関数値再利用および並列事前実行による高速化技術

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[††]

SPARC Application Binary Interfaceに従って記述されたプログラムについて，コンパイラによる専用命令の埋め込みを必要とすることなく，ハードウェアによる関数レベルの値再利用および並列事前実行が可能であることを述べる．また，Stanfordベンチマークでは最大70%，SPEC95ベンチマークでは最大30%のサイクル数を削減できることを示す．

A Speedup Technique with Function Level Value Reuse and Parallel Precomputation

Yasuhiko Nakashima,[†] Katsuya Ogata,[††] Shingo Masanishi,[††]
Masahiro Goshima,[††] Shin-ichiro Mori,[††] Toshiaki Kitamura[†††]
and Shinji Tomita[††]

This paper describes how to apply function-level value-reuse and parallel precomputation against the programs based on SPARC application binary interface without any additional instructions. We show the maximum eliminated cycles reaches to 70% against Stanford-integer and 30% against SPEC95 benchmark programs respectively.

1. はじめに

値再利用とは，一連の命令列において，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．現在，数多くの研究が行われている値予測および投機的実行が，莫大な命令の投機および投棄を行うのに対し，新たに提案されている値再利用は，実行すべき命令列そのものを削減できるという点で，従来とは発想の異なる高速化技術である．ただし，プリフェッチ機構のないキャッシュと同様，過去の実行結果を登録するだけの単純な値再利用では，入力が単調に変化する場合に効果がない．

我々は，コンパイラによる専用命令の埋め込みを必要とせず，既存ロードモジュールの実行において入れ子を含む関数値を再利用するとともに，将来実行される関数およびパラメータを予測し，事前に実行しておくことにより高速化を図る基礎的技術の確立をめざしている．事前実行機構は，再利用表（メモリ）自身がパラメータの時系列変化から将来のパラメータを予測する機構と，複数の命令処理装置（プロセッサ）が予測パラメータに基づいて関数を実行し結果を再利用表に登録する機構の組合せにより構成される．

本稿の前半では，SPARC ABI（Application Binary Interface）[1]に従って記述された，ある一定の条件を満たすプログラムに対して，専用命令を追加することなく関数値再利用を適用できることを示し，再利用機構および事前実行機構の詳細について述べる．後半では，StanfordおよびSPEC95ベンチマークを用いた評価を行う．

2. 関連研究

最近では，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

† 京都大学大学院経済学研究科
Graduate School of Economics, Kyoto University
†† 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University
††† 広島市立大学情報科学部
Faculty of Information Sciences, Hiroshima City University

告されている[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，一般的に，1）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；2）誤った予測に基づく一連の演算結果をすべて無効化する必要があるため，一度に投機的実行できる命令数を多くするには，相応のハードウェアコストを要する；3）命令間の依存関係が多いほど，多重に投機的実行をする必要がある；といった複雑さが生じる．

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，値再利用[7)~10)]（以下，再利用と略する）は，プログラムの一部分に関する入力値および出力値を再利用表に登録しておく．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，1）入力値さえ一致すれば，実行結果を検証する必要がない；2）入力値および出力値の総数によってのみ，ハードウェアコストが決定され，省略可能な命令列の長さを制約しない；3）命令間の依存関係の多少は，再利用機構の複雑さに影響を与えない；ことである．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)]．

ただし，近年報告されている再利用の具体的実現方法[13),14)]は，プロセッサに専用命令を追加し，コンパイラが再利用を行うための命令列を生成することを前提としている．これは，プロセッサが動的かつ効率良く基本ブロックを切り出すことが難しく，簡単化のためには，コンパイラが基本ブロックの範囲をハードウェアに伝達しなければならないためである．残念ながら，専用命令を前提とする場合は，既存ロードモジュールを高速化できない．

命令レベルの投機的実行と値再利用を比較した研究[15)]では，値再利用の適用範囲は狭いものの，予測失敗時のペナルティも考慮した場合，値再利用のほうが有利であることが報告されている．引数，局所変数，大域変数の区別が容易である Java 仮想マシンおよび SpecJVM98 を用いた研究[16)]では，Last Value Prediction による投機的実行の場合 3.8%から 29.1%（平均 17.0%），メソッド単位の値再利用の場合 1.1%から 47.0%（平均 16.7%）のサイクル数を削減できること，値再利用の効果はプログラムによる偏りが大きいこと，予測失敗を考慮すると，やはり値再利用のほうが有利であることが報告されている．

両者を組み合わせた方法としては，コンパイラが値再利用区間の切り出しを行い，実行時に再利用可能である場合には再利用を行い，再利用不可能である場合には再利用区間の出力を予測して後続区間の実行を投機的に開始する研究が報告されている[17)]．

以上のような関連研究に対し，本稿における提案の特長は，1）Java 仮想マシン以外の一般的なプロセッサの既存ロードモジュールに対しても値再利用を適用できること；2）値再利用区間を特定するために，コンパイラにより埋め込まれた専用命令ではなく ABI を利用すること；3）値再利用区間が入れ子の場合にも対応できること；4）値再利用区間を対象とする事前実行は，一般的な投機的実行と異なり事後検証の必要がないため，投機失敗時のミスペナルティが生じないこと；である．

3. SPARC ABI に基づく再利用

既存ロードモジュールに対して再利用を適用するには，命令列から，入力と出力を明確に特定できる命令区間を切り出す必要がある．さらに，命令区間が多くの命令を含むことが望ましい．このようなことから，我々は，関数を再利用の単位とした．引数を特定するために，プログラムが，SPARC ABI に規定されている以下の条件を満たすと仮定した．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．

