エントリを RW の top
    に積み，該 RF/RB エントリをビジー状態にする;
    if (RW が溢れた)
      RW の bottom に対応する RB を無効化;
  }
  プログラムカウンタを関数の先頭へ進める;
}
else {
  /* 該関数は再利用可能 */
  RB から出力を求めレジスタ/主記憶へ書き込む;
  if (他に関数/ループが RW に登録されている)
    { 再利用した関数の RB エントリのうち必要な内
    容を RW に積まれている RB エントリに追加.
    このとき，RW の top から順に登録し，途中で RB
    があふれた場合は，以降，RW の bottom
    までの RB を無効化し RW から降ろす;}
  関数呼び出しは行わず次の命令に進む;
}
```

A.2.2 関数復帰命令

```
if (RW の top から順にたどり，関数に対応する
  RB を検出するまでに，ループに関する RB
```

を検出)
該 RB をすべて無効化し，RW から降ろす;
if (RW 探索中に関数に対応する RB を検出)
該 RB エントリを有効化し，RW から降ろす;
復帰命令を実行する;

A.2.3 後方分岐命令において分岐成立

if (RW の top から順にたどり，関数に対応する RB を検出)
{/* 該後方分岐命令に関する区間は登録できず */}
else if (RW の top から順にたどり，該後方分岐命令自身のアドレスと RB 中の後方分岐命令アドレスが一致する RB がない)
{/* 該後方分岐命令に関する区間は登録できず */}
else
{/* 該後方分岐命令に対応する RB を検出 */
RW の top から該 RB の手前までの RB をすべて無効化し，RW から降ろす;
該 RB エントリを有効化（かつ taken=1）し，RW から降ろす;}
【次ループ再利用検査】:
if (次ループの先頭アドレスおよび入力が RF/RB にない) {
/* 次ループは再利用不可能 */
if (次ループのための既存または空き RF/RB エントリがない) {
/* 登録を開始しない */
RW に登録されている RB をすべて無効化;
RW を空にする;
}
else {
/* 登録を開始する */
既存または空き RF/RB エントリを RW の top に積み，該 RF/RB エントリをビジー状態にする;
RB に後方分岐命令のアドレスを登録する;
if (RW が溢れた)
RW の bottom に対応する RB を無効化;
}
プログラムカウンタを条件分岐先へ進める;
}
else {
/* 次ループは再利用可能 */
RB から出力を求めレジスタ/主記憶へ書き込む;
if (他に関数/ループが RW に登録されている)
{再利用したループの RB エントリのうち必要な内容を RW に積まれている RB エントリに追加.
このとき，RW の top から順に登録し，途中で RB があふれた場合は，以降，RW の bottom までの RB を無効化し RW から降ろす;}
プログラムカウンタは，次ループ先頭ではなく，該 RB 中の taken の値に応じて，taken=1 の場合は自命令，taken=0 の場合は RB 中に記憶しておいた後方分岐命令アドレスの次へ進める;
}

A.2.4 後方分岐命令において分岐不成立

if (RW の top から順にたどり，関数に対応する RB を検出)
{/* 該後方分岐命令に関する区間は登録できず */}
else if (RW の top から順にたどり，該後方分岐命令自身のアドレスと RB 中の後方分岐命令アドレスが一致する RB がない)
{/* 該後方分岐命令に関する区間は登録できず */}
else
{/* 該後方分岐命令に対応する RB を検出 */
RW の top から該 RB の手前までの RB をすべて無効化し，RW から降ろす;
該 RB エントリを有効化（かつ taken=0）し，RW から降ろす;}
プログラムカウンタは次命令へ進める;

A.2.5 その他の命令

レジスタに対する読み書き，および，主記憶に対する読み書きを実行する．この際，RW が空でなければ，以下の手順により，読み書きデータを RW に積まれた RB に対して登録する．なお，図 2 における Args（Implicit および%i0〜%i5）を arg0 および arg[0..5] に読み替え，また，引数以外のレジスタ読み出しを登録する Regs,CC のうち，%g は grr[0..7]，%o は arg[0..7]，%l は lrr[0..7]，%i は irr[0..7]，%f は frr[0..31]，icc は icr，fcc は fcr にそれぞれ読み替える．同様に，Return Value（%i0〜%i1 および%f0〜%f1）を rti[0..1] および rtf[0..1] に読み替え，返り値以外のレジスタ書き込みを登録する Regs,CC のうち，%g は grw[0..7]，%o は rti[0..7]，%l は lrw[0..7]，%i は irw[0..7]，%f は frw[0..31]，icc は icw，fcc は fcw にそれぞれ読み換えて説明する．

A.2.5.1 汎用レジスタ READ

while (RW の top から順にたどり) {
if (該 RB が関数) {
if (該 RB がリーフ関数かつ%o0..5，または，該 RB が非リーフ関数かつ%i0..5) {