- スタック上の有効データは%sp 以上の範囲である．
- %sp から 16 ワードはレジスタ退避空間であり関数の入出力には関連しない．
- %sp+64 の 1 ワードは構造体を返り値とするための暗黙的引数である．
- %sp+68 から 6 ワードは引数の一時退避空間であり関数の入出力には関連しない．
- レジスタ%o0~5，および，%sp+92 以上に関数への明示的引数が格納される．

さらに，大域変数と局所変数（フレーム内変数）を区別するために，一般的に，実行時のデータサイズとスタックサイズにはそれぞれ上限が設けられることを利用し，以下を仮定した．

- 大域変数は Limit（実行時の固定アドレスを仮定）未満の範囲に配置される．
- %sp が Limit 未満になることはない．
- Limit 以上%sp 未満のデータは無効．

以上の条件を満たしながら，関数 A が関数 B を呼び出し，さらに，関数 B が関数 C を呼び出す場合（以後 A，B，C と略する）の，引数およびフレームの概

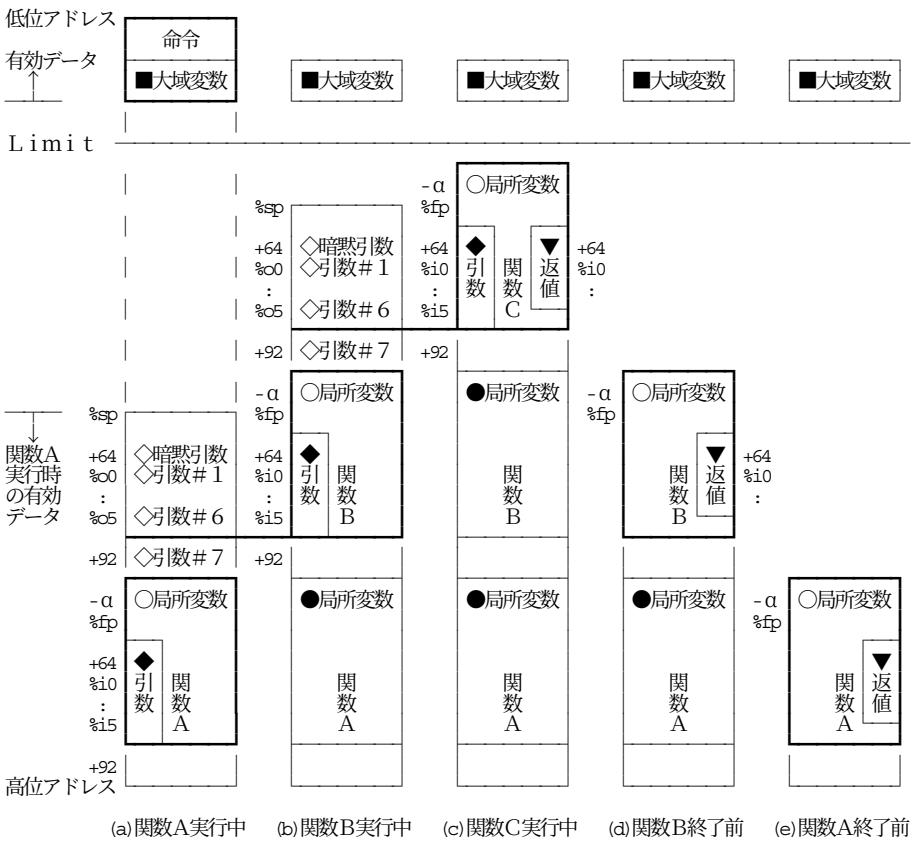

図 1 引数およびフレームの概略
Fig. 1 Overview of parameters and frames.

略を**図 1** に示す．

(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 への明示的な引数は，先頭の 6 ワードがレジスタ%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 が使用された場合，この領域は B への第 7 引数，すなわち B の局所変数である．

(b) は B 実行中の状態である．A と同様に大域変数（■）および引数（◆）を入力とする．ポインタを通じて他の大域変数や A の局所変数（●）も入力となりうる．B の局所変数には，引数の先頭 6 ワードのアドレスを扱うために必ず確保される%fp+68〜91，および，第 7 ワード以降が存在する場合に確保される%fp+92 以上の領域が含まれる．ただし，%fp 相対アドレスにより参照されるとは限らないため，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．(a) においてオペランド%sp+92 以上が出現した場合に，次に呼び出される B に第 7 ワード以降が存在すると考える．出現頻度が低いと予想されることおよび簡単のために，第 7 ワード以降を検出した関数は再利用の対象外とする．

(c) はリーフ関数 C を実行している状況である．C の入力は，大域変数，および，save 命令の有無によりレジスタ%i0〜5 または%o0〜5，同様にアドレス%fp+α または%sp+α ($\alpha \geq 0$) に格納されている引数である．一方 C の出力は，大域変数，および，save 命令の有無によりレジスタ%i0〜1 または%o0〜1 に格納される返り値である．なお，返り値が浮動小数点数の場合は%f0〜1，構造体の場合は%fp+64 または%sp+64 に格納されている構造体先頭アドレスへ書き込まれる．ポインタを通じて，大域変数および A，B を含む上位関数内の局所変数も入出力となりうる．(d) および (e) は同様に B および A の終了直前の状況である．

さて，(b) において，C に対する入力が既知であり，対応する出力が計算済みである場合，再利用により (c) を省略できる．ただし，(b) の時点では C の局所変数が場所として存在しないため，再利用の際に C の局

所変数を参照してはならない．C 実行時に C の入出力として登録すべきフレーム上データは A，B の局所変数である．この区別には，前述のように引数の第 7 ワード以降を扱わない場合，A，B の局所変数が (b) における%sp+92 以上に対応することを利用する．

同様に，(a) において，B に対する入力が既知の場合，(b)，(c)，(d) を一度に省略できる．B 実行時に B の入出力として登録すべきフレーム上データは A の局所変数であり，途中の C 実行時に B の入出力として登録すべきフレーム上データも A の局所変数である．

以上のように，同じく C の実行中であっても，どの関数を登録中であるかにより，B の局所変数が入出力に含まれるか否かが異なる．この区別には，B の局所変数が，(a) では%sp+92 未満，(b) では%sp+92 以上であることを利用する．すなわち，登録開始時点の%sp を記憶しておくことにより，C を実行中に，B, C それぞれの関数として登録すべき入出力を特定でき，複数レベルの登録を同時に行うことができる．

4. 再利用機構の構成と動作

前述した再利用を実現するための再利用表の論理構成を**図 2** に示す．再利用表は，再利用ウィンドウ（RW），関数管理表（RF），本体（RB）からなる．RW は現在実行中かつ登録中である関数呼び出しの入れ子関係を表現しており，各々の関数呼び出しに対応する RF および RB のエントリを指している．RF の各エントリは互いに異なる関数に対応しており，V = 有効エントリを表示；REF = エントリ入れ換えのヒント；関数アドレス = 関数の先頭アドレス；Read = 読み出しアドレス（4 の倍数）；Write = 書き込みアドレス（4 の倍数）；から構成される．RB は RF の各エントリに対応する複数エントリがブロック化されており，V = 有効エントリを表示；REF = エントリ入れ換えのヒント；%sp = 前述の関数呼び出し時の%sp；引数 = 有効エントリを示す V および入力値；Read = RF の各 Read アドレス 4 バイトの有効バイトを示すマスクおよび入力値；Write = 同じく各 Write アドレスに関するマスクおよび出力値；返り値 = 汎用レジスタまたは浮動小数点レジスタに格納される出力値；から構成される．%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 アドレスの内容と RB の複数エントリを一度に比較するために，Read アドレスは RF が一括管理し，RB はマスクおよび値のみを管理する．

次に，RF の 1 エントリ分に対応する再利用表の物理構成を**図 3** に示す．引数および主記憶読み出しデータと RB の内容との比較には CAM を利用する．関数呼び出しが再利用可能か否かを判定するためには，まず，関数アドレスが一致する RF エントリ（1）を特定し，次に，引数がすべて一致する RB エントリ（2）を特定する．さらに，少なくとも 1 つのマスクが有効である Read アドレス（3）を RF から順に選択し，主記憶から読み出した各 4 バイトのデータ（4）と RB の対応する列に属するすべての値との比較を行う．マスクが有効であるすべての値が一致したとき，書き込みデータ（5）および返り値（6）を主記憶およびレジスタへ格納する．引数の予測機構については後述する．

続いて，命令実行手順について詳述する．

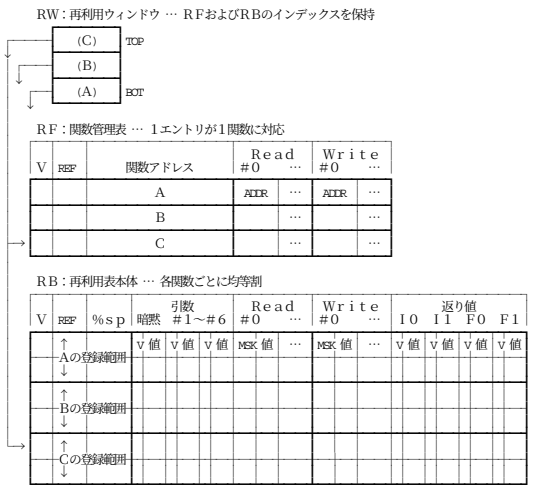

図 2　再利用表の論理構成
Fig. 2　Logical structure of reuse-buffer.

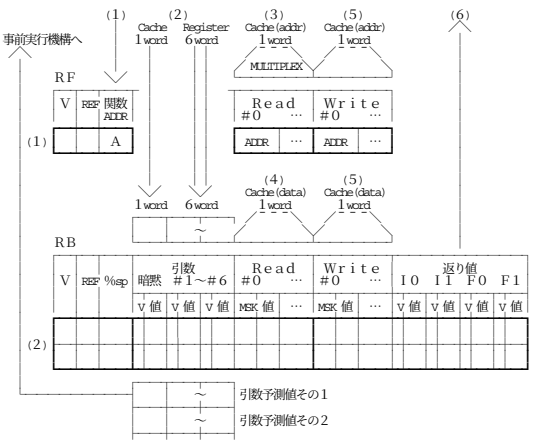

図 3　再利用表の物理構成
Fig. 3　Physical structure of reuse-buffer.