```
    if (arg[0..5].v==0)
      {arg[0..5].v=1;arg[0..5].val=読み出しデータ;}
    while (さらに RW をたどり) {
      if (該 RB が関数)
        break; /* 汎用レジスタ READ 終了 */
      if (arg[0..5].v==0) /* ループである限り */
      {arg[0..5].v=1;arg[0..5].val=読み出しデータ;}
    }
  }
  break; /* 汎用レジスタ READ 終了 */
 }
 else { /* 該 RB がループの場合 */
  if (大域レジスタ%g0..7 の場合) {
    if (grr[0..7].v==0)
      {grr[0..7].v=1;grr[0..7].val=読み出しデータ;}
  }
  else if (OUT レジスタ%o0..7 の場合) {
    if (arg[0..7].v==0)
      {arg[0..7].v=1;arg[0..7].val=読み出しデータ;}
  }
  else if (局所レジスタ%l0..7 の場合) {
    if (lrr[0..7].v==0)
      {lrr[0..7].v=1;lrr[0..7].val=読み出しデータ;}
  }
  else if (IN レジスタ%i0..7 の場合) {
    if (irr[0..7].v==0)
      {irr[0..7].v=1;irr[0..7].val=読み出しデータ;}
  }
  continue; /* 次の RW へ */
 }
}
```

A.2.5.2 汎用レジスタ WRITE

```
while (RW の top から順にたどり) {
 if (該 RB が関数) {
  if (該 RB がリーフ関数かつ%o0..5，または，
    該 RB が非リーフ関数かつ%i0..5) {
    if (arg[0..5].v==0)
      arg[0..5].v=2; /* 以後の読み出しは入力では
                        ないことを示す */
    if (%o0..1 / %i0..1 の場合) {
      rti[0..1].v=1;rti[0..1].val=書き込みデータ;
      while (さらに RW をたどり) {
        if (該 RB が関数)
          break; /* 汎用レジスタ WRITE 終了 */
        if (arg[0..1].v==0) arg[0..1].v=2;
          rti[0..1].v=1;rti[0..1].val=書き込みデータ;
        }
      }
    }
    break; /* 汎用レジスタ WRITE 終了 */
  }
  else { /* 該 RB がループの場合 */
    if (大域レジスタ%g0..7 の場合) {
      if (grr[0..7].v==0) grr[0..7].v=2;
      grw[0..7].v=1;grw[0..7].val=書き込みデータ;
    }
    else if (OUT レジスタ%o0..7 の場合) {
      if (arg[0..7].v==0) arg[0..7].v=2;
      rti[0..7].v=1;rti[0..7].val=書き込みデータ;
    }
    else if (局所レジスタ%l0..7 の場合) {
      if (lrr[0..7].v==0) lrr[0..7].v=2;
      lrw[0..7].v=1;lrw[0..7].val=書き込みデータ;
    }
    else if (IN レジスタ%i0..7 の場合) {
      if (irr[0..7].v==0) irr[0..7].v=2;
      irw[0..7].v=1;irw[0..7].val=書き込みデータ;
    }
    continue; /* 次の RW へ */
  }
}
```

A.2.5.3 浮動小数点レジスタ READ

```
while (RW の top から順にたどり) {
 if (該 RB が関数の場合)
   break; /* 登録不要 */
 else { /* 該 RB がループの場合 */
   if (frr[0..31].v==0)
     {frr[0..31].v=1;frr[0..31].val=読み出しデータ;}
   continue; /* 次の RW へ */
 }
}
```

A.2.5.4 浮動小数点レジスタ WRITE

```
while (RW の top から順にたどり) {
 if (該 RB が関数) {
   if (%f0..1 の場合) {
     rtf[0..1].v=1;rtf[0..1].val=書き込みデータ;
     while (さらに RW をたどり) {
       if (frr[0..1].v==0)
         frr[0..1].v=2; /* 以後の読み出しは入力
                           ではないことを示す */
```

```
      rtf[0..1].v=1;rtf[0..1].val=書き込みデータ;
    }
   }
   break; /* 浮動小数点レジスタ WRITE 終了 */
  }
  else { /* 該 RB がループの場合 */
   if (frr[0..31].v==0) frr[0..31].v=2;
   frw[0..31].v=1;frw[0..7].val=書き込みデータ;
   continue; /* 次の RW へ */
  }
}
```

A.2.5.5 条件コードレジスタ icc-READ

```
while (RW の top から順にたどり) {
 if (該 RB が関数の場合)
   break; /* 登録不要 */
 else { /* 該 RB がループの場合 */
   if (icr.v==0)
     {icr.v=1;icr.val=読み出しデータ;}
   continue; /* 次の RW へ */
 }
}
```

A.2.5.6 条件コードレジスタ icc-WRITE

```
while (RW の top から順にたどり) {
 if (該 RB が関数の場合)
   break; /* 登録不要 */
 else { /* 該 RB がループの場合 */
   if (icr.v==0) icr.v=2;
   icw.v=1;icw.val=書き込みデータ;
   continue; /* 次の RW へ */
 }
}
```

A.2.5.7 浮動小数点条件コードレジスタ fcc-READ

icr の代わりに fcr を用いる以外は，条件コードレジスタ icc-READ と同様である．

A.2.5.8 浮動小数点条件コードレジスタ fcc-WRITE

icr/icw の代わりに fcr/fcw を用いる以外は，条件コードレジスタ icc-WRITE と同様である．

A.2.5.9 主記憶 READ