4.1 関数呼び出し

関数呼び出しの契機は，call 命令または%o7 へ現 PC を書き込む jmpl 命令である．RF を参照し，前述の手順により再利用を試みる．再利用した関数（たとえば C）を含む上位関数 A，B が登録中である場合，C の再利用を行った RB エントリの主記憶参照に関する内容を A，B それぞれの登録中 RB エントリに追加する．ただし前述のように，A，B それぞれの呼び出し時における%sp+92 未満に対する参照は対象外とする．

ところで，以上の方法により入れ子の関数を登録すると，より上位の関数（たとえば B）において登録すべき主記憶参照箇所が RB の各エントリが収容可能な数を超える．この場合は，制限を超えた関数を含む上位関数 A，B それぞれに対応する登録中 RB エントリおよび RW エントリを無効化し，引続き登録可能な C のみを RB および RW に残して登録を続行する．

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

4.2 関数本体の実行

関数値の再利用ができない場合，関数本体の実行を開始する．ただし，以後の再利用に備えて，各命令の実行と同時に，再利用に必要な関数の入力および出力を RB に登録していく．

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

【レジスタ参照】 レジスタ%i0〜5（save 命令をともなわない関数では%o0〜5）には明示的引数の先頭 6 ワードが格納される．関数内においてまず読み出しを行った対象を引数として RB へ登録する．まず書き込みを行った対象は引数ではないため比較対象外として登録する．ただし，%i0〜1（save 命令をともなわない関数では%o0〜1），および，%f0〜1 への書き込みは，返り値の可能性があるので返り値としても RB へ登録する．その他のレジスタ参照は，関数への入力から得られる中間結果であるため，登録は不要である．

【Limit 以上，呼び出し時%sp+64 未満】 前述したように無効領域または関数の局所変数であり，RB への登録は不要である．

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

【呼び出し時%sp+64 以上】 %sp+68〜91 は局所変数である．%sp+92 以上への書き込みを検出した場合，引数の第 7 ワード以降が存在するため，現関数から復帰する前に次の関数が呼び出された場合，次の関数を含む上位関数の登録を中止する．

【その他の主記憶参照】 上記以外の場合，大域変数または上位関数内局所変数であり，関数に対する入出力として登録が必要である．RW に登録されているすべての RF/RB エントリについて，以下を行う．

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

4.3 復 帰

関数からの復帰の契機は，%g0 へ現 PC を書き込み，%o7 または%i7 の内容へ無条件分岐する jmpl 命令である．登録中の RB エントリを有効にし，最後に RW から該当エントリを削除する．

5. 並列事前実行機構の構成と動作

これまでに述べた単純な再利用では，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，通常どおり再利用を行いながら命令列を実行するプロセッサ（Main Stream Processor：以下 MSP と略する）とは別に，先行して RB エントリへの登録を行うプロセッサ（Shadow Stream Processor：以下 SSP と略する）を複数個設けることにより，さらなる高速化が可能ではないかと考えた．

並列事前実行機構の概要を**図 4** 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．R はレジスタやキャッシュからの読み出しおよび RB

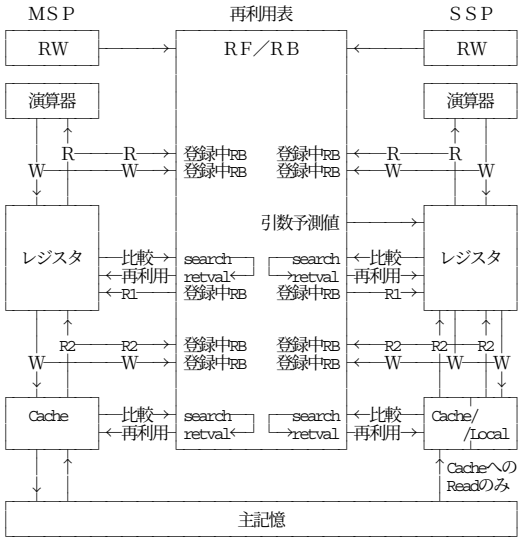

図 4 並列事前実行機構
Fig. 4 Logical structure of precomputation.

への登録，W はレジスタやキャッシュへの書き込みおよび RB への登録を表している．R1 は，すでに RB へ登録されているアドレスからの読み出しは，RB からデータを直接得ることにより，キャッシュを汚さない工夫である．R2 は，RB に未登録のアドレスは，従来どおりキャッシュを参照することに対応する．事前実行のために必要となる引数予測値は，MSP や SSP とは別に，RF の各エントリごとに設けた小さなプロセッサが RB の使用履歴に基づいて予測し，図 3 に示す引数予測値格納領域に，一定数を用意する想定である．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]を行っている．なお，$B + D$ に基づく計算は MSP がすでに開始していると考える．SSP が N 台の場合，用意する予測値は，$B + D * 2$ から $B + D * (N + 1)$ の範囲としている．

4 章に述べたように，MSP はレジスタや主記憶に対する参照を RB に登録しながら通常どおり命令列を実行し，可能であれば関数呼び出し時に再利用を行う．これに対し SSP は，特定の関数のみを実行し，SSP が有する局所メモリおよび RB への読み書きは行うものの，キャッシュおよび主記憶への書き込みは行わない．MSP と異なる点は以下のとおりである．

5.1 関数の決定と引数の受け取り

過去に RB への登録を行ったことがあり，かつ，RB への登録回数が多いにもかかわらず再利用時のヒット率が低い関数を事前実行の対象とする．SSP は，対応する RB の引数予測値から 1 つをレジスタに受け取り，関数の実行を開始する．

5.2 主記憶の参照

関数フレームに対する参照には SSP ごとに設けた局所メモリを用いる．これら以外の主記憶読み出しは，MSP と共有する主記憶から，SSP ごとに設けたキャッシュを経由して行う．主記憶書き込みはいっさい行わず，書き込みアドレスおよびデータは RB に登録する．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には，その関数の実行を打ち切る．また，0 番地の参照など，実行を継続できない例外が発生した場合も，実行を打ち切る．なお，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．

5.3 事前実行の終了

jmpl 命令により，選択した関数の実行が終了した場合，MSP と同様，登録中の RB エントリを有効にし，RW から該当エントリを削除する．そして，次に事前実行する関数を選択し，以上を繰り返す．

5.4 RB エントリの入れ替えアルゴリズム

MSP だけでは再利用できず，パラメータの変化が予測できないために SSP の効果もない関数や，入出力データがきわめて多い，システムコールを含む，入れ子のレベルが RW の容量を超えて深すぎるなど，RB へ登録できない関数を除くと，再利用可能な関数は，入力の与えられ方によって，以下の 3 つに分類できると予想される．

【第 1 種関数】 パラメータの変化が小さく，MSP のみの再利用でも十分高速化が可能なもの．以下に示す，Stanford ベンチマーク Towers 中の Push 関数があげられる．引数 i に 1〜14，s に 1〜3 の値が繰り返し渡される．

```
Push(i, s) int i, s;
{ int errorfound, localel;
  errorfound = false;
  if (stack[s] > 0)
    if (cellspace[stack[s]].discsize ≤ i) {
      errorfound = true;
      Error("disc size error");
    }
  if (!errorfound) {
    localel = Getelement();
    cellspace[localel].next = stack[s];
    stack[s] = localel;
    cellspace[localel].discsize = i;
  }
```

```
}
```

【第 2 種関数】 パラメータの変化が大きく，MSP のみでもある程度高速化が可能であるものの，SSP によりさらに高速化できるもの．同じく Puzzle 中の Place 関数があげられる．引数 i に 0〜12，j に 73〜353 の値が繰り返し渡される．

```
Place (i, j) int i,j;
{ int k;
  for (k=0; k≤piecemax[i]; k++)
    if (p[i][k]) puzzl[j+k] = true;
  piececount[class[i]] = piececount[class[i]] − 1;
  for (k=j; k≤size; k++)
    if (!puzzl[k]) return (k);
  return (0);
}
```

【第 3 種関数】 パラメータが単調変化し，MSP だけではまったく効果がなく，SSP により高速化できるもの．同じく Intmm 中の Rand 関数や Innerproduct 関数があげられる．Rand 関数は引数がない代わりに，入力となる大域変数が単調変化する．Innerproduct 関数の引数 result，a，b は不変であり，（row，column）が（1，1）〜（40，40）の範囲を単調変化する．

```
Rand ()
{ seed = (seed ∗ 1309 + 13849) & 65535;
  return (seed);
}
Innerproduct(result,a,b,row,column) int ∗result;
  int a[rowsize+1][rowsize+1],b[rowsize+1][rowsize+1];
  int row,column;
{ int i;
  ∗result = 0.0;
  for (i=1; i≤rowsize; i++)
    ∗result = ∗result + a[row][i] ∗ b[i][column];
}
```

さて，図 3 に示すように，RB の各エントリにカウンタ（REF）を設けている．MSP が再利用した場合に 1 を加算し，また，RF エントリの参照が一定回数に達したときに，その RF エントリに属する RB の全カウンタを 0 に初期化する．この機構では，REF 値最小のエントリを追い出すことが LRU に相当し，REF 値最大のエントリを追い出すことが FIFO に相当する．また，ある範囲の RB エントリに関する REF 値の論理和が 0 である場合に，その範囲の RB エントリはまったく再利用されていないと判断する．MSP が登録した RB エントリがまったく再利用されない場合，第 3 種関数であると判断し SSP の実行対象に加える．一度 MSP により再利用されたエントリは再び再利用されることはないと考え，SSP は FIFO に従って RB エントリの入れ替えを行う．逆に，SSP が登録した RB エントリがまったく再利用されない場合，第 1 種関数であると判断し SSP の実行対象から除く．MSP，SSP いずれの登録エントリも再利用される場合には，第 2 種関数であると判断し，SSP による実行を継続する．このような状況に対応するために，RB を 2 分割し，MSP，SSP それぞれが使用する領域を分けたうえで，MSP はつねに LRU，SSP はつねに FIFO に基づいて RB エントリを入れ替えている．

6. Stanford ベンチマークを用いた評価

評価には，再利用機構を搭載した単命令発行の SPARC-V8 アーキテクチャ・シミュレータを用いた．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．測定対象は Stanford ベンチマークを 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である．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 5** に，再利用を適用しない場合の総実行命令ステップ数（複数サイクルを要するものはサイクル数を加算）を 1 とした場合の，各構成における総実行命令ステップ数の比（縦軸）を示す．横軸は，MSP1 台を含むプロセッサ数（1，2，4，8）/RF あたりの RB エントリ数（32，64，128，256），また，凡例の MA は，RF に登録可能

表 1 シミュレータの諸元
Table 1 Simulation parameters.