```
if (RW の top から bottom までの RB に
 WRITE データとして登録済)
 その値を READ データとして使用する;
else if (RW の top から bottom までの RB に
 READ データとして登録済)
 その値を READ データとして使用する;
else
 キャッシュを経由して主記憶から読み込む;
while (RW の top から順にたどり) {
 if (アドレスが各 RB の SP+64 に等しい) {
   /* 構造体ポインタの読み出し */
   if (arg0.v==0)
     {arg0.v=1;arg0.val=読み出しデータ;}
 }
 else if (アドレスが LIMIT 以上 SP+92 未満)
   /* 局所変数なので登録不要 */
 else if (WRITE データとして登録済の場合)
   /* 上書きされたあとの READ なので登録不要 */
 else if (READ データとして登録済の場合)
   /* 登録済なので登録不要 */
 else {
   /* READ データとしての登録が必要 */
   RF に主記憶 READ アドレスを確保;
   対応する RB エントリに，有効バイトマスクと
   ともに READ データを登録;
   if (RF に主記憶 READ アドレスを確保できず) {
     その RW エントリから bottom までに
     対応する RB エントリをすべて無効化;
     break; /* 登録打ち切り */
   }
 }
}
```

A.2.5.10 主記憶 WRITE

```
キャッシュを経由して主記憶に書き込む;
if (ベースレジスタが 14（%sp）かつ
 オフセットが 92 以上を検出)
 第 7 引数を検出したことを記憶;
while (RW の top から順にたどり) {
 if (アドレスが各 RB の SP+64 に等しい)
   {if (arg0.v==0) arg0.v=2;}
 else if (アドレスが LIMIT 以上 SP+92 未満)
   /* 局所変数なので登録不要 */
 else if (WRITE データとして登録済の場合)
   登録データを更新する;
 else {
   RF に主記憶 WRITE アドレスを確保;
   対応する RB エントリに，有効バイトマスクと
   ともに WRITE データを登録;
   if (RF に主記憶 WRITE アドレスを確保できず) {
     その RW エントリから bottom までに
```

```
      対応する RB エントリをすべて無効化;
      break; /* 登録打ち切り */
    }
  }
}
```

(平成 14 年 12 月 23 日受付)
(平成 15 年 4 月 16 日採録)

中島　康彦（正会員）

1963 年生．1986 年京都大学工学部情報工学科卒業．1988 年同大学院修士課程修了．同年富士通（株）入社．スーパコンピュータ VPP シリーズの VLIW 型 CPU，命令エミュレーション，高速 CMOS 回路設計等に関する研究開発に従事．工学博士．1999 年京都大学総合情報メディアセンター助手．同年同大学院経済学研究科助教授，現在に至る．2002 年より（兼）科学技術振興事業団さきがけ研究 21（情報基盤と利用環境）．計算機アーキテクチャに興味を持つ．IEEECS，ACM 各会員．

津邑　公暁（正会員）

1973 年生．1996 年京都大学工学部情報工学科卒業．1998 年同大学院工学研究科情報工学専攻修士課程修了．2001 年同大学院情報学研究科博士後期課程単位取得退学．同年より同大学経済学研究科助手，現在に至る．計算機アーキテクチャ，並列処理応用，脳型情報処理等に興味を持つ．人工知能学会，日本神経回路学会各会員．

五島　正裕（正会員）

1968 年生．1992 年京都大学工学部情報工学科卒業．1994 年同大学院工学研究科情報工学専攻修士課程修了．同年より日本学術振興会特別研究員．1996 年京都大学大学院工学研究科情報工学専攻博士後期課程退学，同年より同大学工学部助手．1998 年同大学院情報学研究科助手．高性能計算機システムの研究に従事．2001 年情報処理学会山下記念研究賞，2002 年同学会論文賞受賞．IEEE 会員．

森　眞一郎（正会員）

1963 年生．1987 年熊本大学工学部電子工学科卒業．1989 年九州大学大学院総合理工学研究科情報システム学専攻修士課程修了．1992 年同大学院総合理工学研究科情報システム学専攻博士課程単位取得退学．同年京都大学工学部助手．1995 年同助教授．1998 年同大学院情報学研究科助教授．工学博士．並列/分散処理，可視化，計算機アーキテクチャの研究に従事．IEEE，ACM 各会員．

富田　眞治（正会員）

1945 年生．1968 年京都大学工学部電子工学科卒業．1973 年同大学院博士課程修了．工学博士．同年京都大学工学部情報工学教室助手．1978 年同助教授．1986 年九州大学大学院総合理工学研究科教授，1991 年京都大学工学部教授，1998 年同大学院情報学研究科教授，現在に至る．計算機アーキテクチャ，並列処理システム等に興味を持つ．著書「並列コンピュータ工学」(1996)「コンピュータアーキテクチャ第 2 版」(2000) 等．電子情報通信学会，IEEE，ACM 各会員．平成 7，8 年度，10，11 年度本会理事．平成 13，14 年度同関西支部長．