D-Cache 容量	64 Kbyte
ラインサイズ	64 byte
ウェイ数	4
Cache ミスペナルティ	20 cycle
Register-Window	6 set
Window ミスペナルティ	20 cycle/set
ロードレイテンシ	2 cycle
整数乗算〃	8 cycle
整数除算〃	70 cycle
浮動小数点加減乗算〃	4 cycle
単精度浮動小数点除算〃	16 cycle
倍精度浮動小数点除算〃	19 cycle
RW の最大深さ	6
RF のエントリ数	16
RB(引数) ⇔ Register 比較	1 cycle
RB(Read) ⇔ Cache 比較	4 byte/cycle
RB(Write) ⇒ Cache 書き込み	4 byte/cycle
RB(返り値) ⇒ Register 書き込み	1 cycle
SSP 局所メモリ容量	64 Kbyte

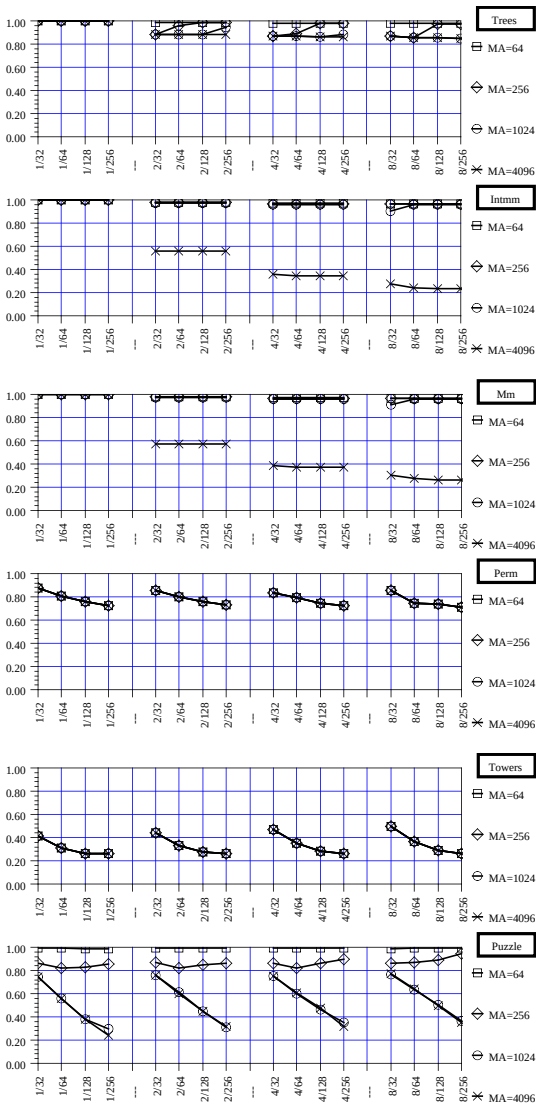

図 5 MSP の実行ステップ数（Stanford）
Fig. 5 Steps of MSP (Stanford).

な Read/Write アドレス各々のエントリ数（64，256，1024，4096）である．なお，FFT および Bubble は乱数発生以外に関数呼び出しがなくステップ数の減少が最大 4%程度，Queens および Quick は最大 10%程度であるため，省略している．FFT，Bubble，Quick における減少は，第 3 種関数による乱数発生が SSP により高速化されることによる．第 3 種関数が多い Trees，Intmm，Mm では，MSP のみでは再利用の効果がないものの，MA = 4096 の場合には SSP により 15〜75%のステップ数を削減している．Intmm および Mm の最内ループが要素数 40 の積和を求める関数を呼び出しており，この部分の並列事前実行が寄与している．第 1 種関数の再帰呼び出しが多い Perm，Towers，Puzzle では，MSP のみでも 30〜75%のステップ数を削減しており，SSP を投入しても効果がない．ところで，Puzzle の MA = 256 において，RB エントリ数の増加につれて再利用の効果が下がっている．これは，後述のように，Read/Write アドレス数がきわめて多い RB エントリが存在することに起因する．RB エントリの総数を増やしたために，このようなエントリが長期間残ると，RF に割り当てられているアドレス領域が枯渇し，かえって有効な RB エントリが減少する．

図 6 は，プロセッサ数 = 8，RB エントリ数 = 256，MA = 4096 の構成（RB 全体は約 256 M バイト）において，再利用可能であった関数呼び出しを入れ子の深さ（L1〜L6）ごとに分類し，再利用しない場合のステップ数に対する削減ステップ数の比（上端 = 0）を左目盛の折れ線グラフ，また，引数（args），Read アドレス（mmrs），Write アドレス（mmws），返り値（retvs）の平均個数を右目盛の棒グラフにより示したものである．なお，削減ステップ数が 0 であった深さは省略している．上端を超える Puzzle-L2/L3 は mmrs = 174/341，mmws = 1/5，retvs = 1/1 である．深さ 4 以上では削減ステップ数の比がほぼ 0 となり，Stanford では入れ子の深さは 3 までを考慮すればよいこと，引数の個数はほぼ 2 以下であること，また，削減ステップ数の比を 2%以上に限っても，Read/Write アドレスの平均個数は 44/8 を超えないことが分かる．

図 7 は，図 6 の折れ線グラフに，1 回の関数呼び出しあたりの平均削減ステップ数を重ねたものである．上端を超える Puzzle-L2/L3 は 1633/3998 ステップである．削減ステップ数の比を 2%以上に限っても，300 ステップ程度の関数呼び出しを再利用している．

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 8** は，命令ステップ数（exec）に，表 1 に示した RB(Read) ⇔ Cache 比較（test），RB(Write) ⇒ Cache 書き込み（write），キャッシュミス（cache），レジスタウィンドウミス（window）の各オーバヘッドを加えたサイクル数の内訳である．左側棒グラフは再利用を適用しない場合，右側棒グラフは再利用を適用した場合の内訳である．Towers では再利用によりレジスタウィンドウミスが低減されている．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 比較を 4 byte/cycle から 8 byte/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．

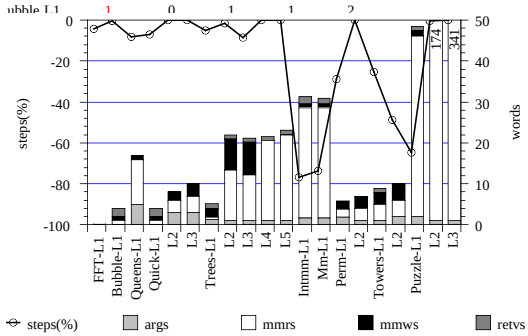

図 6 関数の深さごとの入出力数（Stanford）
Fig. 6 Relation between depth and I/O (Stanford).

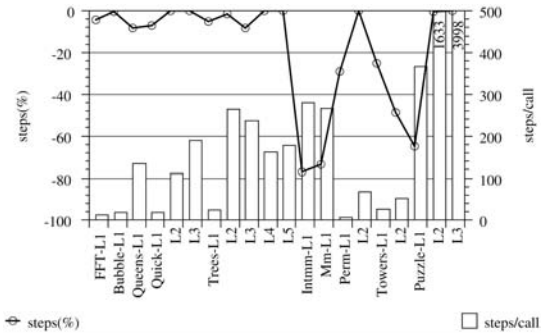

図 7 関数の深さごとの再利用ステップ数（Stanford）
Fig. 7 Relation between depth and reused steps (Stanford).

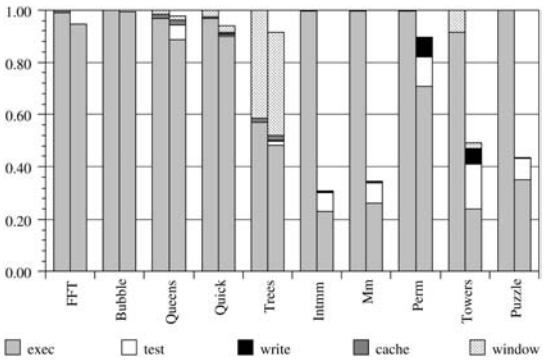

図 8 MSP の実行サイクル数（Stanford）
Fig. 8 Cycles of MSP (Stanford).

7. SPEC95 ベンチマークを用いた評価

Stanford ベンチマークによる評価では，第 1 種関数に対する再利用の効果は，RB エントリ数 = 256 程度で鈍化し，MA = 1024 と 4096 とで大差ないこと，また，第 3 種関数に対する事前実行の効果は，プロセッサ数 = 1 および 2 の比較からおおむね確認できること

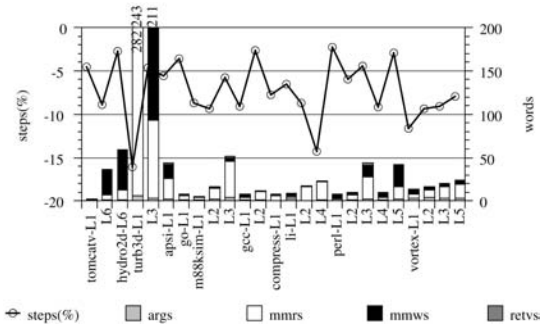

図 9 関数の深さごとの入出力数（SPEC95）
Fig. 9 Relation between depth and I/O (SPEC95).

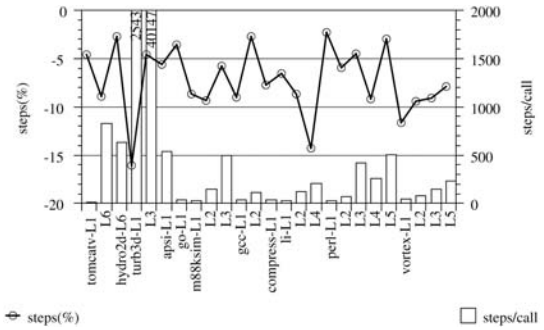

図 10 関数の深さごとの再利用ステップ数（SPEC95）
Fig. 10 Relation between depth and reused steps (SPEC95).

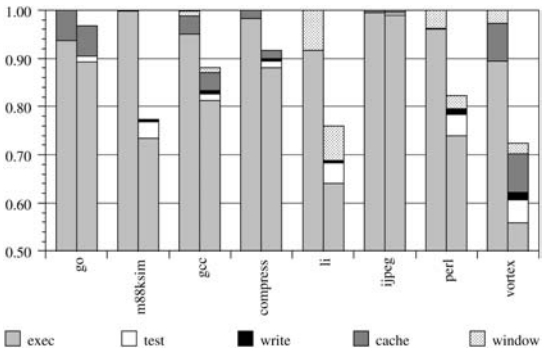

図 11 MSP の実行サイクル数（SPECint95）
Fig. 11 Cycles of MSP (SPECint95).

が分かった．SPEC95 では当然状況が異なると予想されるものの，シミュレーション時間を短縮するために，本結果をふまえて，プロセッサ数 = 1 および 2，RB エントリ数 = 256，MA = 1024 の構成（RB 全体は約 64 M バイト）を仮定し，test 入力による評価を行った．ただし，SSP1 台の投入による効果は最大 3%程度であったためプロセッサ数 = 1 の測定結果を示す．Stanford の結果と同様，削減ステップ数が 2%未満であった深さは省略している．**図 9** において上端を超える

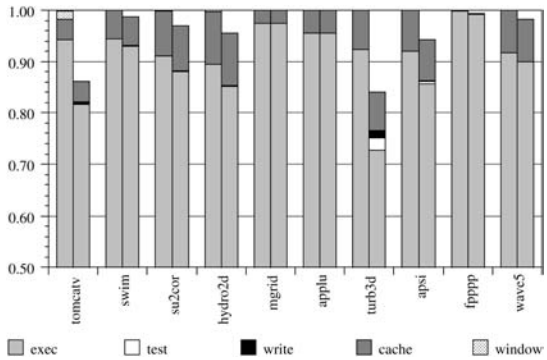

図 12 MSP の実行サイクル数（SPECfp95）
Fig. 12 Cycles of MSP (SPECfp95).

turb3d-L1 は mmrs = 282, mmws = 243, retvs = 2 である．Stanford と異なり，深さ 4〜6 の関数においても 3〜14%のステップ数を削減している．**図 10** において上端を超える turb3d-L1/L3 は 2543/40147 ステップである．深さ 4〜6 の関数においても 200〜800 ステップ程度の関数呼び出しを再利用している．**図 11** および**図 12** に示すように，オーバヘッドを加えた評価でも，m88ksim, li, perl, vortex, tomcatv, turb3d において 20%弱〜30%のサイクル数を削減できている．

8. おわりに

本稿では，入れ子を含む関数値の再利用，および，事前実行による高速化手法を提案した．値再利用に関する制約には，1）再利用手順および連想検索に直接関わる質的制約；と，2）チップ面積および配線遅延に直接関わる量的制約；がある．サイクル数ベースで評価するために，前者に関しては，現実的な仮定を行い，RF ごとに RB をグループ化することにより連想検索の上限を 256 エントリまでとした．一方，理想的条件に近付けるために，シミュレーションが可能な限り RB を大きくし，また，制約の影響を調べるために，RB に登録可能なエントリ数やアドレス数を変化させた．このような仮定のもとで，Stanford では，プログラムにより再利用と事前実行のいずれが有効であるかが明確に分かれたのに対し，SPEC95 では，プロセッサ数 2 までの範囲では再利用の効果のみが現れた．これは，SPEC95 では第 2/3 種関数の出現頻度が低いためと考えられる．1）ストライド予測以外の方法により事前実行の効果をあげられるかどうか；2）基本ブロックなど，明示的な関数呼び出しではない部分を関数として切り出すことにより，効果をあげられるかどうか；3）画像など同じパターンが繰り返し現れる可能性が高いデータについて，再利用機構を前提とした処理を行った場合に，どの程度の効果が期待できるか；4）高速かつ大容量の RB をいかに実現するか；などが今後の課題である．

謝辞 本研究の一部は文部科学省科学研究費補助金（基盤研究（B）(2）課題番号 13480083，特定領域研究「情報学」課題番号 13224050）による．

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 MICRO*, 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. Comput.*, Vol.50, No.1, pp.67–82 (2001).
5) Sohi, G.S. and Roth, A.: Speculative Multithreaded Processors, *IEEE Comput.*, Vol.34, No.4, pp.66–73 (2001).
6) 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).
7) Sodani, A. and Sohi, G.S.: Dynamic Instruction Reuse, *24th ISCA*, pp.194–205 (1997).
8) Huang, J. and Lilja, D.J.: Exploiting Basic Block Value Locality with Block Reuse, *5th HPCA* (1999).
9) González, A., Tubella, J. and Molina, C.: Trace-Level Reuse, *ICPP* (1999).
10) 山田克樹，中島康彦，富田眞治：投機的手法を用いたデータ再利用による Java 仮想マシンの高速化，情報処理学会研究報告 ARC，Vol.139, No.29, pp.169–174 (2000).
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. Comput., Vol.49, No.4, pp.331–347 (2000).
15) Sodani, A. and Sohi, G.S.: Understanding the Differences Between Value Prediction and Instruction Reuse, *31st MICRO* (1998).
16) 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，Vol.41, No.SIG5 (HPS1), pp.13–18 (2000).
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). www.sparc.com/standards/

(平成 14 年 1 月 22 日受付)
(平成 14 年 4 月 25 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．計算機アーキテクチャに興味を持つ．IEEECS，ACM 各会員．

緒方 勝也（学生会員）
1978 年生．2002 年京都大学工学部情報学科卒業．現在同大学大学院情報学研究科修士課程に所属．無線ネットワークシステムに興味を持つ．

正西 申悟（学生会員）
1979 年生．2002 年京都大学工学部情報学科卒業．現在同大学大学院情報学研究科修士課程に所属．オンラインアルゴリズムに関する研究に従事．

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受賞．

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．

北村 俊明（正会員）
1955 年生．1978 年京都大学工学部情報工学科卒業．1983 年同大学大学院博士課程研究指導認定退学．同年富士通（株）入社．汎用コンピュータ，スーパコンピュータ VPP シリーズの VLIW 型 CPU，M アーキテクチャ・命令エミュレーション，米国 HAL 社において SPARC プロセッサ等の研究開発に従事．工学博士．2000 年京都大学総合情報メディアセンター助教授．2002 年広島市立大学情報科学部教授，現在に至る．計算機アーキテクチャに興味を持つ．電子情報通信学会，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．